eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
ZIENTZIA ETA TEKNOLOGIA FAKULTATEA
FACULTAD DE CIENCIA Y TECNOLOGÍA

Gradu Amaierako Lana / Trabajo Fin de Grado
Ingenieritza Elektronikoko Gradua / Grado en Ingeniería Electrónica

Integración de un sistema de control en una red EPICS: diseño y desarrollo del EPICS IOC

Egilea/Autor/a:
Beñat Alberdi Esuain
Zuzendaria/Director/a:
Josu Jugo García

Leioa, 2017ko ekainaren 22a /Leioa, 22 de junio de 2017

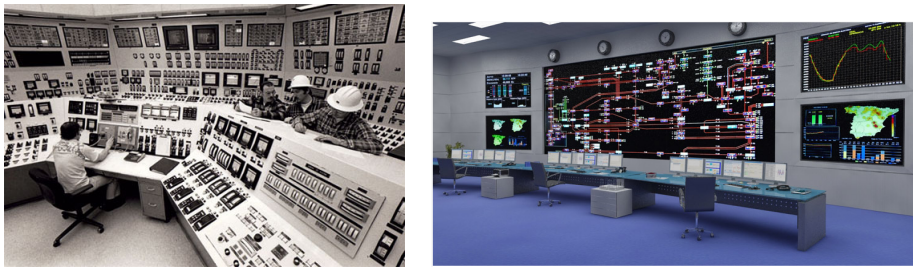
Índice general

Capítulo 1

Introducción

La ingeniería de control es una de las ramas más dinámicas y importantes del mundo tecnológico actual y abarca un gran abanico de posibilidades y aplicaciones. Consiste en la automatización y el control automático de sistemas complejos, sin intervención humana directa. Partiendo de modelos matemáticos, se desarrollan e implementan sistemas de control para conseguir que un sistema se comporte lo más parecido posible a un modelo predeterminado. Desde la medicina hasta la astronáutica o desde los sistemas más pequeños hasta los grandes aceleradores de partículas, en todos los ámbitos tecnológicos la aplicación de teorías de control cobra una importancia vital. Por ello, son numerosas las empresas en la industria y departamentos académicos dedicados al desarrollo de sistemas de control.

El control de procesos de las grandes plantas industriales ha ido evolucionando con el paso de los años. En un principio, el control se realizaba mediante paneles discretos y dispersos. Esto requería una gran cantidad de recursos humanos. No existía una visión global de toda la planta. El siguiente paso lógico era el de transportar todos los componentes al un lugar vigilado permanentemente, una sala de control. Efectivamente, esto significó la centralización de todos los paneles. Así se facilitaba la supervisión de los procesos y requería una menor cantidad de trabajo. No obstante, a pesar de concentrar todos los componentes en el mismo sitio, este método resultaba inflexible al tener cada *loop* de control su propio *hardware* y por ello necesitar un completo rediseño para realizar cambios. Con la llegada de los procesadores electrónicos, canales de alta velocidad e interfaces gráficas se hizo patente la posibilidad de reemplazar los controladores individuales por algoritmos computacionales, alojados en una red de bastidores *Input/Output* con sus propios controladores de procesos. Estos pueden ser distribuidos a lo largo de la planta industrial y comunicarse con el monitor gráfico en la sala de control. Así nació el control distribuido.



(a) Sala de control pre-DCS. (b) Sala de control post-DCS.

Figura 1.1: Control no distribuido y distribuido.

La introducción del control distribuido ha permitido la interconexión y reconfiguración flexible de los sistemas de control, por ejemplo los *interlocks* [14]. La integración de sistemas complejos de gestión de alarmas también se ha hecho posible y se ha eliminado la necesidad de usar grabadoras físicas de datos. En general, el control distribuido ha posibilitado altos niveles de supervisión de los estados de la planta y niveles de producción [21]. En los grandes sistemas de control se acuño el término *Distributed Control Sistem* (DCS) para aquellos sistemas que fuesen modulares, con redes de alta velocidad y un conjunto completo de interfaces gráficas y bastidores de control perfectamente integrados. En la actualidad hay otro punto de vista complementario al DCS, el SCADA, el cual en vez de centrarse en la capa de supervisión y control automáticos que se conectan a nivel de planta, se centra en el sistema de monitorización y supervisión por parte del usuario. Dicho de otro modo, el DCS se centra en construir el sistema de control empezando desde la planta hacia el operario mientras el SCADA se centra en construirlo al reves, empezando desde el sistema de monitorización. Algunas de las aplicaciones típicas de los DCS son por ejemplo el control de plantas químicas, de procesamiento de comida o de reactores nucleares.

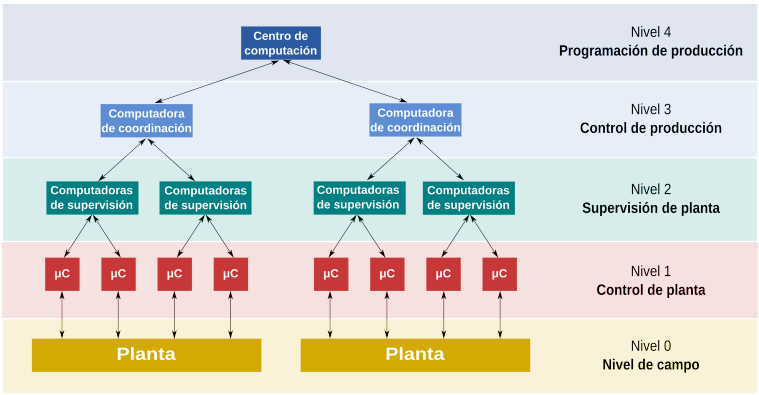


Figura 1.2: Esquema de niveles típicos en un sistema de control.

Entre los sistemas DCS usados actualmente en grandes instalaciones científicas a lo largo del mundo hay dos que destacan por encima de las demás: EPICS y TANGO. Ambos ofrecen las ventajas del control distribuido y también pueden ser integrados con funcionalidades SCADA, pero las características de cada uno los hacen aptos para aplicaciones específicas distintas. Este proyecto se ha llevado a cabo usando EPICS, el cual destaca por su amplio uso en aceleradores de partículas, telescopios o grandes experimentos científicos (tales como BESSY II, IFMIF, FNAL, etc...) para adquisición de datos y control de supervisión [4].

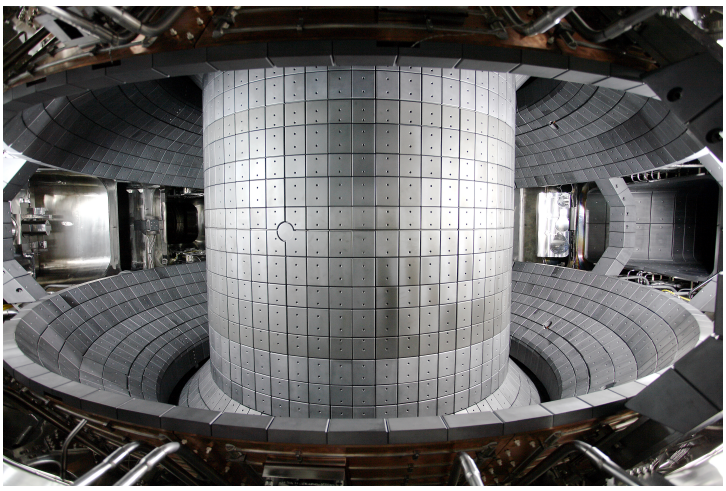


Figura 1.3: Ejemplo del uso de EPICS, experimento KSTAR[5].

1.1. Objetivos

El primer objetivo ha sido entender como funciona y trabaja un sistema EPICS, dado que es esencial para poder desarrollar el proyecto. Una vez entendido su funcionamiento, el objetivo principal de este trabajo ha sido construir un sistema de control basado en EPICS para una tarjeta MicroZed integrada en una I/O Carrier. El sistema de control debe ofrecer la posibilidad controlar todas las entradas y salidas de de la tarjeta, siendo estas salidas y entradas digitales o analógicas. El uso de EPICS permite que este sistema de control pase después a ser parte de un sistema de control distribuido más grande, además de ofrecer un fácil acceso al IOC (*Input Output Controller*) y a sus variables desde cualquier ordenador conectado a la misma red que la MicroZed.

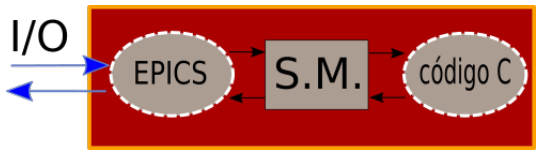


Figura 1.4: Esquema del estado previo del proyecto.

A la hora de comenzar el trabajo ya existía un sistema de control en la MicroZed, pero estaba implementado en dos partes (Figura 1.4). Por un lado, EPICS se encargaba de administrar los datos de entrada y salida de la tarjeta. Por otro lado, el procesamiento de datos y el control se realizaban usando un *loop* independiente de control desarrollado en C que no era integrable en el sistema EPICS. La transferencia de datos entre EPICS y el controlador se hacía mediante una *shared memory*, donde EPICS escribía los datos de entrada para que posteriormente el controlador los leyese. Después de leer los datos el controlador los procesaba. En el sentido inverso, el controlador escribía los datos en la *shared memory* y EPICS se encargaba de leerlos y llevarlos a las salidas correspondientes. El cometido principal del proyecto ha sido llegar a prescindir de la *shared memory* y del controlador independiente, y usar solo EPICS para realizar todo el proceso de adquisición, procesamiento y escritura de datos.

La estructura de la memoria que viene a continuación está dividida en tres capítulos principales además del de las conclusiones. El capítulo que viene a continuación (2: EPICS) trata de dar una breve introducción teórica al sistema de control EPICS, su estructura y sus utilidades. Después, en el proximo capítulo (3: *Hardware*) se da a conocer cual ha sido el *hardware* usado a lo largo de este proyecto, detallando sus características principales y el montaje correspondiente. El capítulo siguiente (4: Desarrollo y implementación del EPICS IOC) es donde se explica cual ha sido el trabajo práctico realizado en el proyecto y que métodos se han utilizado. Por último, en las conclusiones se dan a conocer cuales han sido las valoraciones obtenidas y la proyección al futuro de este trabajo.

Capítulo 2

EPICS

El acrónimo EPICS proviene de *Experimental Physics and Industrial Control System*. Su origen data del año 1988, cuando sus funcionalidades fueron por primera vez definidas en un congreso en *Los Alamos National Laboratory*. En este congreso se debatía cual era el sistema de control más apropiado para construir el GTA (*Ground Test Accelerator*) [17]. En la reunión se definió cuales eran los requisitos que debía cumplir en el futuro un sistema de control apropiado para cualquier experimento de ese tipo. Así nació EPICS.

EPICS es un entorno de *software* usado para desarrollar y implementar sistemas de control distribuidos. El entorno está diseñado para desarrollar sistemas de control que habitualmente contienen una gran red de computadores proporcionando control y *feedback*. Con su creación se buscaba un sistema de control homogéneo para el desarrollador [2], escalable y robusto. EPICS proporciona herramientas de control y obtención de datos para la comunidad de la física experimental. En su desarrollo han participado varias entidades de investigación, principalmente: *Los Alamos National Laboratory*, *Advanced Photon Source* en *Argonne National Laboratory* y *The Computer Systems Group* en el *Lawrence Berkeley National Laboratory*. Hoy en día más de 70 universidades y instituciones lo usan y hacen sus aportaciones al perfeccionamiento del sistema [1].

2.1. Estructura

Se puede afirmar que EPICS esta compuesta por capas: tres principales capas de *software* y otras tres físicas [2]. La capa física frontal, conocida como *Input Output Controller* (IOC), está construida con cajas de *hardware*, tarjetas CPU y tarjetas I/O. Es aquella que está en contacto directo con las magnitudes físicas a controlar. La comunicación entre distintos dispositivos de la capa física se da mediante la capa de red, que se compone de cualquier combinación de medios de transporte (como Ethernet, FDDI, ATM) y repetidores o conmutadores que

soporten el protocolo de internet TCP/IP junto con alguna forma de *broadcast* o *multicast*. Estos forman la segunda capa. Por último están las estaciones del operario conocidas como OPI. Son aquellas estaciones cuyo diseño les permite comunicarse con los distintos IOC de la red y supervisarlos.

Si observamos la estructura física de un sistema de control EPICS dividido físicamente en las distintas estaciones que lo forman, veremos que tiene la siguiente forma:

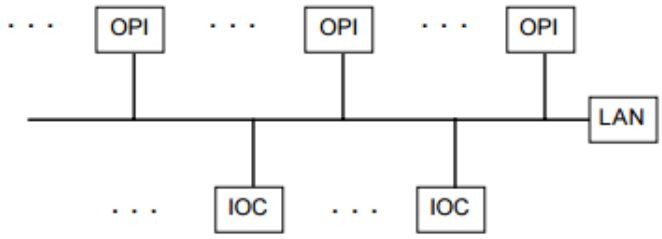


Figura 2.1: Estructura de un sistema EPICS.

Los componentes básicos de la Figura 2.1 y su relación con las capas físicas mencionadas previamente son las siguientes [1]:

- **IOC** (*Input Output Controller*): Es cualquier plataforma que pueda soportar los EPICS *database* junto a alguno de los *softwares* descritos en el manual de EPICS [1]. El IOC agrupa la capa física frontal y la tercera capa de *software*: el servidor (se verá a continuación). Se encarga de recopilar datos y realizar el control. Se comunica con el cliente mediante *Channel Access*.
- **OPI** (*Operator Interface*): Es la interfaz del usuario. Es una estación de trabajo en la cual se pueden usar las herramientas de EPICS. Cada estación OPI sería el equivalente a un cliente del sistema EPICS.
- **LAN** (*Local Área Network*): Red de área local. Formada por la red de comunicación que permite la comunicación entre el IOC y el OPI. Esta red puede estar compuesta por cualquier combinación entre distintos medios de transporte de datos. Posteriormente se verá que EPICS proporciona una herramienta de *software*, el *Channel Access*, para poder realizar comunicaciones transparentes de red entre los IOC y los OPI.

Por otro lado están las capas de *software* [4]. EPICS usa técnicas de cliente/servidor para comunicarse entre varios ordenadores. Un grupo de ordenadores (el servidor) recoge la información experimental y de control necesaria en tiempo real, haciendo uso de los distintos instrumentos que contenga. Esta información se pasa después a otro grupo de ordenadores (clientes) usando el protocolo de red *Channel Access* (CA). CA es un protocolo de red de banda ancha, adecuado

para soportar aplicaciones en tiempo real como los experimentos científicos.

La primera capa es la del cliente, que representa el nivel jerárquico más alto y está típicamente compuesto por monitores de control, paneles de alarmas y archivos de datos. Muchos de estos clientes están configurados mediante simples archivos de texto o editores gráficos. Los clientes de EPICS tienen un rendimiento realmente bueno, por ejemplo los monitores de operación con más de 1000 objetos pueden cargarse en menos de un segundo o pueden actualizar más de 500 objetos por segundo.

La segunda capa de *software* corresponde al *Channel Access*, considerado la espina dorsal de EPICS. Está se encarga de comunicar el cliente con el servidor y esconderles a ambos todos los detalles del protocolo TCP/IP. El *Channel Access* también crea mediante *software* una especie de barrera de independencia entre el cliente y el servidor, permitiéndoles usar distintos procesadores o versiones de EPICS sin obstruirse mutuamente. El diseño del CA proporciona un desempeño que permite acceder a más de 10000 PV por segundo, siendo los PV las unidades mínimas con las que trabaja un sistema EPICS (apartado 2.3.3). Esta capa es también responsable de ofrecer al cliente la posibilidad de actualizar los PV solo cuando estos cambien, algo que se verá más adelante en el apartado 2.3.5. Además, en el apartado 2.3.7 se analizan algunas propiedades más del *Channel Access*.

Finalmente, está la capa del servidor. Esta capa es la encargada de actuar en cada dispositivo físico del nivel más bajo constituyente del sistema de control (el equivalente al nivel de campo de la Figura 1.2). El servidor colabora con todos los clientes *Channel Access* para proporcionar sincronización y actualización de datos.

Se puede observar que cada capa de *software* corresponde en mayor o menor medida a distintos componentes físicos del sistema: la capa del cliente al los OPI, el *Channel Access* a la red de comunicación y el servidor a los IOC.

2.2. Atributos básicos

Se ha visto cuales son las capas sobre las que se construye un sistema de control EPICS y cuales son algunas de sus cualidades. En este apartado, se analizan cuales son las propiedades distintivas del sistema completo [1]. La característica principal de EPICS es que es un sistema completamente distribuido que no requiere un dispositivo central. Por esto logra una escalabilidad y robustez (no existe un punto crítico) muy adecuados. También ofrece funcionalidades SCADA, con el que se integra perfectamente mediante un grupo de herramientas de *software* disponibles en el mismo EPICS. Resumiendo, sus principales características son:

- Esta basado en herramientas: EPICS proporciona diferentes herramientas para crear sistemas de control. Esto minimiza la necesidad de crear códigos personalizados y asegura una interfaz de usuario uniforme.

- Es un sistema distribuido: EPICS es un sistema de control distribuido. Soporta un número arbitrario de IOC y OPI. Un sistema distribuido escala en tamaño sin ningún problema. Si algún IOC se saturase, sus funciones podrían dividirse entre otros IOC. Mediante este método se evita la existencia de cuellos de botella o puntos críticos. En lugar de ejecutar las aplicaciones en un solo usuario, se pueden distribuir entre varios OPI.

- Dirigido por eventos: Los componentes de *software* de EPICS están diseñados para ser dirigidos por eventos tanto como sea posible. Esto significa que en lugar de tener que inquirir al IOC por cualquier cambio que haya ocurrido, un cliente puede pedir que se le notifique cuando ocurra un cambio. Este diseño permite el mejor aprovechamiento de los recursos, así como tiempos de respuesta más cortos.

- Alto rendimiento: Un IOC con procesador 68040 puede procesar más de 10000 *records* en un segundo, incluyendo generaciones de eventos *Channel Access*. Para esto, cada una de las distintas capas de *software* (cliente, *Channel Access* y servidor) están optimizadas y su rendimiento maximizado.

2.3. IOC (*Input Output Controller*)

Un sistema de control EPICS debe tener uno o más IOC. Este proyecto se ha enfocado mayormente en construir un IOC para después supervisarlo desde un OPI. Por ello, es necesario analizar más profundamente cual es la composición interna de una de estas estructuras. Hemos visto que un IOC es la capa frontal de la estructura física de EPICS, cuyo principal cometido es el control de los PV del sistema. Además, dependiendo de la aplicación, también recopila datos experimentales y, después de integrarlos en los PV, los gestiona.

2.3.1. Componentes

A pesar de ser solo una de las capas físicas de EPICS, es muy compleja estructuralmente. Posee un núcleo y varias funcionalidades periféricas, que en su conjunto forman el IOC. Un IOC contiene todos o varios de los componentes *software* de la Figura 2.2:

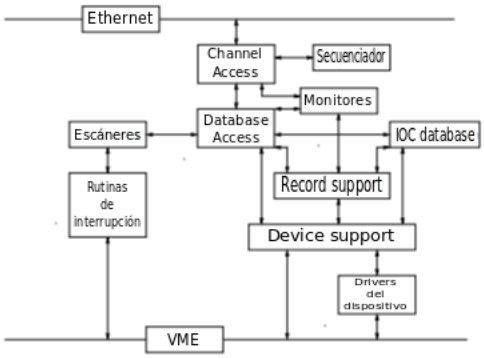


Figura 2.2: Componentes de un IOC.

La Figura 2.2 es muy útil para entender el funcionamiento de un IOC. A continuación, se explica por encima el propósito de cada uno de estos componentes [1].

- IOC *Database*: El *database* es la estructura de datos que define la funcionalidad del IOC. Cada IOC contiene uno o más *databases* que le indican como actuar. Está compuesta por una colección de *records* (apartado 2.3.2) de varios tipos.
- *Database Access*: Contiene las rutinas necesarias para acceder a la *database*. A excepción del *record* y *device support*, todos los accesos al *database* se realizan vía *database access*.
- Escáneres: Es el mecanismo para decidir cuando han de procesarse los *records*.
- *Record Support*: Rutinas de apoyo de los *records*. Cada tipo de *record* tiene su grupo de rutinas de apoyo asociadas.
- *Device Support*: Enlace entre *records* y dispositivos. Cada tipo de *record* puede tener asociados uno o varios grupos de rutinas de apoyo al dispositivo.
- *Drivers*: Los *device drivers* son rutinas para acceder a los dispositivos externos. Un *device driver* puede tener una rutina de interrupción asociada.
- *Channel Access*: Este componente no es exclusivo del IOC. Se ha visto previamente que es la capa de *software* que se encarga de las comunicaciones. Por ello se considera la interfaz entre el mundo externo y el IOC. Proporciona una interfaz independiente a la red para acceder a la base de datos contenida en el IOC.
- Monitores: Los monitores del *database* proporcionan un mecanismo para anunciar los cambios en el *database*. Se invocan cuando el valor de un campo de un *record* perteneciente al *database* cambia.

- Secuenciador: Una máquina de estados que sirve para automatizar procesos del sistema de control mediante la definición de distintos estados y reglas para los cambios de estado.

Entre estos componentes hay algunos que son más importantes que otros, los que juegan un papel esencial en cualquier IOC. Por ello, a continuación se explican más detenidamente cuales son las características principales de varios de estos componentes. Aun así, para mantener un orden y entender las cosas poco a poco, antes hace falta definir con más detalle lo que son los *records* y los PV en EPICS.

2.3.2. *Records*

Los *records* son estructuras de datos que tienen: un nombre, propiedades controlables (campos) y un comportamiento definido por la clase de *record* a la que pertenecen y los campos que contienen. Opcionalmente también pueden tener un *hardware* I/O asociado [20].

EPICS distingue las clases de *record* por la naturaleza de la variable física con la que se relacionan. EPICS soporta una gran cantidad de tipos de *records*, existen cerca de 50 tipos (y se pueden extender) como por ejemplo *analog input* y *output*; *binary input* y *output*; histogramas; realización de cálculos; y otros cometidos.

Cada tipo de record tiene una cantidad fija de campos. Algunos campos son comunes para todos los tipos de *record* pero otros en cambio son específicos para un tipo de *record* en particular. Todos los *records* tienen un nombre único y cada campo tiene un nombre de campo. La primera línea en la definición de cualquier *record* debe contener su nombre, el cual debe ser único en todo IOC conectado a la misma sub-red TCP/IP.

Los *records* son estructuras activas. Esto significa que pueden hacer varias cosas: obtener datos desde otros *records* o desde el *hardware*, realizar cálculos, emitir alarmas dependientes de sus límites, esperar a interrupciones de *hardware*, etc... Los *records* no cambian a no ser que sean procesados [18].

2.3.3. PV (*Process Variable*)

Las variables de proceso (*Process Variable* o PV) son estructuras de datos relacionadas con campos individuales de los *records*. Para identificarlos, a cada PV se le asigna un nombre único en todo el sistema, el cual es conocido como *id*. Un solo *record* puede contener varios PV asociados. Puede ser que un *record* y un PV asociado a él tengan el mismo nombre [1, 20].

En esta estructura de datos, no solo se guarda la información directamente relacionada a un campo de *record*, sino que también varios atributos interesantes que están relacionados con él: última hora de procesamiento, unidades, etc...

Todos estos datos proporcionan toda la información necesaria para evaluar el estado del PV. Los PV son los componentes más interesantes del IOC, siendo el principal objetivo de este último su monitorización y manipulación. Como se ha mencionado en el apartado 2.1 los PV son la unidad mínima con la que trabaja el IOC.

2.3.4. IOC *Database*

Una base de datos es una colección de información organizada de forma que un programa de ordenador pueda seleccionar rápidamente los fragmentos de datos que necesite. En el caso de EPICS, el *database* es el lugar dónde se definen los *records*, es decir, es donde se decide que atributos se le asignan a cada uno y que funcionalidad tiene. Así, la base de datos de un IOC guarda la definición de los tipos de *records* necesarios para hacerlo funcionar. Dicho de otro modo, el *database*, junto a las estructuras que describen su contenido, se considera el corazón del IOC por definir sus funcionalidades [1, 18].

EPICS proporciona estructuras de datos para que el *database* pueda ser accedido eficientemente. La mayoría de los demás componentes de *software* acceden al *database* mediante rutinas de acceso, por lo que no tienen que saber cual es la estructura del *database*.

2.3.5. *Database scanning*

El escaneo, como se ha mencionado previamente, es el mecanismo que decide cuando ha de procesarse un *record*. Entre sus cometidos está el de actualizar el valor de distintos campos de un *record*. Por ejemplo, si se imagina que tenemos un *record* del tipo *ai* (*analog input*) y se quiere medir el valor de una entrada analógica cada T tiempo o cada evento X, se debe procesar el *record* enlazado a esa entrada cada T tiempo o cuando el evento X ocurra. El escaneo es lo que permite fijar esto [1].

Hay cinco tipos de escaneos posibles:

1. Periódico: El *record* se procesa periódicamente. EPICS soporta varios intervalos de tiempo para procesar el *record* periódicamente, por ejemplo: 0.1 segundos, 10 segundos, etc...

2. Evento: El escaneo se basa en el despliegue de un evento por parte de algún componente de *software*.

3. Evento I/O: Este sistema permite procesar el *record* cada vez que se reciba una interrupción externa. Para ello, se debe tener alguna rutina de interrupción de dispositivo que permita aceptar y atrapar (o provocar) estas interrupciones.

4. Pasivo: El *record* se procesa solo cuando algún *record* relacionado se procese o como resultado de cambios externos, tales como ordenes del *Channel Access*.

5. Escaneo unitario: El sistema de escaneo proporciona una rutina que permite escanear un *record* una única vez.

2.3.6. *Record support* y *device support*

El acceso al *database* no requiere conocimientos relacionados a los tipos de *records*, porque cada tipo de *record* tiene unos módulos de *record support* específicos. Del mismo modo, estos módulos *record support* no tienen ningún conocimiento sobre el dispositivo, porque hay módulos de *device support* específicos para cada dispositivo que se encargan de esto. Simplificándolo, se puede entender esto como una cadena, donde cada eslabón de la cadena solamente tiene conocimiento de los eslabones que están en contacto directo con él, nunca de los que están a más de una posición de distancia. Esto puede apreciarse en la Figura 2.2.

El *device support* es la interfaz entre campos específicos del *database* (*records*) o el *record support* mismo y los *drivers* del *hardware*, o el *hardware* mismo. En otras palabras, es el medio de proveer una funcionalidad especifica de acceso al dispositivo. Esto se hace proporcionando varias funciones a las que la capa del *record support* llama cuando es apropiado. Las funciones que deben ser proporcionadas dependen de los tipos de *record* que se quieran soportar.

Como se ha mencionado, el propósito del *device support* es esconder las características especificas del *hardware* al *record support*. Por lo tanto, se puede desarrollar el *device support* para un nuevo *hardware* sin cambiar las rutinas de *record support*. El número de módulos de *device support* que cada tipo de *record* puede tener es arbitrario. Esto significa que un solo tipo de *record* puede tener varios módulos de *device support* distintos asociados. Por ejemplo, para un tipo de *record* concreto pueden definirse dos módulos de *device support*, uno síncrono y otro asíncrono (la diferencia entre ellos se explica en el apartado 4.2.2), y cuando se cree el *record* se elige cual de ellos usar en función del *hardware* al que se quiera acceder.

En caso de que el acceso al dispositivo sea más complicado de lo que se pueda manejar con el *device support*, puede desarrollarse un *driver* del dispositivo. En este trabajo esto no ha sido necesario y gran parte del mismo ha consistido en desarrollar los módulos de *device support* necesarios para el *hardware*.

2.3.7. *Channel Access*

El *Channel Access* proporciona un acceso transparente mediante la red al *database* del IOC [1]. Usa un modelo de servidor/cliente, donde los IOC del

sistema de control actúan como un servidor. Cualquier OPI del sistema puede actuar como cliente, no hay limitaciones en cuanto al número. Además, los distintos IOC del sistema también podrán acceder como clientes a los servidores, siendo al mismo tiempo servidores ellos mismos.

Los servicios básicos que el *Channel Access* proporciona a un cliente son los siguientes:

- *Search*: Localiza en los IOC los PV que se desean y establece comunicación con cada una de ellos.
- *Get*: Obtiene la información de un PV, más la información adicional que se requiera.
- *Put*: Cambia el valor de un PV.

Toda información que transcurre por el *Channel Access* lleva consigo información adicional que puede resultar muy útil a la hora de asegurar la veracidad de la información. Por ejemplo, toda información lleva un sello de tiempo, lo que posibilita a cualquier cliente obtener datos y asegurar su orden cronológico. Generalmente, a la hora de obtener la información de un PV el *Channel Access* permite requerir información adicional sobre sus distintos atributos, entro ellos: estado de alarma, precisión, la hora en la cual se procesó por última vez, etc...

2.3.8. Secuenciador

El secuenciador es la herramienta de *software* que permite construir una máquina de estados basado en un sistema de control construido en EPICS. Para ello se usa el SNL (*State Notation Lenguage*), un lenguaje de dominio especifico cuya construcción depende de la base de EPICS y con el cual se integra perfectamente [22].

Con el SNL se estructura el sistema de control en un grupo de máquinas de estado que corren al mismo tiempo. Cada una de estas máquinas se declaran dando una lista de sus estados, y definiendo bajo que condición un estado pasa a otro y lo que debe hacer el programa en ese caso. De este modo, mediante el secuenciador, se pueden construir relaciones complejas entre distintas PV, y hacerlas dependientes del estado de otras PV. Para entender el funcionamiento del secuenciador y su integración en EPICS lo más apropiado es analizar un ejemplo sencillo.

Ejemplo del secuenciador:

Se va a suponer que hay un IOC EPICS muy simple construido con dos PV. El primero de ellos está conectado a un sensor que mide la temperatura de la atmósfera («TEMP») y el segundo («PEL») a un LED que indica si la temperatura es superior a 40°C, lo que podría ser dañino para algunas cosechas. Es fácil

ver que el primer PV pertenece al campo del valor de un *record* de naturaleza analógica y además es un proceso de lectura, mientras que el segundo pertenece a uno digital y de escritura. Para conectar estas dos variables de proceso entre sí se construirá un sencillo secuenciador usando SNL. Con él se crea una máquina de dos estados dependiendo de la temperatura de la atmósfera: LED encendido o LED apagado.

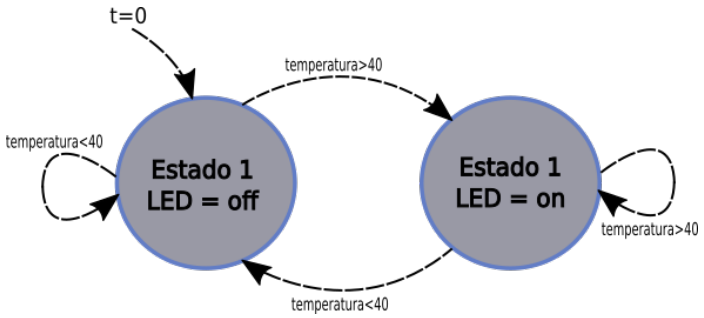


Figura 2.3: Esquema de la máquina de estados.

```
program control_temperatura

float temperatura;
assign temperatura to "TEMP";
monitor temperatura;

short led;
assign led to "PEL";

ss controlar_temp {
  state led_off {
    when (temperatura > 40.0) {
      /* Encender el LED */
      led = TRUE;
      pvPut(led);
    } state led_on
  }

  state led_on {
    when (temperatura < 40.0) {
      /* Apagar el LED  */
      led = FALSE;
      pvPut(led);
    } state led_off
  }
}
```

Arriba puede verse el código correspondiente al programa del control de temperatura. El SNL interactúa con el mundo exterior usando variables que se conectan a los PV del sistema EPICS. Por ello, primero se asigna una variable a cada PV («temperatura» para el PV «TEMP» y «led» para el PV «PEL») y después se le ordena al programa que monitorice la temperatura ambiental, lo que actualiza el valor de la variable «temperatura» cada vez que el valor de «TEMP» cambie en el *database* de EPICS. Posteriormente, se definen los dos estados del sistema y las reglas correspondientes a cada uno, donde se especifica en que condiciones el sistema debe cambiar de estado. Para materializar estos cambios y trasladarlos al mundo real se usan las funciones para los PV previamente construidas como **pvPut(),** el cual transmite el valor de una variable al PV al que está asignado.

2.4. OPI, la interfaz del usuario

Se ha mencionado antes que el OPI es una de las capas físicas de un sistema EPICS, más concretamente es la interfaz que el usuario tiene para interactuar con el sistema EPICS [1]. Para construir dicha interfaz, se han desarrollado varias herramientas que hacen uso del *Channel Access*. El secuenciador, mencionado en el apartado previo, es un caso especial de un OPI. Pero hay muchos más, entre ellos: CSS, EDM, MEDM, ALH,etc...

En este trabajo se ha usado usado el CSS (*Control System Studio*) para montar una interfaz gráfica con la que el cliente puede controlar todos los PV del IOC. CSS es una colección de herramientas para monitorizar y operar sistemas de control a gran escala, por ejemplo los aceleradores de partículas. Es producto de la colaboración entre varios laboratorios y universidades. Se basa en tecnología de *software* actual (Java, Eclipse), con un énfasis especial en la interoperabilidad [7].

Una de las principales virtudes de CSS es que resuelve el problema de la transmisión y recepción de datos haciendo un uso muy eficaz del *Channel Access*. El usuario (o incluso el desarrollador) no necesita saber nada de como se da la transmisión de datos entre el cliente y el servidor, basta con saber el nombre del PV que desea monitorizarse o actualizarse y mediante herramientas gráficas CSS nos permite tener un absoluto control sobre esta PV.

Ejemplo de CSS:

Para entender mejor el funcionamiento de CSS y ver como luce la interfaz se usará el mismo ejemplo que en el caso del secuenciador. Para continuar la expansión de la sencilla estación meteorológica a los PV previos se les añadirá otro que mide la velocidad del viento, llamado «VEL». En este caso, dado que no hay una máquina de estados, la misión de encender el LED que alerta de

temperaturas demasiado altas corresponderá al operador del sistema de control, que deberá hacerlo manualmente mediante CSS. A la hora de construir esta interfaz, lo único que hace falta es saber cuales son los nombres de los PV en nuestro sistema, el *Channel Access* se encarga de establecer comunicación entre cliente y estos.

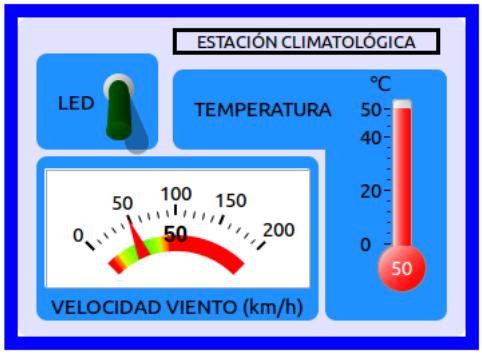
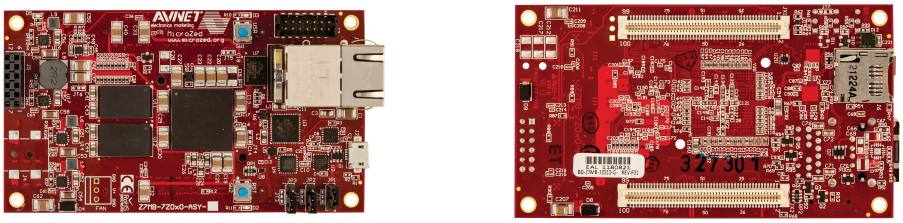


Figura 2.4: Ejemplo de una interfaz CSS. Una sencilla estación meteorológica.

Capítulo 3

Descripción del *hardware*

Este proyecto se ha llevado a cabo usando una tarjeta MicroZed de ZedBoard. La MicroZed es un SoC (*System on Chip*) basado en Xilinx Zynq®-7000 que puede ser usado como tarjeta de evaluación para experimentos básicos de SoC, o combinado con otra tarjeta *carrier* como un sistema embebido SoM (*System on Module*). La tarjeta MicroZed contiene dos bancos de conexiones I/O que están inactivas si la tarjeta se usa sola. En el caso de combinarla con otra tarjeta, los pines I/O de la MicroZed pasan a ser programables. La MicroZed tiene una arquitectura de computadora tipo ARM. La arquitectura ARM es conveniente dado que los computadores con arquitectura ARM suelen requerir menos transistores que los que tienen una arquitectura más compleja (por ejemplo el CISC), lo que reduce el costo energético, aumenta la disipación de calor y reduce el costo monetario. Esto la hace beneficiosa para los dispositivos ligeros y portátiles como es el caso de la MicroZed [11].



(a) Desde arriba. (b) Desde abajo.

Figura 3.1: Vista de la Microzed.

En este caso se ha combinado la MicroZed con una tarjeta I/O Carrier. Esto permite activar y acceder fácilmente a los 100 pins de entrada y salida de la MicroZed más los propios de la tarjeta I/O Carrier (los botones, LED, *switches*,...). Esta tarjeta también da a la MicroZed la energía necesaria para

funcionar, proporcionando los 5V que requiere el núcleo y voltajes configurables por el usuario para las salidas de lógica programable [12].

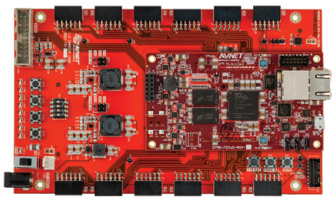


Figura 3.2: Vista de la I/O Carrier con la MicroZed.

De este modo, mediante programación y haciendo uso del sistema de control EPICS, se pueden configurar los puertos de entrada y salida de la I/O Carrier. Cada puerto puede ser configurado para dar una salida binaria o leer una entrada del mismo tipo. Para este proyecto, dada su naturaleza, hacia falta que varios puertos tuviesen la configuración de entrada y salida analógicas. Para ello, en uno de los bloques de puertos de la I/O Carrier se ha adjuntado una tarjeta MAX11300 con funcionalidades de ADC y DAC.

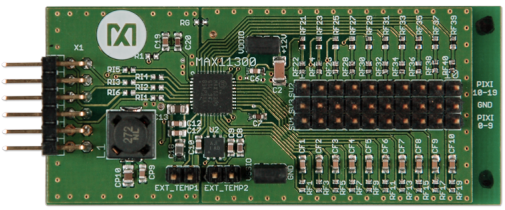


Figura 3.3: MAX11300 PIXI.

El MAX 11300 contiene en un solo circuito integrado un chip PIXI, un conversor de analógico a digital de 12 bits y otro conversor de digital a analógico de otros 12 bits. Para la comunicación con el mundo exterior dispone de 20 puertos individuales configurables, los cuales admiten distintos rangos de voltaje dependiendo de la configuración. Cada puerto hace uso de uno de los conversores, dependiendo de si es de entrada o salida [13]. Para la comunicación con la I/O Carrier, en cambio, la tarjeta contiene un bus que funciona con el protocolo SPI. El protocolo SPI (*Serial Peripheral Interface*) es un protocolo de comunicación en serie y síncrono usado principalmente para la transferencia de datos entre dos circuitos integrados. Se usa un bus con 4 bits para transferir las distintas señales del protocolo.

Así se ha conseguido disponer de varios bloques de entradas y salidas digitales: 20 pins de entradas y salidas analógicas y los distintos I/O de la misma I/O Carrier (botones y LED). Además, para realizar pruebas se dispone de una pequeña tarjeta con varios LED conectable a cualquier bloque de salidas digitales y un potenciómetro para las analógicas.

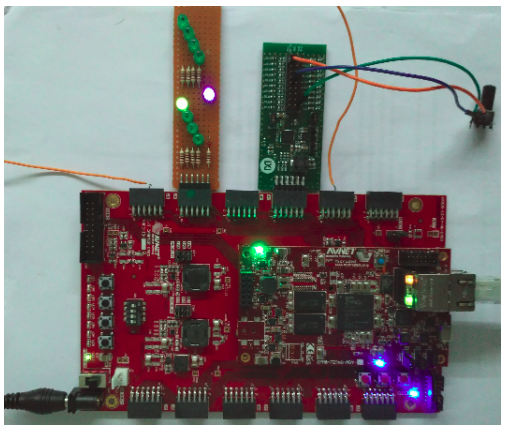


Figura 3.4: Imagen del montaje completo.

3.1. Compilación cruzada

El desarrollo de este proyecto, el cual incluye la escritura, estructuración y compilación de los distintos códigos, se ha llevado a cabo usando un ordenador con una arquitectura x86, pero se ha implementado en un dispositivo (MicroZed) con una arquitectura ARM. Aunque esto pueda parecer sencillo, hay muchos problemas que surgen cuando se implementa una aplicación compilada en una arquitectura concreta en otra distinta. Por ello, ha sido necesario hacer uso de plataformas y sistemas que permiten compilar aplicaciones en una arquitectura para luego usarlas en otra.

El sistema que se ha utilizado en este trabajo ha sido *poky*, la distribución de referencia del proyecto Yocto. El proyecto Yocto busca desarrollar herramientas y procesos para la creación de distribuciones Linux para sistemas embebidos que no dependan de la arquitectura subyacente de los mismos. El uso de *poky*, combinado con las opciones de compilación cruzada que ofrece el mismo EPICS, ha hecho posible compilar el sistema de control en una computadora externa y luego, tras trasladar la aplicación, ejecutarla en la MicroZed.

Capítulo 4

Desarrollo e implementación del EPICS IOC

En este apartado se explica con detalle cual ha sido el camino seguido durante este proyecto. El código desarrollado durante el trabajo no se incluye para no hacer la memoria demasiado pesada, pero sí todo lo necesario para entender como se ha construido y como funciona.

4.1. Proceso de instalación

Al comienzo del proyecto, antes de empezar con el desarrollo de ningún código, es necesario realizar la instalación y configuración de todos los instrumentos que se usan a lo largo del mismo.

Poky Para ello se comienza con la instalación de *poky,* el cual está disponible aquí: *https://www.yoctoproject.org/tools-resources/projects/poky,* y su posterior configuración para adecuarlo a las necesidades de compilación cruzada.

EPICS Al terminar de configurar *poky* se ha descargado e instalado la versión 3.14.12 de EPICS. A la hora de instalar EPICS, el sistema constructor asume algunos hechos sobre el diseño del código fuente. Varias variables de entorno deben ser configuradas antes de empezar con la instalación de EPICS, siendo las más importantes las siguientes dos:

1. EPICS_BASE = *$HOME/epics/base/*
   Por conveniencia indica la dirección donde está la base del sistema EPICS.

2. EPICS_HOST_ARCH = *linux-x86_64*
   Sirve para indicar cual es la arquitectura del computador en el que se instala EPICS.

La fuente para gran parte del *software* de EPICS puede encontrarse en EPICS Home. Se descarga lo necesario en la carpeta *$HOME/epics/*, se descomprime el archivo base_3.14.12 y se le cambia el nombre a *«base»*. Antes de compilar hace falta mencionar que hay varios archivos que contienen opciones de usuario que afectan al proceso de construcción. Estos archivos son CONFIG_SITE y RELEASE en *$EPICS_BASE/configure/*. Es de especial importancia el archivo CONFIG_SITE, donde el usuario especifica cuales son los blancos para la compilación cruzada, en el caso de este trabajo la arquitectura ARM. Para terminar con la instalación de EPICS hay que establecer las variables de entorno como se ha mencionado y solo queda ir al directorio base y ejecutar el comando *make* [10].

CSS Finalmente, se ha instalado una versión previamente compilada de CSS. Esto se debe a que aunque la fuente del programa esté disponible, el proceso de construcción es bastante complejo y es recomendable usar una versión previamente configurada y compilada acorde a los requerimientos de cada uno. Al instalarlo el programa pregunta donde se quiere establecer el directorio de trabajo, es recomendable ponerlo en *$HOME/epics/workspace/*.

4.2. Desarrollo del IOC

El controlador de entrada y salida es un sistema compuesto por varias capas que se han estudiado en el apartado teórico. Para poder ejecutar un sistema de control en la tarjeta MicroZed hace falta tener el control total sobre sus señales de entrada y salida. Esto se consigue construyendo los módulos de *device support*. Para decirlo de algún modo, el *device support* se encarga de traducir las instrucciones del usuario a un idioma que el *hardware* pueda entender y al reves. Construir dicho *device support* ha sido el principal cometido de este proyecto.

Para empezar con el desarrollo del *Input Output Controller* se crea la estructura en la cual se guardan los códigos del IOC y los archivos *Makefile* necesarios para compilarlo. Para ello en la carpeta que se desee crear el IOC se ejecuta un comando parecido al siguiente [10]:

```
makeBaseApp.pl -t ioc test
```

En este caso, el nombre del IOC que se ha creado es *«test»*. Este comando que se acaba de introducir crea varias carpetas, donde se han guardado los archivos creados a lo largo del trabajo. La estructura de estas carpetas puede apreciarse en la Figura 4.1. A partir de ahora, a la localización del IOC se le denominará *$(TOP)*.

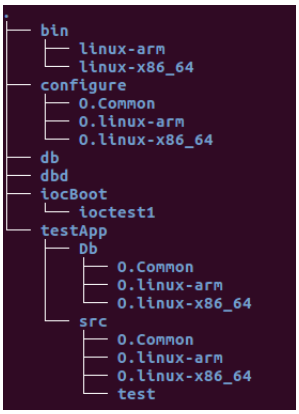


Figura 4.1: Arbol de carpetas. No se muestran los archivos, solo las carpetas.

Los archivos de definición de *records* (.db) creados se guardarán en *$(TOP)/testApp/Db/* y los archivos de *device support* en *$(TOP)/testApp/src/*.

Si se toma un momento para examinar los archivos en *$(TOP)/testApp/src/* (no los ficheros) se puede ver que ahora mismo solo hay dos archivos. El *testMain.cpp* será el punto de entrada al IOC. No es muy interesante ahora mismo dado que solo sirve para invocar el *IOC shell* (más adelante se verá que es crucial para convertir el sistema de control en un sistema *real-time*). El *Makefile*, en cambio, contiene la información necesaria para compilar el IOC.

En este momento el test IOC puede ser compilado y el ejecutable resultante ejecutado. Aun así, no será capaz de hacer nada más que el ejecutable *softIoc*. El siguiente paso consistirá en añadir más funcionalidades al IOC *«test»* para que este funcione como el desarrollador desee.

4.2.1. *Database*

Tal y como se ha mencionado en el apartado teórico, hay varios tipos de *records* predefinidos en EPICS, el objetivo es personalizarlos según las necesidades. Por lo tanto, el primer paso que hay que dar para construir el IOC es definir los *records* correspondientes. Para esto hay que tener en cuenta las diferentes propiedades que pueden adjudicársele a cada tipo de *record* determinado. Para poner un ejemplo, se usará el caso del *record* de entrada analógica (*ai*). Este ejemplo se seguirá usando a lo largo de todo el apartado práctico de la memoria del trabajo.

Primero se debe crear un archivo *$(TOP)/testApp/Db/ai.db* y posteriormente definir el *record*. La forma de declarar el tipo de *record* es la siguiente [10]:

```
record(ai,"$(P)"){
```

```
        field(DTYP,"AnalogInput")
        field(DESC,"Analogic␣Input")
        field(SCAN,"$(SCAN=1␣second)")
        field(INP,"$(S)")
}
```

Esta es seguramente la forma más simple de definir un *record*, donde solo se definen las características indispensables. Por orden estas características son las siguientes [1]:

- En la primera línea aparece el tipo de *record*. Junto al tipo de *record* aparecen unos caracteres (*«$(P)»*) que representan datos que se obtendrán desde el *startup script*, un archivo ejecutable cuya función se explica más adelante. Basta con decir que con estos caracteres se define el nombre de cada *record* creado.
- DTYP (*Device Type*): Este apartado establece la relación del *record* con un dispositivo concreto. Su función se analiza más profundamente en el apartado 4.2.2.
- DESC: Esta es la descripción que se le da al *record*.
- SCAN: Este apartado define en que momento se procesa cada PV asociado a este *record*. Se relaciona con lo visto en el apartado 2.3.5. Puede ser un procesado periódico como en este caso (cada 1 segundo), o se puede definir un escaneo pasivo (procesar los PV solo cuando se le pida) o incluso procesarlos cuando el *hardware* mande una señal de interrupción (este caso se analiza más adelante, ha sido muy útil a lo largo de este trabajo).
- INP: El *link* de entrada del *record*. Esta información se recoge de la *startup script* y cada *record* creado tiene el suyo propio. En el caso de este IOC estos *links* definen de que puerto se quiere leer o en cual escribir la señal correspondiente.

En esta lista se pueden añadir muchas más propiedades y definiciones, pero para este caso basta con dejarlo así. Es aquí donde se pueden definir conversiones o límites para las magnitudes físicas asociadas a cada *record*.

Para incluir un tipo de *record* que se haya definido al *database* del IOC es necesario añadir este archivo al *Makefile* correspondiente (*$(TOP)/testApp/Db/Makefile*) de la siguiente manera y posteriormente compilarlo usando el comando *make* [10].

```
//Una vez se quitan los comentarios el código queda así:

TOP=../..
include $(TOP)/configure/CONFIG

DB += ai.db                          // <- Añadido

include $(TOP)/configure/RULES
```

4.2.2. *Device support*

El *Record Reference Manual* [9] contiene una lista de los tipos de *records* con descripciones y listas de las funciones para el *device support*. Para determinar la forma exacta de las funciones del *device support* perteneciente a un tipo de *record* específico la fuente que encontramos en *$(TOP)/testApp/src/rec* es invaluable. Antes de continuar con la explicación del desarrollo del proyecto se deben analizar algunos aspectos más técnicos relacionados con el *device support* que no se han explicado en el apartado 2.3.6.

Los módulos de *device support* pueden ser divididos en dos clases básicas que se han mencionado por encima anteriormente en el apartado 2.3.6: síncronos y asíncronos [1]. El *device support* síncrono se utiliza para el *hardware* al que puede accederse sin retardo para I/O (*Input/Output*). En cambio, para los dispositivos a los que solo puede accederse vía solicitudes I/O que tardan más tiempo en completarse se debe considerar el *device support* asíncrono. Si se puede acceder a un dispositivo con un retardo inferior a unos pocos microsegundos la opción apropiada es la síncrona. Si el retardo supera los 100 microsegundos entonces la opción asíncrona es la apropiada. Si el tiempo de respuesta está entre unos pocos y un centenar de micro-segundos la elección corre a cuenta del desarrollador. En este trabajo, se han utilizado ambas clases dependiendo del puerto al que se haya ligado el *record*: si la comunicación con *hardware* se realiza en la tarjeta misma se usa la opción síncrona, y si se quiere comunicar con el periférico MAX11300 mediante el protocolo SPI se usa la asíncrona, ya que requiere más tiempo.

En lo que respecta a la relación entre *records* y *device support*, anteriormente se ha visto que en el *database* hay un campo relacionado con el dispositivo (DTYP). Además para relacionar los dos códigos del *device support* (el .dbd y el .c) entre ellos es necesario usar el campo DSET:

- DTYP: *Device Type*.
- DSET: *Adress of Device Support Entry Table*.

El campo DTYP contiene el índice de la opción de menú definida por las definiciones ASCII del dispositivo. Al iniciar el IOC este utiliza el campo DTYP y las estructuras del *device support* definidos en *devSup.h* para inicializar el campo DSET. Así, el *record support* puede encontrar el correspondiente *device support* por medio del campo DSET.

Durante este trabajo, ha sido necesario desarrollar de cuatro tipos de *device support*: uno para las salidas binarias (*bo*) de la I/O Carrier, otro para las entradas binarias (*bi*), un tercero para las salidas analógicas (*ao*) mediante el periférico MAX11300 y el último para las entradas analógicas (*ai*).

Para entender bien las distintas oportunidades que brinda el *device support* es necesario aclarar varios conceptos de programación que han sido muy útiles

a lo largo de la construcción del soporte. Estos conceptos son los ***mutex***, el ***callback*** y el ***I/O Interrupt.***

I/O Interrupt

A la hora de crear el *record support* se debe especificar el tipo de escaneo que se quiera realizar. Hasta ahora se ha supuesto que siempre se realiza un escaneo periódico de los *records*, pero no es así. A veces es necesario usar otros sistemas que permiten actualizar los *records* solamente cuando algo ocurra y no periódicamente. Este es el caso del escaneo de la clase *«I/O Interruption»*, que permite que las rutinas del *device support* correspondientes al procesamiento de un *record* se ejecuten solamente cuando se comunica que ha habido una interrupción de *hardware* [23].

En el contexto de la informática una interrupción es una señal que se envía desde un dispositivo conectado a una computadora o desde un programa al sistema operativo para que se detenga y ejecute un código específico a continuación. Esto sirve, por ejemplo, para poder usar varios programas a la vez en un ordenador. Los ordenadores de hoy en día son susceptibles a las interrupciones (*interrupt-driven*). Básicamente, un ordenador solo puede ejecutar una instrucción en cada momento, pero como puede ser interrumpido, puede turnar los programas o conjuntos de instrucciones que procesa. Esto se conoce como *«multitasking»*.

Durante este trabajo, las interrupciones se han usado para procesar alguno de los *records* solamente cuando ha sido necesario hacerlo. Para ello, cada vez que se quiera procesar uno de estos *records* se crea una interrupción. Para hacer esto, se han incluido unos hilos secundarios en los módulos de *device support* asociados a estos *records*. Estos hilos se denominan *«workers»* y su función es realizar trabajos específicos y crear una interrupción cuando se cumpla una condición previamente fijada. Además, estos hilos pueden pasar información a la rutina de procesión del *record* (el valor de la última medición por ejemplo) para que este lo use si es necesario.

Así, por ejemplo, se puede conseguir que un PV asociado a un *record* de entrada analógica se procese solo cuando los *workers* hayan medido el valor de entrada 100 veces y se haya calculado el valor medio; o cuando una entrada binaria cambie de valor, en vez de hacerlo periódicamente.

Callback

Se denomina *callback* a una función que es introducida como argumento en otra función y es invocada tras algún tipo de evento. La función padre (la

cual recibe el *callback* como argumento) trata al *callback* como parámetro. Esto puede apreciarse más correctamente en el siguiente pseudocódigo como ejemplo:

```
// El método callback:
function sentidoDeLaVida(){
        log("El␣sentido␣de␣la␣vida␣es:␣42");
}

// Un método que acepta la función callback como argumento,
// recoge una función de referencia que se ejecuta
// cuando imprimirNumero se complete
function imprimirNumero(int numero, function funcionCallback)
        {
        print("El␣numero␣que␣has␣introducido␣es:␣" + numero);
}

// Por último, este es el método conductor:
function accion() {
        imprimirNumero(9, sentidoDeLaVida);
}
```

Tras llamar a la función **accion()** en el código anterior se obtiene lo siguiente:

```
El numero que has introducido es: 9
El sentido de la vida es: 42
```

Por lo tanto, cuando se llama al método padre y se completan las acciones descritas en su código, se invoca el método *callback*, o en otras palabras : *«call at the back»*. La principal ventaja es que el *callback* se utiliza como una parte de código que hay que ejecutar cuando un determinado evento tiene lugar, pero sin que el procesador se quede esperando a que ocurra el evento. De este modo, se puede seguir con la ejecución del código. A continuación se expone un ejemplo simple para entender esta propiedad:

```
// En este primer ejemplo no se usa el callback.

fileObject = open(file)
// Una vez abierto el archivo es posible escribir en él.
fileObject.write("Estamos␣escribiendo␣en␣el␣archivo")
// Ahora se puede seguir con lo que viene a continuación,
// cosas sin relación a esto.
```

En el código anterior, el programa se queda esperando a que se abra el archivo para después escribir sobre él. Esto bloquea el flujo de ejecución y el programa no puede hacer ninguna otra cosa que podría resultar importante. Y si en cambio se prueba con lo siguiente:

```
// Pasamos escribirEnArchivo (una función callback!)
// a la funcion que abre el archivo

fileObject = open(file, escribirEnArchivo)
```

```
// La ejecución sigue fluyendo ->
// -> no esperamos a que se abra el archivo
// Una vez abierto el archivo se escribe sobre él,
// pero mientras se pueden hacer otras cosas!
```

Los *callbacks* se han usado para desarrollar un módulo de *device support* asíncrono [23]. De este modo, no importa si hay algún retardo para acceder al *hardware*, porque mientras tanto se puede continuar con la ejecución del programa sin esperar a que el *hardware* conteste.

Mutex

En la programación, un *mutex* (*mutual exclusion object*) es un objeto creado para que los múltiples hilos de un mismo programa puedan acceder al mismo recurso turnándose entre sí. Su uso es habitual en las secciones críticas de la programación concurrente. Normalmente, cuando se inicia un programa, este crea un *mutex* para un recurso dado al principio, pidiéndoselo al sistema. El sistema devuelve entonces un nombre o *ID* único. Después de esto, cualquier hilo que quiera usar el recurso debe bloquearlo de los demás hilos y al terminar de usar el recurso desbloquearlo [8].

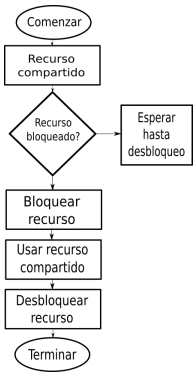


Figura 4.2: Esquema de funcionamiento de un *mutex*.

En este sentido, los *mutex* pueden entenderse como una especie de semáforo para los distintos hilos del programa. Pero a diferencia de los semáforos que se usan en programación, solamente un hilo puede acceder al recurso en cada instante de tiempo. En el *device support* de este trabajo no se ha utilizado un único tipo de *mutex*, sino dos: el *mutex* de EPICS y el *mutex* del propio sistema Unix. A continuación un ejemplo del uso de los *mutex*:

```
#include <pthread.h>

// Primero se crea el mutex (en este caso con el ID "plock")
pthread_mutex_t plock;
```

```
long contador;
incrementar_contador(){
// Cuando se quiera acceder a un recurso primero se bloquea
            pthread_mutex_lock(&plock);
// Se accede al recurso
        contador = contador + 1;
// Y por último se desbloquea
            pthread_mutex_unlock(&plock);
}
```

En este trabajo, el uso de los *mutex* ha sido necesario. Se han definido varios *records* del tipo *ai* y los respectivos PV asociados para leer las distintas entradas de la tarjeta MAX11300. Se realizan medidas de cada pin de entrada cada 2,5 milisegundos. Al iniciar el IOC el acceso simultaneo de varios PV al puerto SPI de la tarjeta colapsaba el proceso. Esto se ha arreglado con el uso de los *mutex*.

4.2.2.1. Creación del *device support* [10]

Tras analizar cuales son las características del *device support* y los distintos mecanismos de programación que se han usado para construirlo, se puede pasar a ver como ha sido el proceso de construcción. El primer paso es añadir el dispositivo al *database*. Para esto, se crea un fichero con el nombre *xxdev.dbd* en *$(TOP)/testApp/src/*, donde *'xx'* representa la clase de *device support* que se quiere realizar, por ejemplo *ai* para el caso de entrada analógica. En este fichero ha de escribirse algo con el siguiente formato:

```
// Para el caso de aidev.dbd:
device(ai,X,Y,"$(D)")
```

Este ejemplo define el dispositivo Y como soporte para un *record* de clase *ai* (*analog input*) con un *link* de entrada X de nombre *«$(D)»*. El significado y los atributos de cada uno de estos elementos se exponen a continuación:

- X = *addrType*, define el tipo de *link* de entrada o dirección del que se va a recibir recibir o trasmitir información. Las opciones que hay aquí son varias, entre ellas: AB_IO, INST_IO, RF_IO, CONSTANT, etc. Para entender en que caso ha de usarse cada uno se debe consultar el *EPICS Application Developer Guide*[1]. En este trabajo se ha usado siempre el tipo INST_IO.

- Y = *dsetName*, el nombre que se le da al dispositivo. El nombre Y debe ser único en todo el IOC. Por convención, los nombres del *device support* deben tomar el nombre *devXxYyy*, donde *Xx* es el tipo de *record* al que se asocian y *Yyy* identifica el *hardware* soportado. Por ejemplo, en uno de los módulos de *device support* creados en este trabajo se ha usado el nombre *«devAiAxi»*.

- Por último tenemos en nombre *«$(D)»* , el cual es simplemente el nombre propio que se da al *link X* y se conoce como *dtypeName*.

Este proceso ha de repetirse por cada clase de *record* que se quiera soportar en el dispositivo o más veces si se quiere asociar más de un *device support* con cada clase de *record*.

A continuación se debe escribir el código correspondiente al *device support*. Siguiendo con el ejemplo de la clase de *record ai*, se ha de crear un fichero en la siguiente ubicación y con el siguiente nombre: *$(TOP)/testApp/src/devai.c*. Aquí es donde se definen las propiedades del *device support*, tales como la manera de iniciar el *record* o el método de procesarlo cuando se requiera hacerlo. Este código tiene la siguiente estructura:

```
#include <stdlib.h>
#include <epicsExport.h>
#include <dbAccess.h>
#include <devSup.h>
#include <recGbl.h>

#include <aiRecord.h>

static long init_record(aiRecord *pao);
static long read_ai(aiRecord *pao);

struct aiState {
// Aquí van las variables de la estructura privada:
   unsigned int address;
   unsigned int offset;
   epicsMutexId lock;
   CALLBACK cb;
};
```

En este ejemplo, el *device support* cuenta con las funciones *init_record* y *read_ai*. La información del estado del cliente se guarda en las instancias de la estructura privada *aiState*. Se deben importar las librerías necesarias en cada caso.

```
struct {
  long num;
  DEVSUPFUN  report;
  DEVSUPFUN  init;
  DEVSUPFUN  init_record;
  DEVSUPFUN  get_ioint_info;
  DEVSUPFUN  read_ai;
  DEVSUPFUN  special_linconv;
} devAiAxi = {                      // Línea 9
  6, /* space for 6 functions */
  NULL,
```

```
  NULL,
  init_record,
  NULL,
  read_ai,
  NULL
};
epicsExportAddress(dset,devAiAxi);  // Línea 18
```

Ahora se asocia el nombre *«devAiAxi»* con las rutinas del *device support*. Este mecanismo es un modo de proporcionar una lista de funciones que el *record support* usa para realizar distintas funcionalidades. En este caso solo se proporcionan las dos funciones mencionadas previamente, las demás son declaradas *NULL*. Hay que recalcar que en el caso del *records* de clase *ai* la rutina *read_ai* es necesaria, mientras que la función *init_record* es opcional [10]. Fijarse bien en como se relaciona el modulo del *device support* con la declaración del dispositivo que le corresponde mediante el DSET de las líneas 9 y 18, esto es esencial.

A continuación, simplemente se definen las funciones declaradas anteriormente. Estas definiciones dependerán de lo que se quiera hacer cuando el *record* se inicia (*init_record*) o cuando el *record* se procese (*read_ai*) y están en manos del programador. Hay varias funciones previamente construidas en EPICS que resultan muy útiles para programar las rutinas del *device support*.

```
static long init_record(aiRecord *pao) {

  // por ejemplo:
  // aquí se inicializa el link ->
  // -> entre el dispositivo y el ordenador:

  parameterCount=sscanf(pao->inp.value.instio.string, &
  "%x:%x",&address,&offset);
  int memfd = getMemoryDescriptor();
  void *mapped_dev_base;
  mapped_dev_base = mmap_device(memfd,address);
  priv->address=mapped_dev_base;
  priv->offset=offset;

  // o se inician los mutex:
  priv->lock = epicsMutexMustCreate();

  return 0;
}


static long read_ai(aiRecord *pao) {

  // A cargo de esta rutina corre la responsabilidad
  // de actualizar el PV cuando el record se procesa:

  struct aiState * priv=pao->dpvt;
```

```
  pao->rval = priv->state;
  return 0;
}
```

Dependiendo de las funcionalidades que se le requieran al *device support* puede que la estructura sea bastante más compleja que la que se ha visto aquí. Esto ocurre por ejemplo en los *device support* asíncronos, donde ha de hacerse uso de los antes mencionados *I/O interrupt* y *callback* o en aquellos que requieran acceder a un recurso compartido, en cuyo caso se deben usar los *mutex*.

Cuando la configuración del *device support* está terminada hay que incluir los ficheros *$(TOP)/testApp/src/aidev.dbd* y *$(TOP)/testApp/src/devai.c* en *$(TOP)/testApp/src/Makefile* para poder compilarlos. Una vez eliminados los comentarios el *Makefile* tiene la siguiente forma:

```
TOP=../..
include $(TOP)/configure/CONFIG
PROD_IOC = test

DBD += test.dbd
test_DBD += base.dbd
test_DBD += aidev.dbd                          // <- Añadido

test_SRCS += test_registerRecordDeviceDriver.cpp
test_SRCS += devai.c                           // <- Añadido
test_SRCS_DEFAULT += testMain.cpp
test_SRCS_vxWorks += -nil-

test_LIBS += $(EPICS_BASE_IOC_LIBS)
include $(TOP)/configure/RULES
```

El proceso completo de construcción del *device support* que se ha visto aquí ha sido replicado 4 veces para este trabajo, configurando cada uno de ellos a medida para la clase de *record* correspondiente.

4.2.3. *IOC shell* y la *startup script*

El EPICS *IOC shell* es un simple interprete de comandos que proporciona parte de las capacidades del *vxWorks* (sistema operativo de sistemas integrados) *shell*. Se usa para interpretar el *startup script* (*$(TOP)/iocBoot/ioctest1/st.cmd*) y para ejecutar comandos que se introducen en la terminal de la consola. En la mayoría de casos los *startup scripts* de vxWorks se puede interpretan con el *IOC shell* sin ningún cambio.

El *IOC shell* lee líneas de entrada, separa la línea en comandos y llama a las funciones correspondientes a dichos comandos descodificados [1]. Los comandos y argumentos se separan por uno o más espacios. Los caracteres que el *IOC shell* identifica como espacios son la barra espaciadora, el tabulador, las comas o el abrir o cerrar paréntesis. Por lo tanto la siguiente linea de comando:

```
dbLoadRecords("db/ai.db","user=benat")
```

sería interpretado por el *IOC shell* como el comando *dbLoadRecords* con los argumentos *db/ai.dbd* y *user=benat*.

Aquí es donde entra en juego el archivo *st.cmd*, ya que contiene las instrucciones que se deben ejecutar cuando se inicie el IOC. Para iniciar el IOC se debe introducir la siguiente línea de comandos desde la propia MicroZed:

```
./bin/linux-arm/test ./iocBoot/ioctest1/st.cmd
```

Esta línea ordena al *IOC shell* que lea y ejecute lo que encuentre en el archivo *st.cmd*. Dicho esto, en el *startup script* se deben incluir los comandos que inician los *records* y sus respectivos módulos de *device support* y las rutinas o funciones que se deseen ejecutar al inicio del IOC.

Para el caso de los *records* y el *device support* correspondiente basta con añadir una línea por cada *record* que se quiera crear en el IOC. El contenido de estas líneas tiene la siguiente forma:

```
dbLoadRecords("db/ai.db","P=test:ai0,D=AnalogInput,␣&
S=@0x41220000:00","user=benat")
```

Si se compara esta línea con la que antes se ha usado de ejemplo, el comando es el mismo. Los que cambian son algunos de los argumentos. El primero se mantiene igual y nos indica a que definición de *record* pertenece el *record* que se crea. El segundo (*«P=test:ai0»*) define el nombre del *record*, mientras que el tercero (*«D=analogInput»*) indica el *dtypeName* del *device support* que se asocia al *record* creado. El siguiente indica la dirección con la que el *record* establece el *link* y su formato varía dependiendo de cual sea la opción que se haya elegido en *addrType* (mencionado previamente en el apartado 4.2.2). En este trabajo, al haberse seleccionado la opción de INST_IO como *addrType*, la dirección debe venir precedida de la cadena de caracteres *«@0x»* [1]. El último argumento carece de importancia en este caso.

Hay que mencionar que al crear un *record* también se crea un PV del mismo nombre asociado al campo de valor del *record*. Dependiendo de la clase de *record* los PV creados pueden ser más de uno, pero en las clases usadas en este trabajo (*ai*, *ao*, *bi*, *bo*) cada *record* solamente tiene un PV asociado. Esto significa que el hecho de crear el *record* arriba mencionado (*«P=test:ai0»*) también crea un PV con el mismo nombre, asociado al valor de la entrada analógica (en este caso el tipo de *record* es *ai*) del *link* que se haya como cuarto argumento.

De esta forma, se pueden crear cuantos PV se deseen, asociados a la clase de *record*, con el *device support* y la dirección que se quieran utilizar. Únicamente se debe tener en cuenta que el nombre de cada *record* (y por ello el del PV) tiene que ser único en el IOC, no pueden haber dos iguales.

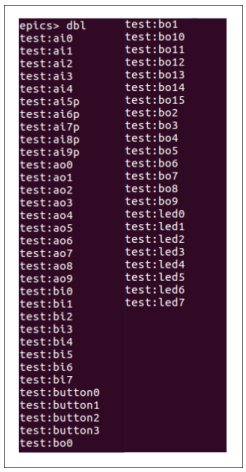


Figura 4.3: Comando *dbl* mostrando todos los PV del IOC.

4.2.3.1. Funciones en el *IOC shell*

En algunos casos es necesario llamar desde el *startup script* a una rutina predefinida. Por ejemplo cuando se desea configurar algún dispositivo o reiniciar los valores de un pin de comunicación. Para que esto tenga efecto, el *IOC shell* debe ser capaz de reconocer la función y ejecutarla. Esto se hace registrando previamente la función. Para ello primero hay que crear un archivo (*$(TOP)/testApp/src/function.dbd*, donde en lugar de poner el nombre *function* se pone el nombre de la rutina) que contenga la siguiente expresión:

```
registrar(functionRegister)
```

A continuación se debe de crear un archivo *$(TOP)/testApp/src/function.c* en el que se define la función y en él se escriben las instrucciones para que el *IOC shell* sea capaz de entenderla y ejecutarla.

```
#include <stdio.h>
#include <epicsExport.h>
#include <iocsh.h>

// Esta es la función al que el IOC shell llama directamente
int spi(char spi_base){

// Aquí se definiría la función que reinicia
// la conexión al puerto con conexión SPI
// y devuelve el valor 0.

return 0;
}
```

```
// Y esta es la información requerida por el IOC shell
// para registral la función:

static const iocshArg  spiArg0 = {"spi_base",iocshArgString};
static const iocshArg *spiArgs[] = {&spiArg0};
static const iocshFuncDef spiFuncDef = {"spi",1,spiArgs};

// Selecciona los agumentos que la función 'spi' necesita
static void spiCallFunc(const iocshArgBuf *args) {
    spi(args[0].sval);
}

// Registro de la rutina, se ejecuta en el startup
static void spiRegister(void) {
    iocshRegister(&spiFuncDef, spiCallFunc);
}
epicsExportRegistrar(spiRegister);
```

Estos archivos se tienen que incluir también en *$(TOP)/testApp/src/Makefile* para que puedan ser compilados.

Después de hacer esto ya sería posible incluir la función *«spi»* en el *startup script* y el *IOC shell* lo entendería correctamente. En este trabajo ha sido necesario configurar la tarjeta MAX11300 al iniciar el IOC y para ello se ha usado este método. La configuración del esta tarjeta determina cuales de sus puertos actúan como ADC o DAC. Haciéndolo de este modo es posible alterar la configuración desde fuera del IOC, para que surja efecto la siguiente vez que este sea iniciado. También se ha optado por limpiar el puerto de comunicación SPI al inicio, para evitar que al iniciar el IOC el bus SPI estuviese saturado.

4.2.4. *Real-time*

Finalmente se ha tratado de darle la máxima prioridad posible al IOC dentro de la I/O Carrier. Esto consiste en que la I/O Carrier ponga cualquier operación que tenga que realizar en el IOC por delante de cualquier otra operación no relacionada con el IOC. Esto es un sistema *real-time*. Para esto simplemente se define una pequeña rutina en el archivo *$TOP/testApp/src/testMain.cpp* que define la prioridad que se le quiera dar al IOC en una escala del 1 al 100, siendo el 100 la prioridad máxima y el 1 la mínima.

Para comprobar la efectividad del *real-time* se le ha pedido al IOC que alterne el valor de un *record* binario continuamente cada T tiempo. La salida se ha conectado a un osciloscopio y se ha analizado como varía la forma de la señal dependiendo del coeficiente de prioridad que se haya asignado al IOC en la rutina *real-time*. Se ha visto que cuando la prioridad es baja hay un pequeño intervalo de incertidumbre *(t)* al rededor de cada cambio de voltaje y que el periodo entre cambios varía entre T-t y T+t. Por otro lado, si se introduce el

valor máximo de prioridad el cambio siempre se da cada T tiempo.

Finalmente, al terminar la construcción del IOC (después de programar el *database*, el *device support* para todas las clases de *records* y definir las funciones a las que se llama desde la *startup script*) el contenido de las carpetas *$(TOP)/testApp/src/* y *$(TOP)/testApp/Db/* es el que puede verse en la siguiente imagen.

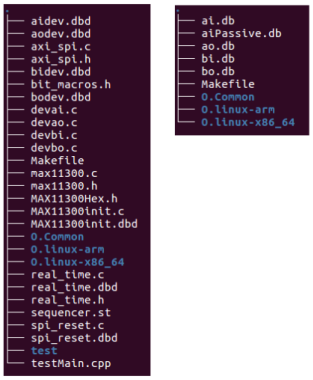


Figura 4.4: *$(TOP)/testApp/src/* (izquierda) y *$(TOP)/testApp/Db/* (derecha).

4.3. Interfaz gráfica

Para terminar se ha intentado construir una OPI los más visual y interactiva posible, que facilite la monitorización y manipulación de todos los PV del IOC. Para ello se ha diseñado una pequeña estación de control mediante CSS. Esta interfaz se ha implementado en un cliente del sistema EPICS. La estación permite al usuario tener acceso fácil a cualquier PV del IOC. La interfaz gráfica del sistema contiene los siguientes instrumentos:

- Monitores LED para monitorizar los 4 botones de la I/O Carrier.
- Botones booleanos para controlar los 8 LED de la I/O Carrier.
- 16 selectores booleanos para controlar las salidas binarias de los bloques JH y JG.
- 8 monitores LED para monitorizar las entradas digitales del puerto JK.
- 10 monitores analógicos para las 10 entradas analógicas del MAX11300.

- 10 deslizadores analógicos para controlas las 10 salidas analógicas del MAX11300.

De este modo, se ha conseguido una interfaz gráfica con el siguiente formato:

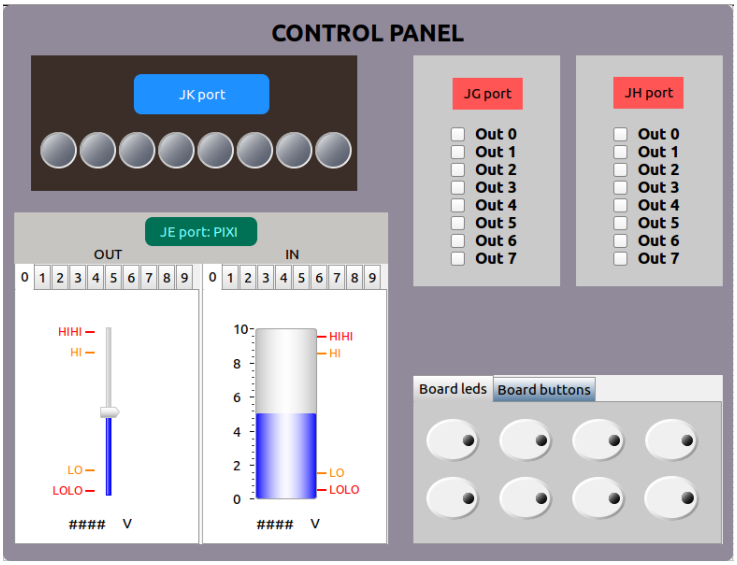


Figura 4.5: Interfaz gráfico de control.

Capítulo 5

Conclusiones

La curiosidad que sentía sobre el desarrollo de un verdadero sistema de control desde cero, cuando el sistema no es más que un *hardware* inerte que requiere ser programado, ha sido la mayor motivación para llevar a cabo este trabajo. Además, obtener un conocimiento de primera mano sobre el funcionamiento de las empresas que se dedican o están parcialmente inmersas en este sector, junto a la oportunidad de participar en sus actividades, se me antojaba realmente interesante. Por todo ello, no tuve mucho que pensar cuando me cruce con la ocasión de desarrollar este proyecto.

Respecto al sistema usado, EPICS ha resultado ser un sistema de control muy versátil. Su utilidad para sistemas de control distribuido ha quedado patente. Aporta una gran homogeneidad para el desarrollador y debido a su sistema modular es fácilmente escalable, pudiendo aumentar el numero de los IOC y los OPI sin muchas complicaciones. Se ha visto que la definición de los *records* es común para todo tipo de dispositivos y que al cambiar de dispositivo solo hace falta cambiar el *device support*. Además un IOC se puede ejecutar en distintos tipos de *hardware*.

Durante este trabajo se ha conseguido construir un IOC en la I/O Carrier. Este IOC ha sido desarrollado íntegramente con las herramientas del sistema EPICS. De este modo, se ha conseguido prescindir de la *shared memory* de la MicroZed y del código independiente y no integrable en EPICS que se tenía al principio. A esto se le ha añadido la creación de la interfaz gráfica del usuario para facilitar el control desde el cliente de EPICS. El sistema EPICS creado es capaz de monitorizar y manipular cualquier señal I/O de la I/O Carrier.

Durante la consecución del objetivo han surgido algunos problemas. Entre estos problemas los hay que han sido causados por la falta de experiencia en el tema y también los que han sido causados por la dificultad de algunas tareas. Entre estos últimos hay que destacar los problemas asociados a la compilación cruzada o el colapso del puerto al que hemos conectado el MAX11300 debido

al protocolo SPI, lo que hemos solucionado usando los *mutex*. Aún así, ha sido posible solucionar la mayoría de los errores encontrados por el camino.

Se ha cumplido el objetivo principal del proyecto, que ha sido diseñar y desarrollar un EPICS IOC. Al final del proyecto se ha intentado complementar este objetivo con algún otro objetivo secundario como el desarrollo de una máquina de estados automática para el sistema. Al intentar desarrollar la máquina de estados usando el secuenciador, a pesar de haber escrito parte del código SNL, hemos tenido problemas a la hora de hacer la compilación cruzada del código. A parte de este último, sería posible añadir otros accesorios y complementos al sistema EPICS, por ejemplo un sistema de alarmas. Por ello, este proyecto tiene un gran potencial de cara al futuro.

Bibliografía

[1] Kraimer, M. R., Anderson, J. B., Johnson, A. N., Norum, et al. (2009). *EPICS application developer's guide*. February 2010. [Online] Disponible: http://www.aps.anl.gov/epics. Última consulta: 16/06/2017.

[2] Lewis, S. A. (2000). *Overview of the Experimental Physics and Industrial Control System: EPICS*. [Online] Disponible: http://csg.lbl.gov/EPICS/OverView.pdf. Última consulta: 16/06/2017.

[3] Dalesio, L. R., Hill, J. O., Kraimer, M., Lewis, S., Murray, D., Hunt, S., ... & Dalesio, J. (1994). *The experimental physics and industrial control system architecture: past, present, and future. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*. (Vol. 352), páginas 179-184.

[4] Dalesio, L. R., Kraimer, M. R., & Kozubal, A. J. (1991, November). *EPICS architecture*. En ICALEPCS (Vol. 91), páginas 92-15.

[5] Kim, K. H., Ju, C. J., Kim, M. K., Park, M. K., Choi, J. W., Kyum, M. C., & Kwon, M. (2006). *The KSTAR integrated control system based on EPICS. Fusion Engineering and Design*. (Vol. 81), páginas 1829-1833.

[6] Bangemann, T., Karnouskos, S., Camp, R., Carlsson, O., Riedl, M., McLeod, S., ... & Stluka, P. (2014). *State of the art in industrial automation. In Industrial Cloud-Based Cyber-Physical Systems* (páginas 23-47). Springer International Publishing.

[7] Kasemir, K. (2007, October). *Control system studio applications*. In Proceedings of ICALEPCS (página 692).

[8] Presler-Marshall, M. J. C. (2001). *Protecting shared resources using mutex striping*. U.S. Patent No. 6,199,094. Washington, DC: U.S. Patent and Trademark Office.

[9] Anderson, J. B., & Kraimer, M. R. (1994). *Record Reference Manual*.

[10] Michael Davidsaver. *Basic EPICS Device Support*. [Online] Disponible: https://mdavidsaver.github.io/epics-doc/epics-devsup.pdf. Última consulta: 21/06/2017.

[11] *Product Briefs: MicroZed.* [Online] Disponible: http://zedboard.org/product/microzed. Última consulta: 18/06/2017.

[12] *Product Briefs: I/O Carrier.* [Online] Disponible: http://zedboard.org/product/microzed-io-carrier-card. Última consulta: 18/06/2017.

[13] *MAX11300 PIXI description & datasheet.* [Online] Disponible: https://www.maximintegrated.com/en/products/analog/data-converters/analog-to-digital-converters/MAX11300.html. Última consulta: 18/06/2017.

[14] Harry Fraser. *The electric kiln: a user's manual (2nd edition) (Página 41).* University of Pennsylvania Press. 2000.

[15] *EPICS training series* (Vol. 1). [Online] Disponible: http://www.aps.anl.gov/epics/docs/APS2015.php. Última consulta: 21/06/2017.

[16] *EPICS training series* (Vol. 2) [Online] Disponible: http://www.aps.anl.gov/epics/docs/APS2014.php. Última consulta: 21/06/2017.

[17] Dalesio, L. R. (1989, March). *The ground test accelerator control system database: configuration, run-time operation, and access.* In Particle Accelerator Conference, 1989. Accelerator Science and Technology., Proceedings of the 1989 IEEE (páginas 1693-1694). IEEE.

[18] Andrew Johnson. *EPICS database principles.* [Online] Disponible: http://www.aps.anl.gov/epics/docs/USPAS2010/Lectures/Database.pdf. Última consulta: 24/06/2017.

[20] Mark Rivers. *EPICS records and Process Variables.* [Online] Disponible: http://www.aps.anl.gov/epics/tech-talk/2012/msg02495.php. Última consulta: 26/06/2017.

[21] Shell, R. (2000). Handbook of industrial automation. CRC Press.

[22] Ben Franksen. *State Notation Language and Sequencer.* [Online] Disponib le: http://www-csr.bessy.de/control/SoftDist/sequencer/. Última consulta: 25/06/2017.

[23] Andrew Johnson. *Writing device support.* [Online] Disponible: http://www.aps.anl.gov/epics/docs/APS2015/01-EPICS-Device-Support.pdf. Última consulta: 26/06/2017.