Trabajo Fin de Grado

Grado en Ingeniería de las Tecnologías Industriales

Desarrollo de un sistema de control en tiempo real para PC-104

Autor: Joaquín García Ordóñez

Tutores: Daniel Limón Marruedo y Mario Pereira Martín

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Desarrollo de un sistema de control en tiempo real para PC-104

Autor:
Joaquín García Ordóñez

Tutor:
Daniel Limón Marruedo
Profesor titular

Cotutor:
Mario Pereira Martín

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Agradecimientos

Quiero agradecer primero a mis padres Joaquín e Inmaculada por darme la oportunidad de llegar hasta aquí y ser mi principal pilar de apoyo durante toda mi vida. A mi abuela Carmenci que se alegraba tanto como yo con cada una de las pequeñas victorias que iba consiguiendo. A mi novia Gloria por ser la energía y el amor que me ha impulsado durante los momentos más difíciles del grado. Y a mis amigos y compañeros por hacer de la vida un camino acompañado.

No pueden faltar todos los profesores que he tenido a lo largo de la carrera, en especial a aquellos que me han motivado y enseñado en la rama de automática y control. Teodoro Álamo, Daniel Rodríguez, y mi primer profesor y también tutor de este proyecto, Daniel Limón. Por su orientación, por su confianza y por abrirme en estos conocimientos: muchas gracias.

Por último, este proyecto no hubiese visto la luz si no fuese por Mario Pereira, quien me ha asistido como cotutor durante todos los meses ante cualquier dificultad y siempre ha estado disponible. Gracias por tu amabilidad, tu gran ayuda y tu interés por sacar el proyecto adelante.

Joaquín García Ordóñez

Sevilla, 2016

Resumen

La ingeniería de sistemas se ha visto muy favorecida en los últimos años gracias a los pasos agigantados a los que han evolucionado las tecnologías eléctricas y eléctronicas, además de la informática. La automatización y el control de sistemas ven cada vez menos límites en cuanto a la resolución de procesos complejos, la necesidad de potencia de cálculo o incluso la accesibilidad al hardware necesario. Pero para que exista este avance se necesita un aspecto esencial: la síntesis de la tecnología y los conocimientos que permite a los nuevos ingenieros adquirir rápidamente el estado actual para seguir mejorando poco a poco el futuro.

En este proyecto se va a realizar una síntesis de la programación de controladores con la informática industrial de tiempo real. Generalmente los microprocesadores electrónicos en los que se programa la ejecución de controladores no disponen de precisión o de garantía suficiente para sistemas de tiempo crítico debido a la incertidumbre que ocasiona el sistema operativo y otros programas o servicios que se realizan en segundo plano. Las funcionalidades de tiempo real ofrecen solución a este tipo de sistemas que necesitan un control muy estricto, preciso y ligado al tiempo.

El logro fundamental de este proyecto consiste en la realización de una librería informática en C++ que facilita un gran número de funciones y propone una estructura de programación sencilla e intuitiva para que el futuro ingeniero de control realice controladores en tiempo real sin que deba conocer ni estudiar la compleja programación informática interna.

Esta librería se ha programado en un PC-104, que es una computadora o sistema empotrado recomendado para trabajar en tiempo real. Además, tanto el sistema operativo Ubuntu como todas las aplicaciones en las que se ha apoyado el desarrollo del proyecto son software libre para maximizar la accesibilidad. También se han realizado ejemplos y programas sencillos que tienen como objetivo facilitar el entendimiento y uso de la librería a sus futuros usuarios. Por último, se han implementado controladores predictivos más complejos que integran la librería del proyecto con la herramienta matemática y de control ACADO Toolkit con el objetivo final de demostrar la utilidad del tiempo real en sistemas mayores.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

1 INTRODUCCIÓN

La ingeniería de sistemas es una rama que con frecuencia necesita estudiar campos muy diferentes del conocimiento. Mientras que el ingeniero es un especialista de ciencias matemáticas, electrónicas y de la computación, se puede encontrar un sistema en cualquier campo de la ciencia: física, biología, química, etc. El trabajo en grupo y la especialización son claves para lograr el éxito en problemas que tocan diversas áreas del conocimiento. Así como es importante presentar una solución al problema, lo es también presentar unos métodos de resolución estándares y la división de los sistemas en bloques de funciones, para que en el futuro lo que ya se haya resuelto una vez no se vuelva a inventar y se sigan contruyendo nuevos proyectos basados en los ya existentes.

La tarea de un ingeniero de control consiste en el estudio y la identificación de sistemas y su traducción al lenguaje matemático, para posteriormente diseñar un control e implementarlo al sistema por medio de herramientas electrónicas o de computación. Todos estos pasos requieren enlaces de diferentes materias de estudio y su completo entendimiento se escapa a las posibilidades del ser humano. Sin embargo, la electrónica y la computación poseen una característica que revolucionó la industria de sistemas: la posibilidad de incorporar algoritmos complejos que ya se han inventado para lograr finalidades muy sencillas de entender pero muy difíciles de programar desde cero.

Hace muchos años, la forma de implementar un controlador PID (proporcional, integrador y derivativo) de forma sencilla era conseguir un ordenador de la época y programar el algoritmo en lenguaje de ensamblador, hecho que podía resultar un reto según la complejidad del proceso y las librerías matemáticas disponibles. Actualmente, implementar un este tipo de controladores en un determinado proceso, por muy complejo que sea, consiste en el empleo de unos bloques muy sencillos de entender y manipular en un programa como Matlab. En la industria, se utilizan sistemas dedicados conocidos como embebidos o empotrados, en los que se busca explotar todos los recursos para realizar las tareas necesarias. Cuando los procesos industriales requieren múltiples tareas críticas y una gestión adecuada de ellas, surge la programación concurrente y el tiempo real.

En este proyecto se pretende dar otro pequeño pasito en la programación de sistemas de control que consiste en facilitar la inclusión de las funcionalidades de tiempo real. Un sistema de control en tiempo real añade muchísima robustez a la resolución de plantas con múltiples procesos que requieren sincronización, tiempos de respuesta críticos e incluso mucha potencia de cálculo. Un procesador que debe resolver múltiples tareas a la vez tiene un comportamiento impredecible para el programador y deriva en un mal funcionamiento, y es precisamente esto lo que la programación en tiempo real se encarga de resolver.

Un ingeniero de control entiende perfectamente este problema y sabría resolverlo si él mismo fuese el procesador de la computadora: sabría qué tareas son prioritarias de ejecutar y cuáles se podrían dejar para cuando las prioritarias terminen, sabría cuáles son los tiempos críticos de respuesta y los retrasos máximos que puede sufrir el control de un lazo, y sabría qué tareas tendría que dejar pausadas en el momento en el que salte otra de mayor prioridad.

Por desgracia, aunque los conceptos de tiempo real sean fáciles de entender, su programación llega a ser muy complicada, y su estudio previo puede llevar mucho tiempo. Y aquí es donde definimos el objetivo del proyecto, que no es solo el de lograr la programación de sistemas de control en tiempo real, sino el de crear una herramienta que facilite la programación al futuro ingeniero interesado hasta el punto de ahorrarle el estudio de la programación de las características de tiempo real. Esto es solo un primer paso en el que se ha intentado avanzar lo máximo posible, pero quedará como una herramienta libre que seguirá mejorando en el futuro.

2 ESTADO DEL ARTE

2.1 Sistemas *embebidos* o empotrados

Un sistema *embebido* (del inglés, *embedded*) es un sistema de computación diseñado específicamente para realizar unas tareas concretas. Se suelen utilizar para otorgar robustez a las funciones o procesos a los que se va a destinar, ya que el sistema estará aislado, desde el punto de vista de potencia de computación, del resto de la planta. Comúnmente, estos sistemas suelen llevar algoritmos que trabajan en tiempo real para asegurar lo máximo posible un funcionamiento correcto y robusto.

Las computadoras más habituales, al contrario que los sistemas empotrados, están diseñadas para poder cubrir una amplia gama de necesidades. Llevan por tanto mucha variedad en hardware, generalmente con más altas prestaciones, pero también requieren de un sistema operativo mucho más complejo y pesado que consume muchos recursos de por sí. Esta complejidad y uso excesivo de recursos dan lugar a errores inesperados y a una degradación física más rápida, desventaja que abre paso a los sistemas empotrados como los mejores dispositivos para realizar tareas específicas.

Otra característica de los sistemas empotrados es su compacidad. Como su finalidad son tareas específicas, suelen traer el hardware mínimo y necesario, bastante menos que las computadoras habituales. Normalmente los componentes informáticos se encuentran todos compactados en una placa base. Su diseño físico también está pensado para evitar ruidos externos y poder disipar el calor por sí solo.

En este proyecto se trabajará con el PC-104[1], que como su nombre indica es un PC que tiene 104 pines, pero estará adaptado a las características anteriormente expuestas para convertirse en un sistema empotrado estándar. El PC-104 se ha usado comúnmente para aplicaciones industriales y adquisición y tratamiento de datos. La estructura del PC-104 viene organizada en módulos que se apilan entre ellos. Se presentará en el capítulo 3 de este proyecto, donde detallaremos sus características más técnicas, así como su montaje.

2.2 Computación en tiempo real

Se entiende que un sistema o programa trabaja en tiempo real cuando este tiene una noción exacta del tiempo en cada momento, y es capaz de realizar diferentes acciones durante su ejecución en base a restricciones temporales que ha impuesto su programador. En otras palabras, un sistema es de tiempo real cuando conoce los tiempos de ejecución de cada una de sus tareas, así como los tiempos de inicio y fin, y puede tomar decisiones en base a restricciones y prioridades que se asignen a las diferentes tareas que componen al programa.

Un sistema en tiempo real se emplea para resolver problemas de computación en los que tanto los tiempos de respuesta como los tiempos de ejecución son críticos. Un ejemplo general consiste en un sistema que deba sincronizar múltiples tareas de control. Una tarea puede ser cualquier algoritmo recursivo que se ejecuta cada cierto tiempo. Pueden existir algoritmos con un periodo de repetición conocido, o pueden existir aquellos que se

[1] http://pc104.org/ [Último acceso: junio 2016]

despierten por medio de una señal externa. Algunos algoritmos son críticos, como la respuesta ante una señal de emergencia, y otros pueden ser ignorados ocasionalmente, como el control de un lazo no crítico que se repita recursivamente, no siendo relativamente grave que se salte un tiempo de muestreo.

En un problema de control sobre una planta, podemos distinguir y separar múltiples tareas: una para cada lazo de control, diferentes tareas para imprimir resultados o pintar gráficos, diferentes tareas pendientes para recibir parámetros que se reciban por un periférico e incluso tareas encargadas de hacer paros de emergencia. Todas estas tareas pueden llegar a compartir tiempo simultáneamente en el procesador, se pelearán entre ellas para obtener los recursos necesarios para ejecutarse y lógicamente afectará a la velocidad de resolución. En situaciones reales de control industrial vamos a encontrarnos con un problema de computación que presenta un gran número de tareas diferentes. Ahora bien, generalmente no se puede permitir que tareas que escriben resultados o lazos de control no importantes afecten a un lazo de control crítico. O que el control sobre una planta continúe incluso después de haber pulsado una seta emergencia, todo porque el procesador esté ocupado con tareas secundarias y no reciba el aviso.

Para resolver este problema computacionalmente es necesario emplear tiempo real. Esta solución es también conocida como programación concurrente. El sistema debe conocer y gestionar las diferentes tareas por orden de prioridad, aprovechando al máximo los recursos disponibles sin detenciones. Una solución a un problema de ejemplo puede ser la siguiente:

1. Ejecutar siempre la tarea de mayor prioridad, justo cuando se recibe su interrupción y en el menor tiempo posible.
2. Si no hay tareas de mayor prioridad, las de menor prioridad se realizan normalmente.
3. Si salta una tarea de mayor prioridad durante la ejecución de una de menor prioridad, la de menor prioridad se pausará inmediatamente y se ejecutará la de mayor prioridad.
4. Las tareas de menor prioridad sufrirán un retraso hasta que terminen las de mayor prioridad.
5. Si existen múltiples tareas de mayor prioridad y la de menor prioridad se ve retrasada hasta llegar al siguiente tiempo de muestreo, esta se descartará y no se acumulará.

La programación en tiempo real añade una capa de dificultad considerable al algoritmo, pero cuando se hace sobre sistemas dedicados se consigue mucha robustez y una solución potente a los problemas de tiempo crítico. Es por esto que se suelen combinar los sistemas empotrados con la computación en tiempo real.

2.2.1 Normas POSIX

POSIX[2] (Portable Operating System Interface) es un conjunto de normas escritas por el IEEE[3] (Institute of Electrical and Electronics Engineers) para estandarizar utilidades esenciales de sistemas operativos relacionados con UNIX. En otras palabras, la programación bajo los estándares de POSIX asegura la funcionalidad en sistemas operativos basados en UNIX y otros derivados compatibles. Estas normas están relacionadas con este proyecto por la parte que aborda las necesidades de tiempo real. Aunque los sistemas operativos UNIX al principio no fueron capaces de adaptar estos servicios, se han ido desarrollando nuevas modificaciones y hoy día encontramos soporte de tiempo real en diversos sistemas operativos.

Las normas de tiempo real que se seguirán en este proyecto son la norma POSIX.1b para señales de tiempo real y temporizadores, y la norma POSIX.1c para hilos, planificación con prioridades y sincronización.

La implementación de las normas se hace con un conjunto de cabeceras que definen múltiples recursos y funciones dedicadas al tiempo real, con lo que se realizará la programación de la librería de tiempo real de este proyecto. Se harán referencias a estas cabeceras y recursos cuando se explique el código fuente en el capítulo 5 de este escrito.

[2] http://en.wikipedia.org/wiki/POSIX [Último acceso: junio 2016]
[3] https://www.ieee.org [Último acceso: junio 2016]

2.3 Desarrollo sobre software libre

Software libre, como su nombre indica, es software que permite total libertad al usuario en su estudio, modificación, desarrollo, copia y distribución. A veces se le suele conocer como software gratuito, pero es solo una de sus características. Hoy día se suele definir el software libre como cuatro libertades esenciales[4]:

- Libertad 0: La libertad de ejecutar el programa como se desea, con cualquier propósito.
- Libertad 1: La libertad de estudiar cómo funciona el programa, y cambiarlo para que haga lo que usted quiera. El acceso al código fuente es una condición necesaria para ello.
- Libertad 2: La libertad de redistribuir copias para ayudar a su prójimo.
- Libertad 3: La libertad de distribuir copias de sus versiones modificadas a terceros. Esto le permite ofrecer a toda la comunidad la oportunidad de beneficiarse de las modificaciones. El acceso al código fuente es una condición necesaria para ello.

Entre sus ventajas podemos destacar su accesibilidad debido a su carácter gratuito y libre distribución, que permite el trabajo cooperativo y la formación grandes comunidades en Internet para la solución de errores y el desarrollo de nuevas aplicaciones. También, su independencia de las condiciones de mercado otorga mucha potencia al software libre en el mundo de la industria, y puede facilitar la incorporación de nuevas entidades.

También tiene numerosas desventajas, que son la principal razón por la que el software libre no domine hoy día. La ausencia de garantía es un factor crítico en el mercado, puesto que nadie se responsabiliza de un error de una aplicación libre. Además, generalmente se requieren conocimientos previos y familiarización con el entorno donde se trabaja, lo que provoca que los usuarios que no tengan conocimientos informáticos se alejen de él. Otra de las desventajas y que vamos a ver claramente durante el desarrollo del proyecto, es que se requiere una monitorización cuidadosa de las actualizaciones y compatibilidades entre muchos programas y librerías. Muchas veces es necesario realizar búsquedas en Internet ante la aparición de errores de compatibilidades, pero sorprendentemente suele resultar fácil encontrar una comunidad donde ya se haya discutido y resuelto el problema en cuestión.

En el ámbito educativo es realmente interesante el desarrollo sobre software libre, ya que se adquieren muchos conocimientos y en general el grado de dificultad es siempre mayor. Además, su accesibilidad es gratuita y es fácilmente trasladable a múltiples sistemas. Después, la portatibilidad hacia otro software de pago suele ser más fácil, siempre y cuando se permita. Por ello, durante todo el desarrollo del proyecto, vamos a utilizar software libre, y por supuesto las aplicaciones que se hayan conseguido también lo serán.

2.3.1 Ubuntu y Xenomai

El sistema operativo elegido para el proyecto es la distribución Ubuntu [1]. Ubuntu es mundialmente conocido como una de las mayores competencias en el mercado de sistemas operativos. Basada en la arquitectura GNU/Linux, Ubuntu es el distribuidor más extendido de sistemas operativos de libre licencia, y es precisamente esta característica de software libre junto a su potencia lo que lo permite competir directamente con los famosos sistemas operativos Windows y MacOS, ambos bajo licencia de pago. Mientras esto puede resultar una gran ventaja económica y de accesibilidad frente a ellos, Ubuntu también trae muchas desventajas: compatibilidad, soporte y diversas complicaciones y problemas directamente relacionados con su distribución gratuita. Lo que sí es cierto es que el éxito de desarrollar aplicaciones sobre Ubuntu es mayor que si se ha desarrollado con la ayuda de las funcionalidades que venden otros sistemas operativos de pago.

Sin embargo, esa no es la principal razón por la que hemos elegido Ubuntu. Puesto que vamos a desarrollar una librería para aplicaciones de tiempo real, es estrictamente necesario que el sistema sea capaz de cumplir todas las restricciones de tiempo que se le plantean. Esta tarea es imposible tanto en Windows como en MacOS, que, a pesar de ser los dos sistemas operativos bajo licencia más populares, su desarrollo no se ha centrado en permitir esta característica.

El único sistema operativo que hemos discutido junto con Ubuntu ha sido QNX, un sistema operativo de tiempo real basado en UNIX que cumple con las normas POSIX de tiempo real y está diseñado principalmente para

[4] https://www.gnu.org/philosophy/free-sw.es.html [Último acceso: junio 2016]

sistemas empotrados. QNX es sin duda el mejor sistema operativo para los objetivos del proyecto, pero tiene la desventaja de no ser software libre, lo cual nos obligó descartarlo para este proyecto de carácter académico. No obstante, la compatibilidad entre Linux de tiempo real y QNX es inmediata gracias a las normas POSIX, por lo que la librería que se va a desarrollar en este proyecto es perfectamente compatible con QNX, solo que en QNX se van a ahorrar muchos pasos de configuración, instalación y configuración que es necesario hacer sobre Ubuntu.

Ubuntu, nativamente, no funciona completamente bien con las aplicaciones de tiempo real, pero sí que existen extensiones o parches que permiten habilitar esta funcionalidad. Para este proyecto se ha utilizado Xenomai, que es uno de estos parches de tiempo real para Linux que también se distribuye como software libre.

Xenomai [2] es una arquitectura de desarrollo para tiempo real basado en el entorno Linux. Su funcionamiento se basa en un *co-kernel*, o segundo núcleo, que trabaja paralelamente con el *kernel* de Linux proporcionando las características de tiempo real. Se puede entender también como una extensión o nueva capa que se instala en el *kernel* para encargarse de las tareas críticas de tiempo real.

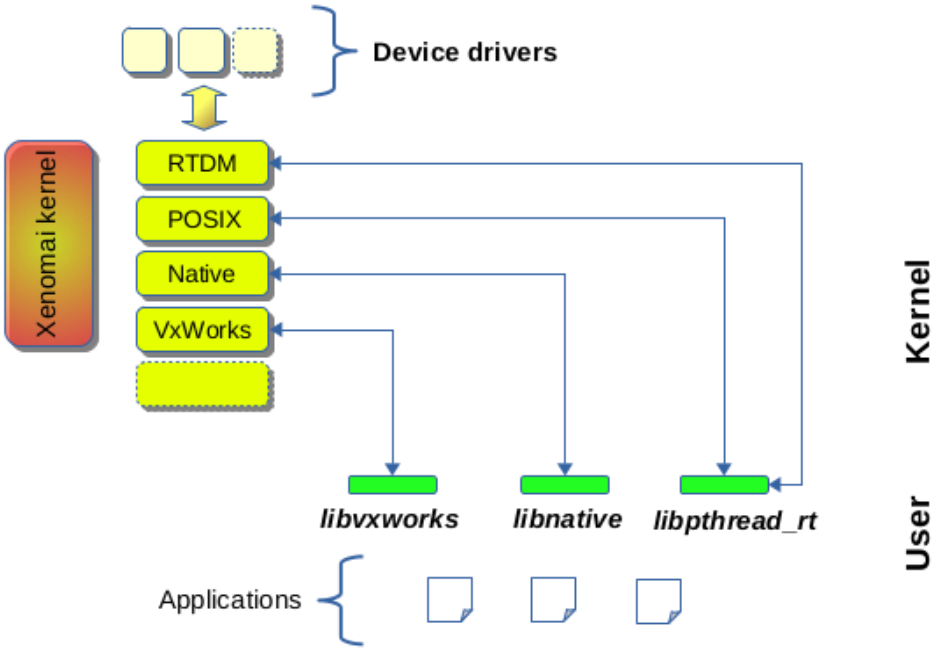

Ilustración 2-1. [5] Arquitectura de Xenomai

Xenomai es una de las arquitecturas más usadas para trabajar con aplicaciones de tiempo real en Linux. Dada su excelente compatibilidad con las normas POSIX nos permitirá un buen desarrollo de la librería que se desarrollará en este proyecto.

2.3.2 ACADO Toolkit

La herramienta ACADO Toolkit [3] es un entorno de desarrollo libre en el ámbito de control automático y optimización de procesos. Proporciona numerosos algoritmos para control predictivo basado en modelo (MPC), estimación de estados y parámetros, y optimización robusta. Posee una potencia ejemplar para resolver problemas no lineales con diferentes tipos de restricciones. Encontramos soporte en C++ y también para Matlab. También disponen de un manual[6] para facilitar el aprendizaje y su forma de uso.

ACADO se ha diseñado siguiendo estos cuatro principios:

- Código abierto: Libre y disponible para todos.

[5] Imagen extraída de la fuente [2]
[6] http://acado.sourceforge.net/doc/pdf/acado_manual.pdf [Último acceso: junio 2016]

- Fácil uso: La sintaxis se ha diseñado para ser lo más intuitiva posible de cara al usuario y que se parezca a las ecuaciones matemáticas habituales.
- Extensibilidad: Debería ser fácil implementar algoritmos ya existentes o acoplarlos a los de ACADO.
- Autocontenido: Salvo paquetes externos para ayudar con el dibujo de gráficas, no se necesitan más instalaciones externas.

ACADO es un software muy importante en el mundo de la ingeniería de control por su capacidad de implementar y resolver problemas que requieren mucha potencia de cálculo y además por su carácter de software libre. Por este motivo, se ha elegido incluir esta herramienta al proyecto con el objetivo de estudiarla, utilizarla e incorporarla a la programación de sistemas de control en tiempo real.

3 PC104 Y PERIFÉRICOS

En este capítulo se introduce el PC-104. Se detallan sus características técnicas, así como se explica su montaje inicial e instalación, además de las modificaciones que han sido necesarias para lograr la funcionalidad necesaria.

3.1 Características técnicas

El PC104 está compuesto por tres módulos o bloques: fuente de alimentación (etapa de regulación), tarjeta de adquisición de datos y el módulo PCM-3708. Se van a presentar las características principales de cada módulo por orden.

El módulo superior es la etapa de regulación de la fuente de alimentación. Recibe una entrada de continua rectificada por un bloque externo que veremos en el próximo capítulo, y adapta el voltaje para dar diferentes salidas que necesitarán el resto de componentes electrónicos. Su tabla de características técnicas es la siguiente:

Módulo	**PM-P005**
Output	5V@12A, 12V@5A, 3.3V@10A, 5VSB@1A Max. Total Output: 60W
Input	[$8V_{DC}$, $36V_{DC}$]
Características de rendimiento	Regulación de línea: <20mV Regulación de carga: <150mV Eficiencia: hasta 90% Corriente en reposo: <2mA
Temperatura de funcionamiento	[–40ºC, 85ºC]

Tabla 3-1. Caracteríticas del módulo de alimentación

El módulo intermedio corresponde a la tarjeta de adquisición de datos. El modelo es el PCM-3718[7] y se ha dejado un datasheet a pie de página. Una tarjeta de adquisición de datos tiene la misión de tomar medidas analógicas del mundo real y generar datos con ellos de forma digital para su tratamiento de datos por parte de una computadora. También puede realizar el proceso contrario: obtener datos digitales de la computadora para mandarlas como salida análogica a un dispositivo externo. Sus características más importantes se recogen en la siguiente tabla.

	Analog input
Canales	16 single-ended / 8 differential

[7] http://support.elmark.com.pl/advantech/pdf/PCM-3718man.pdf [Último acceso: junio 2016]

Resolución	12 bits
Máxima frecuencia de muestreo	100 kHz
Impedancia de entrada	100 MΩ

Tabla 3-2. Características analógicas del módulo de adquisición de datos

	Digital input/output
Canales	16 5V/TTL
Voltaje de entrada	0 lógico: 0.8V máx. 1 lógico: 2.0V mín.
Voltaje de salida	0 lógico: 0.33V máx. @ 6mA 1 lógico: 3.84V mín. @ 6mA
Impedancia de entrada	100 MΩ

Tabla 3-3. Características digitales del módulo de adquisición de datos

El módulo inferior es la placa base y tiene integrada los componentes principales de un ordenador: procesador, memoria RAM, un disco duro interno y conexiones a los periféricos habituales.

Procesador	**Intel® Atom™ Processor N450**
Caché	512K
Conjunto de instrucciones	64-bit
Frecuencia base	1.66GHz
Memoria RAM	1GB
Disco duro interno	2GB

Tabla 3-4. Características del PC-104

3.2 Montaje inicial del PC104

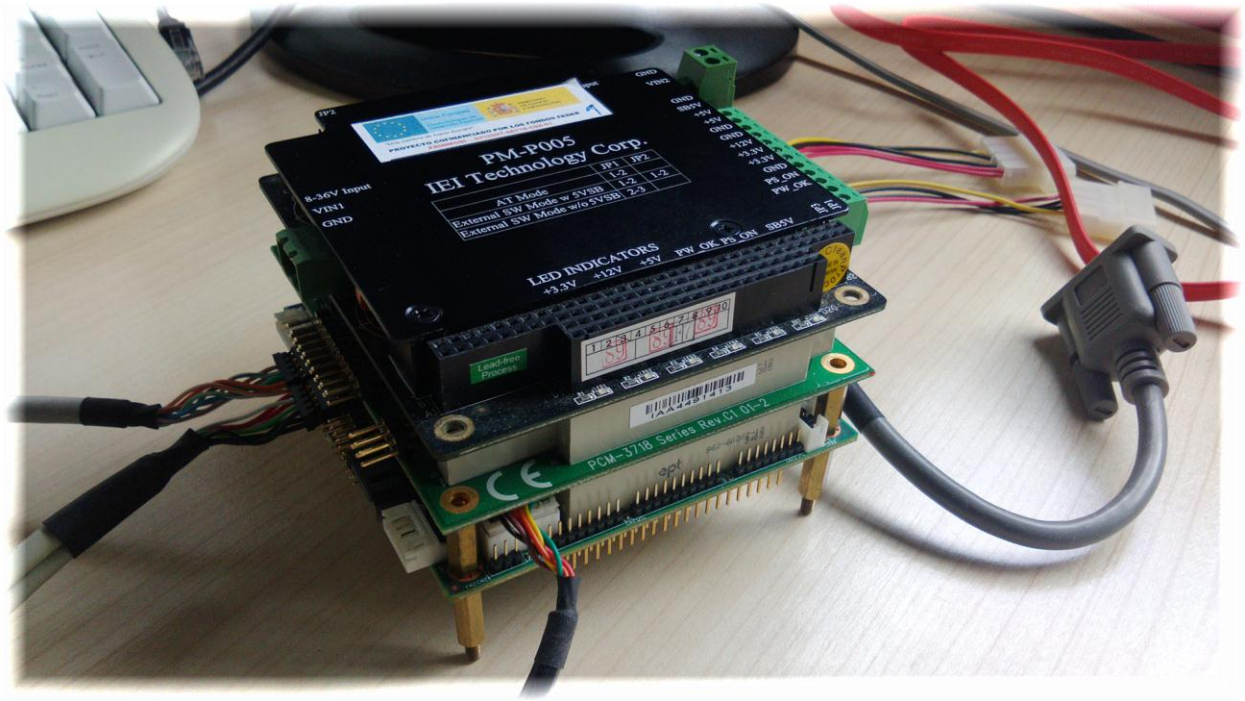

Ilustración 3-1. PC104

El departamento se encargó de atornillar los módulos y de colocar un disipador que cumpliera los requisitos térmicos del micro. Con el bloque compacto montado, se colocan todos los cables de entrada de periféricos que vienen en el embajale según las instrucciones.

3.2.1 Fuente de alimentación

El siguiente paso es instalar la fuente de alimentación y montar el cableado de red eléctrica. La función de este aparato es la de convertir y rectificar los 220 V de alterna de la red a 12 V de continua para la etapa de regulación. Nótese que la etapa de regulación es el primer bloque del PC-104, y es la que se encarga de las transformaciones a los diferentes voltajes necesarios para todos los dispositivos electrónicos. Sin embargo, esta etapa que constituye la fuente de alimentación podría decirse que tiene funciones principalmente eléctricas y no electrónicas, tales como la rectificación, aislamiento al rudio y protección.

El cable que va a la red eléctrica no viene incluido, pero se puede adquirir cualquier cable de alimentación, pelarlo y montar la conexión como se va a explicar a continuación.

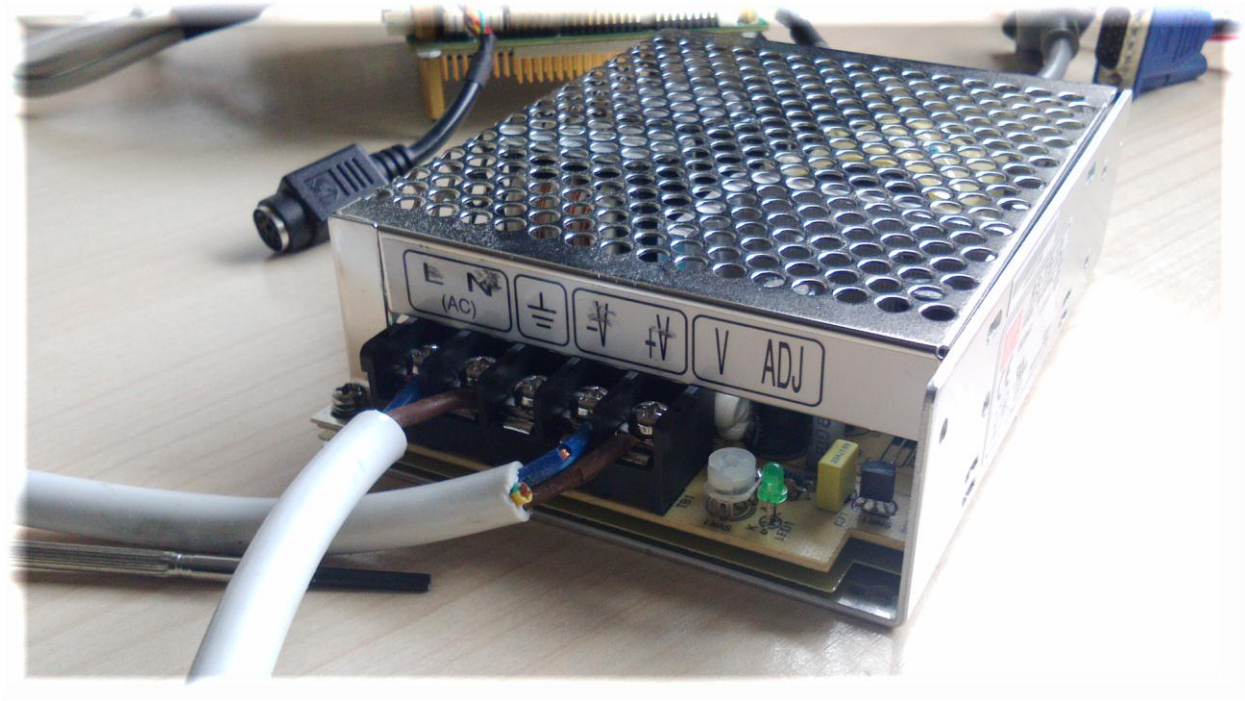

Ilustración 3-2. Fuente de alimentación

La fuente tiene 5 conexiones: las tres primeras a la izquierda para la red de alterna, L (fase), N (neutro), tierra; y las dos últimas, -V y +V. Cogemos un cable de alimentación con toma de enchufe, pelamos los cables, y lo conectamos de la siguiente manera: azul a fase, marrón a neutro, y verde/amarillo a tierra (este es opcional).

Por otro lado, cogemos un trozo de cable de alimentación para conectar +V y –V con VIN1 y GND del PC-104. Los colores del cable no importan siempre y cuando se conecte +V con VIN1 y –V con GND.

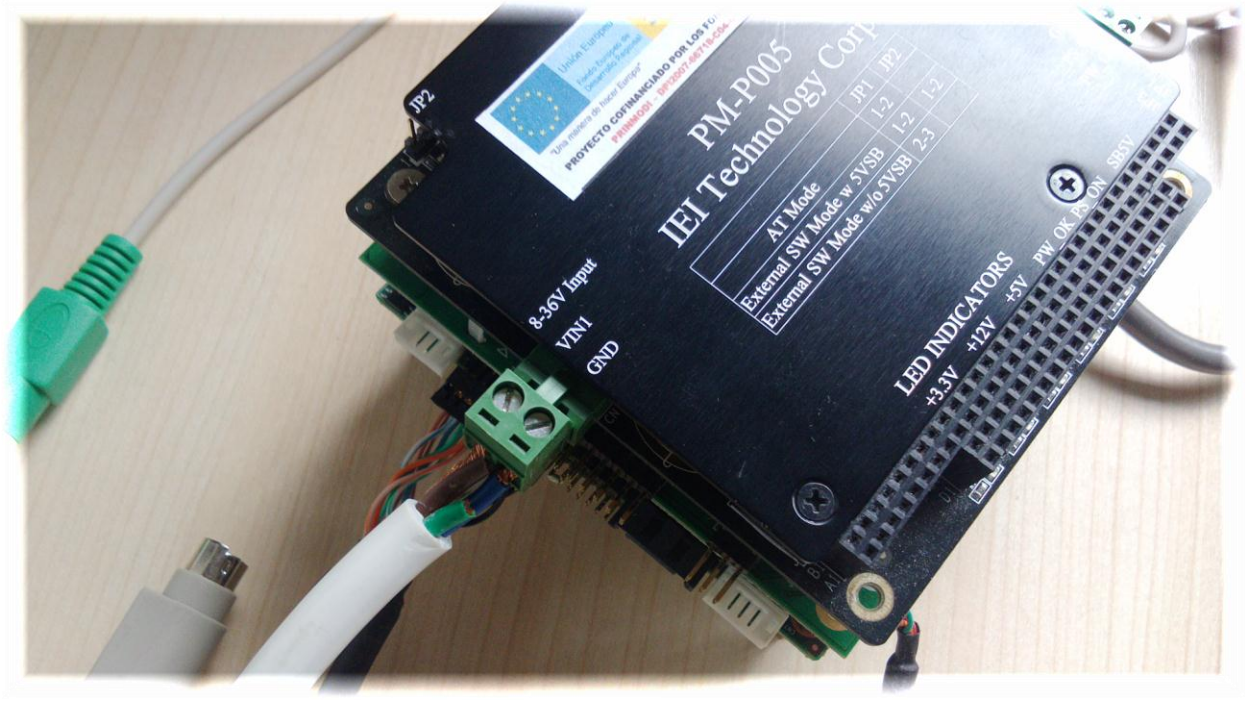

Ilustración 3-3. Alimentación a PC104

3.2.2 Disco duro

Aunque el módulo trae un disco duro interno de 2 GB, nos dimos cuenta que no era suficiente para el sistema operativo Ubuntu y demás software que había que instalar, lo que nos obligó a buscar un disco duro externo con más capacidad.

Ilustración 3-4. Disco duro externo

Utilizamos este disco duro Toshiba de 40 GB que venía en su carcasa gris con una salida mini-USB para conectar después a un PC por USB. Sin embargo, como el PC-104 solo tiene dos entradas USB, conectar el disco duro por USB implica tener uno de los dos puertos en uso permanentemente, lo cual podría ser una molestia en un futuro. Además, el PC-104 tiene una entrada por SATA, lo cual deja bastante claro la siguiente decisión: desmontar la carcasa, extraer el disco duro y utilizar la conexión SATA, con el consiguiente aumento de la velocidad de lectura y escritura del disco con respecto a la conexión USB. En la ilustración anterior puede observarse a la derecha el disco duro y justo debajo el dispositivo que transformaba la conexión SATA a micro-USB.

La conexión SATA es ligeramente más compleja que la conexión USB, pero no deja de ser sencilla una vez conocida. Del disco duro parten dos conexiones en forma de L, una más corta a la izquierda y otra más larga a la derecha (ver ilustración 3-4). La conexión de la izquierda se encarga de la transferencia de datos e irá conectada directamente al módulo inferior del PC-104, concretamente a otra conexión SATA de datos con la misma forma de L.

La conexión de la derecha es la alimentación del disco duro, y conectaremos un adaptador del que parten cuatro cables: amarillo, rojo y dos negros. El código de colores es el siguiente: amarillo +12V, rojo +5V, negro GND (tierra). Vamos a conectar estos cuatro cables al módulo de regulación del PC-104. El resultado final puede verse en la ilustración 3-5.

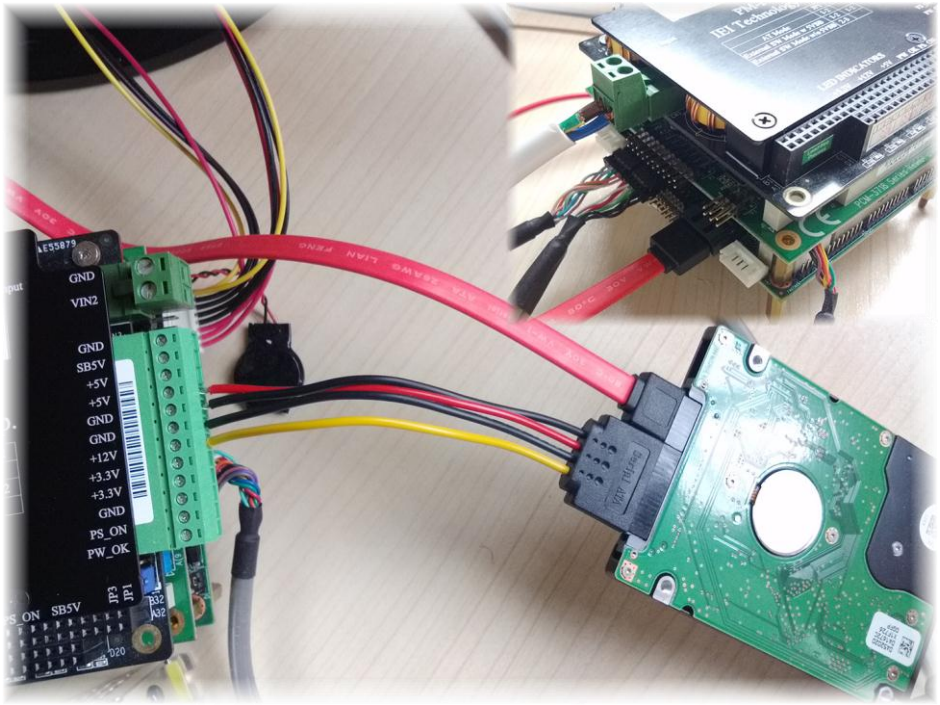

Ilustración 3-5. Conexión del disco duro por SATA

3.2.3 Periféricos

Mostramos el resto de conexiones y de periféricos que hemos utilizado con el PC104.

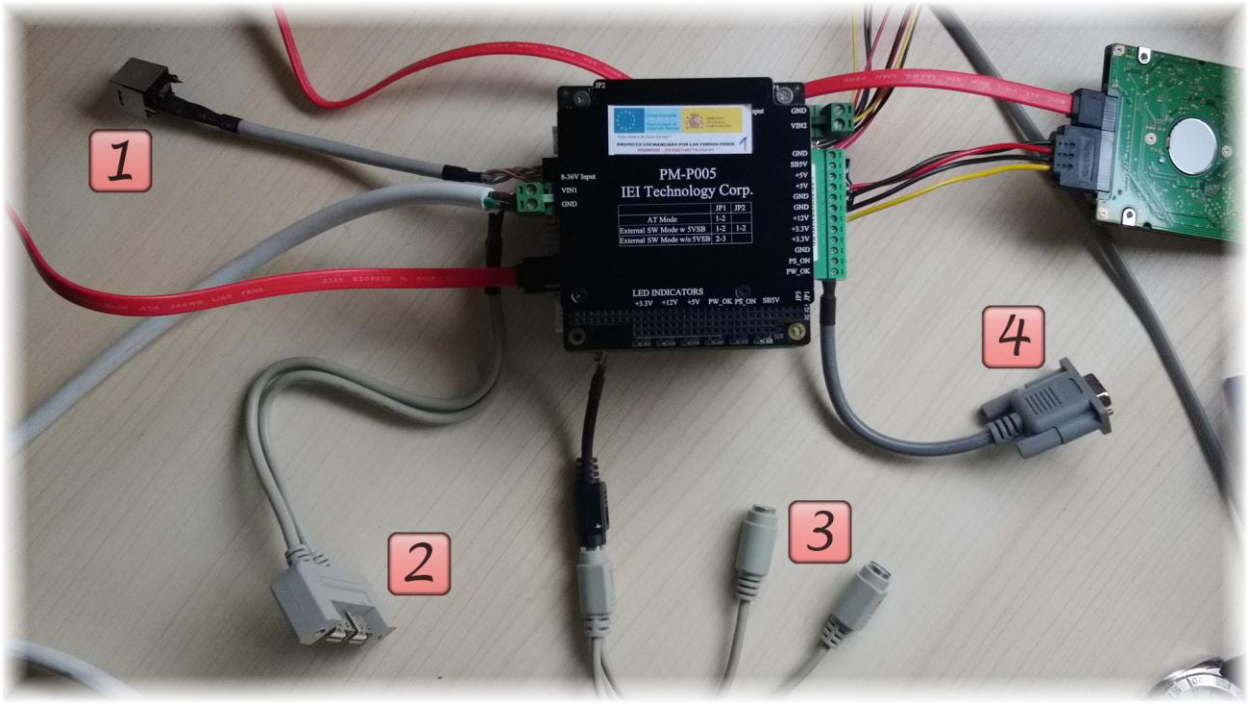

Ilustración 3-6. Conexiones de periféricos

Según la numeración que hemos empleado en la fotografía, el resto de conexiones son las siguientes: (1) cable ethernet hembra para la conexión a internet o red local, (2) doble puerto USB, (3) puerto PS/2 para ratón y teclado y (4) salida VGA hacia monitor. Los periféricos que hemos utilizado, por tanto, son ratón y teclado PS/2, y monitor de entrada VGA. Más tarde hablaremos sobre qué hemos utilizado para instalar el sistema operativo.

Finalmente, mostramos el espacio de trabajo en la siguiente ilustración.

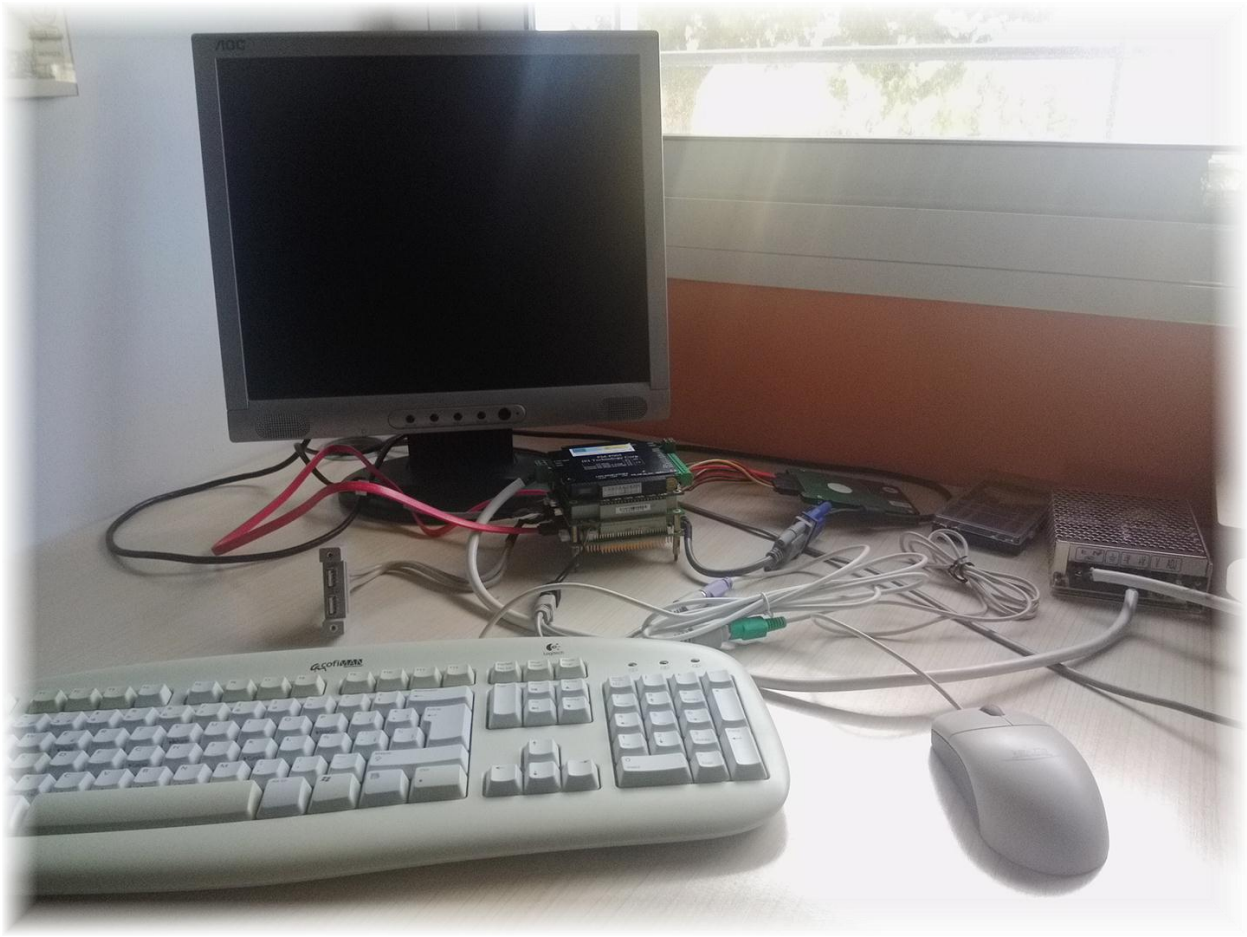

Ilustración 3-7. Lugar de trabajo

4 INSTALACIÓN DEL SISTEMA OPERATIVO EN TIEMPO REAL Y DEL SISTEMA DE CONTROL

En este capítulo explicaremos la elección de la versión y la instalación de Ubuntu, así como los pasos necesarios para actualizarlo e incorporar los paquetes que serán necesarios para el resto del proyecto. Presentaremos también el software adicional que hemos instalado para lograr los objetivos propuestos.

4.1 Versión del sistema operativo

La elección de la versión de Ubuntu suele ser más complicada de lo normal. Mientras que para instalar Windows solo tienes que fijarte en los requerimientos de hardware y normalmente siempre funciona usar la última versión, cuando se tiene software libre hay que fijarse mucho más en el tema de la compatibilidad. Nosotros estamos trabajando con un sistema embebido que requiere unos drivers específicos y al que hay que instalarle un parche de tiempo real. Normalmente, el método a seguir consiste en buscar versiones por su fecha, y comprobar las fechas de soporte que facilite el fabricante.

Dado que el PC104 ya no es tan actual, vamos a movernos por fechas cercanas a 2012, según la última actualización de los drivers de la tarjeta de adquisición de datos del PC104. Por ello hemos elegido Ubuntu 12.10 [3], distribución de finales de 2012, para asegurarnos de la compatibilidad del resto de software que instalemos. Usar una versión anterior provocaría que ACADO no funcionara, ya que requiere versiones más actualizadas.

Ubuntu Linux	12.10 (32-bit)
Kernel Linux	3.5.0-17
GNOME	3.6.0

Tabla 4-1. Versión del SO

4.2 Instalación del sistema operativo

La instalación de cualquier sistema operativo se ha hecho clásicamente desde una unidad CD o DVD autoarrancable, o *bootable* en inglés. Hoy en día esta técnica se está dejando atrás debido a las memorias o "pinchos" USB, que son mucho más accesibles, sencillas y comunes entre todo tipo de usuarios. Existe una gran cantidad de programas gratuitos en internet que te permiten salvar una imagen ISO en una memoria USB y

prepararla como dispositivo autoarrancable. En esta ocasión se usado Rufus[8], programa muy simple y portable en el que se ha seleccionado la imagen ISO de Ubuntu 12.10 descargada desde la fuente [3].

Así pues, teniendo puertos USB en el PC104, podemos usar cualquier memoria USB de al menos 1 GB para la instalación del sistema operativo, puesto que la imagen de Ubuntu para esta versión ocupa 800 MB. Con todo listo, pasamos a insertar la memoria USB en el PC104.

4.2.1 Configurar la BIOS para arrancar desde la memoria USB

La BIOS (*Basic Input/Output System*) es el primer programa que se carga al iniciar un PC. Comprueba e inicia el hardware básico del sistema, y además tiene la función de arrancar el sistema operativo desde los dispositivos de memoria asociados.

Para esta primera vez que entramos en la BIOS, nuestra misión es arrancar desde la memoria USB en la que hemos instalado la imagen ISO del sistema operativo. Antes de encender el PC104, la memoria USB ya debe estar conectada, así como el resto de periféricos que enseñamos en el capítulo 3 (monitor, teclado y disco duro, al menos) para poder configurar la BIOS e instalar el sistema operativo.

Cuando encendemos el PC104, la BIOS va a arrancar automáticamente con sus preferencias por defecto. Para entrar a modificarlas antes de que se ejecuten, hay una pequeña ventana de tiempo poco después de encender el PC en la que puedes pulsar la tecla *Supr* para cancelar el arranque automático y entrar a las opciones. Para asegurarnos de hacerlo bien, podemos pulsar la tecla *Supr* repetidamente desde que conectamos el PC a la red hasta ver la siguiente pantalla.

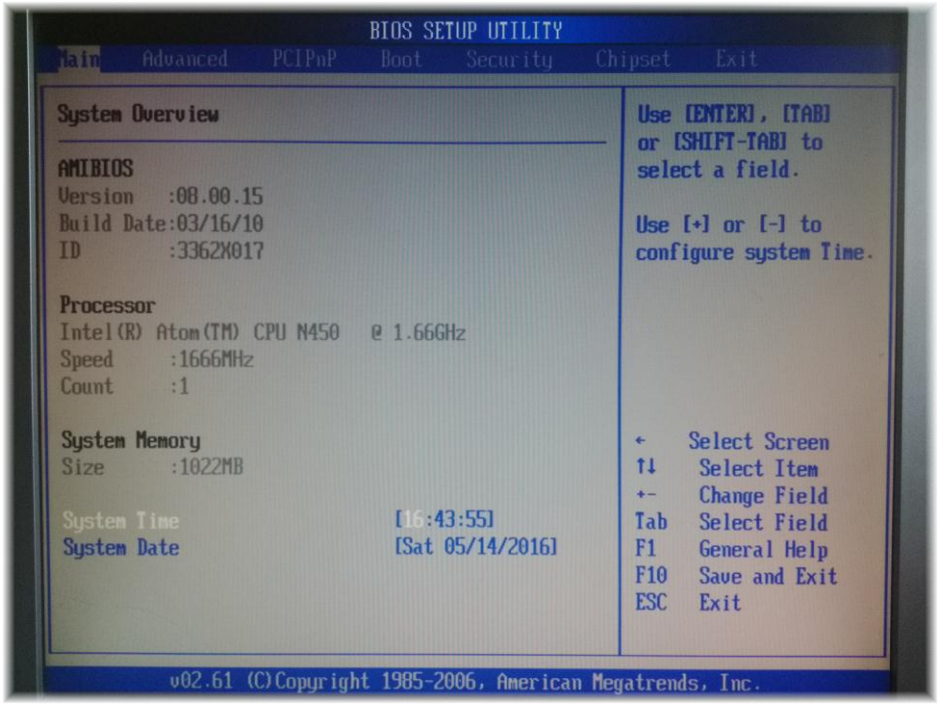

Ilustración 4-1. Pantalla principal de configuración de la BIOS

Esta pantalla solo puede controlarse desde el teclado, y las teclas que te permiten navegar por las opciones son las que se indican abajo a la derecha. Lo que buscamos nosotros está tres pestañas a la derecha, en *Boot*, que son las opciones de arranque.

[8] https://rufus.akeo.ie/ [Último acceso: mayo 2016]

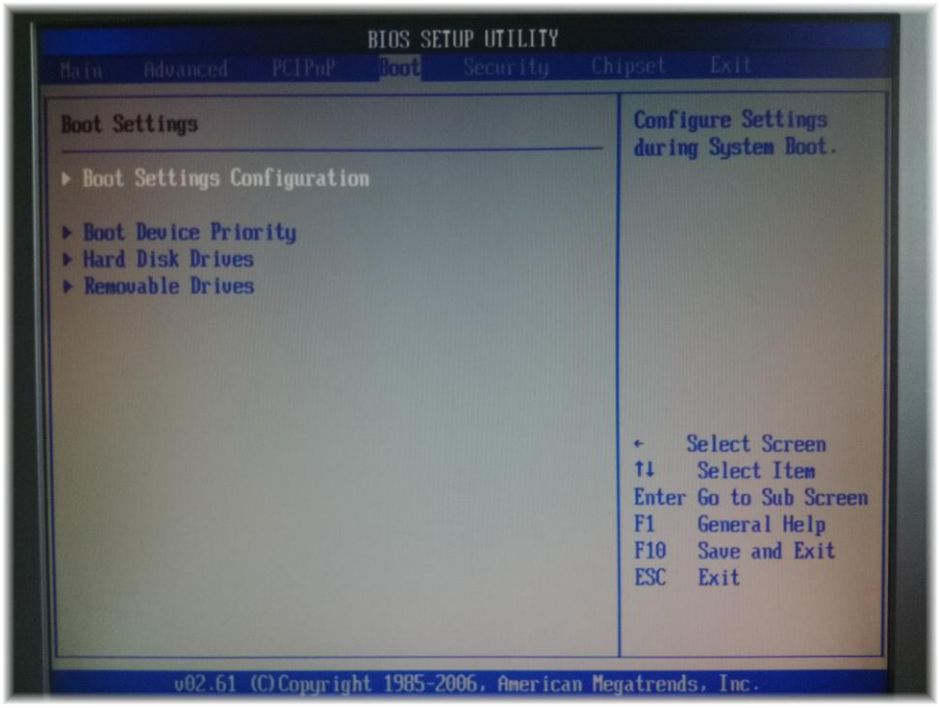

Ilustración 4-2. Pantalla de configuración de arraque de la BIOS

Si seleccionamos *Hard Disk Drives*, se nos proporciona una lista de dispositivos de almacenamiento conectados. Si recordamos lo anterior, en la lista debería aparecer lo siguiente: el disco duro interno de 2 GB del PC104, que no íbamos a utilizar por poco espacio, el disco duro externo que conectamos por SATA, y la memoria USB que hemos insertado antes de arrancar.

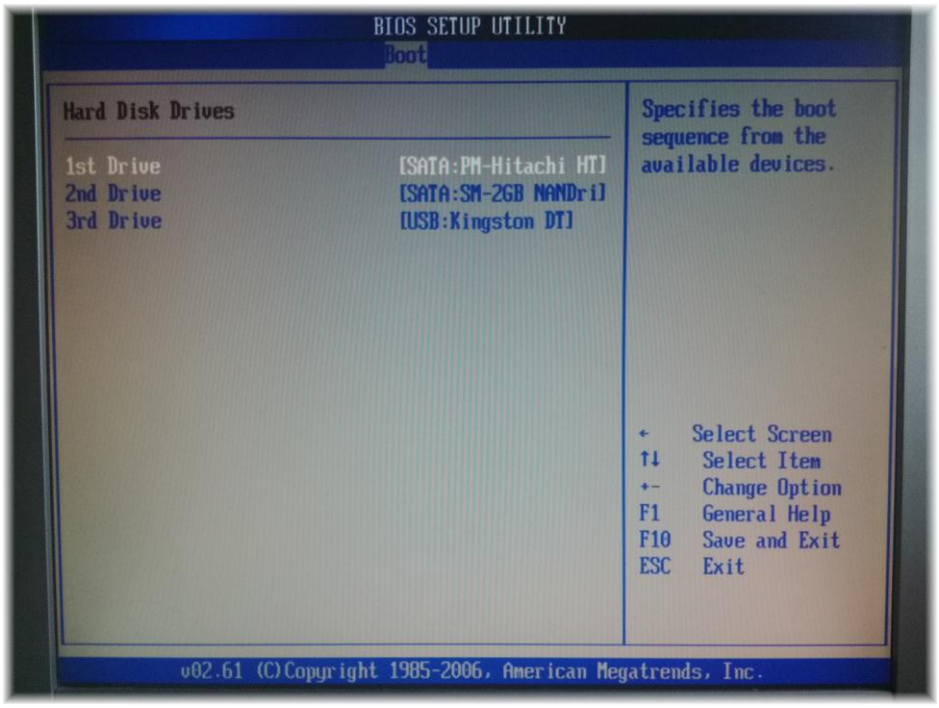

Ilustración 4-3. Dispositivos de almacenamiento disponibles

El primero que aparece es el disco duro externo, luego el interno y por último la memoria extraíble. Normalmente aquí el orden no importa, puesto que la prioridad se establece en la siguiente pantalla, pero para estar seguros se podría poner el dispositivo USB como el primero. Volvemos atrás y seleccionamos Boot Device Priority. Aquí

sí que importa el orden, puesto que en esta pantalla vamos a configurar la prioridad de arranque. Como queremos instalar el sistema operativo desde la memoria USB, vamos a poner este dispositivo como primera prioridad. Como segunda, el disco duro externo. Esto significa que la BIOS arrancará automáticamente desde la memoria USB, y en caso de que no haya una conectada, desde el disco duro. Una vez que se instale el sistema operativo, es recomendable volver a este paso y reestablecer las prioridades.

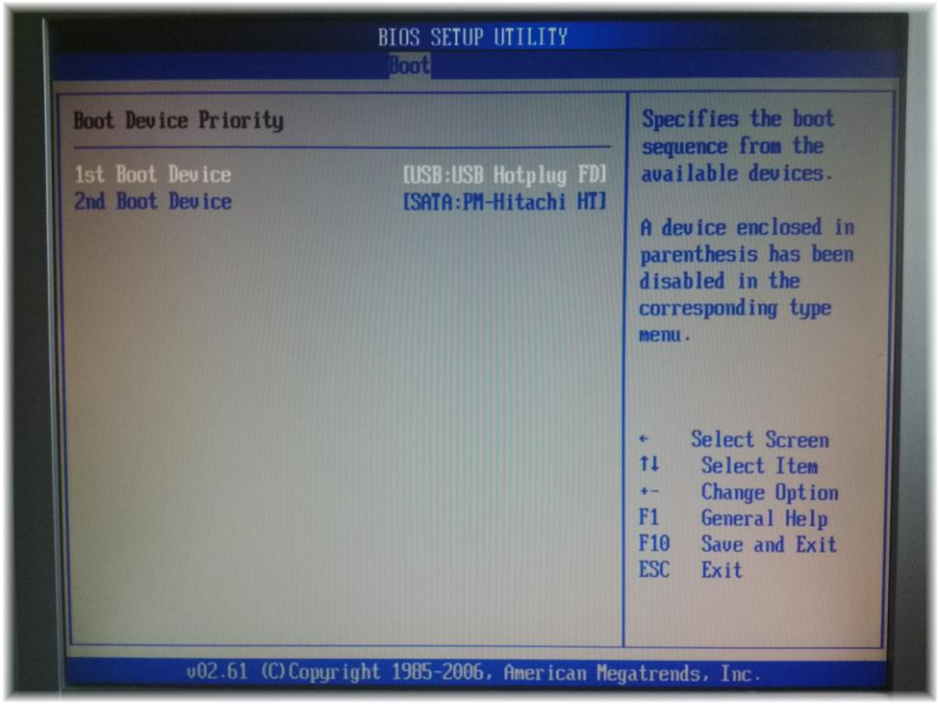

Ilustración 4-4. Ajustes de prioridad de arranque

Con estos ajustes, podemos pulsar F10 para guardar opciones y reiniciar.

4.2.2 Proceso de instalación de Ubuntu 12.10

Como vamos a ver a continuación, una vez arrancada la imagen ISO de instalación, los pasos a seguir son muy intuitivos. Antes de realizar la instalación, además de haber seguido los pasos anteriores, debemos de asegurarnos que tanto la memoria USB como el disco duro por SATA están conectados, puesto que la instalación se hará desde el primero hasta el segundo.

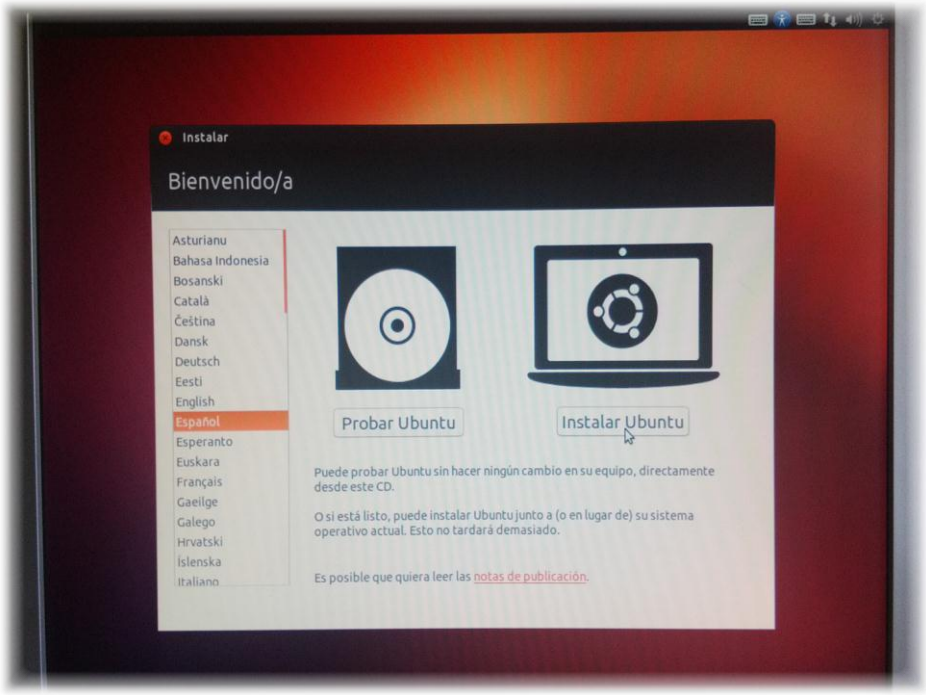

Ilustración 4-5. Instalación de Ubuntu 12.10

Seleccionamos el idioma a la izquierda y hacemos clic en "Instalar Ubuntu".

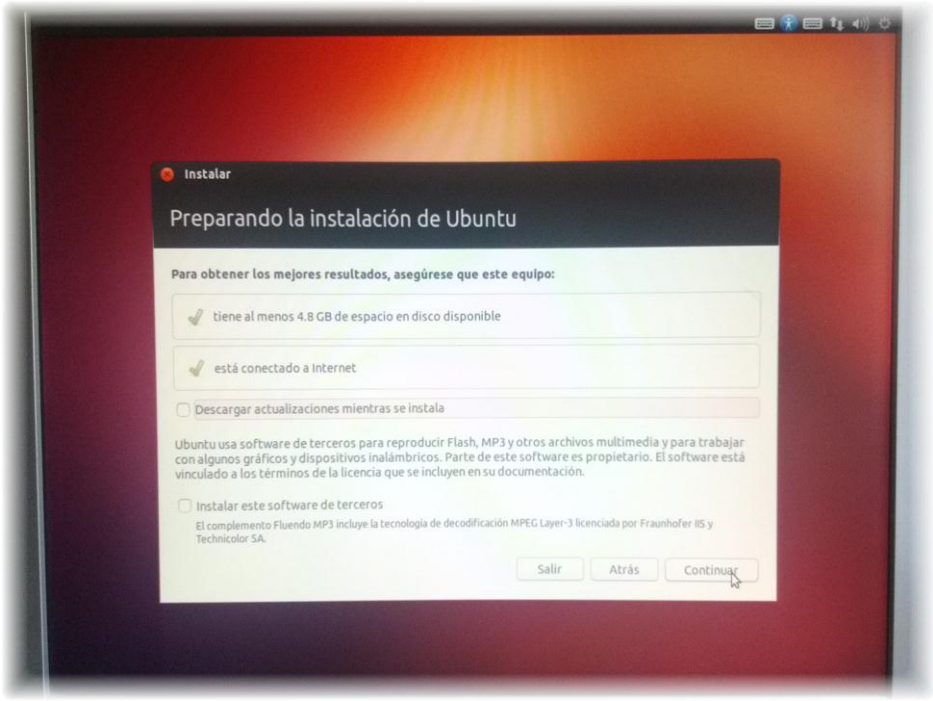

Ilustración 4-6. Requisitos de instalación

Como vemos, se necesitan casi 5 GB de espacio en el disco duro, bastante por encima de la capacidad que traía el disco duro interno del PC104. También se requiere la conexión a internet para recibir las diversas actualizaciones que salieron después. Es posible que este último requisito sea complicado si la conexión a internet necesita una configuración previa, así que vamos a desmarcar las dos casillas que aparecen y vamos a instalar el resto de lo necesario una vez terminemos la instalación del sistema operativo.

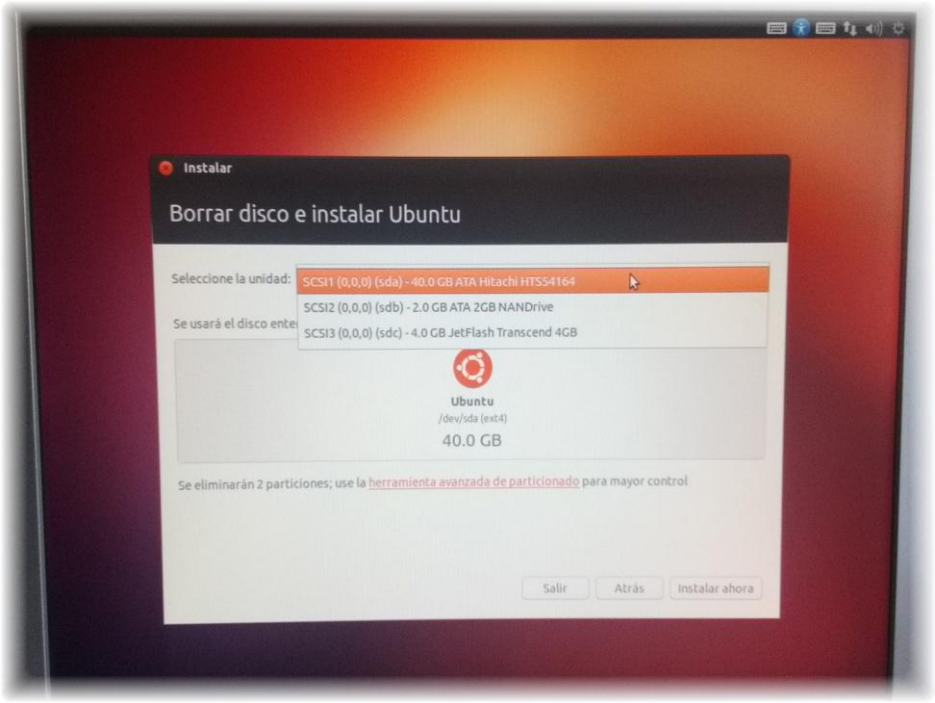

Ilustración 4-7. Selección de la unidad de instalación

Nos aseguramos que se instala en el disco duro externo que hemos colocado. En este caso, también vamos a formatear el disco duro completo y borrar lo que haya previamente.

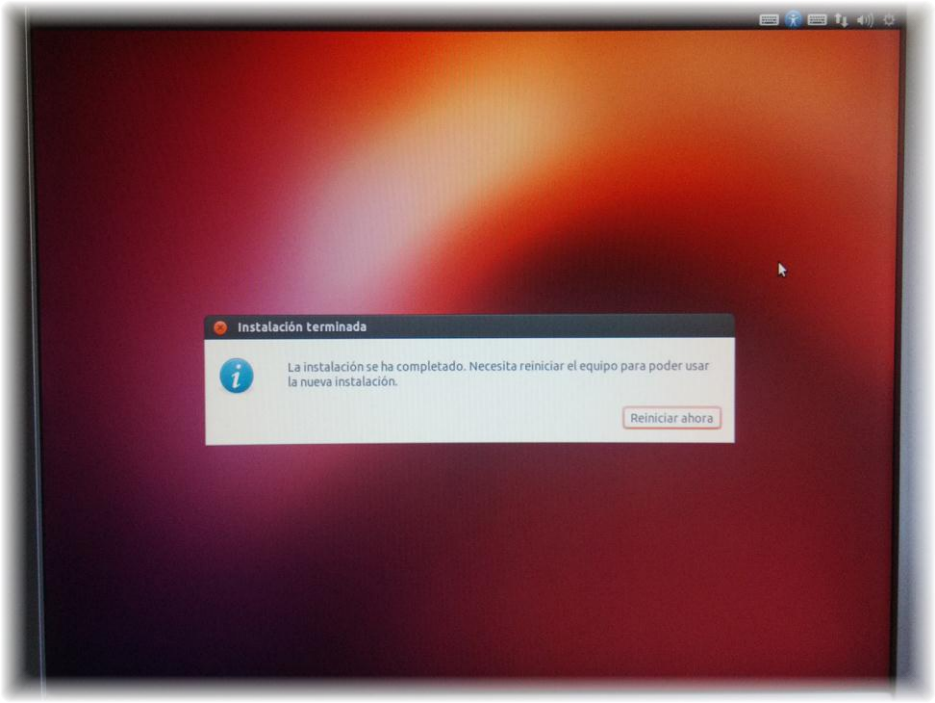

Ilustración 4-8. Pantalla de instalación con éxito

Recordar que cuando se reinicie el ordenador hay que volver a entrar en la BIOS y deshacer los últimos cambios que se realizaron, de lo contrario volveremos a iniciar desde la memoria USB, y de nuevo saldrá la pantalla de instalación de Ubuntu. El disco duro externo tendrá ahora la mayor prioridad de arranque, puesto que Ubuntu ya se ha instalado aquí.

4.3 Actualización e instalación de paquetes necesarios

Tenemos que actualizar diversos paquetes de Ubuntu que serán necesarios no solo para empezar a programar, sino también para poder instalar más tarde Xenomai, ACADO y los drivers de la tarjeta de adquisición de datos. Dado que Xenomai actúa sobre el kernel de Linux, es mejor dejar este paso para el final e instalar primero los paquetes que serán necesarios para el resto del desarrollo del proyecto.

Es necesario avisar que esta parte es la que suele dar más ruido debido a la naturaleza de software libre con la que estamos trabajando. Se podría garantizar que cambiar cualquier versión de cualquier paquete que vayamos a instalar a continuación dará errores, por lo que dada la necesidad se recomienda informarse muy bien de las versiones que exigen los desarrolladores.

4.3.1 Actualización del sistema

En general, este apartado es común a todas las versiones de Ubuntu actuales, salvo una excepción que hay que tener en cuenta para las versiones que han quedado obsoletas, que como es nuestro caso, también detallaremos.

Lo primero que se debe hacer al arrancar Ubuntu es abrir el terminal. Si no aparece como acceso directo por defecto en el escritorio, se debe buscar "terminal" en el buscador de programas.

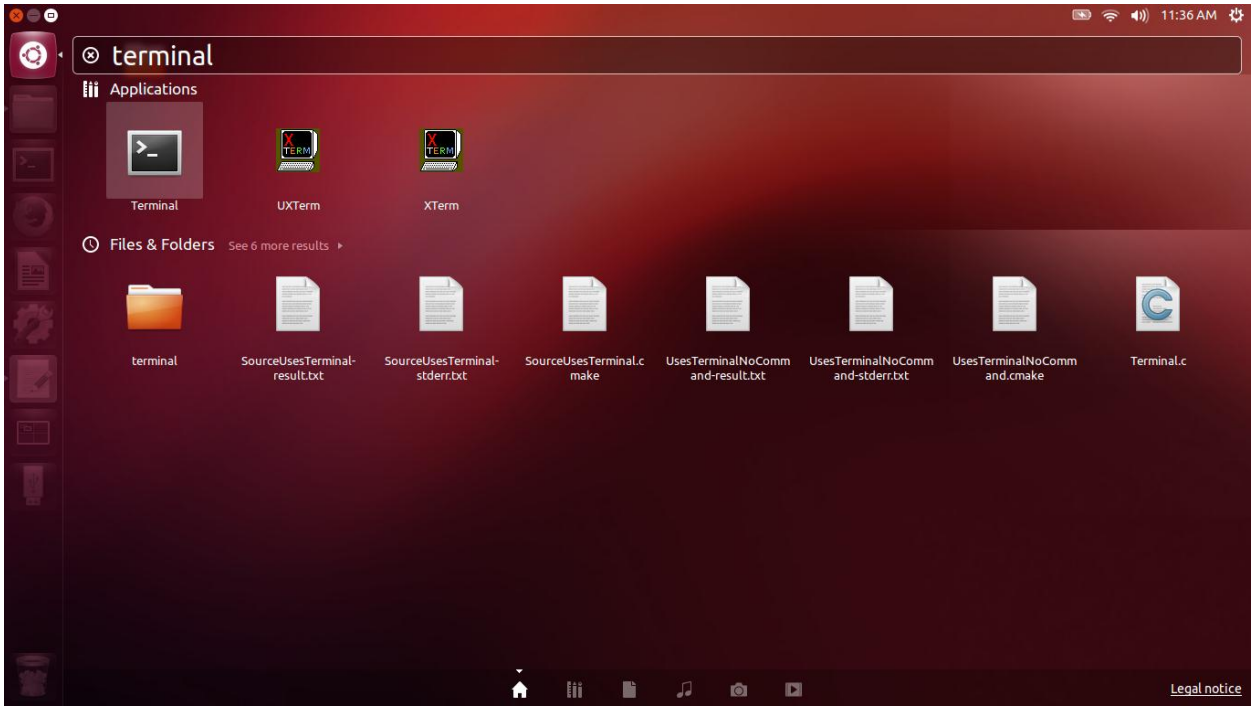

Ilustración 4-9. Terminal de Ubuntu 12.10

Una vez abierto el terminal, escribimos estas líneas que también son comunes en cualquier versión de Ubuntu hasta la fecha.

```
sudo -s
apt-get update
apt-get upgrade
```

Escribimos cada línea por separado, es decir, pulsando la tecla *Enter* entre cada una. La primera línea sirve para pedir permisos de administrador, necesaria para realizar cualquier instalación. Cuando pulsamos *Enter*, el terminal te pedirá escribir la contraseña, la cual es la misma que se escribió en la instalación de Ubuntu. Una vez escrita, volvemos a pulsar *Enter* y tendremos los permisos de administrador. Todos los pasos que se realizarán a continuación necesitan estos permisos de administrador, por lo que si se reinicia el sistema o el terminal en algún momento, debemos escribir esta línea de nuevo.

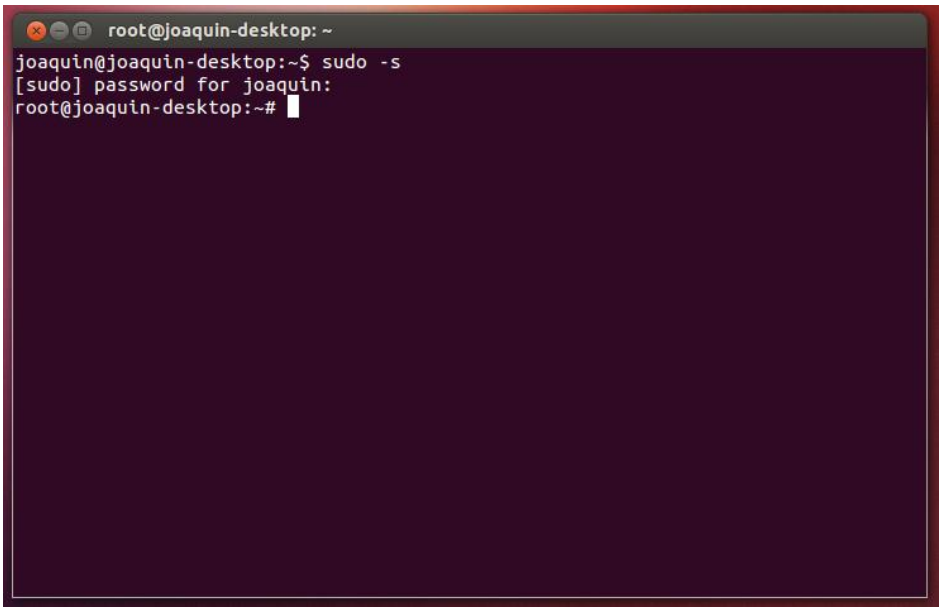

Ilustración 4-10. Permisos de administrador en Ubuntu

La siguiente línea (*update*) sirve para descargar las actualizaciones de los paquetes asociados a nuestra versión de Ubuntu. Primero hará una comprobación, y después pedirá una confirmación del tipo [Y/n], correspondiente a *yes/no*, en la que habrá que pulsar la tecla *y* para confirmar la descarga.

Aquí es donde está la excepción: las versiones obsoletas no pueden ejecutar el comando tal cual y saldrán múltiples errores al ejecutarlo, y es que el servidor donde están alojados los paquetes ha cambiado a otra dirección diferente. La versión 12.10 de Ubuntu a día de hoy está obsoleta, pero no es ningún problema si se sabe solucionar. Lo único que tenemos que hacer es cambiar el servidor de descarga de las actualizaciones, y para eso tenemos que acceder a este archivo:

```
gedit /etc/apt/sources.list
```

Con el comando *gedit* abrimos el editor de archivos de texto, y *sources.list* es el archivo en el cual tenemos que modificar el servidor de descarga. Veremos múltiples enlaces que empiezan así "http://archive.ubuntu.com/" o así "http://security.ubuntu.com/". Tenemos que cambiar ambas direcciones a esta nueva "http://old-releases.ubuntu.com/", sin modificar los subdirectorios o comandos que hay detrás. Esta tarea es muy sencilla, pero hay alrededor de una docena de enlaces que hay que cambiar, y no debe escaparse ninguno. Si no se hace correctamente, volveremos a ver algún error al ejecutar *update*, y tendremos que volver al archivo para asegurarse de que todos los enlaces se han corregido.

En la siguiente captura mostramos cómo debe quedar finalmente el archivo *sources.list*.

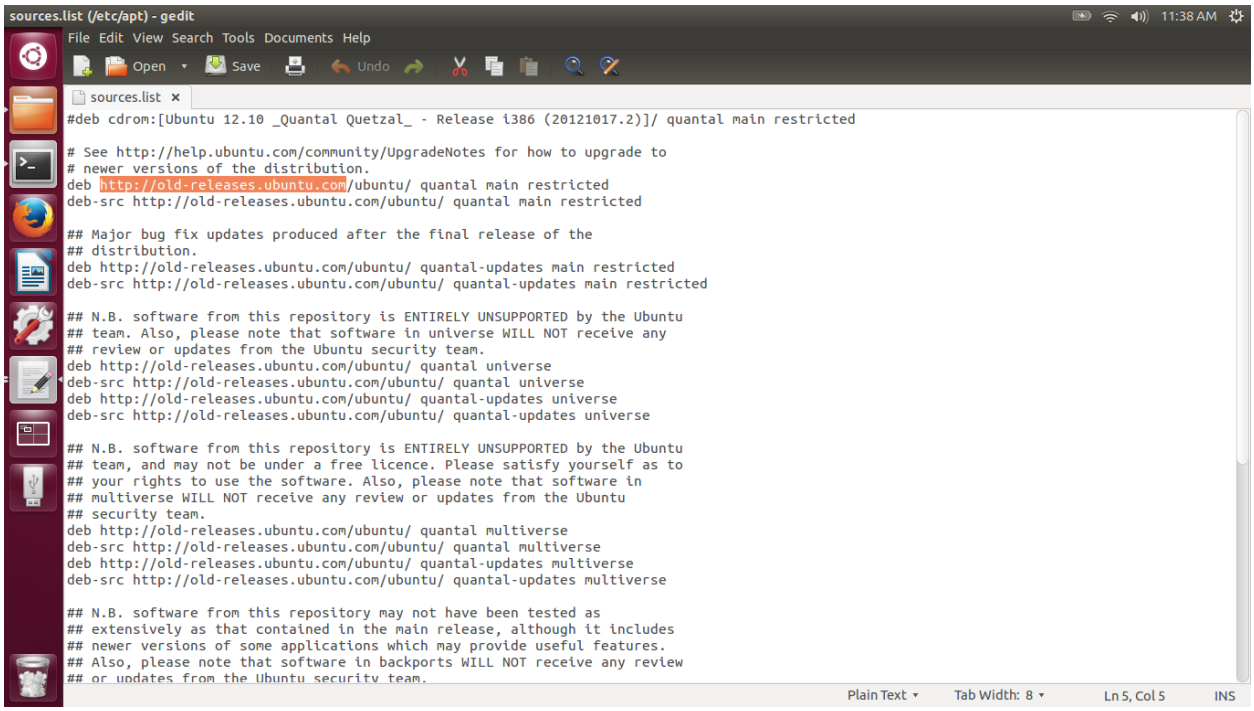

Ilustración 4-11. Corrección de sources.list para actualizar Ubuntu 12.10

Por último, se ejecuta la línea con *upgrade*, que consiste simplemente en la instalación de los paquetes que se han descargado anteriormente. Si el paso anterior se hizo correctamente, esta línea no dará errores.

4.3.2 Instalación de paquetes adicionales

Dado el objetivo final de instalar Xenomai, ACADO y los drivers de la tarjeta de adquisición de datos, tenemos que instalar varios paquetes esenciales para poder compilarlos. Mientras que generalmente estos paquetes se instalan con el mismo comando *apt-get* que hemos visto antes, en este caso vamos a tener que hacerlo manualmente. Y es que el comando *apt-get* no busca exactamente el paquete más actualizado, sino el último paquete que se actualizó durante la vida de la versión de Ubuntu que se esté usando.

El problema al que nos vamos a enfrentar es que los drivers de la tarjeta de adquisición de datos piden un Ubuntu hasta la versión 12 como máximo. Sin embargo, ACADO requiere un par de paquetes paquetes actualizados de Ubuntu 14 o superior, en concreto Cmake [5] y GCC [6]. Por suerte, estos paquetes se pueden instalar todavía en Ubuntu 12 con una instalación manual, así que vamos a detallar por completo el proceso en este apartado.

Primero vamos a instalar dos paquetes básicos que nos servirán para poder instalar Cmake y GCC. Estos dos paquetes sí se pueden conseguir fácilmente con *apt-get*.

```
apt-get install build-essential
apt-get install libmpc-dev
```

El segundo paso es obtener e instalar la versión 3.2.2 de Cmake[9], concretamente bajamos el archivo "cmake-3.2.2.tar.gz". Podemos guardarlo por ejemplo en el mismo escritorio. Hacemos clic derecho, y extraemos el archivo comprimido. Cuando acabe, tendremos una carpeta llamada "cmake-3.2.2". Volvemos al terminal y ejecutamos estas líneas.

```
cd Desktop/cmake-3.2.2
./configure
make
make install
```

[9] https://cmake.org/files/v3.2/ [Último acceso: mayo 2016]

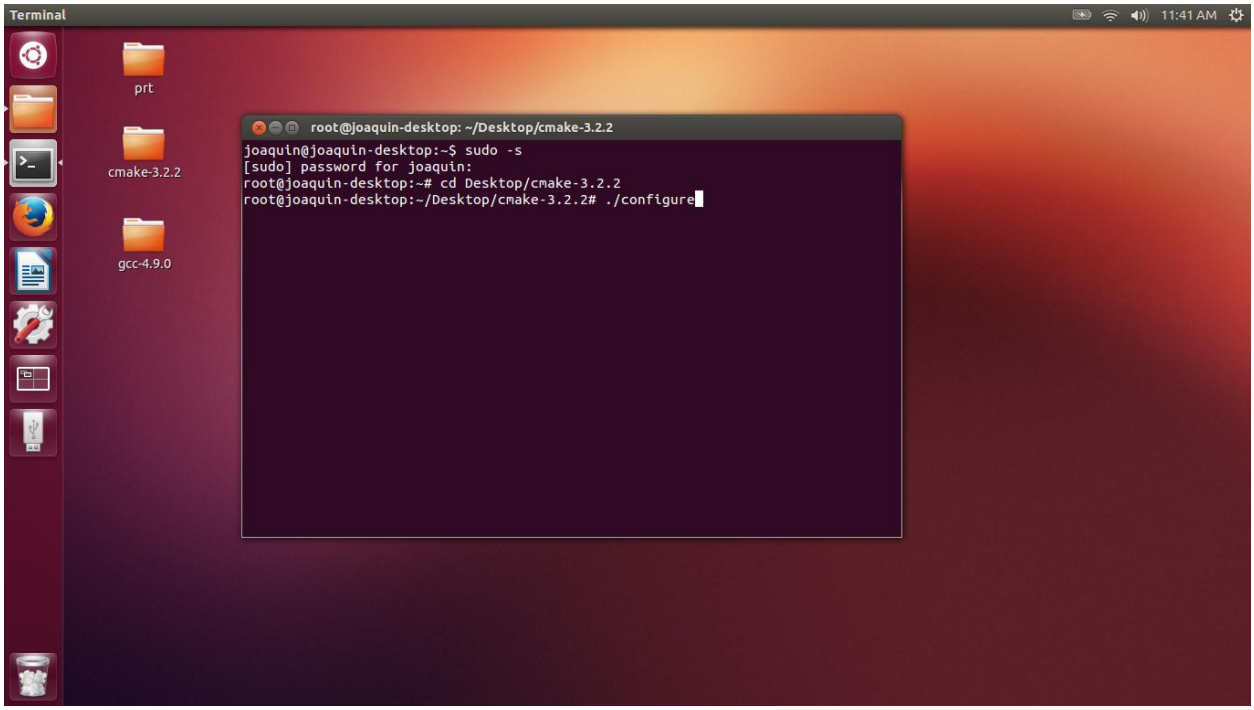

Ilustración 4-12. Instalación de Cmake 3.2

La primera línea sirve para entrar con el terminal a la carpeta donde se encuentran los archivos de instalación. Una vez dentro, ejecutamos por orden las otras 3 líneas, que forman parte del proceso de instalación. Es muy común utilizar estas líneas para instalar programas en Ubuntu. Cada línea lleva un poco de tiempo para ejecutarse, especialmente *make*. No hace falta estar atento a todas las líneas que aparezcan en el terminal, puesto que si hay un error crítico se cancelará la instalación, y podremos leer cuál es la causa. Dependiendo de la versión de Ubuntu pueden ocurrir errores de falta de paquetes de preinstalación, lo cual se soluciona con apt-get y el nombre del paquete al que se hace referencia. Si se utilizan las mismas versiones que en este proyecto y se han seguido todos los pasos anteriores, la instalación se realizará con éxito.

El tercer paso es obtener e instalar la versión 4.9.0 de GCC[10], concretamente necesitamos el archivo "gcc-4.9.0.tar.gz". Su instalación es análoga al paquete Cmake. Guardamos en el escritorio, extraemos y ejecutamos las líneas en el terminal.

```
cd Desktop/gcc-4.9.0
./configure
make
make install
```

[10] ftp://ftp.gnu.org/gnu/gcc/gcc-4.9.0/ [Último acceso: mayo 2016]

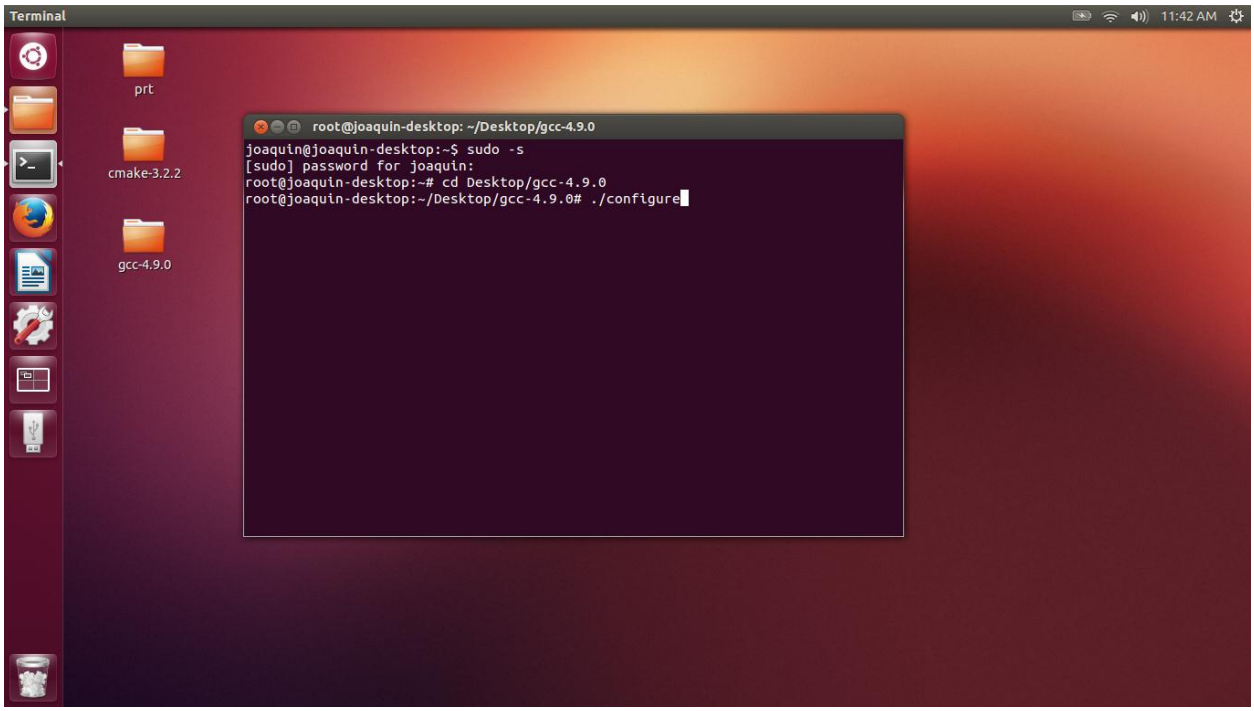

Ilustración 4-13. Instalación de GCC 4.9

Este paquete es mucho más gordo que el anterior, y tras hacer *make* es posible que se requiran hasta 2 horas para compilar todo lo necesario.

En el último paso vamos a hacer una comprobación de que Cmake y GCC se han instalado correctamente y además Ubuntu ha puesto como versión por defecto la que acabamos de instalar. Escribiendo las siguientes dos líneas obtendremos la versión que reconoce Ubuntu.

```
cmake --version
gcc --version
```

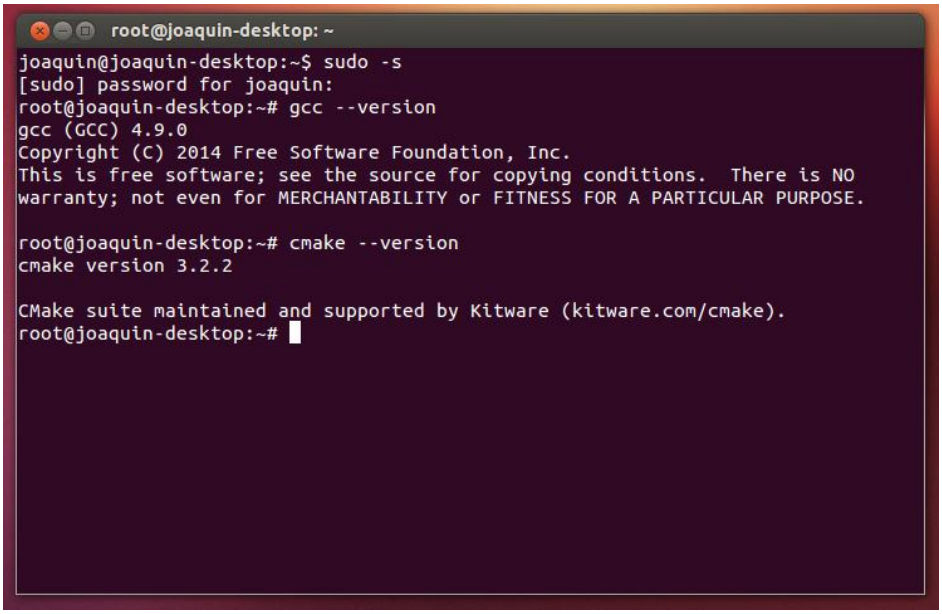

Ilustración 4-14. Comprobación de instalación con éxito

Y el resultado que buscamos es que aparezcan las versions 3.2.2 y 4.9.0, respectivamente. En nuestro caso

aparece correctamente desde el principio, pero vamos a explicar igualmente por qué esto puede no ocurrir y cómo solucionarlo. Si se ha instalado previamente versiones de Cmake o GCC con *apt-get*, se descargará un paquete según la versión de Ubuntu que, aunque pueda ser más antiguo, Ubuntu lo reconocerá como la última versión disponible. Para decirle a Ubuntu que utilice la versión que acabamos de descargar, podemos usar las siguientes líneas (ejemplo para GCC):

```
cd /usr/bin
rm gcc g++ cpp
ln -s gcc-4.9 gcc
ln -s g++-4.9 g++
ln -s cpp-4.9 cpp
```

El comando *ln* sirve para establecer enlaces entre archivos, en concreto con la opción *-s* se crea un enlace simbólico. Esto servirá para que Ubuntu coja la versión 4.9 del compilador y no otra que se haya instalado anteriormente o que venga por defecto en otro paquete.

Existe también un último problema que hemos encontrado y que no debería ocurrir, pero por algún motivo siempre nos lo encontramos cada vez que hemos terminado de instalarlo todo. Se trata de un error con la dirección de una librería que no salta hasta que se ejecuta ACADO. Para mantener estos pasos ordenados, vamos a describir cómo se soluciona el problema en este mismo apartado.

La librería dinámica libstdc++, que viene por defecto en Ubuntu y que se debe actualizar con GCC 4.9, no se actualiza correctamente. Esta librería, en la versión de Ubuntu 12 que utilizamos, se encuentra en la carpeta */usr/lib/i386-linux-gnu/*, sin embargo el paquete GCC 4.9 la instala en */usr/local/lib/*. Además, esta librería consiste en dos archivos, llamados *libstdc++.so.6* y *libstdc++.so.6.0.20* (versión actualizada). En la carpeta donde supuestamente se encuentra la versión antigua, debemos comprobar que vienen estos archivos. Si viene únicamente el primer archivo acompañado de otro con versión 6.0.17 (por defecto en Ubuntu 12), significa que la actualización no se ha realizado donde debería. La solución es simple: copiamos los archivos de la versión actualizada y los ponemos también en la carpeta donde están los antiguos. Este paso debe hacerse desde el terminal.

```
cd /usr/local/lib/
cp -r libstdc++.so.6 /usr/lib/i386-linux-gnu/
cp -r libstdc++.so.6.0.20 /usr/lib/i386-linux-gnu/
```

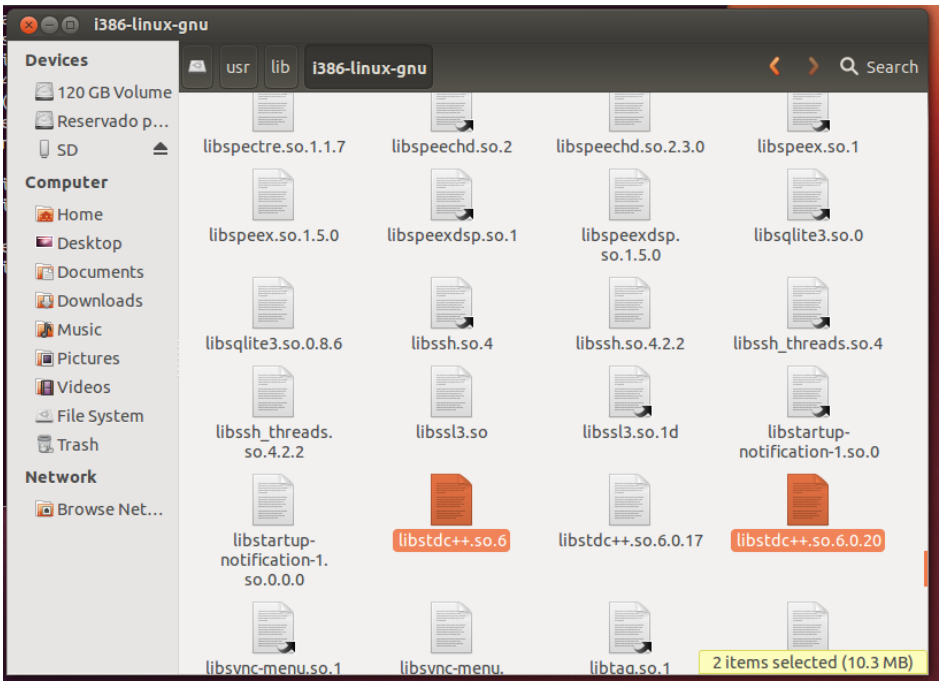

Ilustración 4-15. Actualización de la librería libstdc++

4.4 Instalación de ACADO toolkit

La instalación de ACADO es sencilla si se usa una versión de Ubuntu actualizada, y los pasos a seguir se encuentran en su misma página web[11]. De nuevo, vamos a encontrar más dificultades porque utilizamos una versión obsoleta, pero los pasos siguen siendo lo mismos, solo que en lugar de instalar paquetes automáticamente con *apt-get*, tendremos que hacerlo manual. Con Cmake y GCC instalados en la versión que ellos nos piden, lo que nos queda es lo siguiente.

```
sudo -s
apt-get install git
git clone https://github.com/acado/acado.git -b stable ACADOtoolkit
```

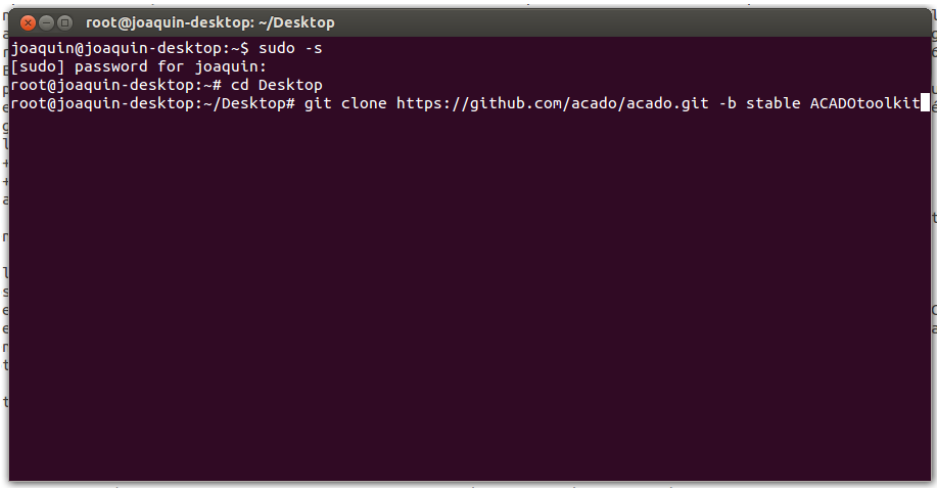

Ilustración 4-16. Descarga de ACADO Toolkit

Se descargará el paquete de instalación de ACADO en el directorio actual en el que nos encontremos (en la captura de pantalla, se ha realizado en el escritorio), en la carpeta ACADOtoolkit. También existen 3 paquetes adicionales que no son necesarios para correr ACADO, pero permiten entre otras cosas la visualización gráfica. La visualización gráfica es realmente interesante así que no la vamos a dejar atrás. El comando que te recomienda utilizar ACADO para instalar estos paquetes es el siguiente.

```
apt-get gnuplot doxygen graphviz
```

Sin embargo, como ya hemos comentado anteriormente, *apt-get* escoge versiones antiguas en una versión de Ubuntu obsoleta, por lo que tenemos que hacer la instalación manual igual que con Cmake y GCC. En Ubuntu 12 y a día de hoy, hemos instalado Doxygen con *apt-get* sin problemas, pero Gnuplot y Graphviz requieren una instalación manual más actualizada.

4.4.1 Instalación de Gnuplot y Graphviz

Gnuplot [7] y Graphviz [8] forman parte del entorno gráfico que nos ayudará a obtener gráficas con los resultados que nos proporcione ACADO. Pero antes de instalarlas, necesitamos instalar la librería básica de gráficos LibGD [9], ya que no venía en ninguna de las actualizaciones esenciales previas. Vamos a instalar primero dos paquetes con *apt-get*, y luego haremos una actualización manual de LibGD que nos servirá para que Gnuplot pueda sacar los resultados en formato PNG, entre otros.

```
apt-get install libgd2-xpm-dev
apt-get install php5-gd
```

[11] http://acado.github.io/install_linux.html [Último acceso: junio 2016]

Para la actualización manual de LibGD, que repetimos, solo será necesario en caso de usar una versión antigua de Ubuntu, visitamos la zona de descargas de su misma página web[12]. Usamos la versión 2.1.0, ya que es una de las versiones más actuales y será compatible con el resto. Los pasos de instalación son los mismos que hemos estado utilizando durante todo este apartado. Accedemos desde el terminal a la carpeta que se obtiene al descomprimir del archivo descargado, y ejecutamos las líneas:

```
./configure
make
make install
```

Ahora realizamos la instalación manual de Graphviz. De nuevo tenemos que coger una de las versiones más nuevas para que no haya problemas de compatibilidad, en este caso la 2.38.0[13]. Se descarga desde su misma página web, se extrae el archivo comprimido, y accedemos a la carpeta desde el terminal. Ejecutamos:

```
./configure
make
make install
```

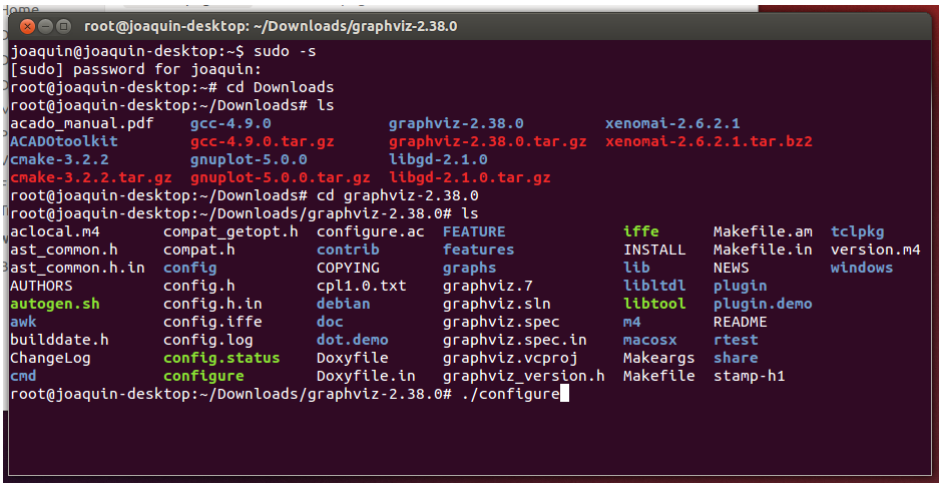

Ilustración 4-17. Instalación de Graphviz

Y, finalmente, la instalación de Gnuplot. Hemos elegido la versión 5.0.0[14]. Descargamos, descomprimimos, accedemos desde el terminal y ejecutamos las mismas líneas de instalación:

```
./configure
make
make install
```

[12] https://github.com/libgd/libgd/releases/download/gd-2.1.0/libgd-2.1.0.tar.gz [Último acceso: junio 2016]
[13] http://www.graphviz.org/pub/graphviz/stable/SOURCES/graphviz-2.38.0.tar.gz [Último acceso: junio 2016]
[14] https://sourceforge.net/projects/gnuplot/files/gnuplot/5.0.0/gnuplot-5.0.0.tar.gz/download [Último acceso: junio 2016]

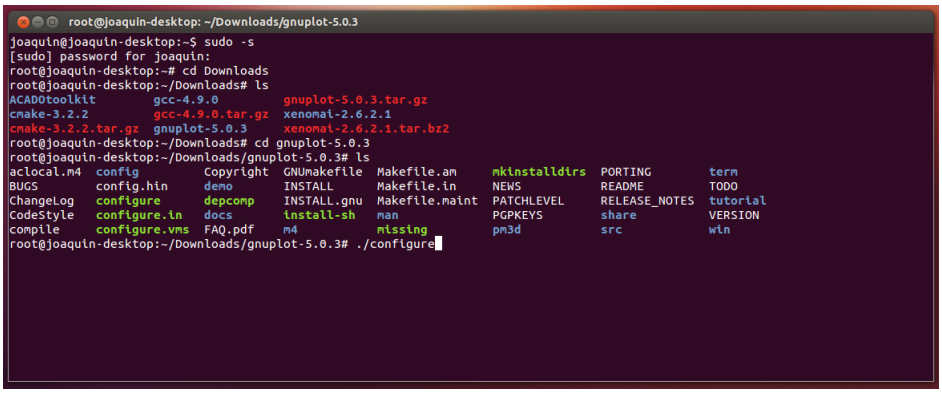

Ilustración 4-18. Instalación de Gnuplot

4.4.2 Compilación de ACADO Toolkit

Una vez instalado lo referente al entorno gráfico, volvemos con el terminal a la carpeta "ACADOtoolkit" donde se descargó ACADO originalmente. Ejecutamos las siguientes líneas para terminar la instalación:

```
mkdir build
cd build
cmake ..
make
```

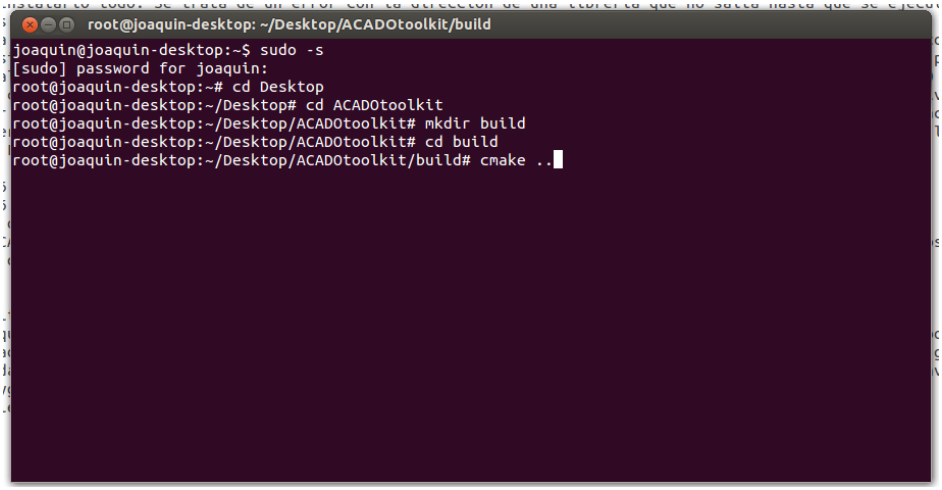

Ilustración 4-19. Instalación de ACADO Toolkit

Cuando se ejecute *make*, la última línea, se instalará ACADO. Normalmente tardará hasta 15 minutos, y además va indicando el progreso con un porcentaje. Si existe un error, la instalación se cancelará y aparecerá el mensaje de error. Si se han realizado todos los pasos anteriores, la instalación llegará al 100% y no devolverá ningún error.

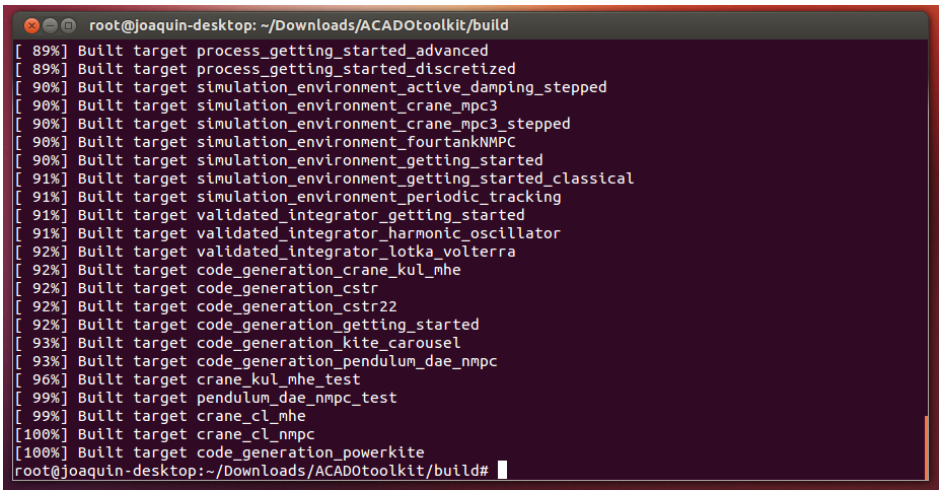

Ilustración 4-20. Éxito de instalación de ACADO Toolkit

Y por ultimo, se pueden ejecutar estas tres líneas para cargar un ejemplo predeterminado y comprobar que en efecto se ha instalado correctamente.

```
cd ..
cd examples/getting_started
./simple_ocp
```

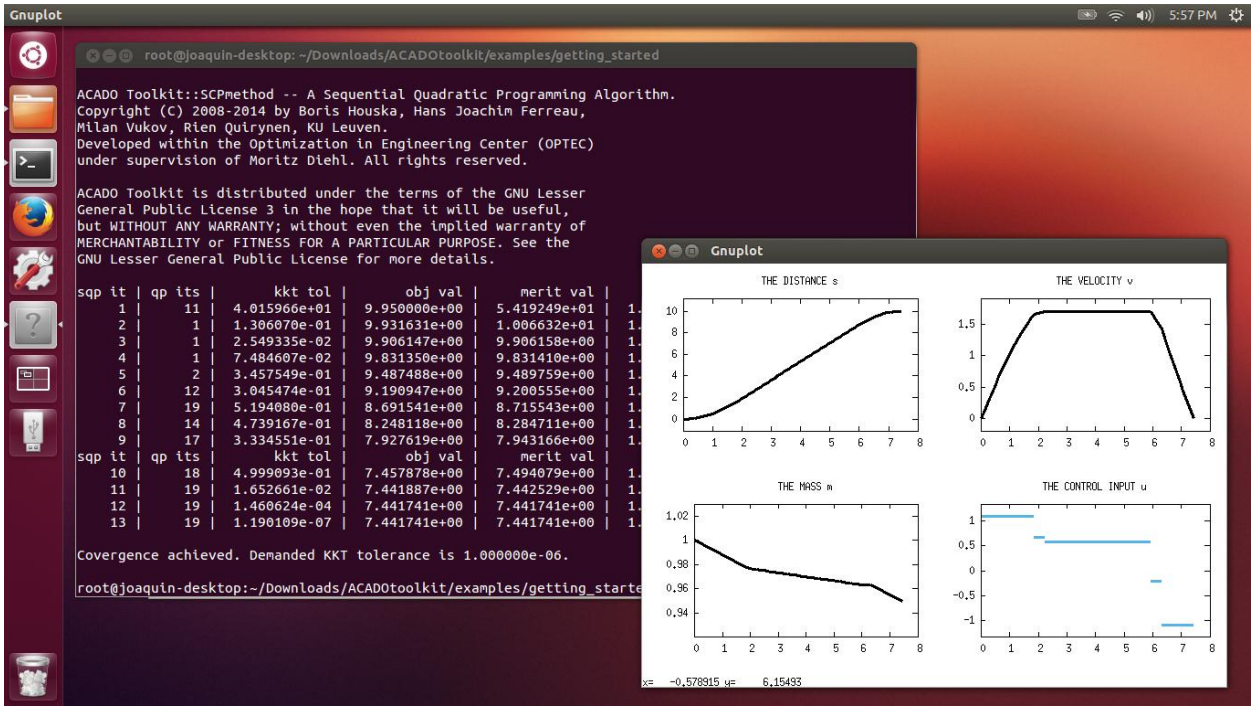

Ilustración 4-21. Cargando un primer ejemplo de ACADO Toolkit

Como podemos observar, el ejemplo se ejecuta exitosamente, y además Gnuplot carga correctamente las gráficas con el resultado.

4.5 Instalación de Xenomai

Puesto que la instalación de Xenomai modifica el kernel de Linux, hemos dejado este paso para el final para así

evitar problemas. El parche que vamos a instalar de Xenomai tiene que corresponder con la versión del kernel de Linux que estemos usando, así que vamos a poder hacer una instalación cómoda usando apt-get. El paquete se llama "linux-patch-xenomai" y lo podemos instalar ejecutando las siguientes líneas:

```
sudo -s
apt-get install linux-patch-xenomai
```

Pedirá una confirmación de instalación, por lo que tendremos que pulsar la tecla *Y* y después *Enter*.

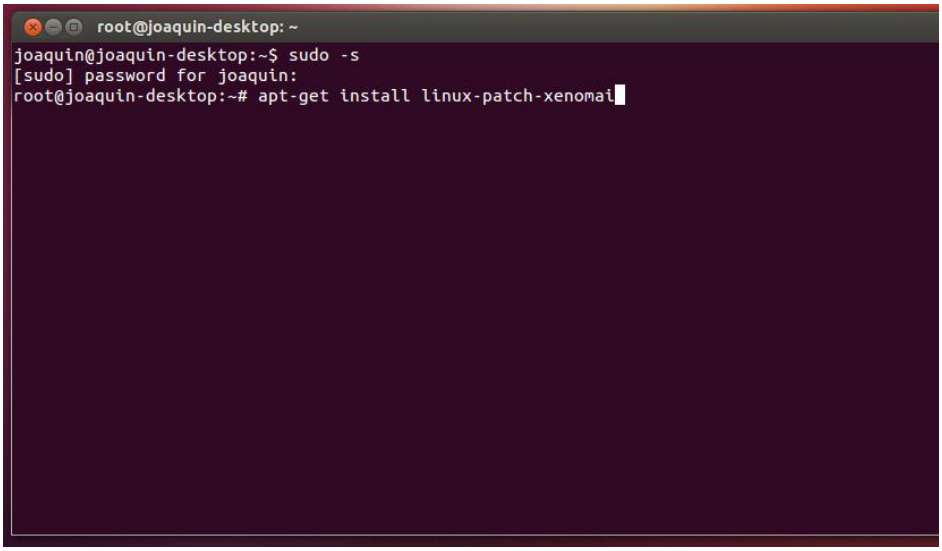

Ilustración 4-22. Instalación de Xenomai

Para comprobar que se ha instalado correctamente basta con correr algunos de los ejemplos que se expondrán en el último capítulo de la memoria.

4.6 Instalación del driver de la tarjeta de adquisición de datos

Por último, para cerrar este capítulo de preparación del software, se va a explicar cómo instalar el driver de la tarjeta de adquisición de datos, ya sea esta la que trae incorporada el PC-104 o una tarjeta USB externa que se conecte adicionalmente. Lo primero será descargar el paquete de drivers DAQNavi para Linux desde la web de Advantech[15]. El archivo que buscamos es "linux_driver_source_3.2.8.0_32bit.zip", que guardaremos en la carpeta "home". Después, extraemos el fichero y obtendremos en el mismo escritorio una carpeta con el mismo nombre.

La instalación sigue los siguientes pasos:

```
sudo -s
cd linux_driver_source_3.2.8.0_32bit
cd drivers/driver_base/src/lnx_ko
make
cd ../../../usb4702_usb4704/scr/lnx_ko
make
```

Los dos *make* sirven para compilar el driver. Nótese que en la carpeta "drivers" se encuentran todas las tarjetas de adquisición de datos para el PC-104. Para instalar la tarjeta propia del PC-104 (módulo PCM-3718) que mostramos en el capítulo 3 de esta memoria, debemos entrar en la carpeta "drivers/pcm3718", aunque en este

[15] http://support.advantech.com.tw/Support/DownloadSRDetail_New.aspx?SR_ID=1-LXHFQJ&Doc_Source=Download [Último acceso: junio 2016]

ejemplo se ha instalado otra tarjeta enterna llamada USB-4702[16]. Su funcionamiento e instalación es totalmente análogo al del módulo PCM-3718, y la única diferencia la vamos a notar en algunas características de toma de datos y en el número de entradas y salidas.

```
cd ../../../bin
insmod biokernbase.ko
insmod bio4704.ko
```

Estas líneas instalan el driver compilado. Una vez que esté instalado ya se puede conectar la tarjeta de adquisición de datos, de lo contrario Ubuntu podría asociar otro driver por defecto que impediría que funcionase. Si eso ocurre, podemos solucionarlo con la siguiente línea:

```
rmmod biokernbase
```

Por último, debemos copiar las librerías del driver al directorio de librerías de Ubuntu.

```
cd ../../
cp libs/* /usr/lib/
cp inc/* /usr/include/
lsusb
```

Con eso quedan los drivers instalados. Para comprobar que la tarjeta funciona, vamos a conectarla al PC-104 (solo en caso de ser la tarjeta USB, si es la PCM-3718 ya estaría conectada) y vamos a ejecutar estos dos comandos.

```
lsusb
lsmod
```

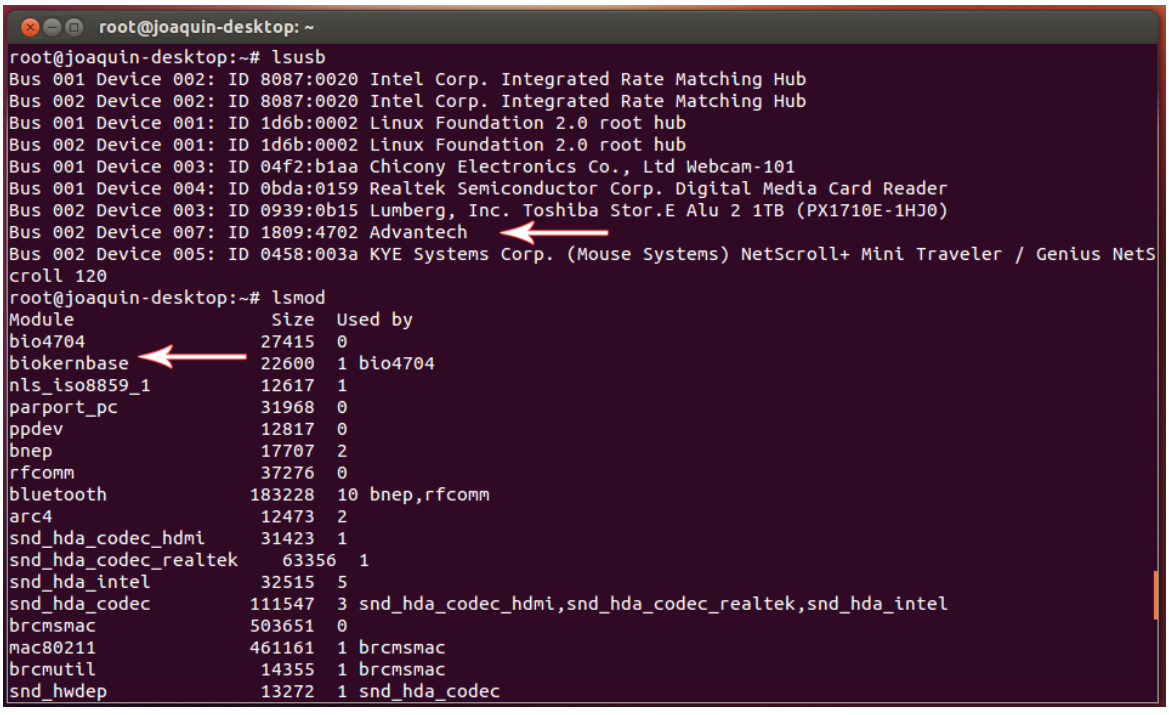

Ilustración 4-23. Instalación del driver de la tarjeta de adquisición de datos

Tras hacer *lsusb*, si está la tarjeta conectada, debería aparecer un dispositivo de descripción "Advantech". Cuando ejecutamos el comando *lsmod*, debemos comprobar que esté instalado *biokernbase* y *bio4704* (este

[16] http://www.advantech.com/products/1-2mlkno/usb-4702/mod_807adf3c-b073-4671-ad72-10d0930dc8f6 [Último acceso: junio 2016]

último para el caso de la tarjeta USB-4702, que es el mismo driver que para la versión 4704).

Para terminar la comprobación, vamos a compilar y ejecutar uno de los ejemplos que trae el driver. Vamos a probar, por ejemplo, la captura instantánea de entrada analógica. Para acceder al ejemplo vamos al siguiente directorio:

```
cd linux_driver_source_3.2.8.0_32bit/examples/C++_Console/AI_InstantAI
```

Abrimos el fichero "InstantAI.cpp" y modificamos lo que nos piden en las instrucciones que trae comentadas al comienzo del código. El siguiente código ya está preparado para que funcione con la tarjeta instalada.

```
#include <stdlib.h>
#include <stdio.h>
#include "../inc/compatibility.h"
#include "../../../inc/bdaqctrl.h"
using namespace Automation::BDaq;

#define      deviceDescription  0
int32        startChannel = 0;
const int32  channelCount = 3;

inline void waitAnyKey()
{
   do{SLEEP(1);} while(!kbhit());
}
int main(int argc, char* argv[])
{
   ErrorCode        ret = Success;

   InstantAiCtrl * instantAiCtrl = AdxInstantAiCtrlCreate();

   do
   {
      DeviceInformation devInfo(deviceDescription);
      ret = instantAiCtrl->setSelectedDevice(devInfo);
      CHK_RESULT(ret);

      printf("Acquisition is in progress, any key to quit!\n\n");
      double   scaledData[channelCount] = {0};
      int32 channelCountMax = instantAiCtrl->getFeatures()->getChannelCountMax();

      do
      {
         ret = instantAiCtrl->Read(startChannel,channelCount,scaledData);
         CHK_RESULT(ret);

         for (int32 i = startChannel; i< startChannel+channelCount;++i)
         {
            printf("Channel %d data: %10.6f\n", i % channelCountMax, scaledData[i-
          startChannel]);
         }
         printf("\n");
         SLEEP(1);
      } while(!kbhit());
   }while(false);

    instantAiCtrl->Dispose();
    if(BioFailed(ret))
    {
        printf("Some error occurred. And the last error code is Ox%X.\n", ret);
        waitAnyKey();
    }
   return 0;
}
```

Finalmente, guardamos el fichero y lo compilamos con la siguiente línea.

```
g++ -o InstantAI InstantAI.cpp -lbiodaq
```

Tras ejecutarlo, sin todavía tener conectada la tarjeta de adquisición a alguna toma de datos, debería imprimir lo siguiente por el terminal:

```
./InstantAI
```

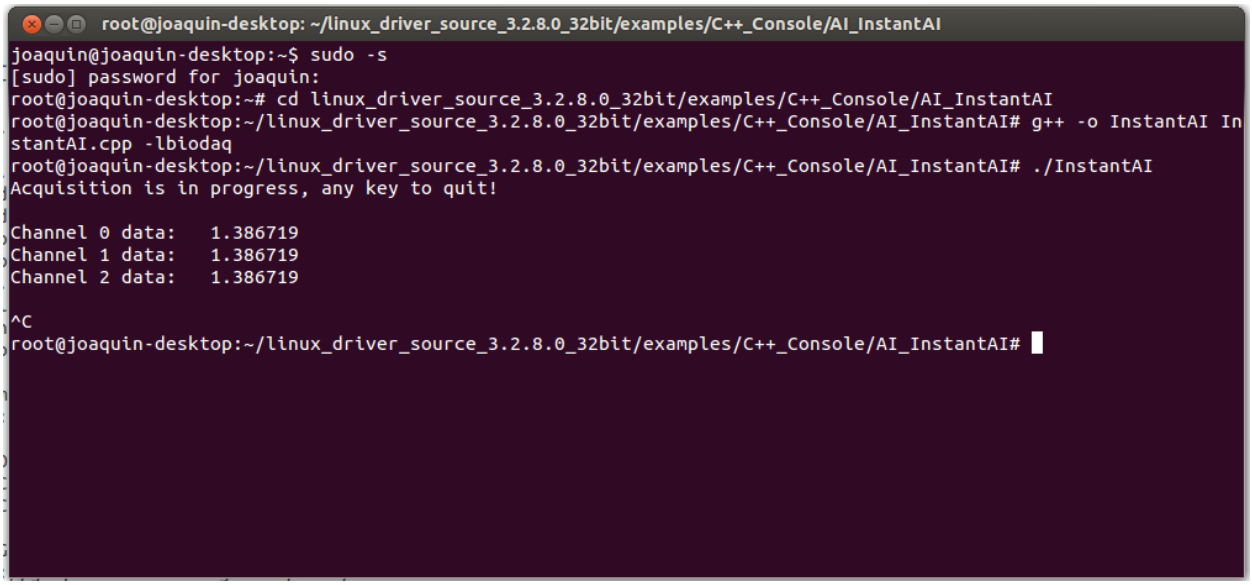

Ilustración 4-24. Ejecutando un ejemplo para la tarjeta de adquisición de datos

5 LIBRERÍA DE TIEMPO REAL

Esta es la parte más importante del proyecto. Vamos a exponer el desarrollo de la librería de tiempo real, justificando su funcionamiento y detallando su modo de uso para crear procesos de control en tiempo real.

5.1 Librería: definición y objetivos

Una librería o biblioteca en C recoge diversos archivos de cabecera y paquetes internos necesarios para que el compilador entienda e interprete el código de programa que se ha escrito. Las librerías básicas contienen información para realizar operaciones esenciales tales como la lectura por teclado o diversas funciones matemáticas. Estas operaciones son muy complicadas si se configuran siempre desde cero. En la informática y en otras muchas áreas de la tecnología existe la ventaja de poder englobar la programación en estructuras o módulos sencillos de manejar sin que sea necesario conocerlos al completo. En otras palabras, estas librerías o cabeceras permiten seguir contruyendo a partir de lo que ya está hecho. Y es por eso por lo que esta es la parte más interesante del proyecto. Hasta ahora hemos mirado atrás a lo que ya está hecho, y a partir de ahora vamos a contruir algo nuevo.

Una librería recoge otros archivos con funcionalidades que ya se han diseñado, y se basa en ellos para construir otras nuevas. Esta librería en concreto recoge las funcionalidades de tiempo real y mejora su accesibilidad al ingeniero de control.

¿Por qué es necesario el desarrollo de esta librería? La programación en tiempo real requiere un estudio muy amplio de la materia para llegar a implementar cualquier proceso. Debido al gran abanico de posibilidades que ofrece, el nivel de dificultad es relativamente alto. Mientras que un ingeniero de control tiene muchos conocimientos de su área, y tiene también muchas habilidades de programación, no es de extrañar que no esté familiarizado con la programación de tiempo real, aunque sí sepa muy bien cómo funciona. El objetivo principal de este proyecto y de la librería es que el ingeniero de control tenga un acceso fácil a las funcionalidades de tiempo real sin necesidad de estudiar su campo de programación.

¿Cómo ayuda esta librería al ingeniero? Aunque la programación en tiempo real ofrezca un sinfín de posibilidades en multitud de aplicaciones, las estructuras de programación en el ámbito de control son, desde un punto de vista que veremos a continuación, muy similares. Entonces, sería posible el desarrollo de una librería que implemente estas funcionalidades sin que haya que programarlas siempre desde cero.

¿Cuánto simplifica esta librería el trabajo al ingeniero de control? Veremos que las características necesarias que pide el ingeniero se realizan con simples llamadas de funciones intuitivas que ocupan una línea y tienen un nombre fácil de recordar. Se pretende que el ingeniero pueda adaptar sus algoritmos de control a programación en tiempo real añadiendo varias nuevas funciones de fácil uso y que se entienden con conocimientos y conceptos básicos de tiempo real, sin entrar en su modo de programación.

A continuación, se irán detallando estas funcionalidades necesarias, con qué función de la librería se solucionan y cómo se han programado internamente.

5.2 Estructura y presentación general de la librería

Cuando se programa un lazo de control simple vemos siempre una estructura que se repite: primero se definen las condiciones iniciales y se inicializan ciertas variables informáticas, y después se entra en un bucle continuo de control que se ejecuta cada periodo de muestreo. En cada ciclo, dependiendo de la complejidad que se haya dado, se van a hacer generalmente estas tres operaciones por orden: lectura de una entrada, cálculo de la variable de control y envío de una salida.

Un programa simple con un único algoritmo de control solo debe cumplir una especificación de tiempo real: las tres tareas realizadas en el bucle deben tardar en conjunto un tiempo menor que el tiempo de muestreo.

Puesto que es una especificación muy simple y fácil de resolver, nunca se va a necesitar de programación en tiempo real para realizar un problema simple. El tiempo real es necesario cuando se ejecutan múltiples tareas a la vez, ya que en este nuevo problema aparecen nuevas especificaciones de tiempo real mucho más difíciles de cumplir, y se requiere de programación más avanzada para resolverlo.

Veamos con más detalle el caso de múltiples lazos de control corriendo a la vez. Cada lazo de control puede compartir o no variables de entrada o salida con el resto, y pueden tener cada uno diferentes tiempos de muestreo. Podemos ejecutar cada uno de ellos por separado y comprobar que funcionan y cumplen la primera especificación de tiempo real. Pero ello no significa que funcionen cuando todos se están ejecutando. Cuando el tiempo de muestreo salta para el primer lazo de control, hay una tarea que aparece y ocupa trabajo en el procesador durante un tiempo que ya hemos comprobado que es menor que su tiempo de muestreo. Pero puede que en este mismo momento salte el segundo lazo de control, pero como todavía se está calculando el primero, el segundo debe esperar a que termine y sufrirá un retraso. Y puede que todavía haya más lazos que salten a la vez. Esta situación presenta una solución que todavía es posible conseguir sin la programación en tiempo real, y consiste en sincronizar los tiempos de muestreo para evitar que los lazos se solapen. Esto solo es posible en el caso ideal de que se conozcan todos los lazos y sus tiempos de muestreo, y que su tiempo en el procesador sean relativamente pequeños y siempre el mismo.

En un caso más real, habrá acciones que se ejecuten con un tiempo desconocido a priori: emergencias, avisos, o diversas entradas manuales; y otras que sí se conozcan: bucles de control. Además, no siempre se cumplirá que los tiempos en el procesador sean muy pequeños y habrá veces que no haya tiempo para realizar una tarea en un momento determinado. Cuando además se añaden controladores más complejos como un control predictivo no lineal, el tiempo de ejecución o el tiempo de procesador que necesita la tarea se vuelve variable. Estos problemas no son realizables desde una programación clásica en la que el sistema no es consciente del tiempo, pero es posible manejarlos con la programación en tiempo real. En concreto, este problema se solucionará con la definición de prioridades, que es una caracterítica propia del tiempo real que permite completar tareas críticas dejando atrás las menos importantes en caso de que el procesador se sature en un momento concreto.

Para facilitar el entendimiento entre el usuario y la librería, se han llamado tareas (*task*) a cualquier proceso simple que se vaya ejecutar. Esta tarea puede ser tanto un lazo de control como una espera de una señal de emergencia. Las tareas podrán llevar asociadas un tiempo de muestreo en caso de ser un lazo de control, o bien una espera a una interrupción en cualquier otro caso. Además, las tareas llevarán asociadas un número de prioridad, y se cumplirá siempre que una tarea de menor prioridad se pausará cuando una tarea de mayor prioridad salte al procesador.

Se han creado dos objetos de tiempo real correspondientes al tiempo de muestro (temporizador, *timer*) y a la sincronización por interrupciones (evento, *event*). Tareas, temporizadores y eventos son tres objetos de tiempo real que se han definido en la librería, y se crean y se manejan con funciones muy sencillas que permiten al programador trabajar con estos conceptos de manera intuitiva. Tanto los temporizadores como los eventos funcionarán con interrupciones.

Se muestra un gráfico a continuación para visualizar el funcionamiento.

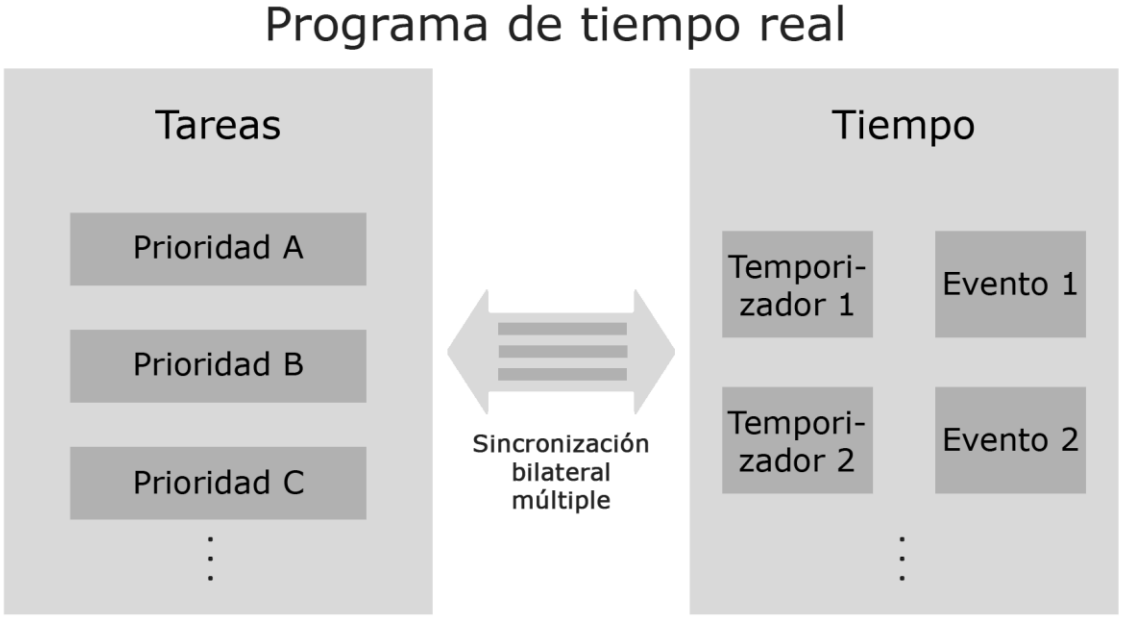

Ilustración 5-1. Esquema de la librería de tiempo real

Cualquier programa que se realice sobre la librería puede descomponerse en dos bloques principales: las tareas y los objetos de tiempo para sincronización.

Las tareas puedes entenderse como funciones, hilos o trozos de código independientes que realizan una función en concreto, que pueden compartir variables en memoria que se definan como globales. Estas tareas tienen un número de prioridad asociado, y se cumplirá siempre que en cualquier instante dado la tarea que se está ejecutando es aquella que cumple estos dos requisitos:

- Ha pedido permiso de ejecución o no está en estado de espera.
- De entre todas las que están en estado de espera, es la que tiene mayor número de prioridad.
- Comparte el mayor número de prioridad con otras tareas. En este caso se ejecutarían en paralelo.

Los otros dos objetos de tiempo real son los temporizadores y los eventos. Estos deben ser iniciados y configurados desde cualquier tarea, y cualquier tarea puede leerlos y recibirlos. Un temporizador es un objeto que se envía una señal determinada con una frecuencia preestablecida. La señal y su frecuencia de envío han de ser inicializadas junto con la creación del temporizador, y cualquier tarea puede hacerlo. Una vez que se envía la señal, cualquier tarea puede recibirla siempre y cuando se haya programado en esta la acción de esperarla. Un evento tiene un comportamiento muy similar, pero en lugar de mandar una señal con un periodo de repetición, solo envía una tras un tiempo de espera.

5.3 Programación con la librería de tiempo real adaptada a control

En este apartado se verá cómo se utiliza la librería de tiempo real de cara al usuario. Se presentarán pseudocódigos y estructuras a seguir para facilitar la compresión, y posteriormente se explicará el uso de las funciones y cómo podemos resolver un problema de tiempo real. La librería está adaptada tanto a C como a C++, pero en este proyecto se datallarán y se mostarán ejemplos únicamente de la versión de C++. Las diferencias entre ambas son mínimas y se pueden apreciar a simple vista en el código fuente, si se desea utilizar la versión en C.

Aunque el proyecto se haya realizado completamente en castellano, se ha decidido que los nombres y las funciones de la librería estén en inglés. No obstante, las palabras en inglés son pocas y ya se han presentado en

el apartado anterior, y toda la explicación a continuación seguirá en castellano.

Un programa general que se base en esta librería tendrá la siguiente estructura:

```
#include "libRTC.hpp"

// Variables globales

main {
     // Funciones de inicialización del programa de tiempo real
}

Tarea1 {
     // Código de tarea 1
}

Tarea2

...
```

En la librería (archivo libRTC.hpp) se incluyen tanto las varias librerías de tiempo real como otras básicas de C y matemáticas. Si se requieren más librerías para implementar más funcionalidades, solo habría que añadirlas con el mismo comando *#include*.

Las variables globales se colocan fuera de main. El comportamiento local de las variables es el de C: las declaradas dentro de main solo sirven para main, las declaradas dentro de cada tarea sirven solo para esa tarea, y las globales sirven para todos. Si los algoritmos de control son independientes, se pueden declaran sus variables dentro de sus tareas, pero si varios algoritmos comparten alguna variable, ha de ser declarada como global.

El bloque main realmente no es una tarea con una prioridad asignada, ni tiene comportamiento de tiempo real. Pero al iniciar el programa se comporta como un bloque de máxima prioridad, y por tanto se usará para inicializar la configuración de tiempo real y luego se quedará completamente dormido para que los recursos se distribuyan al resto de tareas.

Las tareas se programan como si fueran funciones en C++. Sin embargo, la librería de tiempo real entiende estas funciones como hilos (en inglés, *threads*) y requieren un tratado especial, y una de sus restriccciones es que no es posible enviar múltiples parámetros por referencia. En su lugar, se utilizarán variables globales.

5.3.1 Creación de una tarea

Para crear una tarea se necesitan siempre tres pasos: declaración, inicialización y programación.

- La declaración se hará siempre como una variable global al principio del código.
- La inicialización se hará con una función especial dentro y al principio de main.
- La programación de la tarea se hará como una función que se programa debajo de main.

En el siguiente ejemplo simplificado se pueden ver claramente estos pasos:

```
// Declaración
task task_1;                        // nombre
void *control (void*);              // función

// Inicialización en main
void main (void) {
     init_task(&task_1, control, 30);
     // ______^nombre^_^función^_^prioridad
}

// Programación de la tarea
```

```
void *control (void*) {
     // Código de la tarea
}
```

En la declaración, se debe declarar la tarea con un tipo *task*. Se ha elegido como nombre de esta tarea *task_1*. A continuación, se ha declarado la función que hará como código de la tarea, y la hemos llamado *control*. Esta función debe estar declarada necesariamente como puntero, tal y como se muestra en el código de ejemplo. Aunque pueda parecer complicado, la sintaxis es siempre la misma para todos los programas. Nótese que hasta ahora no se ha hecho el enlace entre *task_1* y *control*, solo se han declarado.

Dentro de main aparece la primera función especial de la librería de tiempo real desarrollada. Esta función se llama *init_task*, y recibe los siguientes argumentos:

```
init_task ( task * TASK_NAME , void* (*function)(void*), int
        PRIORITY);
```

Como su nombre indica, es la función que inicializa la tarea. Recibe como primer argumento el objeto inicializado con *task,* con la particularidad de que es un paso por referencia y hay que introducir la dirección del objeto, no el objeto en sí, y por tanto no se debe olvidarse el signo ampersand (&), tal y como se muestra en el ejemplo. Cada tarea nueva que se inicie debe llevar un objeto *task* con nombre diferente para cada tarea. El segundo argumento apunta a la función en la que se encuentra la programación de la tarea. Para introducir este segundo argumento, no es necesario ningún operador (*) o (&), como se ve en el ejemplo. El tercer argumento es un tipo *int* que indica la prioridad de la tarea. Este número solo puede ser un entero comprendido entre el 1 y el 99, siendo el 1 la prioridad mínima y el 99 la prioridad máxima.

Por último, después de main se programará la función *control* que será a su vez la tarea en cuestión. Esta tarea empezará a correr cuando se inicialice con la función correspondiente en main.

5.3.1.1 Funcionamiento de la prioridad

Antes de continuar explicando el resto de funciones, se va a hacer una nota aclaratoria sobre el comportamiento de la prioridad de la librería. Hemos visto en la función de creación de tarea que asignamos un número de prioridad comprendido entre el 1 y el 99, siendo el 99 el número de máxima prioridad. Las tareas se comportarán tal y como se ha detallado hasta ahora: una tarea de máxima prioridad pondrá en pausa una de menor prioridad hasta que termine de ejecutarse, y en cada momento solo se ejecutará la tarea de mayor prioridad de entre las que estén pidiendo permiso de ejecución.

Cuando a varias tareas se les asigna la misma prioridad (el mismo número de prioridad), se entiende que el programador quiere ponerlas a correr a la vez. Por ello, se ha programado internamente en la librería la política de planificación *Round-robin*, en la que múltiples tareas con la misma prioridad tomarán turnos de ejecución para intentar lograr que trabajen paralelamente.

Un aspecto importante a tener en cuenta es el papel de la función main en cuanto a la prioridad entre tareas. Main tiene por defecto asignado una prioridad con el número 0, que no significa prioridad ni máxima ni mínima, sino una política de planificación diferente e independiente a las tareas, cuyo comportamiento no es del todo definido. En numerosas pruebas realizadas, se ha comprobado que main se ejecuta como una tarea de máxima prioridad al principio, siendo ideal para inicializar tareas, objetos y variables iniciales sin que el resto del programa empiece. Pero una vez que sale del procesador, se comporta más bien como una tarea única de política FIFO sin prioridad establecida, en la que entrará en ejecución según el orden de llamada y cuando terminen las tareas en ejecución. Esto quiere decir que no es apropiado utilizar main como tarea de tiempo real.

Como ya veremos más adelante, este detalle no nos va a afectar en la programación, ya la solución es utilizar main para inicializar todo al principio y luego ponerlo a dormir indefinidamente, dejando que las tareas se encarguen del resto cumpliendo con las funcionalidades de tiempo real.

5.3.2 Creación de un temporizador

Un temporizador se define para esta librería como un objeto de tiempo real que manda una señal de interrupción

periódicamente con un periodo determinado. Los temporizadores se usarán para marcar los tiempos de muestreo de diferentes lazos de control.

De un modo similar, para crear un temporizador se necesitan dos pasos, quitando el de programación: declaración e inicialización.

- La declaración se hará siempre como variables globales al principio del código.
- La inicialización se hará con una función especial dentro y al principio de main.

Con un ejemplo simple:

```
// Declaración
timer timer_1;
signal sigtimer_1;

// Inicialización en main
void main (void) {
     create_timer(&timer_1, &sigtimer_1, 0.5);
}
```

Para la declaración completa de un temporizador necesitamos dos tipos: *timer* y *signal*. Tal y como hemos definido más arriba, un temporizador se comunicará con el resto del proceso a través de una señal de interrupción, por lo que de cara a la programación debemos declarar tanto el temporizador como la señal.

En main aparece la función especial *create_timer* que es la que se encarga de crear el temporizador, y recibe los siguientes parámetros:

```
create_timer ( timer * TIMER_NAME , signal * SIGNAL_NAME , double
        TIME );
```

Los dos primeros argumentos reciben los tipos *timer* y *signal* declarados al principio, con la particularidad de que es un paso por referencia y hay que introducir la dirección de los objetos, no los objetos en sí, y por tanto no se debe olvidarse el signo ampersand (&), tal y como se muestra en el ejemplo. El tercer argumento es un tipo *double* que indica el tiempo de repetición o muestreo en segundos. En el ejemplo, se crea un temporizador llamado *timer_1* que envía la señal de interrupción *sigtimer_1* cada 500 milisegundos. Más tarde se mostrará con qué función se asociarán las tareas con los temporizadores.

5.3.3 Creación de un evento

Un evento se define para esta librería como un objeto de tiempo real que manda una señal de interrupción única al transcurrir un tiempo determinado. Puede entenderse como un cronómetro que se inicia con una cuenta, y una vez que llega al tiempo indicado, manda la interrupción. Los eventos tendrán múltiples usos, comúnmente en tareas manuales.

La creación de un evento es análoga a la del temporizador:

```
// Declaración
event event_1;
signal sigevent_1;

// Inicialización en main
void main (void) {
     create_event(&event_1, &sigevent_1, 7.25);
}
```

Tras la declaración, la función especial *create_event* en main es la que se encarga de crear el evento, y recibe los siguientes parámetros:

```
create_event ( event * EVENT_NAME , signal * SIGNAL_NAME , double
        TIME );
```

La función es muy parecida a la de creación de un temporizador. Los dos primeros argumentos reciben los tipos *event* y *signal* declarados al principio, con la particularidad de que es un paso por referencia y hay que introducir la dirección de los objetos, no los objetos en sí, y por tanto no se debe olvidarse el signo ampersand (&), tal y como se muestra en el ejemplo. El tercer argumento es un tipo *double* que indica el tiempo en segundos que tardará en saltar el evento y enviarse la señal de interrupción. Este tiempo empezará a contarse justo en el momento en el que la función *create_event* concluye con éxito. En el ejemplo, se crea un evento llamado *event_1* que envía la señal de interrupción *sigevent_1* tras 7 segundos y 250 milisegundos de ser inicializado. A continuación, se mostrará con qué función se asociarán las tareas con los eventos, así como funciones prácticas que nos permitirán leer, reiniciar o parar la cuenta del evento.

5.3.4 Sincronización entre tareas, temporizadores y eventos

Con las tareas, temporizadores y eventos definidos, queda solo un paso más para poder realizar un ejemplo sencillo de un proceso en tiempo real. Las diferentes tareas han de estar coordinadas y funcionando con el ritmo que se ha marcado con los temporizadores y eventos de tiempo real, así que solo queda presentar la función que se va a encargar de esta sincronización.

En general, queremos que una tarea esté "dormida" en el procesador hasta que una señal le dé permiso para continuar o realizar una iteración de control. Tras esto, la tarea volverá a su estado aletargado hasta nuevo aviso. Por ello, buscamos una línea de función en la que en cuanto se llegue, la tarea no continúe y entre a dormir, y una vez que llegue la señal, la tarea continúe con normalidad. Además, esta función no debe mandar sobre las reglas de prioridad ya establecidas, y aunque llegue una señal para continuar, no se continuará hasta que no haya más tareas prioritarias en el procesador.

La función se define de la siguiente forma:

```
int wait_signal ( signal * SIGNAL_NAME);
```

Cuando se escribe esa función en una tarea, la tarea se queda dormida hasta que llega la señal que se le indica, y después continúa. Poner esa función al principio de un bucle *while* infinito es lo mismo que imponer un tiempo de muestreo determinado por la configuración del temporizador al que va asociado la señal.

```
void *control (void*) {
     // Algoritmo de la tarea
     ...

     // Bucle de control
     while(1) {
          wait_signal(&sigtimer_1);
          ...
     }
}
```

Un detalle a tener en cuenta del funcionamiento de esta función es el caso de que se mande la señal en cuestión antes de que el proceso llegue a la línea donde se encuentra esta función de espera. Este caso generalmente estará debido a retrasos en tareas de menor prioridad cuando hay muchas tareas de mayor prioridad consumiendo todos los recursos del procesador. El comportamiento ante esta situación es que la función *wait_signal* se salta el paso de dormir la tarea y continúa directamente con la ejecución del código, si las reglas de prioridad lo permiten. Además, esta situación no es acumulativa: sucesivas activaciones de una señal sin que la tarea llegue a completar un solo ciclo y vuelva a la función de espera no implica que las señales se encolen, por lo que solo se recuerda el retraso una vez.

5.3.5 Programación de eventos

Los eventos son realmente útiles para la sincronización de tareas y notificaciones de sucesos internos. Se podrían usar, por ejemplo, para avisar del retraso de una tarea o notificar la activación de una variable externa. Por ello, se han añadido varias funciones adicionales para facilitar la programación.

La primera de ellas es *reset_event*, que como su nombre indica, se usará para reiniciar la cuenta interna hasta la activación del evento. La función recibe los siguientes argumentos:

```
reset_event ( event * EVENT_NAME , double TIME );
```

El primer argumento es el evento, como siempre pasado como dirección. En el segundo argumento se introduce de nuevo una variable de tiempo que reiniciará la cuenta interna del evento a ese valor indicado. Un evento que todavía no ha mandado una interrupción puede reconfigurarse para que se active siguiendo la nueva cuenta y olvidar la que ya lleva. Un evento que ya ha mandado una interrupción se quedará en espera hasta que se vuelva a configurar con una nueva cuenta.

En un ejemplo se mostraría de la siguiente manera:

```
void *control (void*) {
     // Algoritmo de la tarea
     ...

     // Bucle de control
     while(1) {
          wait_signal(&sigtimer_1);
          reset_event(&event_1, 0.65);
          ...
     }
}
```

El comportamiento sería el siguiente: cada vez que se reciba la señal que indica el tiempo de muestreo, se reinicia el evento *event_1*. Si el tiempo de muestreo es menor que 650 ms y no se sufre ningún retraso, el evento siempre se reiniciará antes de que se envíe la señal. Puede ocurrir que en el momento de sufrir un retraso (por ejemplo, que la señal del tiempo de muestreo llegue, pero la tarea no tenga prioridad suficiente para continuar con la siguiente línea), el evento no se reinicie a tiempo, y la señal asociada al evento se envíe, notificando a otra tarea de mayor prioridad este retraso.

Otra funcionalidad interesante es la de forzar que un evento envíe la señal sin importar la cuenta de espera. La función encargada es *force_event* y solo recibe como argumento el evento en cuestión.

```
force_event ( event * EVENT_NAME );
```

Esta función envía la señal y elimina cualquier cuenta que haya en curso. Si se quiere volver a iniciar la cuenta tras esta función, habrá que volver a utilizar *reset_event*.

```
void *control (void*) {
     // Algoritmo de la tarea
     ...

     // Bucle de control
     while(1) {
          wait_signal(&sigtimer_1);
          force_event(&event_1);
          ...
     }
}
```

Por último, una función que pare la cuenta pero que, al contrario que la anterior, no envíe la señal. Esta función se llama stop_event y recibe los mismos argumentos que la anterior.

```
stop_event ( event * EVENT_NAME );
```

Su sintaxis es análoga a *force_event*.

5.3.6 Pausas desvinculadas de señales

Hemos visto que *wait_signal* se comporta como una pausa: la tarea se pone a dormir y libera recursos del procesador hasta que llegue la señal indicada. A veces se presenta la necesidad de poner una tarea a dormir un tiempo concreto sin que sea necesario utilizar una señal de sincronización. Además, configurar e iniciar temporizadores o eventos puede requerir más líneas y más trabajo de lo necesario. Para este caso, presentamos la función *sleep_task*.

```
sleep_task ( double TIME );
```

La función solo recibe un argumento tipo *double* que indica el tiempo en segundos que va a dormir la tarea en la que se utilice. Es muy simple y prescinde de la inicialización y configuración que necesitan los temporizadores y los eventos. Tiene, sin embargo, una desventaja muy grande respecto a la sincronización por temporizadores y eventos, y es que dado que la pausa se realiza en cuanto se llega a la función, una tarea programada únicamente con *sleep_task* no tiene en cuenta los efectos de un retraso en ningún punto de su ejecución. Es por tanto una función útil para tareas de alta prioridad, pero para el resto de tareas no asegura un buen comportamiento de tiempo real y se recomienda seguir usando temporizadores o eventos.

```
void *control (void*) {
     // Algoritmo de la tarea
     sleep_task(3.5);
     ...

     // Bucle de control
     while(1) {
          ...
     }
}
```

En el ejemplo, esta tarea esperará 3.5 segundos desde que se inicializa antes de entrar en el bucle. Hay que tener en cuenta que si la tarea es de baja prioridad y se inicializan muchas a la vez, habrá un pequeño retraso junto con los 3.5 segundos.

5.3.7 Lectura de tiempos

Ahora que se han presentado funciones precisas de tiempo real, sería de gran ayuda una función más que pudiera medir tiempos y referencias con respecto a los temporizadores y los eventos. Con estas medidas no solo se podría estudiar el tiempo de ejecución de determinados cálculos o trozos de código, sino también utilizarlo para programar nuevas decisiones en tiempo real. Se define para ello la función get_time, que recibe los siguientes argumentos:

```
double get_time ( timer * TIMER_NAME);
double get_time ( event * EVENT_NAME);
```

Esta función, a diferencia de las demás, no distingue entre eventos y temporizadores: su primer y único argumento puede recibir cualquiera de los dos tipos.

La función devuelve un variable de tipo *double*, el mismo tipo que la variable de tiempo que se utiliza para crear los temporizadores y eventos. En el caso de introducir un temporizador, la variable de tiempo que devuelve la función corresponde con el tiempo restante hasta la activación del siguiente tiempo de muestreo. En el caso de

introducir un evento, la variable que devuelve es el tiempo restante hasta la activación del evento.

```
void *control (void*) {
      // Algoritmo de la tarea
      ...

      // Bucle de control
      while(1) {
            wait_signal(&sigtimer_1);
            /* Algoritmo recursivo de control */

            double lectura_timer = get_time(&TIMER_1);
            printf("tiempo de calculo = %f \n",Tm-lectura_timer);
      }
}
```

En el ejemplo anterior, se ha colocado la función *get_time* al final del bucle de control. Cuando se realicen los cálculos necesarios, comprobaremos el tiempo restante hasta la siguiente señal de muestreo. El tiempo de muestreo es el que se introdujo en la función *create_timer* al principio, por lo que es un valor exacto conocido. Si se hace la resta entre el tiempo de muestreo *Tm* y el valor obtenido por la función *get_time*, se obtiene el tiempo de cálculo.

Como última anotación de esta, el tiempo de cálculo obtenido puede ser engañoso en el caso de que la tarea se haya retrasado por ser de baja prioridad, más aún si la tarea ha tardado más tiempo que el propio periodo de muestreo, puesto que *get_time* nunca devuelve un valor negativo, sino el tiempo hasta la siguiente señal de muestreo. Si se combina esta función con un evento que avise cuando ocurra el caso particular, se obtendrá una información completa del suceso.

5.3.8 Cancelación de tareas

Una funcionalidad importante que debe poseer la programación en tiempo real es la posibilidad de cancelar tareas en un momento dado. Una tarea que realice un cálculo pesado puede, en ocasiones, pasarse del tiempo crítico establecido, y es necesario cancelarla y obtener cualquier resultado que haya podido conseguir en ese tiempo. Por ello se ha implementado en la librería la función *cancel_task*.

```
cancel_task ( task * TASK_NAME);
```

La función recibe como único parámetro el nombre de la tarea, pasado como dirección. La función cancela la tarea que recibe por parámetro y puede escribirse, en principio, desde cualquier otra tarea.

```
void *tarea (void*) {
      // Algoritmo de la tarea
      ...

      while(1) {
            // si llega la señal de emergencia
            wait_signal(&sigevent_1);
            // cancelamos la tarea task_1
            cancel_task(&task_1);
      }
}
```

Para no llegar a un comportamiento anómalo, hay que tener en cuenta dos cosas:

- Se debe cancelar una tarea desde otra tarea de mayor prioridad. Si se busca una cancelación instantánea de una tarea, se debe mandar la orden desde otra tarea de mayor prioridad, de lo contrario la cancelación puede verse retrasada debido a que tiene menos prioridad que la tarea en cuestión.

- La función main no es una tarea de tiempo real, por lo que enviar una cancelación desde ella no asegura que sea inmediata.

Se recomienda, por tanto, que el tratamiento de las cancelaciones se realice desde tareas "vigilantes" de mucha prioridad que se encarguen de vigilar y regular el comportamiento de otras tareas.

5.3.9 Borrado de temporizadores y eventos

Se han incluido dos funciones para eliminar completamente de la memoria los temporizadores o eventos. Se pueden usar opcionalmente para cerrar el programa o bien para paradas de emergencia.

```
delete_timer ( timer * TIMER_NAME );
delete_event ( event * EVENT_NAME );
```

Estas funciones están al mismo nivel y son las contrapuestas a *create_timer* y *create_event*, por lo que tras utilizarlas, habría que volver a escribir las funciones de inicialización o creación si se quiere seguir usando el objeto en cuestión.

```
void main (void) {
     ...

     delete_timer(&timer_1);
     delete_event(&event_1);
}
```

Cuando main acaba, todas las tareas también lo hacen, por lo que no es estrictamente necesario escribir estas líneas al final de main para cerrar las tareas. Sin embargo, son útiles cuando se presenta la necesidad de eliminar temporizadores u eventos durante la ejecución de un programa, ya que hasta que no se ejecuten estas líneas, estos objetos ocuparían un espacio residual en memoria.

5.4 Código fuente de la librería

Por último en este apartado, se va a exponer el código fuente y cómo se han realizado todas las funciones anteriores de acuerdo con las cabeceras de tiempo real de las normas POSIX. Todo el código que se expone a continuación pertenece a la librería *libRTC.hpp*.

```
#define _POSIX_C_SOURCE 200809L

#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <time.h>
```

Estas líneas son obligatorias para tener todas las funcionalidades de la norma en un programa de tiempo real. No solo se incluyen las cabeceras de tiempo real, sino otras básicas necesarias para su utilización. Si se necesitan más cabeceras se han añadir con el comando *include* al programa que se vaya a realizar, o bien modificando el código fuente de "*libRTC.hpp"*.

```
#define timer timer_t
#define event timer_t
```

```
#define task pthread_t
#define signal sigset_t
```

Se definen cuatro macros para los objetos que se utilizan en la librería: tareas, temporizadores, eventos y señales. Consiste en una simple traducción a los tipos que se definen a la norma, y el objetivo únicamente consiste en facilitar la comprensión y el aprendizaje con nombres más fáciles de recordar y asociar.

```
void init_task(pthread_t *TASK, void* (*function) (void*), int
        priority);
void create_timer(timer_t *TIMER, sigset_t *SET, double time);
void create_event(timer_t *EVENT, sigset_t *SET, double time);
void reset_event(timer_t *EVENT, double time);
void stop_event(timer_t *EVENT);
void force_event(timer_t *EVENT);
int wait_signal(sigset_t *SET);
void sleep_task(double time);
double get_time(timer_t *TIMEREVENT);
void cancel_task(pthread_t *TASK);
void delete_timer(timer_t *TIMER);
void delete_event(timer_t *EVENT);
```

Estos son los prototipos de todas las funciones que se han explicado en este capítulo. A continuación, se expondrá el código de cada una de ellas siguiendo el orden.

5.4.1 Función init_task

```
void init_task(pthread_t *TASK, void* (*function) (void*), int
        priority);

   pthread_attr_t attr;
   pthread_attr_init(&attr);
   pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED);
   int error1 = pthread_attr_setschedpolicy(&attr, SCHED_RR);
   struct sched_param setp;
   setp.sched_priority = priority;
   int error2 = pthread_attr_setschedparam(&attr, &setp);
   if (error1 || error2) printf("ERROR ASSIGNING PRIORITY: %d,
        %d\n",error1,error2);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

   int error3 = pthread_create(TASK,&attr,function,NULL);
   if (error3) printf("ERROR CREATING TASK: %d\n",error3);

   pthread_attr_destroy(&attr);

}
```

Para programar la creación de tareas se han utilizado los siguientes recursos:

- pthread_create[17]: Crea una tarea de identificación *pthread_t* (*task*), con atributos *pthread_attr_t* (donde configuraremos la política y la prioridad) y apuntando a la función que hará de tarea. La identificación y la función a la que apunta vienen dadas por el primer y segundo argumento de la función.

[17] http://man7.org/linux/man-pages/man3/pthread_create.3.html [Último acceso: julio 2016]

- pthread_attr_init[18]: Inicializa los atributos para poder modificarlos.
- pthread_attr_setinheritsched[19]: Cambia la forma en la que se heredan los atributos desde la tarea que está creando a otra. En este caso, se quiere que los atributos no se hereden, sino que sean configurados manualmente. Se utiliza para ello el flag *PTHREAD_EXPLICIT_SCHED*.
- pthread_attr_setschedpolicy[20]: Cambia la política de planificación de la tarea. Se utiliza el flag *SCHED_RR* para establecer la política *Round-robin*.
- pthread_attr_setschedparam[21]: Cambia el número de prioridad con una estructura *struct sched_param*. Se asignará el número de prioridad que viene como tercer argumento de la función.
- pthread_attr_setdetachstate[22]: Configura el estado de unión del hilo. Se utiliza el flag *PTHREAD_CREATE_DETACHED*, que indica que el hilo tiene una terminación desvinculada, y por tanto puede ser cancelado externamente desde cualquier otro hilo y lo hará de manera inmediata devolviendo sus recursos al sistema.
- pthread_attr_destroy[23]: Destruye el atributo *pthread_attr_t*, lo cual no es obligatorio ni tiene ningún efecto en la tarea, pero es recomendado por la propia norma.

Esta función imprimirá dos tipos de errores por pantalla, si se dan. Si hay un error con la creación de tareas y la sintaxis es correcta, probablemente el problema se encuentre en que no hay soporte de tiempo real. Si el error solo salta con la asignación de prioridades, lo más probable es que no se ha ejecutado el programa con permisos de administrador (*root*), ya que esta condición es necesaria, al menos en Linux, para que funcione la configuración de prioridades.

5.4.2 Función create_timer

```
int contsignal = 0;
struct itimerspec on;
struct sigevent ev;

void m(int sig) {}

void create_timer(timer_t *TIMER, sigset_t *SET, double time){

   struct sigaction a;
   sigemptyset(SET);
   sigaddset(SET,SIGRTMIN+contsignal);
   a.sa_flags = SA_SIGINFO;
   sigemptyset(&a.sa_mask);
   pthread_sigmask(SIG_BLOCK,SET,NULL);
   a.sa_handler = m;
   sigaction(SIGRTMIN+contsignal,&a,NULL);

   int sec = (int)time;
   double dnanos = time - sec;
   long nanosec = (long)(dnanos*1000000000);

   struct timespec cero = {0, 0};
   struct timespec stime = {sec, nanosec};
   ev.sigev_notify = SIGEV_SIGNAL;
   ev.sigev_signo = SIGRTMIN+contsignal;
```

[18] http://man7.org/linux/man-pages/man3/pthread_attr_init.3.html [Último acceso: julio 2016]
[19] http://man7.org/linux/man-pages/man3/pthread_attr_setinheritsched.3.html [Último acceso: julio 2016]
[20] http://man7.org/linux/man-pages/man3/pthread_attr_setschedpolicy.3.html [Último acceso: julio 2016]
[21] http://man7.org/linux/man-pages/man3/pthread_attr_setschedparam.3.html [Último acceso: julio 2016]
[22] http://man7.org/linux/man-pages/man3/pthread_attr_setdetachstate.3.html [Último acceso: julio 2016]
[23] http://man7.org/linux/man-pages/man3/pthread_attr_init.3.html [Último acceso: julio 2016]

```
        ev.sigev_value.sival_int = contsignal;
        contsignal = contsignal + 1;

        timer_create(CLOCK_REALTIME, &ev, TIMER);
        on.it_value = stime;
        on.it_interval = stime;
        timer_settime(*TIMER, 0, &on, NULL);

        return;
    }
```

Para explicar la programación de los temporizadores y las señales vamos a introducir un concepto que no hemos utilizado anteriormente por el simple motivo de hacer la librería más sencilla y accesible. Además de señales, existe el concepto de conjuntos de señales. Un conjunto de señales es un grupo en el que se puede ir añadiendo y quitando señales, y es con este grupo con lo que se pueden realizar las tareas de sincronización y el tratado de señales, no con las señales en sí. Dicho de otra manera: los temporizadores, eventos, y funciones de espera y tratado de señales no trabajan directamente con las señales, sino con los conjuntos.

Si trabajáramos con los conjuntos y las señales en la librería desarrollada en este proyecto, su comprensión empezaría resultar engorrosa y hemos decidido ocultar el concepto de conjuntos al usuario que la utilice. La forma es que vamos a obviar estos conjuntos es la siguiente: Cuando el usuario define una señal con "*signal sig1*", *sig1* no es en realidad una señal, sino un conjunto que contiene únicamente una señal. El conjunto se define en POSIX como un tipo *sigset_t*, que es con lo que se puede trabajar realmente, mientras que la señal es un tipo *int* con un número que comprende entre los macros SIGRTMIN y SIGRTMAX. Por ello hemos definido un contador global con la variable *contsignal*, que recorrerá todos los números entre las dos macros anteriores para asignar por orden una señal a cada conjunto. El usuario, que en principio no tiene por qué saber la norma ni cómo se realiza la programación de esta, solo tiene que crear un tipo *signal* definido por la librería del proyecto. Aunque, si lo desea, ya que *contsignal* es una variable global, puede utilizarla y modificarla para manejar conjuntos y señales por separado, aunque en general no será necesario.

Para programar la función de creación de temporizadores se han utilizado los siguientes recursos:

- timer_create[24]: Crea un temporizador o *timer*, de tipo *timer_t*, apuntando a un reloj del sistema (en este caso, siempre queremos el reloj de tiempo real, por lo que usamos el flag *CLOCK_REALTIME*) que realiza la acción dada por la estructura *struct sigevent*.
- sigemptyset, sigaddset[25]: Para trabajar con los conjuntos de señales y las señales que contienen.
- pthread_sigmask[26]: Configura la máscara de señales para procesos multihilo. Se utiliza el flag *SIG_BLOCK* para bloquear el manejador de señales y obtener un comportamiento síncrono con las señales.
- sigaction[27]: Cambia la acción que se realiza cuando se recibe una señal. Se utiliza el flag *SA_SIGINFO* que es el que permite la sincronización en tiempo real.
- struct sigevent[28]: Estructura que configura la acción que realiza el temporizador cada vez que salta. El flag *SIGEV_SIGNAL* indica que la notificación se realiza por medio de una señal.
- timer_settime[29]: Arma o desarma el temporizador y configura el tiempo según la estructura *struct itimerspec*, la cual contiene dos campos, un *int* con los segundos y un *long* para los nanosegundos.
- struct itimerspec: Estructura que contiene dos campos, uno para configurar el tiempo hasta el primer salto del temporizador (*it_value*), y otro para configurar el periodo entre saltos (*it_interval*). Un *timer* que tenga un tiempo hasta el primer salto y un 0 en el periodo, se comportará como lo que hemos

[24] http://man7.org/linux/man-pages/man2/timer_create.2.html [Último acceso: julio 2016]
[25] http://man7.org/linux/man-pages/man3/sigfillset.3.html [Último acceso: julio 2016]
[26] http://man7.org/linux/man-pages/man3/pthread_sigmask.3.html [Último acceso: julio 2016]
[27] http://man7.org/linux/man-pages/man2/sigaction.2.html [Último acceso: julio 2016]
[28] http://man7.org/linux/man-pages/man7/sigevent.7.html [Último acceso: julio 2016]
[29] http://man7.org/linux/man-pages/man2/timer_settime.2.html [Último acceso: julio 2016]

definido en este proyecto como un evento. Un *timer* que tenga el mismo tiempo en el primer salto y en el periodo, se comportará como lo que hemos definido en este proyecto como un temporizador. Por esto, el código fuente de creación de un evento, que veremos a continuación, es prácticamente análogo al del temporizador solo que cambiando este campo.

El código también contiene una conversión del tiempo que se recibe como tipo *double* y se traduce a la estructura *struct itimerspec* de la norma. El único propósito es que la programación sea sencilla externamente.

5.4.3 Función create_event

```
void create_event(timer_t *EVENT, sigset_t *SET, double time){

   struct sigaction a;
   sigemptyset(SET);
   sigaddset(SET,SIGRTMIN+contsignal);
   a.sa_flags = SA_SIGINFO;
   sigemptyset(&a.sa_mask);
   pthread_sigmask(SIG_BLOCK,SET,NULL);
   a.sa_handler = m;
   sigaction(SIGRTMIN+contsignal,&a,NULL);

   int sec = (int)time;
   double dnanos = time - sec;
   long nanosec = (long)(dnanos*1000000000);

   struct timespec cero = {0, 0};
   struct timespec stime = {sec, nanosec};
   ev.sigev_notify = SIGEV_SIGNAL;
   ev.sigev_signo = SIGRTMIN+contsignal;
   ev.sigev_value.sival_int = contsignal;
   contsignal = contsignal + 1;

   timer_create(CLOCK_REALTIME, &ev, EVENT);
   on.it_value = stime;
   on.it_interval = cero;
   timer_settime(*EVENT, 0, &on, NULL);

   return;
}
```

Su programación es análoga a la función *create_timer*, así como los recursos utilizados. La única diferencia se encuentra en la configuración de la estructura *struct itimerspect on*, en la que antes se programó como un salto entre intervalos, y ahora es un único salto.

5.4.4 Función reset_event

```
struct itimerspec reset;

void reset_event(timer_t *EVENT, double time){

   int sec = (int)time;
   double dnanos = time - sec;
   long nanosec = (long)(dnanos*1000000000);
   struct timespec cero = {0, 0};
   struct timespec stime = {sec, nanosec};
```

```
    reset.it_value = stime;
    reset.it_interval = cero;
    timer_settime(*EVENT, 0, &reset, NULL);

    return;
}
```

Para la reprogramación de un evento solo tenemos que utilizar parte de los recursos utilizados anteriormente. Se realiza una conversión del tiempo desde double hasta *struct timespec*, se reconfigura *struct itimerspect reset* y se utiliza *timer_settime* para rearmar el evento.

5.4.5 Funciones stop_event, force_event

```
struct itimerspec off;

void stop_event(timer_t *EVENT){

    struct timespec cero = {0, 0};
    off.it_value = cero;
    off.it_interval = cero;
    timer_settime(*EVENT, 0, &off, NULL);

    return;
}
```

Análogo al anterior, pero configurando *struct timespec* con un 0, lo cual desarma el evento.

```
void force_event(timer_t *EVENT){

    struct timespec force = {0, 1};
    struct timespec cero = {0, 0};
    reset.it_value = force;
    reset.it_interval = cero;
    timer_settime(*EVENT, 0, &reset, NULL);

    return;
}
```

Sigue siendo análogo a lo anterior, pero vamos a aprovechar para realizar una aclaración del comportamiento del reloj de tiempo real. Hemos visto que la estructura *struct timespec* tiene una resolución de 1 nanosegundo, sin embargo esto no quiere decir que la precisión del reloj del sistema lo sea. Un procesador común actual puede tener un "tick" de reloj del orden de 100 microsegundos, lo que asegura una precisión de hasta cuatro décimas de segundo. Todas las décimas extras que se utilicen, aunque sea posible utilizarlas, acaban siendo redondeadas internamente al tick del sistema. Pero se redondea a cualquier número menos al cero, por lo que al configurar force_event, véase el código, se indica un salto de 1 nanosegundo, pero verdaderamente el comportamiento que se consigue es un salto en el próximo tick de reloj, que es precisamente lo que se busca con esta función.

5.4.6 Función wait_signal

```
int wait_signal(sigset_t *SET){

        siginfo_t info;
        sigwaitinfo(SET,&info);
        int inf = info.si_value.sival_int;

        return inf;
```

```
}
```

Para esta función se ha utilizado el siguiente recurso:

- sigwaitinfo[30]: Suspende la ejecución del hilo que la llama hasta que llega una señal que esté contenida en el conjunto que se introduce por su primer argumento. El segundo argumento devuelve información que contenga la señal.

Aunque no se ha explicado anteriormente, debido a que se intenta que el usuario de la librería no deba conocer los conceptos de señales y conjuntos de señales, la función *sigwaitinfo* trabaja también con conjuntos en vez de con señales, solo que ha asignado una única señal al conjunto a través del contador global *contsignal*. Esta función wait_signal es de tipo *int* y devuelve (opcionalmente) precisamente este contador *contsignal*, que es realmente el número identificador de la señal, que se ha configurado para que se envíe junto con la señal y se reciba en *sigwaitinfo*. Este detalle no hace falta saberlo, y de hecho está oculto para el usuario de la librería, ya que en general no será necesario trabajar con señales y conjuntos, y esta solución adoptada simplifica el entendimiento y no limita las posibilidades.

5.4.7 Función sleep_task

```
void sleep_task(double time){

        int sec = (int)time;
        double dnanos = time - sec;
        long nanosec = (long)(dnanos*1000000000);
        struct timespec sleep = {sec, nanosec};
        nanosleep(&sleep,NULL);

        return;
}
```

Para esta función se ha utilizado el siguiente recurso:

- nanosleep[31]: Suspende la ejecución del hilo que la llama durante el tiempo dado por el primer argumento de tipo *struct timespec*.

También se ha realizado la misma conversión desde *double* para facilitar la sintaxis al usuario.

5.4.8 Función get_time

```
double get_time(timer_t *TIMEREVENT){

        struct itimerspec timerem;
        timer_gettime(*TIMEREVENT,&timerem);
        int sec = timerem.it_value.tv_sec;
        long nanosec = timerem.it_value.tv_nsec;
        double dnanosec = (double) nanosec;
        double rem = (double) (sec + dnanosec/1000000000);

        return rem;
}
```

Para esta función se ha utilizado el siguiente recurso:

[30] http://man7.org/linux/man-pages/man2/sigwaitinfo.2.html [Último acceso: julio 2016]
[31] http://man7.org/linux/man-pages/man2/nanosleep.2.html [Último acceso: julio 2016]

- timer_gettime[32]: Devuelve el tiempo que falta hasta el siguiente salto de un *timer_t*.

El tiempo lo devuelve en una estructura *struct itimerspect*, que se convierte de nuevo a *double* para facilitarle el trabajo al usuario.

5.4.9 Funciones cancel_task, delete_timer, delete_event

```
void cancel_task(pthread_t *TASK){

        pthread_cancel(*TASK);
}

void delete_timer(timer_t *TIMER){

        timer_delete(*TIMER);
}

void delete_event(timer_t *EVENT){

        timer_delete(*EVENT);
}
```

Se han utilizado los siguientes recursos:

- pthread_cancel[33]: Envía una petición de cancelación al hilo que se envía por argumento.
- timer_delete[34]: Desarma y elimina el objeto *timer_t* que recibe como argumento.

La función *pthread_cancel*, a pesar de parecer muy sencilla en cuanto a programación, requiere una lectura completa del comportamiento de las cancelaciones, puesto que hay varios tipos y se debe configurar cuando se crean los hilos o bien dentro de ellos. Para esta librería los flags que están configurados son: *PTHREAD_CANCEL_ENABLE*, el cual se encuentra activado por defecto y activa la cancelación del hilo, y *PTHREAD_CANCEL_ASYNCHRONOUS*, que permite la cancelación inmediata desde cualquier hilo con mayor prioridad que el que se va a cancelar. Si la cancelación no llegase a funcionar tal y como se ha explicado en este capítulo, se recomienda estudiar las funciones *pthread_setcancelstate* y *pthread_setcanceltype*[35], las cuales permiten reconfigurar estos flags relacionados con la cancelación.

[32] http://man7.org/linux/man-pages/man2/timer_settime.2.html [Último acceso: julio 2016]
[33] http://man7.org/linux/man-pages/man3/pthread_cancel.3.html [Último acceso: julio 2016]
[34] http://man7.org/linux/man-pages/man2/timer_delete.2.html [Último acceso: julio 2016]
[35] http://man7.org/linux/man-pages/man3/pthread_setcancelstate.3.html [Último acceso: julio 2016]

6 EJEMPLOS DESARROLLADOS

En este capítulo final se expondrán ejemplos de programación con la librería de tiempo real. El objetivo es doble: demostrar que las características descritas en el capítulo anterior funcionan tal y como se han detallado, y servir como base o referencia para implementar nuevos programas sobre la librería, ya que se puede apreciar la sintaxis completa y una idea general de la estructura de programación.

6.1 Ejemplo básico de tiempo real

Se va a presentar un ejemplo utilizando las funciones explicadas en el anterior capítulo para afianzar su empleo y demostrar sus características de tiempo real. Además, se pretende que la forma y estructura del código que se va a presentar sirva como referencia a la hora de realizar nuevos programas con la librería. Se van a crear diferentes tareas, así como temporizadores y eventos que se encargarán de la sincronización entre tareas. Finalmente se mostrará el resultado que imprime por pantalla, con el cual se puede comprobar que las características de tiempo real funcionan y la instalación del software fue correcta.

Empezamos incluyendo la librería y declarando los objetos de tiempo real a utilizar.

```
#include "libRTC.hpp"

task task_1; // Tarea 1 principal de cálculo
void *task1(void*);

task task_2; // Tarea 2 para notificar retrasos
void *task2(void*);

task task_3; // Tarea 3 para imprimir resultados por pantalla
void *task3(void*);

// 1 temporizador y 3 eventos
timer TIMER_1;
event EVENT_1;
event EVENT_2;
event EVENT_3;

signal SIGTIMER_1;
signal SIGEVENT_1;
signal SIGEVENT_2;
signal SIGEVENT_3;
```

Se van a crear 3 tareas: una principal que contiene un cálculo, una segunda que avisa si la tarea principal sufre un retraso y una última que imprime por pantalla los resultados del cálculo de la primera. También se utilizará un temporizador para la tarea de cálculo, y otros tres eventos para la sincronización entre tareas. Los objetos que

se han declarado pueden verse en el siguiente diagrama de bloques. Encontramos las tres tareas con su orden de prioridad, y los cuatro objetos de tiempo real que actúan en el funcionamiento y la sincronización de estas.

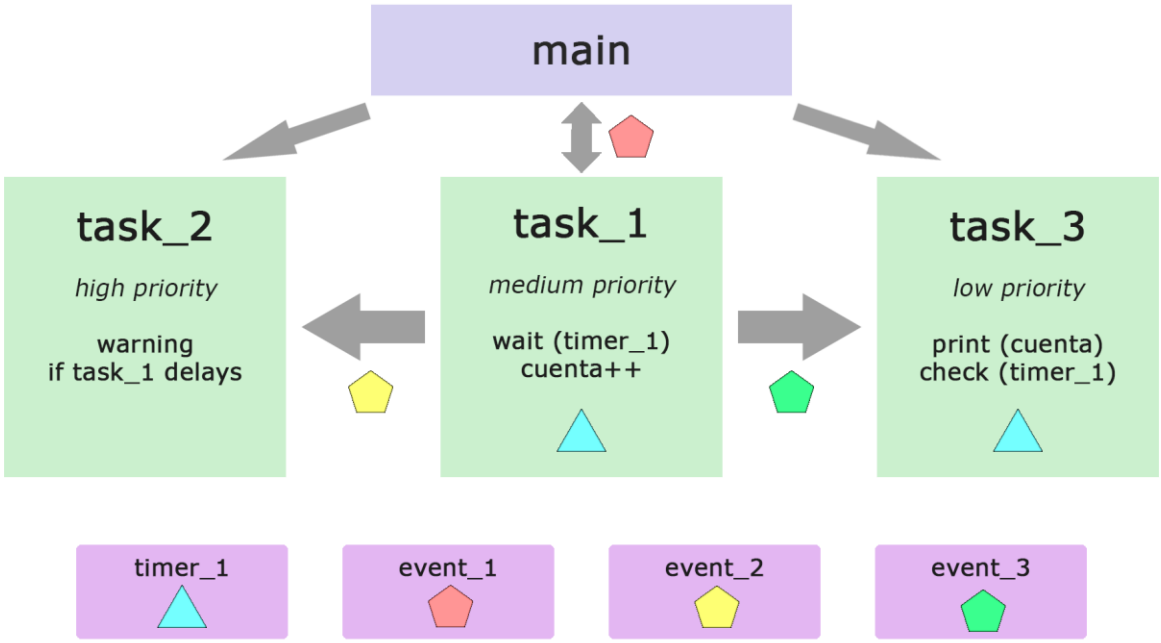

Ilustración 6-1. Esquema de funcionamiento del programa de ejemplo

El código del programa se expone a continuación, empezando por main.

```
int main () {

        /* Inicializacion */

        create_timer(&TIMER_1, &SIGTIMER_1, 0.8);
        create_event(&EVENT_1, &SIGEVENT_1, 0);
        create_event(&EVENT_2, &SIGEVENT_2, 0);
        create_event(&EVENT_3, &SIGEVENT_3, 0);

        init_task(&task_1,task1,60); // Prioridad intermedia
        init_task(&task_2,task2,80); // Prioridad alta
        init_task(&task_3,task3,40); // Prioridad baja

        printf("main: tareas iniciadas\n");


        /* Condicion de espera */

        wait_signal(&SIGEVENT_1);
        printf("main: finalizando...\n");
        sleep_task(1);


        /* Terminacion */

        delete_timer(&TIMER_1);
        delete_event(&EVENT_1);
        delete_event(&EVENT_2);
```

```
        delete_event(&EVENT_3);
        printf("main: fin del programa\n");

        return 0;
}
```

Hemos dividido main en tres bloques principales: Uno de inicialización, otro de espera y otro de finalización. En la inicialización hemos creado primero un temporizador de 800 ms que será el tiempo de muestreo de la tarea número 1, y 3 eventos inicializados a 0 para después reprogramarlos a nuestro gusto. Después, hemos creado las tres tareas con la función *init_task*, hemos enlazado las variables globales con la función que hará de tarea y hemos dado un número de prioridad. Recordando de lo explicado anteriormente, el número de prioridad es un entero del 1 al 99, y cuanto mayor sea el número mayor será la prioridad. En este programa, hemos dado mayor prioridad a la tarea que avisa de retrasos, prioridad intermedia a la tarea principal, y prioridad mínima a la tarea que imprime resultados por pantalla.

Un aspecto clave del bloque de inicialización es que los temporizadores y eventos se crean antes que las tareas. Esto es porque las tareas se compilan en cuanto aparece su función de creación (*init_task*), y en el código de las tareas aparecen múltiples referencias a los temporizadores y eventos. Si estos no se han compilado antes de que se compilen las tareas, las tareas estarán asociadas a objetos no existentes y el programa no funcionará o no hará nada: las tareas se quedarían bloqueadas en cualquier línea de espera porque estarían esperando a señales que aún no se han inicialiado.

En el bloque de espera se ha impuesto como condición que main duerma hasta que llegue una señal del evento número 1. Cuando llegue, se pone a dormir dos segundos más con la función *sleep_task*, que como ya hemos explicado, recibe un tipo *double* con el tiempo en segundos y pone a dormir ese tiempo a la tarea en la que se encuentre.

En el bloque de finalización hemos escrito las funciones que eliminan los objetos de tiempo real de la memoria, aunque como ya se dijo en el capítulo anterior, son opcionales puesto que al acabar main se debería realizar esa acción por defecto.

El siguiente paso es la programación de las tareas.

```
int cuenta;
int fin;

void *task1(void*){

        printf("task_1: inicio\n");
        int delay = 2;
        fin = 0;
        cuenta = 0;

        while(!fin){

           wait_signal(&SIGTIMER_1);
           reset_event(&EVENT_2,0.801); //llega a tarea 2

           if (delay == 0) sleep_task(1.2);
           delay--;
           cuenta++;

           force_event(&EVENT_3); //llega a tarea 3

           if (cuenta == 6) {
              force_event(&EVENT_1); //llega a main
              cancel_task(&task_3); //cancela tarea 3
              fin=1;
```

```
                printf("task_1: cuenta = %d, fin = %d\n",cuenta,fin);
            }
        }

}
```

Primero, hemos declarado dos variables globales: una llamada *cuenta* que será modificada en cada iteración, y otra llamada *fin* que servirá para salir del bucle cuando se dé una condición. Dentro de la primera tarea, se imprime por pantalla un aviso de inicio que solo saltará una vez y luego se entra en el bucle principal. En el bucle, se espera al principio la señal del temporizador cada 800 ms. Cuando llega la señal, se reinicia el evento número 2 con una cuenta de 801 ms. Si este evento se reinicia cada 800 ms dado por el temporizador, nunca saltará, pero sí lo hará si ocurre un pequeño retraso de 1 ms. Después se sumará una unidad al valor de *cuenta*, y se restará una unidad al valor de *delay*, que es otra variable que se inició a 2. Como esta acción se realiza en cada iteración del bucle, estas dos variables actuarán como contadores, uno ascendente y otro descendente. Hay también una condición antes de la actualización de las variables, en la que si *delay* vale 0 (ocurrirá tras dos iteraciones), la tarea se pondrá a dormir 1.2 segundos, lo cual provocará un retraso mayor que el tiempo de muestreo.

Al final de cada iteración, se envía una señal por el evento número 3, y si la cuenta llega a 6, se enviará también una señal por el evento número 1 y se activará la condición para salir del bucle y cancelar la tarea 3. Este evento número 1, como ya hemos visto en main, desembocará también en la terminación del programa.

A continuación, la segunda tarea.

```
void *task2(void*){

        printf("task_2: inicio\n");

        while(1){

            wait_signal(&SIGEVENT_2);
            printf("task_2: task_1 se ha retrasado\n");
        }

}
```

Al igual que la primera, esta tarea va a imprimir un aviso de inicio solo al principio, y luego entra en un bucle infinito. El bucle es simple: cada vez que llegue una señal por el evento número 2, prueba de que la tarea 1 se ha retrasado, imprimirá un aviso. Esta tarea es de máxima prioridad.

Por último, la tercera tarea.

```
void *task3(void*){

        printf("task_3: inicio\n");

        while(!fin){

            wait_signal(&SIGEVENT_3);
            printf("task_3: cuenta = %d, fin = %d\n",cuenta,fin);
            double queda = get_time(&TIMER_1);
            printf("task_3: tiempo hasta TIMER_1: %f\n",queda);
        }

}
```

Imprime un aviso de inicio solo al principio y entra en un bucle que se ejecuta hasta que *fin* vale 1, por lo que

esta variable va a terminar las tareas 1 y 3. En el bucle se espera una señal del evento número 3, la cual se envía al final del bucle de la tarea 1. Este bucle imprimirá las variables *cuenta* y *fin*, y después imprimirá cuánto tiempo queda hasta la próxima activación del temporizador.

En general, el código funciona así: La tarea 1 actualiza el contador de variable *cuenta* cada 800 ms, y después la tarea 3 imprime la variable por pantalla. La tarea 2 notifica si ocurre un retraso, y se ha programado uno manualmente para cuando el programa lleve dos iteraciones.

Vamos a ver el resultado final que se imprime por el terminal y vamos a discutir punto por punto qué ha ocurrido y por qué.

```
joaquin@joaquin-desktop:~$ sudo -s
[sudo] password for joaquin:

root@joaquin-desktop:~# cd Desktop/prt
root@joaquin-desktop:~/Desktop/prt#  g++  -o  ejemplo  ejemplo.cpp  -I
/home/joaquin/Desktop/prt -lrt -lpthread
root@joaquin-desktop:~/Desktop/prt# ./ejemplo

main: tareas iniciadas
task_2: inicio
task_1: inicio
task_3: inicio
task_3: cuenta = 1, fin = 0
task_3: tiempo hasta TIMER_1: 0.799752
task_3: cuenta = 2, fin = 0
task_3: tiempo hasta TIMER_1: 0.799855
task_2: task_1 se ha retrasado
task_3: cuenta = 4, fin = 0
task_3: tiempo hasta TIMER_1: 0.399762
task_3: cuenta = 5, fin = 0
task_3: tiempo hasta TIMER_1: 0.799855
task_1: cuenta = 6, fin = 1
main: finalizando...
task_2: task_1 se ha retrasado
main: fin del programa
```

Compilación del programa: Como vemos en el ejemplo, empezamos por la línea *sudo -s* que nos pedirá la contraseña de la cuenta de administrador y nos otorgará permisos de administrador para el resto de cosas que hagamos hasta cerrar el terminal. Sin permisos de administrador no funcionará la planificación de prioridades, por lo que se recomienda empezar por esta línea cada vez que vayamos a abrir el terminal. Después compilamos el programa con la línea *g++ -o ejemplo ejemplo.cpp -I /home/joaquin/Desktop/prt -lrt -lpthread*, donde *gcc* hace referencia al compilador, "ejemplo" al nombre del ejecutable, "ejemplo1.cpp" al código del programa, el comando *-I* añade la dirección donde se encuentra la librería "libRTC.hpp" y *-lrt, -lpthread* añaden las librerías dinámicas de tiempo real de la norma. Por último, *./ejemplo* corre el ejecutable. Dependiendo de la versión de Ubuntu es posible que no permita correr el ejecutable por defecto, por lo que debe usarse la línea *chmod +x ejemplo1* para desbloquearlo.

Nos dirigimos ahora a la ejecución en sí del programa. En main, hemos visto que primero se inicializan todas las tareas y luego se imprime por pantalla el aviso. Luego, en cada una de las tareas, la primera línea que se ejecutaba era un aviso por pantalla de que se habían iniciado. En un programa que no fuese de tiempo real, el orden de los avisos sería el siguiente: tarea 1, tarea 2, tarea 3 y main, por orden de ejecución de las líneas. Si volvemos al resultado que vemos en el terminal, el orden es diferente: main, tarea 2, tarea 1 y tarea 3, que coincide exactamente con el orden de prioridad establecido. Hasta que main no llega a la función *wait_signal* que lo pone a dormir, las otras tareas no pueden empezar a ejecutarse. Lo mismo ocurre con todas las tareas: la de mayor prioridad comienza, imprime por pantalla que se ha iniciado, y cuando llega a la línea de pausa deja entrar a la siguiente tarea.

El programa sigue con su transcurso normal: tarea 1 actualiza el valor de *cuenta*, y posteriormente tarea 3 lo imprime e indica el tiempo hasta la próxima señal del temporizador. Podemos observar que este tiempo es de

0.799 y otros decimales, por lo que las tareas 1 y 3 tardan menos de 1 ms en ejecutarse. Hay que tener en cuenta que *printf* no es una función de tiempo real del todo a pesar se que se pueda utilizar perfectamente con el código, lo que quiere decir que el instante en el que se imprime la línea por pantalla no tiene porqué ser exacto, aunque la ejecución de la línea sí se haga de acuerdo con las normas establecidas. Siempre, para tomar referencias reales de tiempo se ha de usar la función *get_time*.

Cuando vamos por la segunda iteración, ocurre el retraso que hemos programado manualmente. El retraso ocurre al principio de la tarea 1, y dura 1.2 segundos. El tiempo de repetición es de 0.8 segundos. Detallamos lo que va a ocurrir paso a paso. Vamos a establecer como referencia que este instante es t = 0.

- (t = 0) Salta el temporizador para el bucle de tarea 1. La variable *cuenta* vale 2.
- (t = 0+) Se reinicia el evento 2, que saltará a los 0.801 segundos.
- (t = 0+) Tarea 1 entra en pausa durante los próximos 1.2 segundos.
- (t = 0.8) Salta el temporizador para tarea 1, pero esta no recoge la señal puesto que duerme sin esperarla.
- (t = 0.801) Salta el evento 2, que lo recoge tarea 2 de máxima prioridad e imprime un aviso por pantalla.
- (t = 1.2) Tarea 1 deja de dormir, y continúa el bucle. Se actualiza *cuenta* a 3. Se envía una señal por evento 3 a tarea 3 para imprimir por pantalla y termina el bucle.
- (t = 1.2+) Cuando tarea 1 termina el bucle y vuelve a la línea de espera, no entra a dormir porque debido al retraso ya se había activado el temporizador anteriormente, por lo que continúa sin dormir. Tarea 3, que tiene baja prioridad, a pesar de haber recibido la señal para imprimir *cuenta* por pantalla, tiene que esperar a que tarea 1 se duerma.
- (t = 1.2+) Tarea 1 actualiza *cuenta* a 4 y vuelve a enviar una señal a tarea 3. Termina el bucle y se va a dormir.
- (t = 1.2+) Como tarea 3 ya tiene una señal pendiente que no ha podido tratar por no tener prioridad, el aviso se sobreescribe y no se acumula. Ahora que tarea 1 está dormida, ya puede ejecutarse. Imprime *cuenta* a 4 sin pasar por el valor 3. El temporizador se activará ahora en t = 1.6 (0.8 x 2), por lo que tiempo que queda es 1.6 – 1.2, menos algún decimal por el tiempo de ejecución. La cuenta debería salir 0.399 más otros decimales.

Estos pasos que se han descrito se han basado en el comportamiento que se ha explicado de la librería en el anterior capítulo. Si volvemos al resultado que ha imprimido el terminal, vemos que coincide pefectamente, señal de que el tiempo real está funcionando.

El programa continúa su curso hasta que *cuenta* llega al valor 6. La situación que debe ocurrir es la siguiente. De nuevo, ponemos como referencia que estamos en el instante t = 0 para mayor claridad.

- (t = 0) Salta el temporizador, tarea 1 actualiza *cuenta* a 6. Se reinicia el evento 2 a 0.801 segundos. Se envía una señal por el evento 3 a tarea 3. Tarea 3 tiene menos prioridad que tarea 1, por lo que debe esperar y tarea 1 continúa. Al final del bucle, se cumple la condición de *cuenta* igual a 6, por lo que también se envía una señal por el evento 1.
- (t = 0+) Main es la tarea que espera por el evento 1, pero recordemos que ya no tiene la máxima prioridad y tiene un comportamiento FIFO, por lo que tiene que esperar a que tarea 1 y 3, que están en cola, terminen de ejecutarse. Tarea 1 sigue con el control, cancela inmediatamente la tarea 3, actualiza *fin* a 1 e imprime por pantalla el valor de *cuenta* y *fin* (6 y 1). No vuelve a entrar en el bucle porque la condición es que *fin* sea 0 y la tarea se termina.
- (t = 0+) Aunque tarea 3 estaba en cola, fue cancelada por tarea 1, la cual tiene mayor prioridad y la tarea queda cancelada inmediatamente sin que pueda volver a imprimir por pantalla.
- (t = 0+) Main entra en ejecución, imprime un aviso de que va a finalizar y se encuentra con una línea que lo duerme durante 1 segundo más.
- (t = 0.801) Tarea 2 recibe el evento de tarea 1, la cual ya había terminado, pero reinició el evento una vez más como parte del último bucle. Tarea 2 imprime el aviso por pantalla.
- (t = 2) Main se despierta y termina el programa.

El resultado del terminal coincide exactamente con el comportamiento descrito.

Este ejemplo, a pesar de ser sencillo, pone de manifiesto todas las características de tiempo real que hemos programado en la librería (manejo muy preciso y estricto del tiempo y prioridades entre tareas), por lo que resulta de gran ayuda a la hora de comprobar si un ordenador y su sistema son compatibles o se han instalado correctamente para soportar estas funcionalidades. Si el resultado al ejecutar este programa devuelve por el terminal las mismas líneas que se mostraron anteriormente, es buena señal de que todo funciona.

6.1.1 Compilación con ayuda de Cmake

Se propone una forma alternativa, probablemente más sencilla, para compilar programas con la librería de tiempo real mediante Cmake, el cual ya se había instalado para este proyecto. Para ello solo necesitamos crear una carpeta en cualquier directorio, donde introduciremos el archivo .cpp del programa, la librería .hpp y un fichero de Cmake llamado "Makefile". Para realizar este último fichero se ha utilizado como referencia el manual[36] que proporciona GNU.

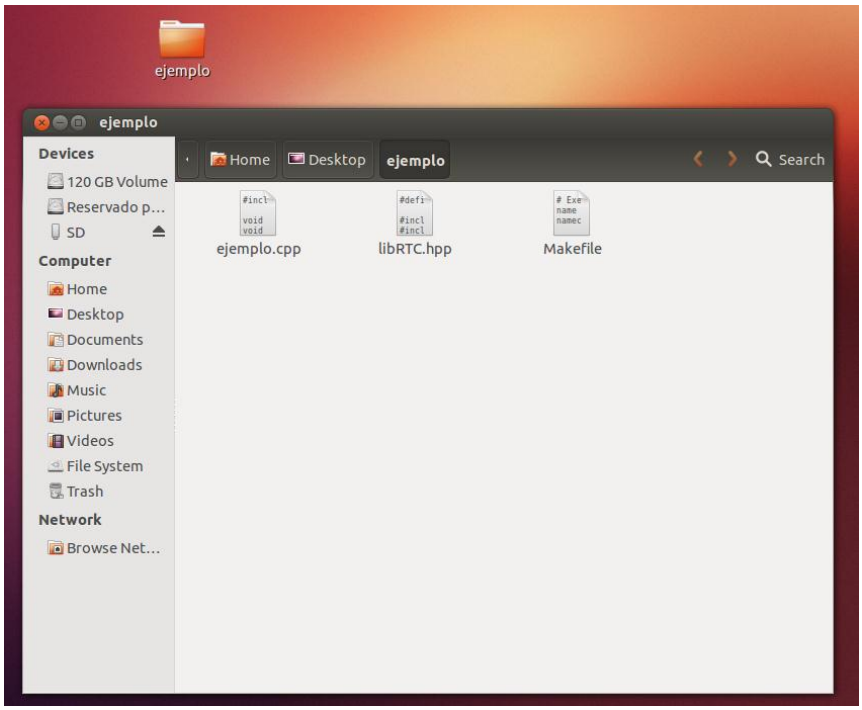

Ilustración 6-2. Compilando un ejemplo de tiempo real (I)

En este ejemplo, hemos creado una carpeta "ejemplo" en el escritorio y hemos introducido el programa "ejemplo.cpp", explicado en este mismo apartado 6.1, la librería "libRTC.hpp" y el "Makefile". Este último fichero es un archivo de texto que contiene lo siguiente:

```
# Executable name and CPP program name
name = ejemplo                  # MODIFICAR
namecpp = ejemplo.cpp           # MODIFICAR

# RT Flags
FLAGS = -lrt -lpthread

all: $(name)

$(name): $(namecpp)
```

[36] https://www.gnu.org/software/make/manual/make.html [Último acceso: julio 2016]

```
g++ $(namecpp) -o $(name) $(FLAGS)
```

En este archivo de texto solo han de modificarse las primeras dos líneas, sin contar con el comentario inicial. La variable *name* debe estar asociada con el nombre del ejecutable y la variable *namecpp* con el nombre del fichero .cpp a compilar.

Finalmente, abrimos el terminal, accedemos a la carpeta donde hemos guardado estos ficheros y simplemente ejecutamos la línea *make* para compilar. Como siempre, no debe olvidarse pedir permisos de administrador.

```
sudo -s
cd Desktop/ejemplo
make
```

Con esto quedaría el programa compilado y podríamos correrlo con la siguiente línea:

```
./ejemplo
```

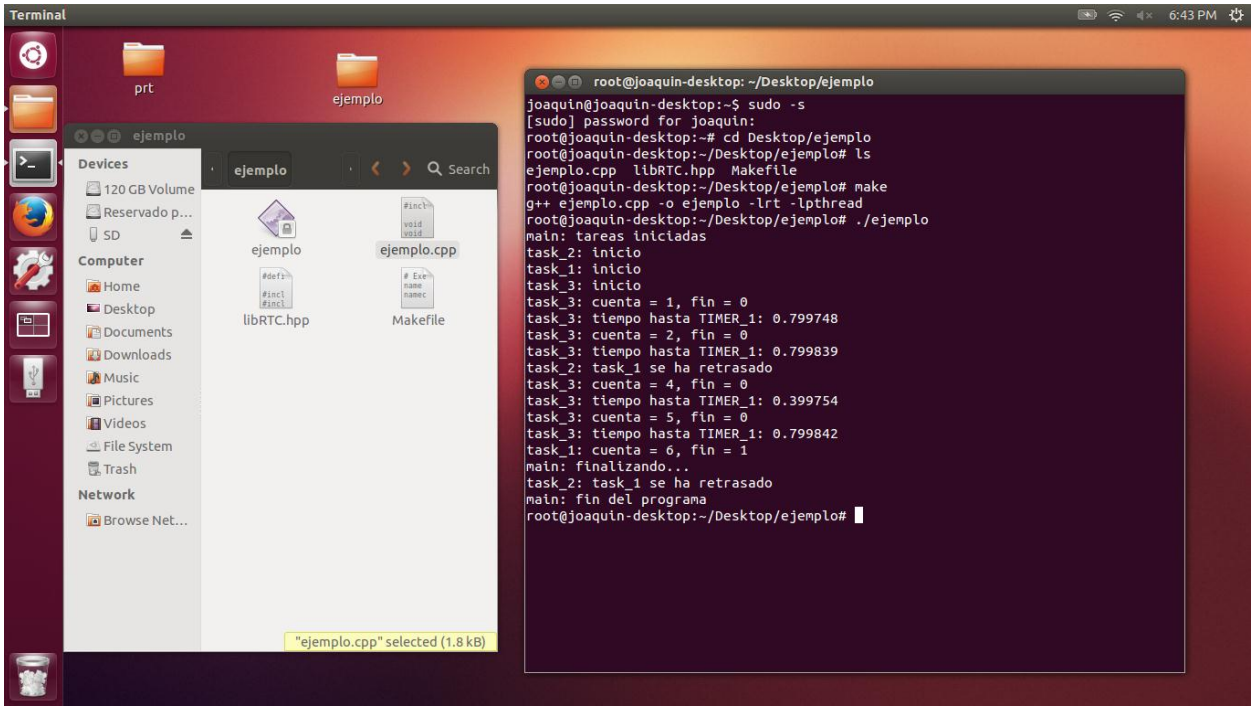

Ilustración 6-3. Compilando un ejemplo de tiempo real (II)

6.2 Controlador PI

Se va a presentar la implementación de un control PI (proporcional integral) discreto que utiliza diferentes tareas para realizar el algoritmo cálculo, para imprimir por pantalla y para escanear la referencia. Aunque no es un ejemplo que demuestre el funcionamiento ante situaciones críticas de tiempo real (debido a que es un único control simple), se pretende que sirva como primera referencia para programar controladores con la librería de tiempo real y utilizar los conceptos de tareas, temporizadores, eventos y sincronización que se han visto en el apartado anterior. Se pueden obtener programas más complejos y de tiempo crítico realizando múltiples controladores de múltiples tareas, y la sintaxis de programación seguiría siendo igual se sencilla, solo que con un código más largo.

Empezamos la programación añadiendo la librería y declarando de forma global las tareas, temporizadores y eventos que utilicemos.

```
#include "libRTC.hpp"
```

```
task task_1; // Tarea 1 para el algoritmo de control
void *controlPI(void*);

task task_2; // Tarea 2 para escanear la referencia
void *scanPI(void*);

task task_3; // Tarea 3 para imprimir por pantalla
void *printPI(void*);

timer TIMER_1;
event EVENT_1;

signal Tm; // Tiempo de muestreo para TIMER_1
signal SIGEVENT_1;
```

Hemos creado 3 tareas, 1 temporizador para el tiempo de muestreo, y 1 evento que veremos más adelante cómo los hemos utilizado para la sincronización entre las tareas.

Mostramos ahora cómo sería la función principal main.

```
int main () {

        /* Inicializar tareas, temps, y eventos */

        create_timer(&TIMER_1, &Tm, 0.6); // Tm = 0.6
        create_event(&EVENT_1, &SIGEVENT_1, 2);

        init_task(&task_1, controlPI, 50); // Prioridad intermedia
        init_task(&task_2, scanPI, 70); // Prioridad alta
        init_task(&task_3, printPI, 30); // Prioridad baja


        /* Espera */

        printf("main: iniciando ejemplo de control PI\n");
        printf("main: durmiendo 1 minuto\n");

        sleep_task(60);


        /* Finalización */

        delete_timer(&TIMER_1);
        delete_event(&EVENT_1);
        cancel_task(&task_1);
        cancel_task(&task_2);
        cancel_task(&task_3);
        printf("main: fin del programa\n");

        return 0;
}
```

Hemos dividido main en tres bloques principales: Uno de inicialización, otro de espera y otro de finalización. En la inicialización hemos creado primero un temporizador de 600 ms que será el tiempo de muestreo del lazo de control, y un evento programado al principio para que se active a los 2 segundos y que veremos más adelante cuál es su uso. Después, hemos creado las tres tareas con la función *create_task*, hemos enlazado las variables

globales con la función que hará de tarea y hemos dado un número de prioridad. Recordando del capítulo anterior, el número de prioridad es un entero del 1 al 99, y cuanto mayor sea el número mayor será la prioridad. En este programa, hemos asociado la prioridad más alta a la tarea que lee la referencia, después al algoritmo de control, y por último y menos importante la tarea de mostrar datos por pantalla.

Repetimos que es muy importante que los temporizadores y eventos se creen en el código antes de que se inicialicen las tareas, de lo contrario las tareas se compilarán haciendo referencia a objetos no existentes y no funcionarán.

En el bloque de espera típicamente irá una condición que cuando se active termine main. Cuando main acaba, todo acaba, por lo que es necesario escribir algo que pause o duerma a main mientras el resto de tareas trabajan. En este ejemplo, en lugar de colocar una condición, que puede ser la de recibir un evento concreto, se ha utilizado simplemente la función *sleep_task*, que como ya hemos explicado, recibe un tipo *double* con el tiempo en segundos y pone a dormir ese tiempo a la tarea en la que se encuentre. Por tanto, main durará un minuto, y tras ese tiempo las tareas que estaban en ejecución o en espera terminarán, y luego el programa acabará.

En el bloque de finalización hemos escrito las funciones que eliminan los objetos de tiempo real de la memoria, aunque como ya se dijo en el capítulo anterior, son opcionales puesto que al acabar main se debería realizar esa acción por defecto.

Continuamos con la programación de las tareas.

```
// Variables globales del PI y sus valores iniciales
float ref = 5;
float act = 0;
float est = 2;

void *controlPI(void*){

        // Constantes y valores iniciales del PI
        float K = 1.42;
        float Ti = 0.65;
        float err_act = 0;
        float sum_err = 0;

        while(1){

            wait_signal(&Tm); // Espera señal muestreo

            // Algoritmo de control PI
            err_act = ref - est;
            sum_err = sum_err + err_act;
            act = K*(err_act + 0.6/Ti*sum_err);

            // Modelo simple para la simulación
            est = est + act/2;

            force_event(&EVENT_1); // Avisa del fin del bucle
        }

}
```

Primero, hemos definido como variables globales la referencia (*ref*), la actuación (*act*) y el estado o salida del sistema (*est*), puesto que estas variables también deben ser conocidas por las otras dos tareas. Dentro de la tarea *controlPI*, hemos definido variables necesarias para el controlador y después se entra directamente al bucle. En el bucle encontramos primero la espera a la señal del tiempo de muestreo con wait_signal, seguido de las operaciones del algoritmo de control, una actualización del estado de la planta con un modelo simple (puesto que el objetivo es explicar la parte de la librería) y terminando con una función que fuerza la activación del

evento.

El evento se programó inicialmente a 2 segundos, pero como a los 0.6 segundos se realiza la primera iteración del bucle, poco después se forzará la activación del evento, eliminando la cuenta de 2 segundos. Por lo tanto, el evento es un aviso que se envía cada vez que se termina de recoger el bucle. Usaremos esta señal posteriormente en la tarea de imprimir por pantalla.

Enseñamos ahora la segunda tarea, encargada de imprimir por pantalla los resultados tras cada iteración.

```
int cnt = 0;

void *printPI(void*){

        // Imprime por pantalla el estado inicial
        printf("printPI: ref=%f; est=%f; act=%f;\n",ref,est,act);

        while(1){

           wait_signal(&SIGEVENT_1);

           // Cuenta cuántas veces seguidas ref-est < 0.001
           if (ref-est < 0.001) cnt++;
           else cnt = 0;

           // Si la cuenta supera 8, se ha llegado a RP
           if (cnt <= 8)
              printf("printPI: ref=%f; est=%f; act=%f;\n",
                      ref,est,act);
           else printf("printPI: cambie la referencia\n");
        }

}
```

La tarea imprime por pantalla una primera vez el estado inicial de las variables, y luego entra en un bucle infinito. En el bucle espera primero que le llegue la señal del evento, la cual indica que se ha terminado de realizar una iteración de control. Después, actualiza un valor de cuenta *cnt* que se ha declarado como global porque también se utilizará en la siguiente tarea. Si la diferencia entre la referencia y la salida del sistema es menor de 0.001, se añade una unidad a la cuenta, y si no se reinicia la cuenta a cero. Esto quiere decir que la cuenta en un determinado instante indica cuántas veces seguidas ha sido esta diferencia menor de 0.001, lo cual es muy útil para averguar si se ha llegado a régimen permanente.

En una siguiente condición, se decide qué imprimir por pantalla según si se ha llegado o no a régimen permanente. Si aún no se ha llegado, se imprimen resultados por pantalla, y en caso contrario saldrá un mensaje que indique que se cambie el valor de referencia.

Por último, la tarea que lee la referencia por teclado.

```
void *scanPI(void*){

        while(1){

           scanf("%f",&ref);
           cnt = 0;
        }

}
```

El bucle es muy simple. La función *scanf* ya de por sí actúa como una línea que duerme la tarea hasta que se

introduce un valor. Aunque es la tarea de mayor prioridad, la mayor parte del tiempo estará dormida dejando paso a las otras dos. También reinicia la variable de cuenta que introdujimos en la tarea anterior para un mejor funcionamiento, ya que la referencia puede introducirse en cualquier momento independientemente de lo que se esté ejecutando debido a que esta tarea tiene la prioridad máxima de las tres.

Ejecutando el programa desde el terminal obtenemos este resultado:

```
joaquin@joaquin-desktop:~$ sudo -s
[sudo] password for joaquin:

root@joaquin-desktop:~# cd Desktop/prt
root@joaquin-desktop:~/Desktop/prt# ./controlPI

main: iniciando ejemplo de control PI
main: durmiendo 1 minuto
printPI: ref=5.000000; est=2.000000; act=0.000000;
printPI: ref=5.000000; est=6.096154; act=8.192307;
printPI: ref=5.000000; est=6.565636; act=0.938965;
printPI: ref=5.000000; est=5.675692; act=-1.779888;
printPI: ref=5.000000; est=4.974770; act=-1.401844;
printPI: ref=5.000000; est=4.788038; act=-0.373464;
printPI: ref=5.000000; est=4.872802; act=0.169528;
printPI: ref=5.000000; est=4.980747; act=0.215890;
printPI: ref=5.000000; est=5.024669; act=0.087844;
printPI: ref=5.000000; est=5.021239; act=-0.006860;
printPI: ref=5.000000; est=5.006325; act=-0.029828;
printPI: ref=5.000000; est=4.997854; act=-0.016941;
printPI: ref=5.000000; est=4.996804; act=-0.002100;
printPI: ref=5.000000; est=4.998594; act=0.003580;
printPI: ref=5.000000; est=5.000034; act=0.002881;
printPI: ref=5.000000; est=5.000430; act=0.000791;
printPI: ref=5.000000; est=5.000263; act=-0.000334;
printPI: ref=5.000000; est=5.000042; act=-0.000441;
printPI: ref=5.000000; est=4.999950; act=-0.000183;
printPI: ref=5.000000; est=4.999957; act=0.000012;
printPI: ref=5.000000; est=4.999987; act=0.000060;
printPI: ref=5.000000; est=5.000004; act=0.000035;
printPI: cambie la referencia
printPI: cambie la referencia
3
printPI: ref=3.000000; est=2.269231; act=-5.461545;
printPI: ref=3.000000; est=1.956242; act=-0.625978;
printPI: ref=3.000000; est=2.549538; act=1.186593;
printPI: ref=3.000000; est=3.016820; act=0.934563;
printPI: ref=3.000000; est=3.141308; act=0.248977;
printPI: ref=3.000000; est=3.084799; act=-0.113019;
printPI: ref=3.000000; est=3.012835; act=-0.143927;
printPI: ref=3.000000; est=2.983554; act=-0.058562;
printPI: ref=3.000000; est=2.985841; act=0.004574;
printPI: ref=3.000000; est=2.995784; act=0.019886;
printPI: ref=3.000000; est=3.001431; act=0.011293;
printPI: ref=3.000000; est=3.002131; act=0.001400;
printPI: ref=3.000000; est=3.000937; act=-0.002387;
printPI: ref=3.000000; est=2.999977; act=-0.001921;
printPI: ref=3.000000; est=2.999713; act=-0.000527;
printPI: ref=3.000000; est=2.999825; act=0.000223;
printPI: ref=3.000000; est=2.999972; act=0.000294;
printPI: ref=3.000000; est=3.000033; act=0.000122;
7
printPI: ref=7.000000; est=8.461567; act=10.923069;
printPI: ref=7.000000; est=9.087524; act=1.251914;
```

```
printPI: ref=7.000000; est=7.900920; act=-2.373208;
printPI: ref=7.000000; est=6.966356; act=-1.869129;
printPI: ref=7.000000; est=6.717382; act=-0.497948;
printPI: ref=7.000000; est=6.830403; act=0.226042;
printPI: ref=7.000000; est=6.974330; act=0.287854;
printPI: ref=7.000000; est=7.032893; act=0.117124;
printPI: ref=7.000000; est=7.028318; act=-0.009149;
printPI: ref=7.000000; est=7.008432; act=-0.039772;
printPI: ref=7.000000; est=6.997139; act=-0.022587;
printPI: ref=7.000000; est=6.995739; act=-0.002800;
printPI: ref=7.000000; est=6.998126; act=0.004773;
printPI: ref=7.000000; est=7.000046; act=0.003841;
printPI: ref=7.000000; est=7.000573; act=0.001053;
printPI: ref=7.000000; est=7.000350; act=-0.000445;
printPI: ref=7.000000; est=7.000056; act=-0.000588;
printPI: ref=7.000000; est=6.999934; act=-0.000244;
printPI: ref=7.000000; est=6.999942; act=0.000015;
printPI: ref=7.000000; est=6.999982; act=0.000081;
printPI: ref=7.000000; est=7.000006; act=0.000046;
printPI: cambie la referencia
printPI: cambie la referencia
printPI: cambie la referencia
printPI: cambie la referencia
printPI: cambie la referencia
printPI: cambie la referencia
1printPI: cambie la referencia
.3
printPI: ref=1.300000; est=-0.782694; act=-15.565388;
printPI: ref=1.300000; est=-1.674710; act=-1.784032;
printPI: ref=1.300000; est=0.016185; act=3.381789;
printPI: ref=1.300000; est=1.347937; act=2.663504;
printPI: ref=1.300000; est=1.702728; act=0.709582;
printPI: ref=1.300000; est=1.541676; act=-0.322105;
printPI: ref=1.300000; est=1.336580; act=-0.410191;
printPI: ref=1.300000; est=1.253128; act=-0.166903;
printPI: ref=1.300000; est=1.259646; act=0.013036;
printPI: ref=1.300000; est=1.287984; act=0.056675;
printPI: ref=1.300000; est=1.304077; act=0.032186;
printPI: ref=1.300000; est=1.306072; act=0.003990;
printPI: ref=1.300000; est=1.302671; act=-0.006802;
printPI: ref=1.300000; est=1.299934; act=-0.005474;
printPI: ref=1.300000; est=1.299184; act=-0.001501;
printPI: ref=1.300000; est=1.299501; act=0.000635;
printPI: ref=1.300000; est=1.299920; act=0.000838;
printPI: ref=1.300000; est=1.300094; act=0.000348;
printPI: cambie la referencia

...

main: fin del programa
```

Se ha destacado en color azul lo que se ha introducido por teclado. A veces, puede entremezclarse lo que se introduce por teclado con lo que el programa está imprimiendo, pero solo es una mezcla visual, al final el programa solo recibe los números introducidos por teclado. La tecla de retroceso para borrar lo introducido sigue funcionando, aunque visualmente no se aprecie que se borre el número. Una vez introducido el número correctamente, se debe pulsar *Enter* para que el programa lo reciba.

7 SIMULACIONES DE CONTROL EN TIEMPO REAL

En este capítulo se va a integrar ACADO Toolkit con la librería de tiempo real para realizar simulaciones de control predictivo no lineal con restricciones. Se utilizarán y se demostrarán las características de la librería con cálculos más complejos que requieren más tiempo de procesador.

7.1 Simulación de control con ACADO en bucle abierto

En este apartado se va a realizar el primer ejemplo de control con ACADO. Hasta ahora solo se había descrito desde un punto de vista general para qué sirve esta herramienta, y también se ha explicado cómo se instala en Ubuntu. Como aún no se ha entrado a programar con ACADO y el objetivo final es integrar esta herramienta con la librería de tiempo real, se va a presentar primero un ejemplo realizado únicamente con ACADO con el objetivo de explicar rápidamente su programación y cómo obtener resultados. Esencialmente, se va a implementar un control predictivo basado en modelo (MPC).

El sistema sobre el que se va a trabajar es una planta de cuatro tanques presentada en la *17th IEEE International Conference on Control Applications* [10].

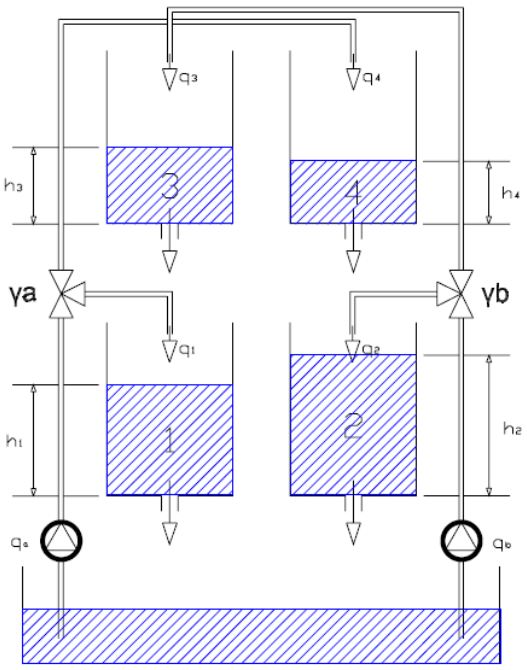

Ilustración 7-1. Proceso de cuatro tanques

Los parámetros de la planta son:

- A_i: Sección transversal del tanque i.
- a_i: Constante de descarga del tanque i.
- h_i: Nivel de agua en el tanque i (estado del sistema).
- q_a, q_b: Flujo producido por las bombas de agua a y b.
- g: Aceleración de la gravedad.
- q_i: Afluencia del tanque i.
- γ_i: Parámetros de las válvulas de tres vías.

Las ecuaciones del modelo del sistema son las siguientes:

$$\frac{dh_1}{dt} = -\frac{a_1}{A_1}\sqrt{2gh_1} + \frac{a_3}{A_1}\sqrt{2gh_3} + \frac{\gamma_a}{A_1}q_a$$

$$\frac{dh_2}{dt} = -\frac{a_2}{A_2}\sqrt{2gh_2} + \frac{a_4}{A_2}\sqrt{2gh_4} + \frac{\gamma_b}{A_2}q_b$$

$$\frac{dh_3}{dt} = -\frac{a_3}{A_3}\sqrt{2gh_3} + \frac{(1-\gamma_b)}{A_3}q_b$$

$$\frac{dh_4}{dt} = -\frac{a_4}{A_4}\sqrt{2gh_4} + \frac{(1-\gamma_a)}{A_4}q_a$$

Sus restricciones:

$$0.3 < h_1 < 1.36\ (m)$$

$$0.3 < h_2 < 1.36\ (m)$$

$$0.3 < h_3 < 1.30\ (m)$$

$$0.3 < h_4 < 1.30\ (m)$$

$$0 < q_a < 5.1\ (m^3/h)$$

$$0 < q_b < 4.85\ (m^3/h)$$

Y el valor de los parámetros:

$$a_1 = 1.341e\text{-}4\ (m^2)\ ;\ a_2 = 1.533e\text{-}4\ (m^2)\ ;\ a_3 = 9.322e\text{-}5\ (m^2)\ ;\ a_4 = 9.061e\text{-}5\ (m^2)\ ;$$

$$A_i = 0.06\ (m^2)\ ;\ \gamma_a = 0.3\ ;\ \gamma_b = 0.4\ ;$$

Con estos datos ya se puede realizar una simulación de control con ACADO, ya que permite perfectamente añadir ecuaciones en tiempo continuo con sus restricciones y trabajar directamente con eso. Se empleará un control predictivo basado en modelo (MPC).

Para comenzar un programa es necesario empezar con las siguientes líneas, que corresponden con las librerías y la inicialización de ACADO:

```
#include <acado_toolkit.hpp>
#include <acado_gnuplot.hpp>

using namespace std;

USING_NAMESPACE_ACADO
```

El bloque main principal tiene la siguiente estructura:

```
int main() {

  // Definición de variables
```

```
    // Definición de constantes
    // Implementación de las ecuaciones del modelo
    // Implementación de la función de costes y la referencia
    // Condiciones de contorno y restricciones
    // Llamada a GNUplot para imprimir los resultados en gráficas
    // Configuración y llamada al solver
    // Obtención de resultados

    return 0;
}
```

Vamos a mostrar punto a punto la programación y la sintaxis para implementar correctamente el ejemplo.

```
// Definición de variables
 DifferentialState h1, h2, h3, h4;
 Control qa, qb;
 DifferentialEquation f(0.0, 200);
        //ecuación diferencial y tiempo de simulación

// Definición de constantes
 const double a1 = 1.341e-4;
 const double a2 = 1.533e-4;
 const double a3 = 9.322e-5;
 const double a4 = 9.061e-5;
 const double A = 0.06;
 const double ya = 0.3;
 const double yb = 0.4;
 const double g = 9.81;

// Implementación de las ecuaciones del modelo
 f << dot(h1) == -a1/A*sqrt(2*g*h1) + a3/A*sqrt(2*g*h3) + ya/A*qa;
 f << dot(h2) == -a2/A*sqrt(2*g*h2) + a4/A*sqrt(2*g*h4) + yb/A*qb;
 f << dot(h3) == -a3/A*sqrt(2*g*h3) + (1-yb)/A*qb;
 f << dot(h4) == -a4/A*sqrt(2*g*h4) + (1-ya)/A*qa;

// Implementación de la función de costes y la referencia
 Function h;
 h << h1;
 h << h2;
 h << h3;
 h << h4;
 h << qa;
 h << qb;

 DMatrix Q(6, 6); //matriz de pesos (diagonal)
 Q(0, 0) = 100;
 Q(1, 1) = 10;
 Q(2, 2) = 1;
 Q(3, 3) = 1;
 Q(4, 4) = 0.01;
 Q(5, 5) = 0.01;

 DVector r(6); //vector referencia
 r(0) = 0.8;
 r(1) = 0.8;
 r(2) = 0.8;
 r(3) = 0.8;
```

```
r(4) = 0;
r(5) = 0;

OCP ocp(0, 200, 10); //horizonte de tiempo
ocp.minimizeLSQ(Q, h, r); //optimización

// Condiciones de contorno y restricciones
ocp.subjectTo(f);

 //alturas y caudales iniciales
ocp.subjectTo(AT_START, h1 == 0.5);
ocp.subjectTo(AT_START, h2 == 0.5);
ocp.subjectTo(AT_START, h3 == 0.5);
ocp.subjectTo(AT_START, h4 == 0.5);
ocp.subjectTo(AT_START, qa == (0.5 / 3600));
ocp.subjectTo(AT_START, qb == (0.5 / 3600));

 //restricciones
ocp.subjectTo(0.3 <= h1 <= 1.36);
ocp.subjectTo(0.3 <= h2 <= 1.36);
ocp.subjectTo(0.3 <= h3 <= 1.30);
ocp.subjectTo(0.3 <= h4 <= 1.30);
ocp.subjectTo(0 <= qa <= (5.1 / 3600));
ocp.subjectTo(0 <= qb <= (4.85 / 3600));

// Llamada a GNUplot para imprimir los resultados en gráficas
GnuplotWindow window;
window.addSubplot(h1, "h1");
window.addSubplot(h2, "h2");
window.addSubplot(h3, "h3");
window.addSubplot(h4, "h4");
window.addSubplot(qa, "qa");
window.addSubplot(qb, "qb");

// Configuración y llamada al solver
OptimizationAlgorithm algorithm(ocp);

algorithm.set(MAX_NUM_ITERATIONS, 200);
algorithm.set(HESSIAN_APPROXIMATION, GAUSS_NEWTON);
algorithm.set(INTEGRATOR_TYPE, INT_RK12);
algorithm.set(KKT_TOLERANCE, 3e-3);
algorithm.set(ABSOLUTE_TOLERANCE, 1e-2);
algorithm.set(INTEGRATOR_TOLERANCE, 1e-2);
algorithm.set(DISCRETIZATION_TYPE, SINGLE_SHOOTING);
algorithm.set(LINEAR_ALGEBRA_SOLVER, GAUSS_LU);
algorithm.set(NUM_INTEGRATOR_STEPS, 500);

algorithm << window;
algorithm.solve();

// Obtención de resultados
VariablesGrid controls;

algorithm.getControls(controls); //obtiene qa y qb
controls.print(); //imprime controls, que contiene qa y qb
```

```
DVector U1;
U1 = controls.getVector(1);

double DU1 = U1(0, 0); //pasamos qa a un tipo double
double DU2 = U1(1, 0); //pasamos qb a un tipo double

printf("\nControl: qa = %f qb = %f\n", DU1, DU2);
```

Para implementar un MPC es necesario definir el estado (*DifferentialState*) y las variables de control (*Control*), además de los parámetros que sean necesarios para las ecuaciones. Después, se escribe las ecuaciones del modelo del sistema en el objeto *DifferentialEquation*, que recibe como argumentos el horizonte de predicción. También se debe implementar la matriz de pesos y el vector de referencia. Todo va asociado a un objeto *OCP*, que debe recibir por argumentos el horizonte de predicción y el tiempo entre las iteraciones que va a realizar. Después, se asocia todo en la función *minimizeLSQ*. Al sistema definido por el objeto *OCP*, se le añaden condiciones de contorno y restricciones, se configuran las opciones del solver y se hace una llamada a este. Por último, se recogen los resultados del cálculo obtenido.

Este programa se ha guardado como "tanques.cpp". Para compilarlo debemos guardarlo dentro de la raíz del directorio de ACADO, en la carpeta "examples/my_examples". Es necesario permisos de administrador para escribir archivos dentro de ese directorio, por lo que tenemos que abrir el terminal para ello. Primero accedemos a la carpeta donde tenemos "tanques.cpp", y escribimos la siguiente línea.

```
sudo -s
cp tanques.cpp /ACADOtoolkit/examples/my_examples
```

Es necesario añadir también la ruta donde se encuentra ACADOtoolkit. Una vez copiado, accedemos a la raíz de ACADO y compilamos.

```
cd ACADOtoolkit
cd build
cmake ..
make
```

Tras este paso, aparecería un ejecutable del mismo nombre que el fichero .cpp en la carpeta donde se ha guardado ("ACADOtoolkit/examples/my_examples"). Accedemos a la carpeta desde el terminal y lo ejecutamos.

```
cd ../examples/my_examples
./tanques
```

El resultado tras ejecutar "tanques*"* sería el siguiente:

```
root@joaquin-desktop: ~/Downloads/ACADOtoolkit/examples/my_examples
ACADO Toolkit is distributed under the terms of the GNU Lesser
General Public License 3 in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Lesser General Public License for more details.

sqp it | qp its |       kkt tol |       obj val |     merit val |      ls param |
     1 |      6 |  3.179966e+01 |  3.088126e+02 |  3.088126e+02 |  1.000000e+00 |
     2 |      6 |  3.508774e-03 |  3.087558e+02 |  3.087558e+02 |  1.000000e+00 |
     3 |      6 |  2.102868e-07 |  3.087561e+02 |  3.087561e+02 |  1.000000e+00 |

Covergence achieved. Demanded KKT tolerance is 3.000000e-03.

[       0.0000000000000000e+00  1.3888888888888889e-04  1.3888888888888889e-04  ]
[       2.0000000000000000e+01  1.4166666662032842e-03  1.3472222222222221e-03  ]
[       4.0000000000000000e+01  1.4166666666666663e-03  1.3472222222222221e-03  ]
[       6.0000000000000000e+01  1.4166666666666668e-03  1.3472222222222221e-03  ]
[       8.0000000000000000e+01  6.4106868666711798e-04  5.4757415590074977e-04  ]
[       1.0000000000000000e+02  4.6870019660909089e-04  4.6882778988308283e-04  ]
[       1.2000000000000000e+02  4.9473560743222588e-04  4.7913374335237675e-04  ]
[       1.4000000000000000e+02  5.1780992300286971e-04  4.8557851204814777e-04  ]
[       1.6000000000000000e+02  5.3853642150208697e-04  4.8842987345084450e-04  ]
[       1.8000000000000000e+02  5.5749551598015495e-04  4.8785887704595011e-04  ]
[       2.0000000000000000e+02  5.5749551598015495e-04  4.8785887704595011e-04  ]

Control: qa = 0.001417 qb = 0.001347
root@joaquin-desktop:~/Downloads/ACADOtoolkit/examples/my_examples#
```

Ilustración 7-2. Control en bucle abierto con ACADO (I)

En la pantalla del terminal podemos ver que la solución ha convergido. Además, ha imprimido la matriz de resultados, tal y como pedimos al final del código, siendo la primera columna el tiempo, la segunda la predicción de *qa* y la tercera la predicción de *qb*. Además, ha impreso el primer resultado de *qa* y *qb* como tipo *double* correspondientes a la siguiente iteración de control, que sería lo que buscamos para realizar un control en bucle cerrado y en tiempo real, que será el siguiente y último apartado de este capítulo.

Vamos a mostrar también la gráfica que imprime del resultado.

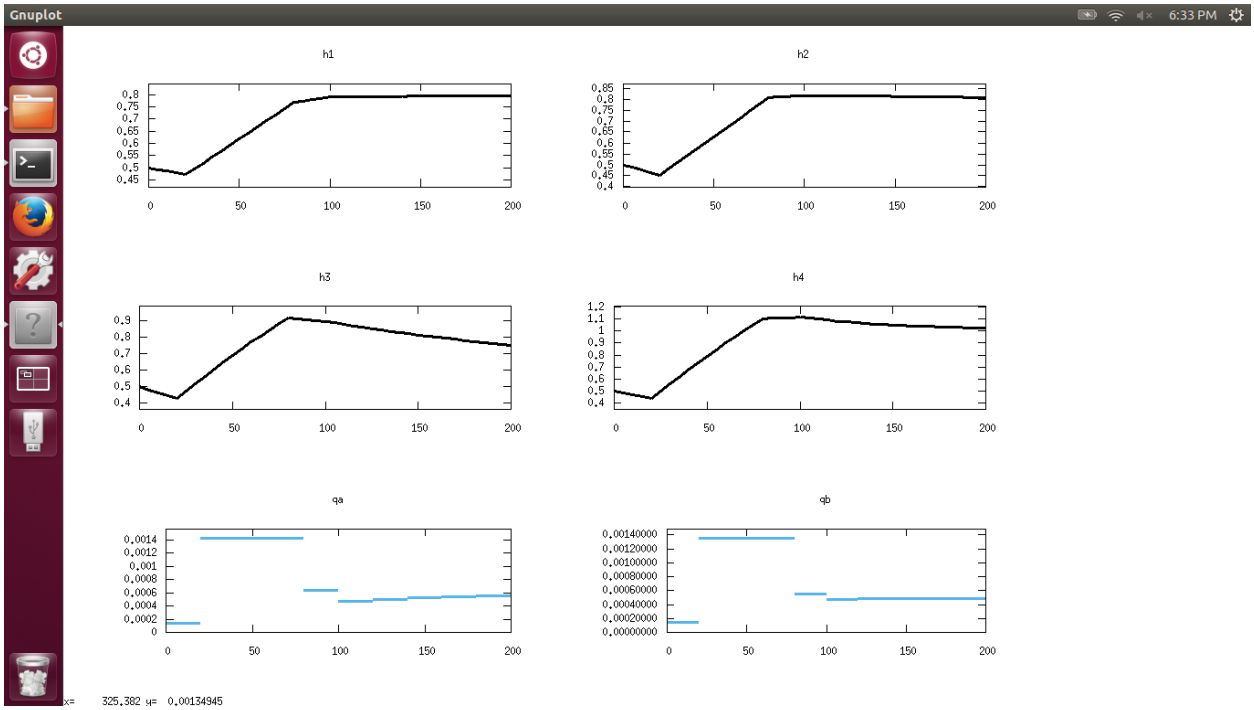

Ilustración 7-3. Control en bucle abierto con ACADO (II)

Podemos observar que las alturas parten del estado inicial 0.5 y aproximadamente llegan a 0.8, tal y como se programó en las condiciones de contorno y el vector de referencia, respectivamente. Si volvemos a fijarnos en la matriz de pesos, se les dio mucha más importancia a los tanques 1 y 2, que serían las dos gráficas superiores

de la imagen. En efecto, el control sobre esos dos tanques es mucho mejor y más preciso en régimen permanente que sobre los tanques 3 y 4, los cuales tienen peso.

Este control predictivo basado en modelo está realizado en bucle abierto y se ha hecho en 10 pasos, correspondientes al horizonte de predicción dado. Se pueden observar perfectamente estos 10 pasos en la gráfica de tipo discreta de *qa* y *qb*.

7.2 Simulación de control en tiempo real con ACADO en bucle cerrado

En esta segunda simulación de este capítulo se va a integrar finalmente la librería de tiempo real desarrollada en el proyecto con ACADO, buscando un control en bucle cerrado sobre el sistema anterior de cuatro tanques. Básicamente, lo que se intenta realizar es la mezcla del primer ejemplo desarrollado de tiempo real con la simulación de ACADO en bucle abierto.

Hemos visto que la simulación con ACADO en bucle abierto nos devolvía una matriz con las futuras actuaciones que debíamos aplicar al sistema, de acuerdo con la predicción realizada sobre el modelo de ecuaciones diferenciales y el horizonte de predicción dado. En bucle cerrado, en cada iteración de cálculo, vamos a ignorar todas las futuras actuaciones y nos vamos a quedar solamente con la primera, que correspondería con la siguiente actuación que se debe aplicar al sistema. Así, se recalcularía la predicción en cada muestreo del sistema.

El código que vamos a presentar a continuación tiene una estructura muy similar a los ejemplos que ya se han descrito en este capítulo.

- Main: Inicializa tareas y objetos de tiempo real.
- Calculo (tarea 1): Tarea principal de cálculo. Simplemente vamos a copiar el código de ACADO realizado en el anterior apartado dentro de esta tarea, con ligeras modificaciones para adaptarlo a bucle cerrado.
- Modelo (tarea 2): Como es una simulación y no un control sobre planta real, vamos a utilizar una tarea extra que haga del modelo del sistema. En principio se van a usar las mismas ecuaciones del modelo, aunque se podrían añadir perturbaciones y otras modificaciones.
- Warning (tarea 3): Vigila y avisa de un retraso en la tarea de cálculo.
- Print (tarea 4): Imprime los resultados tras cada iteración.

Como siempre, vamos a trocear y presentar el código, explicando los nuevos detalles más interesantes.

```
#include "libRTC.hpp"
#include <acado_toolkit.hpp>
#include <acado_gnuplot.hpp>
#include <iostream>
#include <fstream>

using namespace std;

USING_NAMESPACE_ACADO

void *calculo(void*);
void *modelo(void*);
void *warning(void*);
void *print(void*);
```

```
task task_1;
task task_2;
task task_3;
task task_4;
timer TIMER_1;
event EVENT_1;
event EVENT_2;
event EVENT_3;
signal SIGTIMER_1;
signal SIGEVENT_1;
signal SIGEVENT_2;
signal SIGEVENT_3;
```

Hemos incluido la librería de tiempo real, las de ACADO, y otras dos para manejar ficheros en C++. En esta ocasión vamos a imprimir los resultados en un fichero .txt en lugar de por el terminal, ya que ACADO tiene muchos *printf* internos que van a molestarnos y por el método de los ficheros vamos a poder obtener únicamente los resultados que nos interesan. También hemos declarado las cuatro tareas que se van a utilizar, así como un temporizador para el periodo de muestreo y otros tres eventos.

```
int main () {

        /* Inicializacion */

        create_timer(&TIMER_1, &SIGTIMER_1, 5);
        create_event(&EVENT_1, &SIGEVENT_1, 0);
        create_event(&EVENT_2, &SIGEVENT_2, 0);
        create_event(&EVENT_3, &SIGEVENT_3, 0);

        init_task(&task_1,calculo,60);
        init_task(&task_2,modelo,90);
        init_task(&task_3,warning,80);
        init_task(&task_4,print,40);

        /* Condicion de espera */

        printf("main: tareas iniciadas\n");

        while(1) {
           wait_signal(&SIGEVENT_1);
        }

        printf("main: finalizando...\n");
            sleep_task(2);


        /* Terminacion */

        delete_timer(&TIMER_1);
        delete_event(&EVENT_1);
        delete_event(&EVENT_2);
        delete_event(&EVENT_3);
        printf("main: fin del programa\n");

        return 0;
}
```

El código de main es muy similar a los que hemos presentado hasta ahora. Destacamos que hemos asignado a *modelo* la máxima prioridad, por ello de que en la práctica el modelo no está sujeto a un proceso del ordenador, después a *warning*, *calculo* y por último *print*. En la espera, main se queda en un bucle infinito esperando a la señal del evento 1. En realidad no vamos a utilizar el evento 1 hasta el siguiente apartado 6.5, así que de momento se queda esperando indefinidamente.

```
ofstream out("salida.txt");

int fin;
int npaso = 0;
double tiempo = 0;

// Estado inicial
//modelo
double H1 = 0.5;
double H2 = 0.5;
double H3 = 0.6;
double H4 = 0.6;
double QA = (0.5 / 3600);
double QB = (0.5 / 3600);

//referencias
double REF1 = 0.8;
double REF2 = 0.7;
double REF3 = 0.8;
double REF4 = 0.9;

//calculo
double Dqa;
double Dqb;
double Dh1;
double Dh2;
double Dh3;
double Dh4;

// Constantes adicionales
const double a1 = 1.341e-4;
const double a2 = 1.533e-4;
const double a3 = 9.322e-5;
const double a4 = 9.061e-5;
const double A = 0.06;
const double ya = 0.3;
const double yb = 0.4;
const double g = 9.81;


void *calculo(void*){

        printf("calculo: inicio\n");
        fin = 0;

        /* ACADO */

        DifferentialState       h1,h2,h3,h4;
        Control                 qa,qb;
```

```
// Implementando las ecuaciones del sistema
DifferentialEquation     f(0.0,50);
f << dot(h1) == -a1/A*sqrt(2*g*h1) + a3/A*sqrt(2*g*h3) +
ya/A*qa;
f << dot(h2) == -a2/A*sqrt(2*g*h2) + a4/A*sqrt(2*g*h4) +
yb/A*qb;
f << dot(h3) == -a3/A*sqrt(2*g*h3) + (1-yb)/A*qb;
f << dot(h4) == -a4/A*sqrt(2*g*h4) + (1-ya)/A*qa;

Function h;
h << h1;
h << h2;
h << h3;
h << h4;

DMatrix Q(4,4);
Q.setAll(0.0);
Q(0,0) = 100;
Q(1,1) = 10;
Q(2,2) = 1;
Q(3,3) = 1;

DVector r(4);

out << tiempo << " " << QA << " " << QB << " " << H1 << " "
<< H2 << " " << H3 << " " << H4 << endl;

while(!fin){

    wait_signal(&SIGTIMER_1);
    reset_event(&EVENT_2,5.001);
    tiempo = tiempo + 5;

    r.setAll(0.0);
    r(0) = REF1;
    r(1) = REF2;
    r(2) = REF3;
    r(3) = REF4;

    OCP ocp( 0, 50 , 5 );
    ocp.minimizeLSQ(Q,h,r);


    // Condiciones de contorno
    ocp.subjectTo( f                    );
    ocp.subjectTo( AT_START, h1 == H1 );
    ocp.subjectTo( AT_START, h2 == H2 );
    ocp.subjectTo( AT_START, h3 == H3 );
    ocp.subjectTo( AT_START, h4 == H4 );

    // Restricciones
    ocp.subjectTo( 0.3 <= h1 <= 1.36 );
    ocp.subjectTo( 0.3 <= h2 <= 1.36 );
    ocp.subjectTo( 0.3 <= h3 <= 1.30 );
    ocp.subjectTo( 0.3 <= h4 <= 1.30 );
    ocp.subjectTo( 0.0 <= qa <= (5.1 / 3600) );
```

```
            ocp.subjectTo( 0.0 <= qb <= (4.85 / 3600) );

            // Configuración y llamada al solver
            OptimizationAlgorithm algorithm(ocp);

            algorithm.set(MAX_NUM_ITERATIONS, 300 );
            algorithm.set(HESSIAN_APPROXIMATION, GAUSS_NEWTON );
            algorithm.set(INTEGRATOR_TYPE, INT_RK45 );
            algorithm.set(KKT_TOLERANCE, 1e-4 );
            algorithm.set(ABSOLUTE_TOLERANCE, 1e-2 );
            algorithm.set(INTEGRATOR_TOLERANCE,1e-2);
            algorithm.set(DISCRETIZATION_TYPE,MULTIPLE_SHOOTING);
            algorithm.set(LINEAR_ALGEBRA_SOLVER,GAUSS_LU        );
            algorithm.set(NUM_INTEGRATOR_STEPS, 10000 );

            algorithm.solve();

            // Obtencion de resultados
            VariablesGrid controls;
            VariablesGrid DSs;

            algorithm.getControls (controls);
            algorithm.getDifferentialStates (DSs);

            DVector U1;
            U1 = controls.getVector(0);
            Dqa = U1(0,0);
            Dqb = U1(1,0);

            DVector U2;
            U2 = DSs.getVector(1);
            Dh1 = U2(0,0);
            Dh2 = U2(1,0);
            Dh3 = U2(2,0);
            Dh4 = U2(3,0);

            npaso++;

            force_event(&EVENT_3);
        }

        return 0;

}
```

Esta tarea de cálculo tiene el casi el mismo código que el ejemplo anterior de ACADO. Hemos separado la inicialización de variables de las líneas del solver, ya que la inicialización se realiza solo una vez y el resto del código ha de incluirse en el bucle *while* para realizar las iteraciones.

Es muy importante también configurar correctamente el horizonte de predicción del MPC y el tiempo de muestreo. Cuando se realiza un cambio de referencia, el horizonte de predicción determinará cuándo empieza a reaccionar la planta ante el futuro cambio. Además, el tiempo entre cada cálculo de iteración debe coincidir con el tiempo de muestreo con el que se configure el temporizador de tiempo real para que el MPC quede bien configurado. En este caso, el tiempo de muestreo es de 5 segundos.

Destacamos que para trabajar con ficheros de salida hemos declarado la variable global *out* de tipo *ofstream*, que servirá para acceder al fichero de salida "salida.txt".

```
ofstream out("salida.txt");
```

Para escribir en el fichero de salida la sintaxis es diferente que para *prinft*.

```
out << tiempo << " " << QA << " " << QB << " " << H1 << " " << H2
        << " " << H3 << " " << H4 << endl;
```

Esta línea imprimirá una matriz de salida con la siguiente estructura, en la que cada variable es un vector columna de tamaño el número de iteraciones realizadas.

Tiempo	QA	QB	H1	H2	H3	H4

Se debe empezar llamando a *out* y después se incluyen variables o cadenas de caracteres entre comillas. Todos los elementos deben estar separados por un doble signo menor que (<<). Para incluir espacios, debe añadirse una cadena de caracteres que lo incluya (" "). Para terminar la línea se añade *endl* al final. El motivo por el que vamos a imprimir los resultados de esta forma matricial es para que pueda ser reconocido fácilmente después por programas que representen datos gráficamente. Comúnmente se utiliza Matlab, pero en este proyecto enseñaremos a hacerlo con Gnuplot ya que ya está instalado y es una herramienta libre.

En la obtención de resultados, esta vez vamos a sacar tanto las actuaciones (caudales) como las salidas (alturas), ya que el modelo para la simulación del sistema serán sus mismas ecuaciones que ya se han implementado para la predicción. Al terminar el cálculo, se avisa por el evento 3, que lo recogen tanto *modelo* como *print*.

```
void *modelo(void*){

        printf("modelo: inicio\n");

        while(1){

            wait_signal(&SIGEVENT_3);

               H1 = Dh1;
               H2 = Dh2;
               H3 = Dh3;
               H4 = Dh4;
               QA = Dqa;
               QB = Dqb;

               // Cambio de referencia en t = 300 = 60x5
               if (npaso == 60) {
                  REF1 = 0.7;
                  REF2 = 1;
               }
        }

}
```

En la tarea del modelo solo se han pasado los resultados del cálculo a las variables globales del proceso, suponiendo que el sistema se comporta idealmente como sus ecuaciones, cosa que no va a ocurrir. Como este es primer ejemplo de integración de la librería con ACADO no se pretende realizar un programa complejo, sin embargo, posteriormente se puede reutilizar fácilmente esta tarea del modelo para añadir perturbaciones y otras modificaciones para comprobar los resultados y la efectividad del cálculo.

```
void *warning(void*){

        printf("warning: inicio\n");

        while(1){

            wait_signal(&SIGEVENT_2);
            out << "#warning: calculo se ha retrasado" << endl;
            cancel_task(&task_1);
            force_event(&EVENT_1);
        }

}
```

La tarea que avisa de un retraso en el cálculo es análoga a las que hemos programado anteriormente. Esta vez, también cancela a la tarea de cálculo y avisa a main de ello para reiniciarla.

```
void *print(void*){

        printf("print: inicio\n");
        fin = 0;

        while(!fin){

            wait_signal(&SIGEVENT_3);
            out << tiempo << " " << QA << " " << QB << " " << H1 <<
            " " << H2 << " " << H3 << " " << H4 << endl;
            double queda = get_time(&TIMER_1);
            printf("print:  PASO  %d  -  tiempo  hasta  TIMER_1:
            %f\n",npaso,queda);
        }

}
```

Esta tarea imprimirá los resultados del cálculo tras cada muestreo en “salida.txt”. Además, imprimirá en el terminal el número de iteraciones que lleva, así como el tiempo de cálculo.

7.2.1 Compilación de un programa con tiempo real y ACADO

Compilar un programa con ACADO manualmente y fuera del directorio que ellos te imponen es una tarea muy complicada debido a la complejidad de la estructura de las librerías de ACADO, y ha resultado imposible con el tiempo limitado que tenía este proyecto. Pero por otro lado existe una solución más fácil, no del todo elegante, que sí se ha conseguido llegar a implementar, que es la de incluir las librerías de tiempo real dentro de ACADO. El archivo de cabecera realizado en este proyecto “libRTC.hpp” debe aparecer en el código como un *include* y entre comillas, así es reconocido si se encuentra en la misma carpeta que el programa .cpp. Sin embargo la llamada a las librerías de tiempo real internas del sistema no se encuentran si el programa se compila con la configuración de Cmake que trae ACADO por defecto.

En este apartado se va a explicar cómo cambiar esta configuración y poder compilar un programa que integre ACADO con el tiempo real. Se requieren permisos de administrador para poder modificar los archivos internos de ACADO, por lo que tendremos que acceder desde el terminal.

```
sudo -s
cd /ACADOtoolkit
```

Es necesario añadir también la ruta donde se encuentra ACADOtoolkit. Una vez dentro, accedemos al archivo de configuración de Cmake.

```
cd examples
gedit CMakeLists.txt
```

Dentro de este fichero de texto, bajamos hasta el final y añadimos las siguientes dos líneas:

```
SET(CMAKE_CXX_FLAGS_DEBUG "-lpthread -lrt")
SET(CMAKE_CXX_FLAGS_RELEASE "-lpthread -lrt")
```

Guardamos y cerramos el editor de texto. Con esto, ya podemos compilar programas de ACADO que integran el tiempo real. Los programas deben encontrarse en la carpeta "ACADOtoolkit/examples/my_examples", en el caso de un programa de tiempo real deben añadirse tanto el .cpp como "libRTC.hpp". Después, volvemos a la carpeta "ACADOtoolkit/build" y ejecutamos las siguientes dos líneas.

```
cmake ..
make
```

Finalmente, aparecerá el ejecutable en "ACADOtoolkit/examples/my_examples".

```
cd ../examples/my_examples
./ejemploacado
```

Una vez compilado el ejemplo de este apartado, lo ejecutamos y lo dejamos correr durante unos 20 segundos o cuando se observe que el sistema ha llegado a régimen permanente. Como no hemos programado ninguna parada automática, debemos hacer *Ctrl+C* en el terminal para pararlo manualmente. Obtendremos, en el mismo directorio, el archivo "salida.txt".

```
0 0 0 0.5 0.5 0.6 0.6
5 0.00141667 0.00134722 0.554023 0.561535 0.679671 0.711118
10 0.00141667 0.00134722 0.607722 0.622758 0.756081 0.817907
15 0.00141667 0.00134722 0.660989 0.683488 0.829524 0.920813
20 0.00141667 0.00074318 0.712665 0.704661 0.840946 1.02019
25 0.00141667 0.000334867 0.76106 0.701257 0.811876 1.11633
30 0.00128244 0.000326799 0.799181 0.700219 0.783103 1.19405
35 0.00056315 0.000326569 0.799187 0.700269 0.755405 1.18677
40 0.000584399 0.000329334 0.799186 0.700317 0.729053 1.18215
45 0.00060502 0.000330915 0.799178 0.700364 0.703897 1.18003
50 0.000625086 0.000331374 0.799165 0.700411 0.679797 1.18029
55 0.00064468 0.000330773 0.799146 0.700458 0.656623 1.18279
60 0.000663885 0.000329168 0.799122 0.700504 0.634255 1.18743
65 0.000682778 0.00032661 0.799092 0.700551 0.612583 1.19409
70 0.000701436 0.000323153 0.799058 0.700599 0.591505 1.2027
75 0.000719936 0.000318841 0.799019 0.700647 0.570927 1.21317
80 0.000738349 0.000313716 0.798975 0.700695 0.550765 1.22545
85 0.000751931 0.000313827 0.798706 0.701124 0.531526 1.23892
90 0.000762613 0.000317019 0.798115 0.702135 0.513487 1.25322
```

```
95 0.000772954 0.000320167 0.797245 0.703707 0.4966 1.26829
100 0.000782866 0.000323317 0.796128 0.705822 0.480826 1.28405
105 0.00078842 0.0003265 0.794606 0.708457 0.466125 1.3
110 0.000653731 0.000331223 0.785925 0.711449 0.452608 1.3
115 0.000653731 0.000335481 0.777071 0.714521 0.44018 1.3
120 0.000653731 0.000339505 0.7681 0.717653 0.428771 1.3
125 0.000653731 0.000343283 0.759064 0.720827 0.418315 1.3
130 0.000653731 0.000346812 0.750009 0.724024 0.408744 1.3
135 0.000653731 0.00035009 0.740978 0.727226 0.399993 1.3
140 0.000653731 0.00035312 0.73201 0.730419 0.392001 1.3
145 0.000653731 0.000355908 0.723139 0.733586 0.384707 1.3
150 0.000653731 0.00035876 0.714394 0.736735 0.378084 1.3
155 0.000653731 0.000361056 0.705803 0.739832 0.372045 1.3
160 0.000653731 0.000363103 0.697388 0.742863 0.366536 1.3
165 0.000653731 0.000364913 0.689166 0.745819 0.361508 1.3
170 0.000653731 0.000366487 0.681154 0.748688 0.356912 1.3
175 0.000653731 0.000367975 0.673362 0.751472 0.352718 1.3
180 0.000653731 0.000369319 0.665801 0.754167 0.34889 1.3
185 0.000653731 0.000370532 0.658478 0.75677 0.345397 1.3
190 0.000653731 0.000371503 0.651399 0.759272 0.342197 1.3
195 0.000653731 0.000372271 0.644566 0.761666 0.339254 1.3
200 0.000653731 0.000373151 0.63798 0.763966 0.336566 1.3
205 0.000653731 0.000373938 0.631643 0.766174 0.334109 1.3
210 0.000653731 0.000374645 0.625552 0.768288 0.331862 1.3
215 0.000653731 0.000375279 0.619706 0.770312 0.329807 1.3
220 0.000653731 0.000375847 0.614103 0.772245 0.327927 1.3
225 0.000653731 0.000376355 0.608737 0.774091 0.326206 1.3
230 0.000653731 0.00037681 0.603606 0.775852 0.324629 1.3
235 0.000653731 0.000377166 0.598702 0.777525 0.32318 1.3
240 0.000653731 0.000376952 0.59402 0.779081 0.321795 1.3
245 0.000653731 0.000377297 0.58955 0.780563 0.320524 1.3
250 0.000653731 0.000377594 0.585289 0.781972 0.319358 1.3
255 0.000653731 0.00037786 0.581228 0.78331 0.318286 1.3
260 0.000653731 0.000378097 0.577362 0.784581 0.317301 1.3
265 0.000653731 0.00037831 0.573683 0.785788 0.316395 1.3
270 0.000653731 0.000378501 0.570186 0.786931 0.31556 1.3
275 0.000653731 0.000378671 0.566864 0.788016 0.314793 1.3
280 0.000653731 0.000378823 0.563709 0.789043 0.314085 1.3
285 0.000653731 0.000378959 0.560715 0.790015 0.313433 1.3
290 0.000653731 0.000379081 0.557875 0.790935 0.312831 1.3
295 0.000653731 0.00037919 0.555183 0.791806 0.312276 1.3
300 0.000653731 0.000379288 0.552632 0.792629 0.311764 1.3
305 0.000653731 0.00134722 0.552947 0.855949 0.405283 1.3
310 0.000653731 0.00134722 0.558016 0.91551 0.493861 1.3
315 0.000653731 0.00134722 0.566865 0.971648 0.578196 1.3
320 0.000653732 0.00134722 0.578766 1.02465 0.658792 1.3
325 0.000653731 0.000535767 0.591529 1.02215 0.656559 1.3
330 0.000653731 0.000532333 0.6034 1.01956 0.654083 1.3
335 0.000653731 0.000532726 0.614438 1.01714 0.651748 1.3
340 0.000653731 0.000533082 0.624707 1.01487 0.649545 1.3
345 0.000653731 0.000533415 0.634265 1.01275 0.647467 1.3
350 0.000653731 0.000533724 0.643162 1.01077 0.645507 1.3
355 0.000653731 0.00053402 0.651448 1.00891 0.643657 1.3
360 0.000653731 0.000534291 0.659167 1.00717 0.641912 1.3
365 0.000653731 0.000534554 0.666358 1.00555 0.640265 1.3
370 0.000653731 0.000534791 0.673059 1.00403 0.638712 1.3
375 0.000653731 0.000536236 0.679307 1.00268 0.637365 1.3
```

```
380 0.000653731 0.000540242 0.685145 1.00167 0.636467 1.3
385 0.000653731 0.000544415 0.690615 1.00099 0.636015 1.3
390 0.000653731 0.00054877 0.69576 1.00062 0.636009 1.3
395 0.000617466 0.000553372 0.698857 1.00052 0.636453 1.29583
400 0.000557405 0.000558137 0.698888 1.00051 0.637345 1.28488
405 0.000556465 0.000562927 0.698919 1.0005 0.638668 1.27413
410 0.000555166 0.000567662 0.698948 1.00049 0.640399 1.26355
415 0.000553557 0.000572324 0.698978 1.00048 0.642513 1.2531
420 0.000551616 0.000576972 0.699008 1.00047 0.644994 1.24273
425 0.000549375 0.000581608 0.699037 1.00045 0.647824 1.23241
430 0.000546847 0.000586249 0.699067 1.00044 0.65099 1.22211
435 0.000544046 0.00059091 0.699098 1.00043 0.65448 1.21179
440 0.000540982 0.000595603 0.699128 1.00041 0.658285 1.20142
445 0.000537667 0.000600342 0.699159 1.0004 0.662395 1.19099
450 0.000534108 0.000605138 0.69919 1.00039 0.666805 1.18047
455 0.000530312 0.000610005 0.699222 1.00037 0.67151 1.16982
460 0.000526286 0.000614953 0.699254 1.00035 0.676506 1.15904
465 0.000522034 0.000619992 0.699287 1.00034 0.68179 1.1481
470 0.00051756 0.000625134 0.69932 1.00032 0.687363 1.13698
475 0.000512867 0.000630388 0.699354 1.0003 0.693223 1.12567
480 0.000507957 0.000635765 0.699389 1.00028 0.699373 1.11414
485 0.000502832 0.000641274 0.699425 1.00026 0.705815 1.10239
490 0.000497492 0.000646925 0.699461 1.00024 0.712553 1.09039
495 0.000491937 0.000652727 0.699498 1.00022 0.719589 1.07813
500 0.000486166 0.000658691 0.699537 1.0002 0.726931 1.0656
```

En este fichero de resultados hemos imprimido únicamente los resultados separados por un espacio para poder representarlos fácilmente con Gnuplot. En cada línea, aparece *tiempo*, *qa*, *qb*, *h1*, *h2*, *h3* y *h4* por orden, y Gnuplot leerá el fichero como una matriz de siete columnas.

La simulación se ha cortado a los 500 segundos.

7.2.2 Representación del resultado con Gnuplot

La sintaxis de Gnuplot es muy similar a la de Matlab, aunque los nombres de los comandos son diferentes. Para este ejemplo, vamos poner en una misma imagen las seis gráficas correspondientes a las cuatro alturas y los dos caudales. Se ha utilizado como referencia los ejemplos y demostraciones que propone Gnuplot en su misma página web[37].

Debemos encontrarnos desde el terminal en el mismo directorio que el fichero que contiene la matriz de datos. Accedemos con el comando *gnuplot*, y a continuación escribimos las líneas que configurarán la pantalla donde se mostrarán las gráficas.

```
gnuplot
set multiplot layout 3,2
plot “salida.txt” using 1:4 title ‘h1’ with lines
plot “salida.txt” using 1:5 title ‘h2’ with lines
plot “salida.txt” using 1:6 title ‘h3’ with lines
plot “salida.txt” using 1:7 title ‘h4’ with lines
plot “salida.txt” using 1:2 title ‘qa’ with steps
plot “salida.txt” using 1:3 title ‘qb’ with steps
```

[37] http://gnuplot.sourceforge.net/demo/ [Última visita: julio 2016]

La segunda línea abre una imagen blanca donde se irán insertando las gráficas por orden en una estructura 3x2. Es importante saber que antes de empezar a pintar las gráficas, se ha de ajustar la ventana de Gnuplot o hacerla más grande, porque una vez que se empiecen a pintar gráficas el tamaño de la ventana no se puede modificar. Quizás se corrija en futuras versiones.

Las líneas que empiezan por *plot* abren el fichero de datos, y luego se introducen las dos columnas que harán de ejes X:Y. En este caso, se utiliza siempre como eje X el tiempo (columna 1). El comando *lines* dibuja las gráficas uniendo los puntos representados, mientras que *steps* destaca su naturaleza discreta.

En el fichero "salida.txt" se puede añadir comentarios con una almohadilla (#) que no serán leídos por Gnuplot.

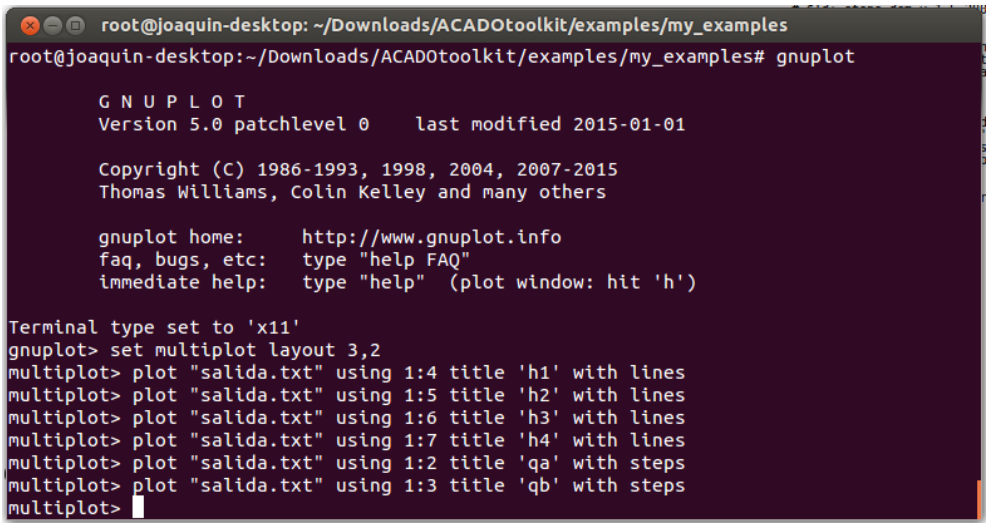

Ilustración 7-4. Representación de datos con Gnuplot

Finalmente, mostramos el resultado que hemos obtenido de la simulación que integra ACADO con la librería de tiempo real, que sería la simulación de un control predictivo MPC en bucle cerrado sobre el proceso de cuatro tanques. En morado quedan representados los resultados, mientras que en rojo se indican las referencias programadas.

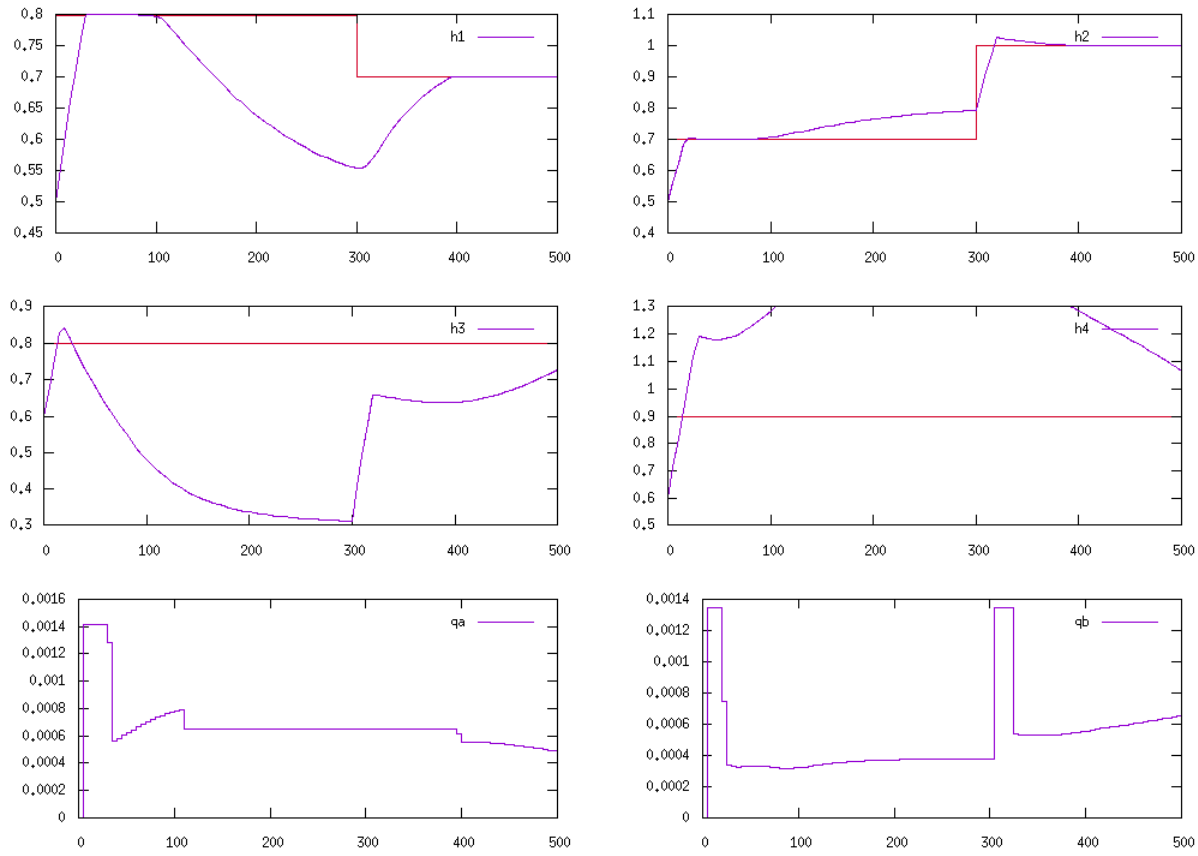

Ilustración 7-5. Control en bucle cerrado con ACADO y en tiempo real

Del resultado gráfico podemos comentar lo siguiente:

- Los dos primeros tanques tienen muy buen comportamiento mientras que los otros están prácticamente sujetos a las interacciones. Tiene sentido porque la matriz de pesos se realizó para asegurar un buen control en los tanques 1 y 2.
- En los tanques 1 y 2 se observa que el control se anticipa al cambio de referencia más de 200 segundos antes, que dividido por el tiempo de muestreo dan casi las 50 iteraciones programadas como horizonte de control.
- Las alturas de los tanques 1 y 2 empiezan en 0.5, mientras que la de los tanques 3 y 4 comienzan en 0.6, tal y como se había establecido en las condiciones iniciales.
- El régimen permanente de los tanques 1 y 2 es prácticamente exacto y sin fluctuaciones.
- Las entradas del sistema (caudales) tienen un comportamiento muy estable y parecen converger a un régimen permanente.
- El tanque 4 llega al máximo de altura (1.3 metros), por lo que rebosaría, suceso que suele ocurrir cuando se le da poca importancia en la matriz de pesos.

7.3 Simulación estricta de tiempo real sobre la planta de cuatro tanques

Queda por demostrar que la integración de la librería con ACADO cumple los requisitos más complicados de tiempo real que ya se habían demostrado con ejemplos más sencillos. Por tanto, esta va a ser la última simulación del proyecto y consistirá en una demostración de la prioridad de las tareas y la cancelación.

La priemra vez que se realizó esta simulación se comprobó que las prioridades funcionaban correctamente, pero había un problema adicional con la cancelación. La cancelación de tareas por parte de la librería funciona correctamente, pero realmente no sabemos cómo va a reaccionar ACADO ante una cancelación. En efecto, en determinadas ocasiones, la cancelación se produce en un momento que ocasiona problemas de memoria e impide que ACADO vuelva a iniciarse. Como ACADO trabaja y llama a librerías externas que no son posibles de estudiar y modificar dado el alcance de este trabajo, tuvimos que buscar una solución al nivel con el que se está tratando ACADO en este proyecto.

Conseguimos hacer funcionar la cancelación reiniciando ciertas variables internas de ACADO cada vez que se iniciaba la tarea de cálculo, además de tratar los objetos de cálculo con memoria dinámica. El código será muy similar al utilizado en el anterior apartado, solo que añadiendo estas dos modificaciones y provocando una cancelación mientras ACADO está calculando.

La inicialización, variables globales, main y las tareas *modelo* y *print* no cambian con respecto al código de la simulación del apartado 7.2. Los cambios se han introducido en las tareas *calculo* y *warning*.

```
void *calculo(void*){

        printf("calculo: inicio\n");
        fin = 0;


        /* ACADO */

        DifferentialState       h1,h2,h3,h4;
        Control                 qa,qb;

        // Reinicio de contadores internos para cancelacion
        h1.clearStaticCounters();
        h2.clearStaticCounters();
        h3.clearStaticCounters();
        h4.clearStaticCounters();
        qa.clearStaticCounters();
        qb.clearStaticCounters();

        // Implementando las ecuaciones del sistema
        DifferentialEquation    f(0.0,50);
        f << dot(h1) == -a1/A*sqrt(2*g*h1) + a3/A*sqrt(2*g*h3) +
        ya/A*qa;
        f << dot(h2) == -a2/A*sqrt(2*g*h2) + a4/A*sqrt(2*g*h4) +
        yb/A*qb;
        f << dot(h3) == -a3/A*sqrt(2*g*h3) + (1-yb)/A*qb;
        f << dot(h4) == -a4/A*sqrt(2*g*h4) + (1-ya)/A*qa;

        Function h;
        h << h1;
        h << h2;
        h << h3;
```

```
h << h4;

DMatrix Q(4,4);
Q.setAll(0.0);
Q(0,0) = 100;
Q(1,1) = 10;
Q(2,2) = 1;
Q(3,3) = 1;

DVector r(4);
r.setAll(0.0);
r(0) = 0.9;
r(1) = 0.9;
r(2) = 0.9;
r(3) = 0.9;

out << tiempo << " " << QA << " " << QB << " " << H1 << " "
<< H2 << " " << H3 << " " << H4 << endl;

while(!fin){

     wait_signal(&SIGTIMER_1);
     reset_event(&EVENT_2,5.001);
     tiempo = tiempo + 5;
     npaso++;

     // Objeto de memoria dinámica para cancelacion
     OCP *ocp;
     ocp = new OCP( 0, 50 , 5 );
     ocp->minimizeLSQ(Q,h,r);


     // Condiciones de contorno
     ocp->subjectTo( f                    );
     ocp->subjectTo( AT_START, h1 == H1 );
     ocp->subjectTo( AT_START, h2 == H2 );
     ocp->subjectTo( AT_START, h3 == H3 );
     ocp->subjectTo( AT_START, h4 == H4 );

     // Restricciones
     ocp->subjectTo( 0.3 <= h1 <= 1.36 );
     ocp->subjectTo( 0.3 <= h2 <= 1.36 );
     ocp->subjectTo( 0.3 <= h3 <= 1.30 );
     ocp->subjectTo( 0.3 <= h4 <= 1.30 );
     ocp->subjectTo( 0.0 <= qa <= (5.1 / 3600) );
     ocp->subjectTo( 0.0 <= qb <= (4.85 / 3600) );

     // Configuración y llamada al solver
     // Objeto de memoria dinámica para cancelacion
     OptimizationAlgorithm* algorithm;
     algorithm = new OptimizationAlgorithm(*ocp);

     algorithm->set(MAX_NUM_ITERATIONS, 100 );
     algorithm->set(HESSIAN_APPROXIMATION, GAUSS_NEWTON );
     algorithm->set(INTEGRATOR_TYPE, INT_RK45 );
     algorithm->set(KKT_TOLERANCE, 1e-9 );
```

```
            algorithm->set(ABSOLUTE_TOLERANCE, 1e-2 );
            algorithm->set(INTEGRATOR_TOLERANCE,1e-2);
            algorithm->set(DISCRETIZATION_TYPE,MULTIPLE_SHOOTING);
            algorithm->set(LINEAR_ALGEBRA_SOLVER,GAUSS_LU       );
            algorithm->set(NUM_INTEGRATOR_STEPS, 1000 );
            algorithm->set( HOTSTART_QP, NO );

            // Programacion manual de retraso
            if (npaso == 3) {
               sleep_task(4.95);
            }

            algorithm->solve();

            // Obtencion de resultados
            VariablesGrid controls;
            VariablesGrid DSs;

            algorithm.getControls (controls);
            algorithm.getDifferentialStates (DSs);

            DVector U1;
            U1 = controls.getVector(0);
            Dqa = U1(0,0);
            Dqb = U1(1,0);

            DVector U2;
            U2 = DSs.getVector(1);
            Dh1 = U2(0,0);
            Dh2 = U2(1,0);
            Dh3 = U2(2,0);
            Dh4 = U2(3,0);

            delete ocp;
            delete algorithm;

            force_event(&EVENT_3);
        }

        return 0;

}
```

Los cambios más importantes que se han hecho para permitir la cancelación son los relaciones con los objetos *ocp* y *algorithm*, el reinicio de los contadores internos y la configuración del flag *HOTSTART_QP* a *NO*. Este último flag que configura el solver permite a ACADO realizar unos cálculos extras en la primera iteración de cálculo para ahorrárselos después y reducir el tiempo de cálculos en las iteraciones posteriores. Sin embargo, cuando cancelamos la tarea de ACADO, estos cálculos extras se pierden con el resto de la tarea, y al reiniciar el control se encontraría con un error de memoria.

Se ha programado manualmente un retraso de 4.95 segundos antes de la tercera iteración de cálculo. Se ha elegido esta cifra porque en anteriores simulaciones se comprobó que el tiempo de cálculo variaba entre 0.1 y 0.2 segundos. Si retrasamos la tarea 4.95 segundos antes de realizar el cálculo, tendría solo 0.05 segundos para realizarlo, y cuando se pasa 0.001 segundos del tiempo estipulado, la tarea *warning* saltará cancelando el hilo. Así nos aseguramos de que la cancelación se produce durante el cálculo de ACADO.

La tarea warning es similar a la anterior, solo que cancela y además vuelve a iniciar el hilo seguidamente.

```
void *warning(void*){

        printf("warning: inicio\n");

        while(1){

            wait_signal(&SIGEVENT_2);
            out << "#warning: calculo se ha retrasado" << endl;
            cancel_task(&task_1);
            init_task(&task_1,calculo,60);
        }

}
```

Compilando y ejecutando el ejemplo, de la misma forma que se explicó en el apartado 7.2, obtenemos este fichero de salida.

```
0 0.000138889 0.000138889 0.5 0.5 0.6 0.6
5 0.00141667 0.00134722 0.607722 0.622758 0.756081 0.817907
10 0.00141667 0.00134722 0.713752 0.743608 0.900241 1.02019
#warning: calculo se ha retrasado
15 0.00141667 0.00134722 0.713752 0.743608 0.900241 1.02019
20 0.00141667 0.00134722 0.713752 0.743608 0.900241 1.02019
25 0.000655276 0.000366542 0.879146 0.897228 1.03991 1.3
30 0.000520667 0.000457393 0.890424 0.899563 0.997929 1.2847
35 0.000520518 0.000475274 0.891121 0.904093 0.964954 1.25322
40 0.00054055 0.000487264 0.891664 0.908055 0.936549 1.22821
45 0.000558107 0.000496726 0.892076 0.911551 0.911991 1.20873
50 0.000573536 0.000503979 0.892377 0.914668 0.890655 1.19396
55 0.000587156 0.000509312 0.892583 0.917476 0.872005 1.18319
60 0.000599247 0.000512983 0.89271 0.920036 0.855579 1.17583
65 0.000610058 0.00051522 0.892768 0.922399 0.840989 1.17137
70 0.000619806 0.000516221 0.892768 0.924607 0.827902 1.16938
75 0.000628679 0.000516157 0.892718 0.926693 0.816042 1.16951
80 0.000636838 0.000515173 0.892625 0.928689 0.805174 1.17146
85 0.000644422 0.000513394 0.892494 0.930618 0.795101 1.17497
90 0.000651548 0.000510923 0.892331 0.932501 0.78566 1.17985
95 0.000658317 0.000507846 0.892139 0.934354 0.776713 1.18592
100 0.000664812 0.000504236 0.891921 0.936192 0.768145 1.19304
```

La cancelación se produce y se imprime antes de la tercera iteración, tal y como se había programado, y demostrando que las prioridades funcionan ya que la cancelación se realiza desde una tarea con más prioridad que el cálculo. Tras eso, la tarea de cálculo se reinicia y debe esperar al siguiente tiempo de muestreo, perdiendo dos iteraciones. Si se fuerza el cálculo tras el reinicio se recuperaría una iteración. Pero lo importante de esta simulación es que el control continúa después de haber realizado la cancelación durante el cálculo de ACADO, que era el objetivo de este apartado.

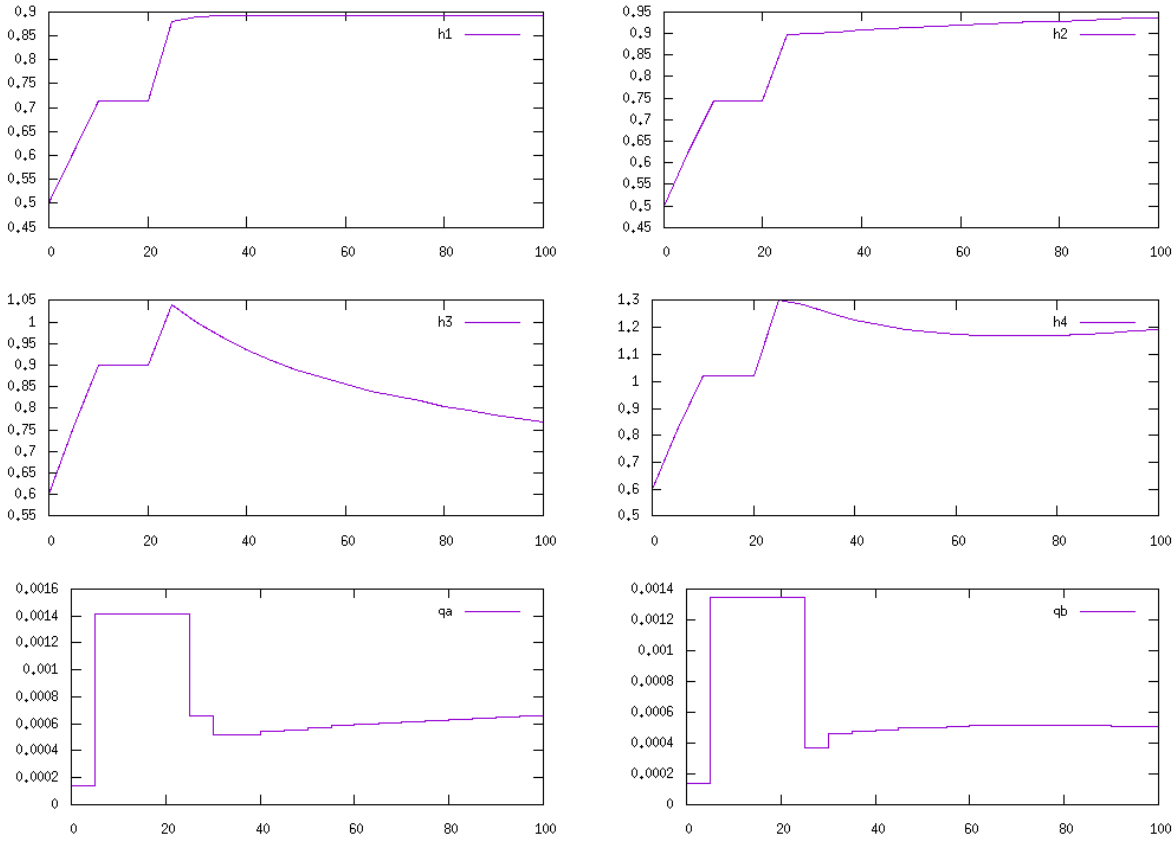

Ilustración 7-6. Control en bucle cerrado en tiempo real con cancelación

Durante la cancelación, se mantienen las actuaciones del sistema constantes. A pesar de ello, el sistema se sigue controlando a las referencias de 0.9 metros, y se realiza mejor en los tanques 1 y 2 porque no hemos variado la matriz de peso con respecto al anterior apartado. El comportamiento del sistema es idéntico al del anterior apartado salvo por la cancelación, que no afecta en absoluto al resultado final, y por tanto la simulación ha sido exitosa.

8 CONCLUSIÓN Y DISCUSIÓN FINAL

En este proyecto se han alcanzado los siguientes objetivos:

- Desarrollo de una librería en C++ que simplifica y facilita la programación de algoritmos de control en tiempo real.
- Integración de la librería con ACADO Toolkit, una herramienta de código libre muy potente para la ingeniería de control, usada en este proyecto para la implementación y simulación de controladores predictivos no lineales en sistemas multivariable.
- Realización total del proyecto bajo software libre, incluyendo el sistema operativo, aplicaciones y paquetes instalados.
- Utilización del sistema embebido PC-104, para el cual se ha dedicado el estudio de este trabajo.

La principal ventaja de haber realizado el proyecto de esta manera es que todo su desarrollo es perfectamente portable a otros sistemas embebidos y es compatible tanto con distribuciones libres de Linux como con QNX. Esto facilitará que el proyecto sea usado como referencia para seguir avanzando en la programación de aplicaciones de control en tiempo real.

Queremos comentar también las principales dificultades por las que hemos pasado para terminar los objetivos propuestos:

- Software libre y compatibilidades entre los diferentes paquetes y librerías internas: Dado que hemos usado bastantes programas que deben funcionar y coordinarse entre ellos, las versiones que utilicemos de cada uno de ellos son críticas. Como estamos usando software libre, este problema se acentúa y la mayoría de las veces acabamos en errores muy difíciles de identificar y solucionar. Aquí queremos agradecer la ayuda de Stack Overflow[38], una comunidad internacional de programadores donde hemos visto solución a muchos de los obstáculos que hemos encontrado durante el desarrollo del proyecto. Es muy recomendado el uso de máquinas virtuales o particiones para instalar y probar diferentes versiones de software y sus compatibilidades.
- Cancelación en tiempo real de ACADO: Integrar la librería de tiempo real con ACADO no ha sido demasiado difícil excepto por la parte de la cancelación. Aunque la cancelación de hilos ha funcionado correctamente en el resto de ejemplos de tiempo real, cuando se hace mientras está ACADO ejecutándose provocaba conflictos de memoria internos por la parte de ACADO y no del tiempo real. Como ya se ha visto, conseguimos solucionar el problema mediante la declaración de objetos de ACADO con memoria dinámica, lo que parece dar una solución funcional en las simulaciones realizadas. No obstante, si se desean utilizar cálculos y funciones más complejas con ACADO, habría que seguir vigilando si la cancelación funciona correctamente.

[38] http://stackoverflow.com/ [Último acceso: julio 2016]

- Disipación de calor del PC-104: Dada su naturaleza de computadora compacta, es muy importante la función del disipador y la temperatura ambiente en la que se trabaja. En épocas de verano se podía llegar al límite de temperatura, lo que provocaba el bloqueo del sistema y posibles errores posteriores. Por este motivo parte del proyecto se ha realizado conectando el disco duro a otros ordenadores comerciales refrigerados, sobre todo para largas sesiones de programación.

En cuanto al trabajo futuro, existen múltiples caminos por los que continuar, entre los cuales destacan:

- Librería de tiempo real: Se pueden añadir nuevas funciones, mejorar las que ya hay, o incluso rediseñar la sintaxis de manera que sea más fácil de comprender. También se podría añadir un tratamiento de errores en los que se avise de los posibles fallos de programación con la librería.
- Integración con ACADO Toolkit: Se ha solucionado correctamente la inclusión de ACADO en una tarea de control y se ha permitido la cancelación externa cuando se producen retrasos en el cálculo. Sin embargo, la cancelación es siempre delicada y pueden surgir contratiempos si se realizan ejemplos muy diferentes al MPC simulado en esta memoria. Además, es posible que la ejecución de múltiples tareas con ACADO no sea trivial. Esta herramienta, dado que es bastante compleja, necesita mirarse a fondo si se van a implementar otros tipos de problemas de control.
- Implementación de lo anterior en nuevos sistemas empotrados más actuales. Hoy día encontramos dispositivos más actualizados y de mayor potencia, por lo que sería interesante adaptar los diferentes capítulos de este proyecto. Lo que dará más problemas será siempre la parte de instalación del software necesario que corresponde con el capítulo 4 de la memoria, ya que las versiones admitidas y compatibles varían, lo cual una es de las desventajas de trabajar con software libre.
- Estudio y uso de la tarjeta de adquisición de datos para implementar los controladores diseñados en plantas reales. Sería el trabajo que mejor complementaría a este, ya se ha tenido que poner el punto y final en las simulaciones debido al tiempo limitado que disponen los proyectos de fin de grado.

REFERENCIAS

[1] «Ubuntu,» [En línea]. Available: http://www.ubuntu.com/. [Último acceso: mayo 2016].

[2] «Xenomai,» [En línea]. Available: https://xenomai.org/. [Último acceso: mayo 2016].

[3] «ACADO Toolkit,» [En línea]. Available: http://acado.github.io/. [Último acceso: junio 2016].

[4] «Ubuntu 12.10 (Quantal Quetzal),» [En línea]. Available: http://old-releases.ubuntu.com/releases/12.10/. [Último acceso: mayo 2016].

[5] «Cmake,» [En línea]. Available: https://cmake.org/. [Último acceso: mayo 2016].

[6] «GCC, GNU Compiler Collection,» [En línea]. Available: https://gcc.gnu.org/. [Último acceso: mayo 2016].

[7] «Gnuplot,» [En línea]. Available: http://www.gnuplot.info/. [Último acceso: junio 2016].

[8] «Graphviz,» [En línea]. Available: http://www.graphviz.org/. [Último acceso: junio 2016].

[9] «LibGD,» [En línea]. Available: http://libgd.github.io/. [Último acceso: junio 2016].

[10] I. Alvarado, D. Limon, A. Ferramosca, T. Alamo and E.F. Camacho, «Robust tubed-based MPC for tracking applied to the quadruple-tank process,» de *2008 Multi-Conference on Systems and Control*, San Antonio, Texas, USA, September 3-5, 2008.

[11] B. Houska and H.J. Ferreau and M. Diehl, «ACADO Toolkit -- An Open Source Framework for Automatic Control and Dynamic Optimization,» *Optimal Control Applications and Methods,* vol. 32, nº 3, pp. 298--312, 2011.

[12] B. Houska and H.J. Ferreau and M. Diehl, «An Auto-Generated Real-Time Iteration Algorithm for Nonlinear MPC in the Microsecond Range,» *Automatica,* vol. 47, nº 10, pp. 2279--2285, 2011.

[13] B. Houska and H.J. Ferreau and M. Vukov and R. Quirynen, «ACADO Toolkit User's Manual,» 2009--2013. [En línea]. Available: http://www.acadotoolkit.org.