Índice

1. Introducción

Continuamos con la descripción del entorno de desarrollo iDigi DIA [1]para Python. En el comentario técnico sobre dicho entorno (CoTC-003) nos habíamos enfocado en los aspectos de diseño, implementación y pruebas para la generación de aplicaciones mediante la interfaz de alto nivel de los proyectos del framework DIA.

Recuérdese que al utilizar dicha interfaz no necesitábamos escribir ni una sola línea de código, siempre y cuando, utilicemos dispositivos XBee para los cuales, a priori, ya existiera el driver correspondiente dentro del entorno DIA.

Ahora tomaremos el enfoque complementario, es decir, asumiremos que tenemos nuestro propio hardware basado en XBee y que aquel no está correctamente representado por ninguno de los drivers existentes. De tal forma, nos veremos obligados a escribir el propio driver para poder aprovechar luego el resto de las facilidades que ofrece el framework. Sin embargo, como la idea de este último es facilitar la tarea del desarrollador, podremos tomar como base a alguno de los drivers ya presentes entre la lista de dispositivos disponibles.

1.1. Objetivos

El objetivo de este trabajo es generar un driver de dispositivo para el entorno de desarrollo iDigi DIA utilizando como base un driver preexistente en dicho sistema.
Mediante el driver a construir vamos a abstraer las particularidades más relevantes para dar soporte a las placas de evaluación XBoard de Cika Electrónica[2].

1.2. Aclaraciones

Este trabajo está estructurado en dos secciones principales: una teórica y una práctica. La teórica ha sido tomada de la ayuda en línea del entorno de desarrollo y comprende la traducción del tema "Drivers" de la documentación para el desarrollador:
Help> Help Contents> iDigi Dia Documentation> iDigi Dia 1.3.8> 2. Developer Documentation > 2.2. Drivers
La parte práctica está centrada en la modificación del driver `XBib Development Boards` generando a partir de aquel un driver propio y en la posterior utilización de este último.

1.3. Fuera de alcance

Esta nota menciona y utiliza en casi toda su extensión varios conceptos propios de la programación orientada a objetos y lo hace tanto en forma teórica como práctica utilizando el lenguaje de programación Python para las implementaciones.
Queda fuera de alcance cualquier temática relacionada con los fundamentos de la programación orientada a objetos o con el aprendizaje del lenguaje Python.

[1] Device Integration Application
[2] Cika Electrónica comercializa estas placas como parte del ZKit: Kit de evaluación XBee ZB (ZigBee-PRO).

2. Teoría

2.1. Introducción a los Device Drivers en Dia

Un **device driver** en Digi Dia es código Python que realiza la abstracción de un dispositivo físico real (como un termómetro o un sensor de luz) o algún control lógico (como un termostato virtual) en un set de propiedades llamadas **canales**. Los canales son exportados a la **base de datos de canales** para ser consumidos por otros servicios del sistema como pueden ser otros device drivers, el logging sub-system, o varias instancias de presentaciones. Los canales son referenciados en la base de datos de canales como "`<driver_instance>.<channel_name>`".

Por ejemplo, un driver para un GPS en Dia actúa como interface para un módulo GPS físico abriendo un puerto serie. La responsabilidad primaria del device driver's será leer strings con el formato GPS NMEA desde el puerto serie, parsearlo y formatearlo como objetos Sample definidos en Dia, y depositar estas samples en los canales correspondientes (propiedades). Las propiedades de canal aplicables para este caso podrían llamarse "latitude", "longitude", "num_satellites", etc. y serían accesibles desde la base de datos de canales usando una instancia de este driver "gps0" como "gps0.latitude", "gps0.longitude", "gps0.num_satellites", etc. Un device driver puede ser un dispositivo puramente virtual que importe canales de otro dispositivo (seguramente via seteos configurables) y tome decisiones sobre esos datos. Un ejemplo de dispositivo virtual es el "geofencing" driver:

Este objeto define una región sobre un mapa dentro de la cual cierto recurso debe permanecer. Bajo una forma simplificada, un "geofence" tiene dos estados: dentro de la región o fuera de ella. Un driver virtual "geofencing" debería configurarse para aceptar *settings* para definir una región geofencing como una lista de coordinadas y una fuente de canales de un driver GPS. El driver geofencing produciría entonces como salida, una propiedad Booleana que indicaría si la ubicación reportada por el driver GPS está dentro o fuera del *geofence*. Como este driver opera solo sobre un flujo de muestras de otro driver, es considerado un driver "virtual". Un virtual driver así normalmente contiene porciones de lógica de control para una aplicación compleja.

Todo device drivers en Dia debe presentar los siguientes requerimientos:

a. Debe heredar de `DeviceBase`[3].
b. Tiene cero o mas configuraciones (`Settings`).
c. Tiene cero o mas propiedades para crear cero o mas `Channels` en la `Channel Database`.
d. Debe implementar los siguientes métodos de interfaz de la `DeviceBase`:
    1. `apply_settings()`, llamada cuando se aplican nuevos *settings*.
    2. `start()`, llamada cuando el driver se inicia.
    3. `stop()`, llamado cuando el driver se detiene .

Aquí se hablará de dos arquitecturas de drivers de dispositivos:
a. Genérico heredando de *DeviceBase.*
b. Arquitectura *XBee*

[3] `DeviceBase` es una clase virtual que, como su nombre lo indica, sirve como base para la realización de cualquier driver de dispositivo dentro del frame DIA.

2.2. Device Drivers estándar

2.2.1. Dónde se ubican físicamente los Device Drivers

Los Device Drivers en Dia están almacenados en el directorio *src/devices*. Este directorio contiene unos pocos sub-directorios. Algunos de ellos (como "gps") contienen archivos para un driver específico. Otros directorios sirven para propósitos de organización general:

- El directorio "vendor" contiene un sitio para ubicar libraries de drivers de los diferentes fabricantes.
- EL directorio "xbee" contiene el stack de drivers para XBee y demás device drivers específicos de Digi. Esto será cubierto en detalles en la sección 3: XBee Device Drivers.

2.2.2. El Device Driver Template

El template del device driver está en *src/drivers/template_device.py*. El template device driver sirve como un ejemplo y punto de inicio para aquellos que quieran aprender a implementar sus propios drivers en el framework iDigi Dia.

El driver *template* sirve como base para aprender sobre la estructura de un driver en el Dia framework usándolo obviamente como plantilla para crear nuevos drivers.

Es además, un virtual driver que no conecta con ningún periférico de hardware y comprende las dos características siguientes:

- Un contador que se actualiza a intervalos programables.
- Dos canales cuyas propiedades se pueden escribir, y cuando esto sucede ambas se suman estableciendo el valor de una tercera propiedad que puede leerse.

Este driver comienza con una descripción al comienzo del archivo.

Debajo de aquella le sigue la sección de importaciónes. Esta sección incluye otros módulos Python dentro del alcance del driver *template* con el propósito de incluir funcionalidades en forma modular. El conjunto mínimo de "includes" para un device driver es el siguiente:

```
from devices.device_base                    import DeviceBase
from settings.settings_base                 import SettingsBase, Setting
from channels.channel_source_device_property import *
```

Todo device drivers se implementa definiendo una clase derivada de *DeviceBase*. Una clase *DeviceBase* contiene objetos *Settings* (accedidos via métodos definidos en la clase *SettingsBase*) y un set de propiedades de canales para ser expuestos en la *Channel Database*. Estas propiedades de canal son del tipo *ChannelSourceDeviceProperty*.

El template implementa un contador que se actualiza asíncronamente y requiere capacidad de funcionamiento por *threading* (método de ejecución multi-tareas) y la habilidad de demorar esta tarea

con la función `sleep()` perteneciente al módulo de tiempo estándar de Python[4]. Estas *libraries* también se importan, tal como puede verse en las siguientes líneas:

```
import threading
import time
```

Debajo de la sección de importaciones del template driver es creada la clase que define al driver. Ella se deriva de sendas clases (*DeviceBase* y de la clase *Thread* del módulo *threading*):

```
class TemplateDevice(DeviceBase, threading.Thread):
```

Cuando una instancia de este driver es creada desde el *Device Driver Manager*, la ejecución comenzará con el método `__init__()`, donde las siguientes tareas son completadas:

1. Los valores `count_init` y `update_rate` son creados como objetos *Setting* y son agregados a una lista llamada `settings_list`.
2. The `ChannelSourceDeviceProperty` objects `counter`, `adder_total`, `counter_reset`, `global_reset`, `adder_reg1`, y `adder_reg2` son creadas y agregadas a una lista llamada `property_list`.
3. El *constructor DeviceBase* es llamado y se le pasan referencias al objeto instanciado (`self`), al nombre de instancia actual, al *core services object*, y a las *settings_list* y *property_list*.
4. El *Thread* se inicializa llamando al método `threading.Thread's __init__()`
   Cuando `__init__()` retorna, el *Device Driver Manager* intentará iniciar el driver. Antes que un driver sea iniciado, sus configuraciones deben establecerse. Esto ocurre en el método `apply_settings()`.

Los Settings se almacenan dentro de un registro especial gestionado por el objeto *DeviceBase* (que hereda esta funcionalidad del objeto *SettingsBase*). El objeto *Settings* tiene dos estados: *pending settings* y *running settings*.

La implementación del método `apply_settings()` dentro del driver template realiza como mínimo estas operaciones para importar nuevos seteos:

1. Un llamado a `SettingBase.merge_settings()` se hace para unir los nuevos *settings* del registro *pending* y los existentes *settings* del registro *running* dentro del registro *pending*.
2. Un llamado a `SettingsBase.verify_settings()` se hace para parsear los *settings* desde el registro *pending* dentro de tres listas de settings: una lista de aceptados, una lista de rechazados, y una lista de requeridos que no se encontraron.

Luego de que los settings son aplicados, es llamado el método `start()`. En el caso del driver template el método `start()` es muy simple: este llama a la función `start()` del módulo

[4] Para mayor información sobre funciones de la library estándar de Python, vea: http://docs.python.org/library/

`threading.Thread` que iniciará el *thread* de nuestro device driver's iniciando la ejecución desde el método `run()` del driver.

Cuando el driver comienza su *loop* de ejecución con el método `run()`, lo primero que hace es llamar a la función `prop_set_global_reset()`. La implementación de esta función resetea las propiedades `counter`, `adder_total`, `adder_reg1`, and `adder_reg2` usando el método `property_set()`. La función `prop_set_global_reset()` es también llamada como un *channel callback*.

Continuando con la ejecución de `run()`, un *while loop* maneja la continua ejecución del *thread*. En el comienzo del loop, un flag se chequea para ver si el thread debe detenerse. Si el loop continúa, el contador está listo para usarse desde el canal *counter* con el método `property_get()`. Luego, una nueva muestra es construida conteniendo el valor del contador mas uno y este nuevo objeto sample es escrito de vuelta al canal usando el método `property_set()`. Finalmente, el método `time.sleep()` es llamado por el valor de tiempo update_rate en segundos antes que el loop se ejecute otra vez. Así es como el *counter channel* es actualizado. Por supuesto, hay más bloques de ejecución dentro del driver y varios métodos callback: `prop_set_counter_reset()`, `prop_set_global_reset()`, and `prop_set_adder()`.

Cada uno de esos métodos callback es llamado en respuesta a un método `set()` utilizado sobre un canal por otro driver o presentación. Los callbacks son registrados sobre un canal durante la definición del canal (en este driver, esto está hecho en el método `__init__()`) pasando un argumento `set_cb` al constructor de un objecto *ChannelSourceDeviceProperty*. Como estos métodos callback son todos muy similares, solo se analizará uno: `prop_set_adder()`.

El método `prop_set_adder()` es usado dos veces como un callback: una para el canal *adder_reg1* y otra para el canal *adder_reg2*. El método toma dos parámetros: el nombre del registro a ser seteado y un objeto *Sample* conteniendo el valor floating-point a ser almacenado en el canal. Cómo puede el mismo método callback ser usado para dos canales distintos?

Si brevemente nos referimos a la definición del canal para *adder_reg1* y *adder_reg2* en el método `__init__()` de este driver, veremos que el argumento `set_cb` pasado al objeto *ChannelSourceDeviceProperty* no es simplemente un nombre de una función callback sino un expresión ***lambda***. Una expresión ***lambda*** [5] retorna una nueva función. En este caso, una nueva función es definida para los canales *adder_reg1* y *adder_reg2* que puede llamar a `prop_set_adder()` con el respectivo nombre del registro correcto como argumento.

Con respecto al método `prop_set_adder()`, su operatoria es simple. Primero, el nuevo objeto *Sample* de punto flotante es seteado al canal correcto. Luego, el valor para `adder_reg1` y `adder_reg2` se recupera de cada canal. Finalmente, el canal `adder_total` es seteado usando un nuevo objeto *Sample* construido poniendo ambos valores de esos canales juntos.

El último metodo que consideraremos en este driver es el método `stop()` que opera seteando un flag para interrupir el loop en el método `run()`. El método *`stop`*() potencialmente puede ser llamado por el *driver manager* en respuesta a un requerimiento del usuario. Si el método `stop()` retorna `True`, el *Device Driver Manager* asume que el driver fue detenido, los recursos son liberados y el driver puede reiniciarse de nuevo llamando a su método `start()`.

[5] Para mas información ver : http://www.python.org/doc/2.5.2/ref/lambdas.html.

2.2.3. Resúmen de cómo se ejecuta un Device Driver

1. Un device driver es cargado y su método `__init__()` es llamado.
2. Las *Settings* son aplicadas llamando al método `apply_settings()`.
    1. Un llamado a `SettingBase.merge_settings()` se hace para unir los nuevos *settings* del registro *pending* y los existentes *settings* del registro *running* dentro del registro *pending*.
    2. Un llamado a `SettingsBase.verify_settings()` se hace para parsear los *settings* desde el registro *pending* dentro de tres listas de *settings*: una lista de aceptados, una lista de rechazados, y una lista de requeridos que no se encontraron.

2.3. XBee Device Drivers

2.3.1. Introducción sobre el stack de drivers XBee

Un conjunto de abstracciones basadas en eventos se proveen con el propósito de realizar drivers que estén conectados a través de el módulo XBee integrado en el gateway. Esta colección de eventos y abstracciones es conocida como *"iDigi Dia XBee Device Driver Stack"*.

El *XBee Device Driver Stack* está dividido en los siguientes componentes:

- ***XBee Device Drivers :*** son instancias que representan sensores XBee individuales, *XBee adapters*, u otros dispositivos wireless conectados a la red XBee.
- El ***XBee Device Manager*** es una instancia para gestionar el estado y comunicación para todos los dispositivos XBee que son administrados por el gateway.
- ***XBee Configuration Blocks***: abstracciones especializadas usadas por los *XBee Device Drivers* y pasados a los *XBee Device Manager* para definir configuraciones y parametros de *sleeping* para dispositivos wireless.

Estos componentes son organizados en el file system de Dia de la siguiente forma:

- "`src/devices/xbee/xbee_devices`": implementaciones de drivers XBee específicos.
- "`src/devices/xbee/xbee_device_manager`": archivos relacionados con el *XBee Device Manager*, incluyendo la especificación de eventos XBee.
- "`src/devices/xbee/xbee_config_blocks`": implementación de la abstracción *XBee Configuration Block*.

2.3.2. Teoría de Operación

Un driver que usa el stack XBee esencialmente no es diferente de cualquier otro driver del sistema DIA. Aquel contendrá *settings y propiedades de canal* igual que cualquier otro driver. En realidad solo diferirá en que utilizará el stack de drivers XBee.

Un driver XBee tiene tres fases distintivas de operacion:

1. La fase de inicialización, en donde:
   - La instancia del **XBee Device Driver** es registrada con una instancia de **XBee Device Manager**.
   - Uno o mas objetos de **especificación de eventos** son dados al **XBee Device Manager**; estos objectos informan al Manager de que tipos de eventos se trata y de que manera el driver debe ser invocado.
   - Los **Configuration blocks** se pasan **al XBee Device Manager** informando la manera en que el *Device Manager* debería configurar el XBee: ej. que parámetro *DDO*[6] debería setear y para que hardware, que parámetro de *sleep* debería usar, etc.
2. La fase de configuración donde el método `xbee_device_configure()` del *XBee Device Manager* es llamado desde el device driver informando que el XBee Device Manager debe empezar a planificar e intentar aplicar los bloques de configuración al dispositivo remoto.
3. La fase de ejecución, que comienza luego de que todos los bloques de configuraciones sean marcadas como compleadas con éxito. Desde la fase de ejecución, los *callbacks* son hechos desde el *XBee Device Manager* a la instancia del XBee Device sobre la base de las **especificaciones de eventos** que fuera registrada durante la fase de inicialización o por *callbacks* adicionales o eventos programados registrados durante la ejecución.

2.3.3. Application Programming Interface (API)

Las instancias de los drivers XBee pueden utilizar una API predefinida cuyos métodos se obtienen de una instancia de la clase *XBeeDeviceManager*. Estos se explican abajo agrupados por funcionalidad:

2.3.3.1. XBee Device Driver Registration

Estas funciones son llamadas cuando una instancia de XBee Device Driver necesita registrarse o anular su registración ella misma .

- `xbee_device_register()`
- `xbee_device_unregister()`

2.3.3.2. XBee Event Callback Specifications

Las especificaciones de eventos informan a la instancia del *XBee Device Driver Manager* para que evento una instancia *XBee Device* solicita ser invocada mediante un *callback*.

La especificación de eventos está definida en
`"xbee_device_manager/xbee_device_manager_event_specs.py".`

- `xbee_device_event_spec_add()`
- `xbee_device_event_spec_remove()`

[6] DDO: Digi Device Object

2.3.3.3. XBee Configuration Blocks

Los bloques de configuración definen que parámetros del Digi Device Object (DDO) o parámetros sleep, necesitan ser especificados o que llamados callbacks a un driver necesitan ser realizados en tiempo de configuración. En este sentido, el sistema está diseñado para continuar y/o reintentar luego en los casos en que un dispositivo no debe configurarse, o cuando un dispositivo es muy difícil de configurar (ej.: si está en sleep la mayor parte del tiempo).

Todos los bloques de configuración están definidos en el directorio "*xbee_config_blocks*".

2.3.3.4. XBee DDO Inmediatos

El seteo de un parámetro DDO sobre un dispositivo consume temporariamente un recurso DDO sobre el XBee en el gateway. Realizando el DDO requests a través de estas funciones, los recursos utilizados en los requerimientos a dispositivos remotos pueden optimizarse:

- `xbee_device_ddo_get_param()`
- `xbee_device_ddo_set_param()`

2.3.3.5. Eventos programados

Algunos drivers necesitan que el gateway inicie un *polling* (como el envío de serial data requests periodicamente a un *XBee RS-485 adapter* remoto). Las siguientes funciones proveen primitivas para planificar estos requests.

- `xbee_device_schedule_after()`
- `xbee_device_schedule_cancel()`

2.3.3.6. Transmision de datos programada

Los datos se reciben asíncronamente via *callbacks* desde las especificaciones de eventos. La transmición de datos es también realizada asíncronamente (por ej. con llamados no bloqueantes).

- `xbee_device_xmit()`

2.3.4. Tipos de especificaciones de eventos

Las especificaciones de eventos se registran ante una instancia de *XBee Device Manager* creando primero una instancia de especificación de evento y a continuación llamando el método `xbee_device_event_spec_add()` de la instancia del *XBee Device Manager*.

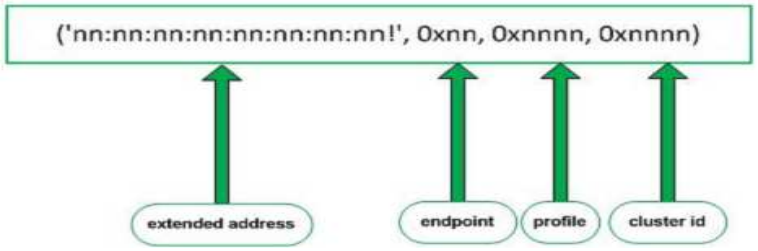
Los diferentes tipos de especificaciones de eventos se detallan abajo:

2.3.4.1 XBeeDeviceManagerRxEventSpec:

Registrar uno o mas de estos objetos de especificacion de eventos de recepción causará que el *XBee Device Manager* realice un llamado a una función *callback* cuando una trama sea recibida por el dispositivo XBee interno y aquella sea coincidente con un criterio de direccionamiento especificado. La función *callback* será procesada si y solo si el device driver ya ha entrado en la fase de *run-time*.

El *callback* se establece usando el método `cb_set()` de una instancia de `XBeeDeviceManagerRxEventSpec`.

El criterio de coincidencia de la dirección es establecido por el método `match_spec_set()` que toma dos parámetros: un registro de direcciónamiento (*address tuple*), y una máscara. El *address tuple* toma la siguiente forma:



La máscara especifica que porción de aquella tupla de dirección debe coincidir. Es especificada como una tupla de cuatro valores booleanos. Por ejemplo, para coincidir solo el campo *extended address* la tupla debería decir:

```
(True, False, False, False)
```

Para exigir la coincidencia del *extended address*, *endpoint*, *profile* y *cluster id* :

```
(True, True, True, True)
```

2.3.4.2 XBeeDeviceManagerConfigRxEventSpec:

Esta especificación de evento es idéntica a la de arriba (clase XBeeDeviceManagerRxEventSpec) con excepción de que procesará *callbacks* si y solo si el dispositivo está en la fase de configuración. Esta funcionalidad se agrega solamente para ser usada en conjunto con una XBeeConfigBlockConfigCb (*XBee Driver Stack XBee Custom Configuration Block)* (ver sección E: Configuration Block Types).

2.3.4.3 XBeeDeviceManagerRunningEventSpec:

El registro de esta especificación de evento XBee causará que el XBee Driver Stack llame a la función callback espeficicada cuando el driver pase de la fase de configuración a la fase de ejecución. Este evento es útil para realizar una acción cada vez, justo después de que un dispositivo sea inicializado y configurado. Un ejemplo en el cual esto es útil es en la interrogación de un dispositivo que normalmente solo reporta excepciones con la intención de generar una muestra inicial para un canal de un device driver.

2.3.5. Configuration Block Types

Hay un número de tipos de bloques de configuración que pueden ser agregados a un driver XBee. Los bloques de configuración se guardan en el directorio *src/devices/xbee/xbee_config_blocks*.

Cuando un bloque de configuración es creado y configurado, este es agregado a una instancia de dispositivo *XBee Device Manager* mediante el llamado al método `xbee_device_config_block_add()` del *XBee Device Manager*. Después de que todos los bloques de configuración fueron agregados, el driver debe llamar al método `xbee_device_configure()` para promover al dispositivo desde la fase de inicialización a la fase de configuration por eso, el XBee Device Manager puede empezar a ejecutar los bloques de configuración de los dispositivos. Los bloques de configuración puede verse listados a continuación.

2.3.5.1. XBeeConfigBlockDDO

El bloque XBeeConfigBlockDDO acepta cualquier número de pares de letras como parámetros representando registros XBee y sus correspondientes valores. Cualquier mnemonico DDO excepto los correspondientes a parámetros de sleep (ej.: “SP”, “SN”, etc.) pueden aplicarse usando este tipo de bloque de configuración. Los parámetros se agregan al bloque usando la función `add_parameter()`.

2.3.5.2. XBeeConfigBlockSleep

El bloque de configuración XBeeConfigBlockSleep provee un útil *front-end* para el cálculo y aplicación de parámetros de *sleep* a los dispositivos remotos XBee. Todos los modos de *sleep* XBee pueden setearse usando este tipo de bloque incluyendo *pin hibernation*, *cyclic sleep*, *cyclic sleep with pin wake*, y desabilitar el *sleep*.

Para configurar un XBee para funcionar en modos *sleeping* solamente se debe llamar a `sleep_mode_set()`. La función toma tres parámetros: `time_asleep_ms`, `time_awake_ms` y `enable_pin_wake`. `time_asleep_ms` especifica el número de milisegundos que el XBee debe dormir entre ciclos de actividad, `time_awake_ms` especifica el número de milisegundos que el XBee permanece despierto después de un período de inactividad, y `enable_pin_wake` informa al XBee si debería atender el pin `Sleep_RQ` para despertar o dormir.

El codigo en el `XBeeConfigBlockSleep` se dedica a enviar los parámetros DDO intentando hacer coincidir los requerimientos con los tiempos de sleep lo mas exactamente posible. Para hacerlo factoriza el tiempo de sleep solicitado en los parámetros SP y SN y setea el parámetro SO para habilitar el *deep sleep* en el módulo de XBee.

2.3.5.3. XBeeConfigBlockConfigCb

El XBeeConfigBlockConfigCb habilita a crear un callback personalizado que será llamado cuando el *Manager* promueva un dispositivo dentro de la fase de configuración (ej.: cuando el dispositivo está listo para ser configurado). Esto es un *callback* de utilidad si el dispositivo necesita ser inicializado con un número de parámetros no-DDO (ej.: comandos seriales o parámetros ZigBee ZDO o ZCL).

Aunque este *callback* será llamado en el contexto de su propio *thread* es útil para registrar un `XBeeDeviceManagerRxConfigEventSpec` para llamar al driver al recibir un mensaje, poner el mensaje en una cola de Python, y bloquear esa cola en la función *callback* para sincronizar los mensajes recibidos. Igualmente, las funciones `xbee_device_xmit()` de las instancias `XBeeDeviceManager` pueden usarse para transmitir mensajes desde el *callback*.

El *callback* debería completarse lo más rápido posible, es decir, solo tareas de configuración deberían realizarse con estos llamados para dejar los recursos de los *thread* de configuración disponibles para otros drivers.

2.3.5.4. XBeeConfigBlockFinalWrite:

Este bloque de configuración no está pensado para ser usado por el desarrollador del driver. Se añade automáticamente al extremo de la cadena de bloques de configuración. Agrega un último parámetro DDO: “WR” (write) como comando para salvar todos los parámetros DDO en la memoria no volátil de los XBee’s.

3. Práctica

3.1. Comenzar a escribir el propio XBee Device Driver

Una buena forma de comenzar a escribir el código de su propio driver dentro del framework Digi DIA sería utilizando como base un driver ya existente. Algunos de estos objetos podrían ser los drivers para la placa *XBIB* (*XBee XBIB development board)* o el driver para el *XBee DIO adapter*, o cualquier otro que se aproxime a nuestras necesidades. Los drivers citados se encuentran respectivamente en la ruta:

- `src/devices/xbee/xbee_devices/xbee_xbib.py`
- `src/devices/xbee/xbee_devices/xbee_dio.py`

En estos archivos se puede encontrar todo lo necesario para registrarse con un *XBee Device Manager*, crear y registrar bloques de configuración, crear y registrar eventos, y los métodos necesarios para el procesamiento de registros callbacks, creación de objetos Sample, e inserción de esos samples dentro de los canales.

1º) Comenzamos nuestro trabajo creando un nuevo proyecto[7] para el framework DIA. Para esto, desde la interfaz de usuario del IDE vamos a *File* **->** *New* **->** *iDigi Dia Project* . Luego, además de indicar el nombre del proyecto y elegir el intérprete Python 2.4.x debemos tildar la opción *Include iDigi Dia source Code in project* de esta manera tendremos la posibilidad de acceder a todos los archivos fuente del proyecto desde el IDE.

2º) A continuación, desde el explorador de proyectos abrimos el directorio `src/devices/xbee/xbee_devices` y eligiendo el driver **`xbee_dio.py`** tal como lo habíamos sugerido, creamos una copia del mencionado archivo fuente en la misma ubicación en que este se encuentra. Dado que vamos a crear un driver para la placa *XBoard* (ver 1.1. Objetivos) llamaremos al nuevo archivo **`xbee_xboard.py`**

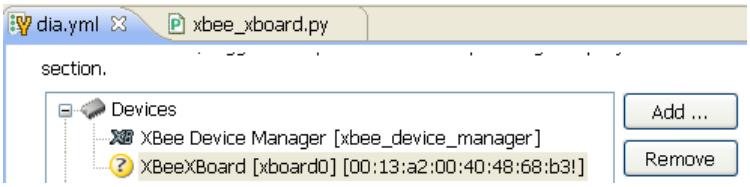
3º) La primera acción que realizaremos dentro del archivo será la de modifiar el nombre de la clase que este contiene, entonces buscamos la línea "`class` **`XBeeDIO`**`(XBeeBase):`" y reemplazamos el nombre de la clase por **`XBeeXBoard.`**

4º) Abrimos el archivo *dia.yml* con el editor - no el gráfico, sino su versión para texto - y bajo la sección `devices` agregamos:

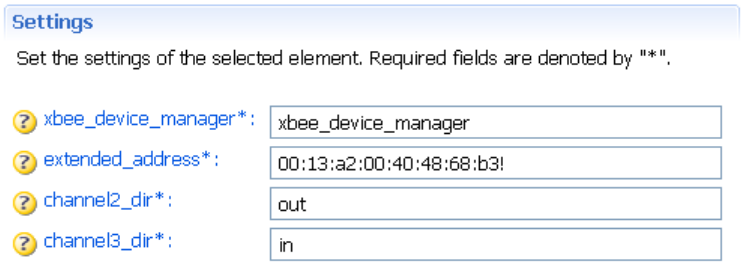
```
- name: xboard0
  driver: devices.xbee.xbee_devices.xbee_xboard:XBeeXBoard
  settings:
      xbee_device_manager: "xbee_device_manager"
      extended_address: "00:13:a2:00:40:48:68:b3!"
      channel2_dir: "out"
      channel3_dir: "in"
```

estos parámetros los podemos editar luego desde la ventana gráfica, pero en este punto ya tenemos todas las piezas en posición para comenzar a definir el comportamiento del driver. Ahora, si pasamos al editor gráfico de este mismo archivo, veremos aparecer nuestro nuevo driver con un feo signo de pregunta que no debe alarmarnos para nada, pero ahí lo tenemos:

[7] Para mayores detalles de configuración del proyecto y otros aspectos de configuración del IDE consultar la nota CoTC-003 o bien la ayuda en línea del entorno de desarrollo.



Un poco a la derecha se presentarán los parámetros que acabamos de agregrar mediante el editor de texto:



De ahora en más debemos realizar la definición del driver de acuerdo a las necesidades del hardware. Recordemos una vez mas que estamos abstrayendo las particularidades de un dispositivo físico; en esta oportunidad se trata de la placa *XBoard* conteniendo a un *XBee ZB End Device*, por lo tanto analizando las características de esta placa y del módulo que le instalamos, observamos que debemos contemplar la manipulación de la interfaz de I/O, por lo tanto al menos debemos manejar los registros D0, D1, D2, D3, D4, D5 y P0.
Ya que nuestra plaquita maneja dos entradas analógicas (D0, D1), dos líneas configurables (D2,D3), una entrada fija (D4), y dos salidas fijas de monitoreo (D5 y P0)

Otro aspecto que tendríamos que observar es que al ser un *End Device*, tenemos que admitir entre sus configuraciones la del modo de bajo consumo, es decir que habría que considerar a los registros SM, ST, SP, SN y SO al momento de definir el driver, sin embargo, los detalles de la operatoria de estos registros están ocultos para el desarrollador, y si así lo queremos, podemos ignorar los detalles de configuración siendo necesario solamente el suministro de los siguientes valores:
`sample_rate_ms, sleep_time_ms, awake_time_ms`

Finalmente, también nos interesa la posibilidad de configurar el dispositivo remoto para la transmisión de muestras de I/O, para lo cual debemos considerar a los registros IR e IC para las configuraciones.

3.2. Las configuraciones de inicialización del driver

Ya tenemos que saber que canales manejará nuestro driver y debemos tener decidido también que parámetros serán configurables por el usuario y cuales serán fijos. Entonces separamos, por un lado los canales fijos[8], en nuestro caso: D0, D1, (analógicos ambos) y D4.
Por el otro lado nos quedan los configurables: D2 y D3, y también son configurables los parámetros de *Sleep*, a saber: `sample_rate_ms, sleep_time_ms, awake_time_ms.`
Ahora bien, todos estos parámetro configurable estarán representado por respectivos objetos *Setting*, como por ejemplo:

```
Setting(name='channel2_dir',type=str,required=True,default_value="out"),
```

[8] Las líneas D5 y P0 también son fijas, pero en este caso no constituyen canales para el driver, sino que son funcionalidad interna propia del módulo remoto.

Y todos los objetos *setting* se ubican dentro de una lista :

```
settings_list = [
      Setting(...),
      Setting(...),
      ... ]
```

Estos objetos Setting interactúan desde el driver de manera tal que permiten que en tiempo de ejecución, el driver pueda tomar los parámetros suministrados por el usuario bajo la forma de un archivo de configuración, el Dia.YML. Entonces nótese cómo encajan las piezas[9]:

Una vez resuelto el asunto de los canales configurables nos resta ocuparnos, de los canales fijos, recordemos que eran D0, D1, y D4. Para esto disponemos de otra lista, la de propiedades, cada canal definido a priori se abstrae en una propiedad establecida por un objeto `ChannelSourceDeviceProperty` tal como puede verse a continuación:

```
property_list = [
    ChannelSourceDeviceProperty(
        name = 'channel0', type=int,
        initial = Sample(timestamp=0, value=0, unit='int'),
        perms_mask = DPROP_PERM_GET,
        options = DPROP_OPT_AUTOTIMESTAMP),
    ChannelSourceDeviceProperty(
        name = 'channel1', type=int,
        initial = Sample(timestamp=0, value=0, unit='int'),
        perms_mask = DPROP_PERM_GET,
        options = DPROP_OPT_AUTOTIMESTAMP),
```

[9] En favor de la claridad se han repetido intencionalmente imágenes de este mismo trabajo.

```
            ChannelSourceDeviceProperty(
                name = 'channel4', type=bool,
                initial = Sample(timestamp=0, value=False, unit='bool'),
                perms_mask = DPROP_PERM_GET,
                options = DPROP_OPT_AUTOTIMESTAMP),
        ]
```

Ya tenemos todo lo necesario para la inicialización del driver, pero seguramente nos estaremos preguntando donde va toso esto:
Dentro de la clase **`XBeeXBoard`** (recuérdese el 3º paso en 3.1.) que define las instancias de nuestro driver se encuentra la función **`__init__`**`()` y dentro de esta se encuentran los elementos que acabamos de describir. Y para finalizar con esta etapa de inicialización, en el final de esta función se realiza un llamado a la función homónima de la clase base, pasandole las referencias a los objetos locales que acabamos de crear:

```
XBeeBase.__init__(self, self.__name, self.__core, settings_list, property_list)
```

3.3. Las acciones de inicialización del driver

Hasta acá, y dentro del método `__init__` nos vimos ocupados en tareas de configuración estáticas. Ahora nos ocuparemos del código mediante el cual se realizan las acciones de inicialización del driver, esto es, dentro de la función **`start`**`()`, allí se realizan las siguientes acciones que serán utilizadas tal como se nos presentan en el driver origial[10]:

1. Obtener el nombre del XBee Manager desde las configuraciones (*Settings Manager*)
2. Autoregistrar la propia instancia del driver con la instancia del *XBee Device Manager*.
3. Obtener la dirección extendida del dispositivo
4. Crear una especificación *callback* para nuestro *address*, perfil *Digi XBee endpoint* y *sample cluster id*.
5. Crear una especificación *callback* que llame a este driver cuando nuestro dispositivo haya dejado el estado de configuración y haya pasado al estado de ejecución.
6. Crear un bloque de configuración DDO (Digi Device Object) para el dispositivo.
7. Obtener la dirección extendida del gateway
8. Establecer el gateway como destino de las muestras de IO.

Reiteramos que todo el código involucrado en la realización de estos 8 pasos nos sirve tal cual está escrito en el driver original sobre el cual estamos trabajando, pero a continuación de este código comienza la sección de definición de las líneas de I/O y aquí es donde debemos dedicar la mayor atención. Las tareas que nos ocupan aquí son tres:

a) Recuperar los datos de configuración desde los objetos Settings. Lo hacemos con llamados a métodos como estos:

```
SettingsBase.get_setting(self, '<setting_name>')
```

b) Aplicar las configuraciones para crear y definir los canales restantes (los configurables[11]). Lo cual realizamos con llamados similares a estos:

[10] No detallaremos el código involucrado para no extendernos demasiado. Solo nos detendremos en el código cuando introduzcamos nuestras propias modificaciones.
[11] Recordar que en (3.2.) definimos una `property_list[..]` conteniendo a los canales fijos, ahora hay que agragar los canales restantes (los configurables) a esa misma lista.

```
self.add_property(
    ChannelSourceDeviceProperty(
         name = '<channel_name>', type=<channel_type>,
         initial = Sample(timestamp=0, value=False,
         unit='channel_unit'),
         perms_mask = DPROP_PERM_GET,
         options = DPROP_OPT_AUTOTIMESTAMP)
    )
```

c) Aplicar las configuraciones a los registros del dispositivo remoto, tanto las obtenidas de los objetos *Settings* como las conocidad a priori. Lo cual haremos fácilmente invocando al método `xbee_ddo_cfg.add_parameter()`. Por ejemplo:

```
xbee_ddo_cfg.add_parameter('IR', SettingsBase.get_setting(self, 'sample_rate_ms')  )
```

Otro ejemplo:

```
# Set D5 as ASSOCIATED INDICATOR
xbee_ddo_cfg.add_parameter('D5', 1)
```

Para resumir, debemos entonces recoger todas las *Settings* que fueran definidas al comienzo en la función `__init__`:

```
SettingsBase.get_setting(self, '<setting_name>')
```

Y con ellas definir todos los canales restantes:

```
self.add_property( ChannelSourceDeviceProperty([...] )
```

Y todas las configuraciones de registros XBee según corresponda:

```
xbee_ddo_cfg.add_parameter(<XBee_register_name>, <register_value>)
```

Y al finalizar con todas las llamadas "**`xbee_ddo_cfg`**`.add_parameter()`" necesarias debemos invocar al método:

```
self.__xbee_manager.xbee_device_config_block_add(self, xbee_ddo_cfg)
```

Con este método estamos enviando todas las configuraciones al *Device Manager* para que este se encargue del trabajo pesado de configurar el equipo, recuérdese que muy probablemente los equipos *End Devices* podrían estar dormidos y gracias a las facilidades que ofrece el framework podemos desentendernos de esos detalles.

Lo último que queda por mencionar sobre esta función está relacionada con toda la configuración de *Sleep* para ello existen un bloque de configuración especializado que se configura con los parámetros mencionados más arriba: `sample_rate_ms, sleep_time_ms, awake_time_ms.` Los cuales se obtienen una vez más de los objetos *Setting* .
No nos extenderemos explicando este bloque ya que dentro de las intenciones de este trabajo no requerimos ninguna modificación al respecto y podemos entonces mantener el código tal cual está en el driver original.

3.4. Tareas en tiempo de ejecución

Una vez que completamos el código necesario para configurar el dispositivo remoto y ponerlo en marcha con sus tiempos de Sleep y todo, nos queda ocuparnos del comportamiento en tiempos de ejecución del driver. Para esto nos concentraremos conceptualmente en dos tipos de eventos. Por un lado, el arribo de muestras provenientes del XBee remoto, por el otro, el envío de comandos hacia el dispositivo remoto. Tenemos entonces sendas funciones para definir el comportamiento deseado:

a) `sample_indication(self, buf, addr, force=False)`

b) `set_output(self, sample, io_pin)`

Comenzamos por describir la función **`sample_indication()`** . Este método es invocado por el Device Manager con el cual está registrado el driver. La llamada se realiza cada vez que, precisamente, ha arribado una nueva muestra al dispositivo gateway. Como puede inferirse, es la instancia del objeto *Device Manager* la que maneja estos eventos y mediante llamadas *callback* notifica a nuestro driver, pasándole, como puede observarse los siguientes parámetros:

- `buf`: conteniendo el string con la muestra que nos interesa
- `addr`: la dirección del dispositivo remoto
- `self`: una referencia a la propia instancia
- `force`: indicando si hay que forzar una muestra

Para que todo este procedimiento de recepción de eventos y notificación no quede en la oscuridad podemos agregar que este mecanismo de llamadas *callback* se ha definido dentro de la función "**`start`**`(self)`"
Véase a continuación cómo se realiza la creación de una instancia de objeto de *especificación de eventos* desde el *Device Manager* y luego mediante el método `.cb_set()` (*callback se*t) se establece una referencia al método "`sample_indication`" correspondiente a la instancia del driver actual (`self`).
Finalmente, se indica cómo fltrar las samples que nos pertenecen mediante .match_spec_set() specificando nuesta dirección MAC (`self.__extended_address`) y además el número de *endpoint*, el *perfil de XBee de Digi* y el *sample cluster id*. Cabe aclarar que no todos estos parámetros son necesarios para definir la máscara de filtrado del emisor del mensaje, de hecho sería suficiente con especificar `self.__extended_address` y a continuación `(True, False, False, False)` en la máscara:

```
# Create a callback specification for our device address, endpoint
# Digi XBee profile and sample cluster id:
 xbdm_rx_event_spec = XBeeDeviceManagerRxEventSpec()
 xbdm_rx_event_spec.cb_set(self.sample_indication)
 xbdm_rx_event_spec.match_spec_set(
     (self.__extended_address, 0xe8, 0xc105, 0x92),
     (True, True, True, True))
```

Volviendo a la función **`sample_indication()`**: lo que haremos aquí, tal como se desprende de los prámetros es recibir un string con la muestra actual (`buf`) y básicamente realizaremos el parsing de ese string mediante funciones especializadas disponibles en la API como por ejemplo la función `parse_is(<str>)` que nos devuelve un string conteniendo una lista de pares "nombre=valor"

De tal forma, mediante la sentencia: `io_sample = parse_is(buf)` obtenemos dicha lista con todos los pares nombre=valor de todos los canales disponibles. Un ejemplo de nombre=valor sería D1=573. Corresponiente al canal analógio 1 (D1) con valor raw 573.

Con `io_sample.has_key("D1")` podemos preguntar por ejemplo, si el string a parsear contiene la clave `"D1"`. Y en caso afirmativo, obtener el valor con algo así como:
`val = int(io_sample["D1"])`

Pues bien, lo que tenemos que completar aquí es entonces, la extracción de todos los valores que nos interesan para hacer algo con ellos y a continuación actualizar el valor de la propiedad, es decir, del canal correspondiente mediante el método siguiente:

```
self.property_set(name,  Sample(now, val, unit))
```

Así, el valor estará siempre actualizado, por ejemplo, para el uso de las presentaciones que acceden a estos valores en forma asíncrona.

Por último, y con esto terminamos el análisis de nuestro driver, nos quedaría examinar la función **`set_output(`***`self`***`, sample, io_pin).`** Tal como su nombre lo indica, este método se utiliza para establecer el valor de salida para los canales que fueron configurados como tales.
El método recibe una referencia a la instancia del propio objeto que lo contiene (*`self`*), un string con el valor a establecer en el canal (`sample`) y el número del canal en cuestión (`io_pin`).
Con todo esto lo que hacemos es convertir la cadena de caracteres *sample* al formato adecuado, por ejemplo:

```
new_val = bool(sample.value)
```

Y convertir el número de pin en el nombre del registro XBee a escribir, por ejemplo:

```
cmd = 'D%d' % io_pin
```

También podemos obtener el valor último de la propiedad correspondiente con algo similar a esto:

```
property = "channel%d" % io_pin
old_val = self.property_get(property).value
```

para después modificar el valor, solo si este ha cambiado (`if new_val != old_val:`):

```
self.__xbee_manager.xbee_device_ddo_set_param(
                                                self.__extended_address,
                                                cmd,
                                                ddo_val,
                                                apply=True )
```

4. Conclusión

Este trabajo se estructuró en dos grandes bloques: uno teórico y otro práctico. En el primero recorrimos los fundamentes esenciales sobre los cuales se apoya un device driver dentro del framework Digi DIA. A su vez, dentro de esta división teórica abarcamos la descripción de dos tipos de drivers, uno estándard y otro, que además de las características estándar, agrega la arquitectura del stack de drivers XBee. Para este último driver existe una API (application programming interface) que provee las funcionalidades necesarias para el trabajo con redes XBee. En el segundo bloque, encaramos la realización de nuestro propio driver construyéndolo sobre la base de algún driver preexistente en el framework. Enfocamos el análisis del código sobre las funciones del driver original que requerían ser modificadas para adaptarlas a nuestras necesidades. Obviamos intencionalmente el análisis del código que fuera genérico o que nos fuera útil en su forma original. De esta manera perseguimos el propósito de entrar rápidamente en contacto con la arquitectura del framework, seleccionando y dejando de lado los detalles que puedan esperar, para concentrarnos así en los aspetos que faciliten la familiarización con las herramientas de desarrollo.