ESCUELA POLITÉCNICA DEL EJÉRCITO

DEPARTAMENTO DE ELÉCTRICA Y ELECTRÓNICA

CARRERA DE INGENIERÍA EN ELECTRÓNICA
REDES Y COMUNICACIÓN DE DATOS

PROYECTO DE GRADO

EMULADOR DE UN SISTEMA DE COMUNICACIONES UTILIZANDO TECNOLOGIA SDR.

JUAN FRANCISCO QUIROZ TERREROS

SANGOLQUI - ECUADOR

2010

CERTIFICACIÓN

Certificamos que el presente proyecto de grado fue realizado en su totalidad por el Sr. Juan Francisco Quiroz Terreros bajo nuestra dirección.

________________	________________
Ing. Carlos Romero	Ing. Julio Larco
DIRECTOR	**CODIRECTOR**

RESUMEN

El presente proyecto consiste en la implementación de una plataforma universal de procesamiento banda base y radiocomunicaciones utilizando herramientas de desarrollo libre, tanto en *hardware* con la USRP así como en *software* con GNU radio.

La USRP realiza las funciones de etapa frontal de radiofrecuencia mediante las tarjetas opcionales RFX900 y RFX2400, gracias a esto la USRP es capaz de trabajar en las bandas de frecuencias de 900Mhz y 2400Mhz. El tratamiento de las señales recibidas por la etapa de RF se realiza en forma digital mediante conversores ADC, para posteriormente ser procesadas en banda base por los DDCs implementados en un FPGA Altera Cyclone. Para los datos a ser transmitidos, el proceso se invierte, esta vez, utilizando DUCs retornamos al valor de frecuencia intermedia, para luego mediante DACs volver al dominio analógico y enviar la señal a la antena.

GNU radio provee todas las herramientas necesarias para generar, modular, demodular, y filtrar digitalmente las señales recibidas por la USRP, permitiendo implementar un radio definido por software, el equipo es capaz de transmitir y recibir señales moduladas digitalmente mediante dbpsk, dqpsk, d8psk y gmsk.

La plataforma se implementó en tres distribuciones de Linux, Mandriva, openSUSE y Ubuntu, para determinar cuál es la más amigable para futuras investigaciones.

A mis padres, Miguel y Fátima,
mis hermanos Byron y Leslie de todo
$r = 1 - \sin\theta$

AGRADECIMIENTOS

Agradezco a los ingenieros Carlos Romero y Julio Larco por haberme dado la oportunidad de realizar este trabajo tan entretenido, curioso e interesante. No puedo olvidar a mi amigo Diego de la Torre por su apoyo a todas mis consultas con LaTeX, de verdad, gracias.

Juan Quiroz

PRÓLOGO

Los radios definidos por software aparecen como respuesta a la constante evolución de la tecnología, que día a día busca nuevas y mejores maneras de transportar la información, sin embargo, esta busqueda tiene como consecuencia una habitual batalla de nuevas tecnologías que buscan instituirse como estándar y dominar un segmento de mercado. Mediante la *Universal Software Radio Peripheral* y el software GNU radio se espera solucionar el problema ya expuesto, gracias a que podemos implementar una plataforma flexible, capaz de cambiar de acuerdo a nuestras necesidades, el mismo equipo puede funcionar como un radio AM, FM, GPS, GSM, etc, todo en uno, evitando comprar un equipo propietario para cada tecnología.

Aunque el proyecto GNU radio pone a nuestro alcance múltiples herramientas para realizar investigaciones en el campo de las comunicaciones inalámbricas, existe un gran problema debido a la falta de información acerca de este tema, y en caso de conseguirla, esta no es actualizada. Es por esto que el presente trabajo busca incentivar el estudio y divulgación de aplicaciones generadas con GNU radio, implementando y siguiendo la filosofía de enseñar mediante el ejemplo.

Para esto, se ha dividido este trabajo en cinco capítulos:

En el primer capítulo revisamos un poco de historia, como nació la tecnología de radios definidos por software, cuáles fueron los primeros proyectos orientados a determinar su factibilidad, para luego describir la USRP, que es uno de los equipos más utilizados para el desarrollo de radios definidos por software.

El capítulo 2 nos presenta una introducción al sistema operativo Linux, de tal manera que nos familiaricemos con la organización de su sistema de archivos y tareas básicas como la

instalación de programas, todo esto comparando algunas de las distribuciones más populares basadas en Debian y Red Hat.

Una vez que sabemos como trabajar en Linux, en el capítulo 3 estudiamos como está estructurado GNU radio, como realizar su instalación y que herramientas posee para la creación de un SDR. También se describe paso a paso la instalación del toolbox simulink-usrp para Matlab en WIndows XP.

En el capítulo 4 analizamos el programa tunnel.py, que nos permite establecer la comunicación entre dos computadoras mediante una interfaz ethernet virtual anexada a la USRP. Comparamos los resultados del enlace realizado en las bandas de frecuencias de 900 y 2400Mhz con distintos tipos de modulaciones.

Finalmente en el capítulo 5 presentamos las conclusiones y recomendaciones derivadas de cada una de las experiencias obtenidas con GNU radio funcionando en distintas distribuciones de Linux.

Adicionalmente se incluye como anexos una pequeña guía de Python, el código fuente de los programas y las mediciones con un analizador de espectros para verificar la frecuencia y potencia de la transmisión.

ÍNDICE GENERAL

Glosario XIV

1. Radio definido por software **1**

1.1. Definición . . . 1
1.2. Historia de la tecnología SDR . . . 1
1.3. USRP . . . 4
1.4. Estructura de la USRP . . . 4
1.4.1. ADC . . . 5
1.4.2. DAC . . . 6
1.4.3. Entradas y salidas analógicas auxiliares . . . 6
1.4.4. FPGA . . . 8
1.4.5. Puertos digitales auxiliares . . . 9
1.4.6. Interfaz USB . . . 10
1.4.7. Alimentación . . . 10
1.5. Módulos adicionales . . . 11
1.5.1. Basic TX/RX . . . 11
1.5.2. LFTX/LFRX . . . 11
1.5.3. TVRX . . . 11
1.5.4. DBSRX . . . 12
1.5.5. Transceivers . . . 12

2. Sistema operativo Linux **14**

2.1. Distribuciones . . . 14
2.2. Sistema de archivos . . . 15
2.3. Particiones . . . 16
2.4. Gestores de arranque . . . 17
2.4.1. LILO . . . 18

2.4.2. Grub 18
2.5. Escritorios 18
2.5.1. KDE 18
2.5.2. Gnome 19
2.6. Gestores de paquetes 20
2.6.1. Paquetes RPM 20
2.6.2. DPKG y APT 21
2.7. Archivos TAR 22
2.8. Herramientas de programación 23
2.8.1. autoconf 24
2.8.2. automake 24
2.8.3. libtool 25
2.8.4. Python 25
2.8.5. Eric 26
2.8.6. Módulo TUNTAP 26

3. Software de desarrollo 28
3.1. GNU Radio 28
3.1.1. Instalación en openSUSE 29
3.1.2. Instalación en Ubuntu 33
3.1.3. Estructura 34
3.1.4. Tipos de datos 37
3.1.5. Nomenclatura de los bloques 38
3.1.6. Bloques jerárquicos 38
3.2. Simulink-USRP 40
3.2.1. Controlador 40
3.2.2. Instalación del controlador 40
3.2.3. Instalación de simulink-USRP 42

4. Implementación y pruebas de la plataforma 46
4.1. Pruebas con la USRP 46
4.2. Descripción del programa 47
4.2.1. Módulo tunnel.py 49
4.2.2. Módulo transmit_path.py 56

4.2.3. Módulo receive_path.py 60
4.3. Transmisión a 900 Mhz 62
4.4. Transmisión a 2.4 Ghz 63
4.5. Análisis de resultados 65

5. Conclusiones y recomendaciones **66**
5.1. Mandriva 66
5.2. Ubuntu 67
5.3. openSUSE 67
5.4. Windows 68
5.5. Enlace 69
5.6. Trabajo futuro 69

Anexos **71**

A. Python **72**

B. Código fuente **76**
B.1. tunnel.py 76
B.2. transmit_path.py 82
B.3. receive_path.py 88

C. Mediciones del espectro **96**
C.1. 900Mhz 96
C.1.1. DBPSK 96
C.1.2. DQPSK 97
C.1.3. D8PSK 97
C.2. 2400Mhz 98
C.2.1. DBPSK 98
C.2.2. DQPSK 99
C.2.3. D8PSK 99

ÍNDICE DE FIGURAS

1.1. SDRConsole. 2
1.2. High Performance Software Defined Radio. 3
1.3. USRP. 3
1.4. Idea general de un radio definido por software. 4
1.5. Estructura de la tarjeta madre USRP. 5
1.6. Conversor analógico a digital. 6
1.7. Conversor digital a analógico. 7
1.8. FPGA. 9
1.9. Valores del registro de control para el multiplexor. 9

2.1. Distribuciones de Linux. 15
2.2. Particiones del disco duro. 17
2.3. Escritorio KDE 4.1 implementado en openSUSE. 19
2.4. Escritorio Gnome implementado en Ubuntu. 20
2.5. Gestor de software en Yast. 21
2.6. Gestor de paquetes Synaptic. 22
2.7. Herramientas de programación en Yast. 24
2.8. Archivos ejecutados durante la instalación. 25
2.9. Eric, entorno de desarrollo gráfico para Python. 26

3.1. Diagrama de bloques. 34
3.2. Localización del controlador. 41
3.3. Administrador de dispositivos. 41
3.4. Instalación Microsoft SDK. 43
3.5. Compiladores soportados por Matlab. 44
3.6. Simulink library browser. 45

4.1. Diagrama de bloques del programa. 48

4.2. Enlace mediante GNU radio. 49
4.3. GMSK a 950Mhz. 63
4.4. GMSK a 2412Mhz. 64

C.1. DBPSK a 950Mhz. 96
C.2. DQPSK a 950Mhz. 97
C.3. D8PSK a 950Mhz. 97
C.4. DBPSK a 2412Mhz. 98
C.5. DQPSK a 2412Mhz. 99
C.6. D8PSK a 2412Mhz. 99

ÍNDICE DE TABLAS

1.1. Asignación de entradas analógicas auxiliares 7
1.2. Asignación de salidas analógicas auxiliares 7
1.3. Señales de control en el puerto USB 10

2.1. Comandos para comprimir y descomprimir archivos 23

4.1. Resultados de transmisión a 950Mhz 63
4.2. Resultados de transmisión a 2412Mhz 64

GLOSARIO

A

ADC Analog to Digital Converter, conversor de analógico a digital, p. 5.

AGC Automatic Gain Control, sistema de control que mantiene una salida de amplitud constante mediante retroalimentación, p. 10.

ALSA Advanced Linux Sound Architecture, componente del núcleo de Linux para controlar tarjetas de sonido de manera automática, p. 29.

API Application Programming Interface, programa que sirve de interfaz para intercambiar datos o funciones con otros programas, p. 29.

Asterisk Proyecto de software libre para el desarrollo de una central telefónica basada en Voz sobre IP, p. 70.

B

BlueTooth Estándar para redes de área personal, p. 63.

D

DAC Digital to Analog Converter, conversor de digital a analógico, p. 6.

daughterboard Módulo adicional intercambiable para la etapa de RF en la USRP, p. 11.

DDC Digital Down Convertion, es el proceso de llevar a banda base una señal digital, que se encontraba en IF, p. 8.

DECT Digital Enhanced Cordless Telecommunications, estándar ETSI para teléfonos inalámbricos digitales, p. 12.

DUC Digital Up Convertion, es el proceso de llevar a IF una señal digital, que se encontraba en banda base, p. 60.

E

EOF End of File, indicador de que no existe más información a ser leída en un archivo, p. 58.

F

FIFO First In First Out, es el nombre que reciben las estructuras de datos donde el primer elemento en llegar, es el primero en ser procesado, p. 65.

FPGA Field Programmable Gate Array, circuito integrado programable ampliamente utilizado para el desarrollo de prototipos, puede llegar a tener hasta 4 millones de compuertas, p. 8.

G

Galileo Sistema de navegación por satélite de uso civil desarrollado por la Comunidad Europea, p. 12.

GPIF General Programmable Interface, interfaz configurable mediante software para modificar los descriptores del bus USB, p. 10.

GPS Global Positioning System, sistema que permite obtener las coordenadas de un objeto o persona sobre cualquier lugar del planeta, basados en una red de 32 satélites, p. 12.

GSM Global System for Mobile communications, es el estándar europeo de telefonía móvil de segunda generación que provee soporte para voz, datos y mensajes de texto, p. 70.

I

IEEE Instituto de Ingenieros Electricistas y Electrónicos, p. 1.

ISM Industrial, Scientific and Medical, banda de frecuencias de uso no comercial, su uso se volvió popular por ser aplicado en el estándar WiFi, p. 12.

J

JACK Jack Audio Connection Kit, servidor de sonido que permite multiples conexiones independientes de audio de entrada y salida hacia el driver de la tarjeta de sonido, p. 29.

M

MAC Media Access Control, subcapa del modelo OSI que define la manera en que los usuarios acceden al medio físico para realizar una comunicación, p. 47.

MIMO Multiple Input Multiple Output, tecnología que aprovecha las forma en que las ondas electromagnéticas se reflejan, para mejorar el área de cobertura y la tasa de transmisión, p. 12.

MTU Maximum Transmit Unit, es el tamaño máximo en bytes que puede tener un paquete de datos según la capa del modelo OSI en donde se forma el paquete, p. 65.

O

OSI Open System Interconnection, modelo de referencia para la definición de arquitecturas, p. 48.

P

PCS Personal Communications Service, es el nombre que reciben los servicios de telefonía celular que trabajan en las frecuencias de 1850 a 1990MHz, p. 12.

R

RSSI Abreviatura en inglés de Receive Signal Strength Indication, p. 6.

RTT Round-Trip delay Time, es el tiempo que demora un paquete de datos en ir y volver de su destino, p. 63.

S

SDR Radio Definido por Software, equipo de radio donde una gran parte de las operaciones como modulación y demodulación se ejecutan mediante software en computadores de alto desempeño, p. 1.

T

TAP Programa para crear una interfaz virtual a nivel de capa 2, p. 27.

TUN Programa para crear una interfaz virtual a nivel de capa 3, p. 27.

W

WiFi Nombre comercial que recibe la familia de estándares 802.11, p. 63.

CAPÍTULO 1

RADIO DEFINIDO POR SOFTWARE

1.1. Definición

En pocas palabras, según la IEEE un radio definido por software (SDR), es un radio en el cual algunas o todas las operaciones de la capa física son realizadas por software, es decir tareas como filtrado, amplificación, modulación y demodulación de señales de radio son manipuladas en el dominio digital por programas que pueden ejecutarse sobre computadores de propósito general. Comúnmente lo que se busca es optimizar el diseño de la etapa frontal de radiofrecuencia del radio para obtener una señal que pueda ser fácilmente tratada por un computador, para lograr esto se implementan conversores de analógico a digital y viceversa en procesadores programables de alto desempeño como los FPGA con el objetivo de reducir el tamaño y costo de los circuitos. La ventaja de un SDR es que este puede recibir y transmitir nuevos protocolos de comunicaciones simplemente mediante la actualización de software sobre el hardware existente, evitando incurrir en el cambio total de la infraestructura de comunicaciones que ya se encuentra implementada, como un ejemplo más sencillo podemos imaginar un teléfono celular de segunda generación siendo actualizado a tercera generación solo con el cambio de su *firmware*.

1.2. Historia de la tecnología SDR

La tecnología de radios definidos por software siempre ha sido importante en el aspecto militar, donde equipos de radio nuevos deben interoperar con equipos antiguos, muchos de los cuales permanecían en uso aún cuando su tiempo de vida había terminado, además casos como el ejército norteamericano, que a menudo realiza operaciones conjuntas con aliados que poseían equipos de comunicaciones incompatibles con sus radios de tecnología avanzada.

Es así que en 1991 nace el proyecto SpeakEasy desarrollado por el Air Force Research Laboratory, sin embargo en su primera etapa SpeakEasy solo era capaz de trabajar como un modem en un rango de frecuencias entre 2Mhz y 2Ghz. En su segunda etapa SpeakEasy II mejora las prestaciones del proyecto inicial emulando un radio completo y aumentando el rango de frecuencias a 45.5Ghz e incluyendo 22 formas de onda programables a una tasa de transferencia de hasta 10 Mbps[1]. Continuando con las investigaciones militares el Departamento de Defensa (DoD) en 1997 emprende el proyecto *Joint Tactical Radio System* (JTRS) que proveería a los aviones de combate capacidad de transmisión convergente de voz, datos y video, esta familia de radios cubren un espectro de funcionamiento de 2 a 2000 MHz, con terminales de bajo coste y apoyo limitado a múltiples canales de radio de banda estrecha y ancha acoplados a redes de computadoras[2].

A partir del año 2000 comienzan los primeros proyectos de origen no militar orientados a radio aficionados, entre ellos tenemos el SDR-1000, liberado al mercado por FlexRadio Systems en Abril del 2003, considerado el primer radio definido por software de código abierto, la etapa frontal de radiofrecuencia se encontraba dividida en tres tarjetas que realizaban las tareas de *Direct Digital Synthesizer*, filtrado y alimentación respectivamente[3] e incluía el paquete de software SDRConsole que observamos en la figura 1.1 desarrollado en Visual Basic 6.

Figura 1.1: SDRConsole.

En Octubre del 2005 se inicia el proyecto HPSDR (*High Performance Software Defined*

Radio) que tuvo sus inicios en un grupo de discusión en Yahoo para en la actualidad reunir a más de 800 integrantes que han desarrollado una arquitectura modular basada en una tarjeta madre conocida como Atlas y módulos de adicionales para transmisión (Penelope) y recepción (Mercury) con aplicaciones escritas en C para Linux y C# para Windows[4].

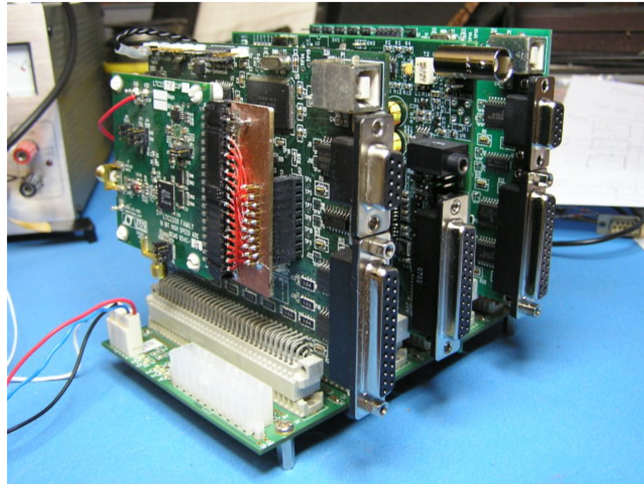

Figura 1.2: High Performance Software Defined Radio.

GNU Radio comienza el 2001 como un proyecto de software libre liderado por Eric Blossom con el objetivo de generar herramientas que permitan procesar señales para posteriormente transmitirlas por el aire[5], luego se suma a la iniciativa Matt Ettus para proveer la etapa frontal de radiofrecuencia, obteniendo como resultado la tarjeta de desarrollo que hoy conocemos como la USRP *Universal Software Radio Peripheral* con la cual se realizará esta tesis, en la figura 1.3 se observa la tarjeta madre con su respectiva caja.

Figura 1.3: USRP.

1.3. USRP

La *Universal Software Radio Peripheral* (USRP) es un equipo que permite el desarrollo de radios definidos por software soportados principalmente por el software de desarrollo GNU Radio, en resumen es nuestro *front end* en lo que se refiere a la etapa de radiofrecuencia, permitiendo a cualquier computador que cuente con puertos USB convertirse en un SDR, en la figura 1.4 observamos la idea principal de un SDR. Los diagramas de la USRP y sus distintos módulos adicionales se encuentran disponibles para su descarga gratuita en la página del proyecto GNU Radio, y nos serán de gran ayuda debido, que para utilizar la USRP en proyectos de comunicaciones inalámbricas es necesario comprender su estructura y funcionamiento.

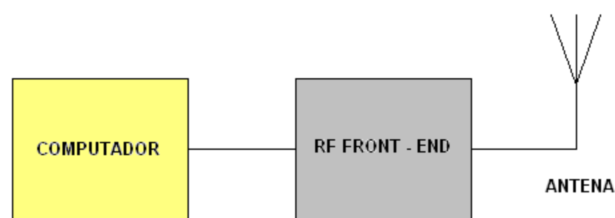

Figura 1.4: Idea general de un radio definido por software.

1.4. Estructura de la USRP

Podemos decir que la USRP posee un diseño modular basada en una tarjeta madre con cuatro ranuras de expansión, cada una de las ranuras se encuentra etiquetada como TXA, RXA, TXB, RXB, respectivamente y la organización de los buses *Serial Peripheral Interface* SPI es realizada de tal manera que si se ocupa las cuatro ranuras de expansión, las tarjetas se observarán, una invertida con respecto a la otra. Esto nos permite realizar múltiples configuraciones, es decir, podemos conectar dos tarjetas con capacidad de transmisión y dos tarjetas para recepción o también se puede conectar dos tarjetas *transceiver*. A continuación enlistamos los principales elementos que constituyen la USRP *motherboard* y los detallamos en la figura 1.5:

- ADC *Analog to Digital Converter*.

- DAC *Digital to Analog Converter*.
- FPGA *Field Programmable Gate Array*.
- Controlador USB

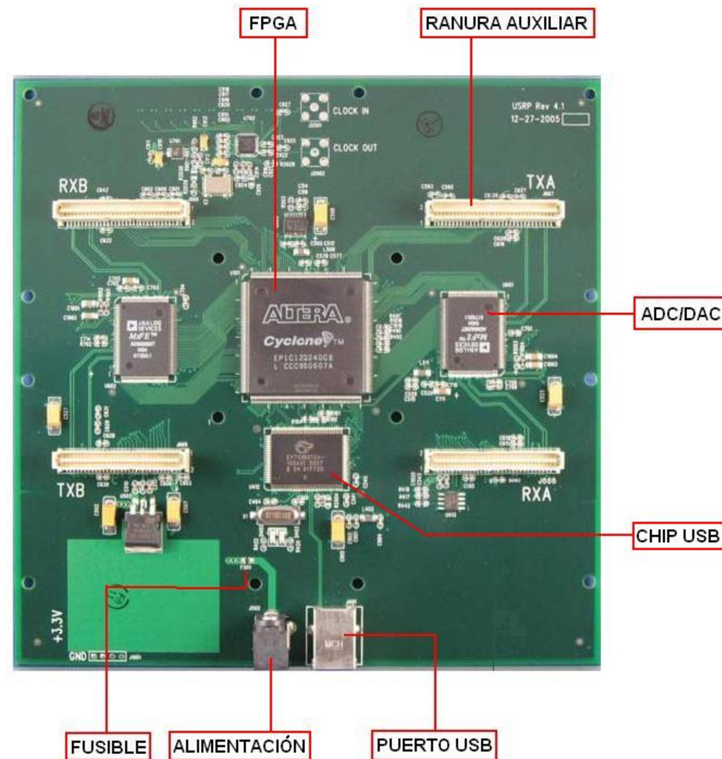

Figura 1.5: Estructura de la tarjeta madre USRP.

1.4.1. ADC

La USRP posee 4 conversores de analógico a digital (ADC) cada uno de ellos con una resolución de 12 bits y una tasa de muestreo de 64MS/seg[1]. Aplicando el teorema de Nyquist sabemos que los conversores están en capacidad de muestrear una señal con un ancho de banda igual a 32Mhz a una frecuencia máxima de 200Mhz, sin embargo es posible ampliar el rango de la frecuencia hasta 500Mhz obteniendo pérdidas de algunos decibeles a la salida. Los ADCs tienen un rango de 2 voltios pico pico y una entrada diferencial de 50 ohmios con un *Programmable Gain Amplifier* (PGA) para variar la entrada hasta 20dB[6], en la figura

[1]Normalmente medimos las tasas de muestreo en millones de muestras por segundo

1.6 apreciamos como se conectan a las ranuras de expansión.

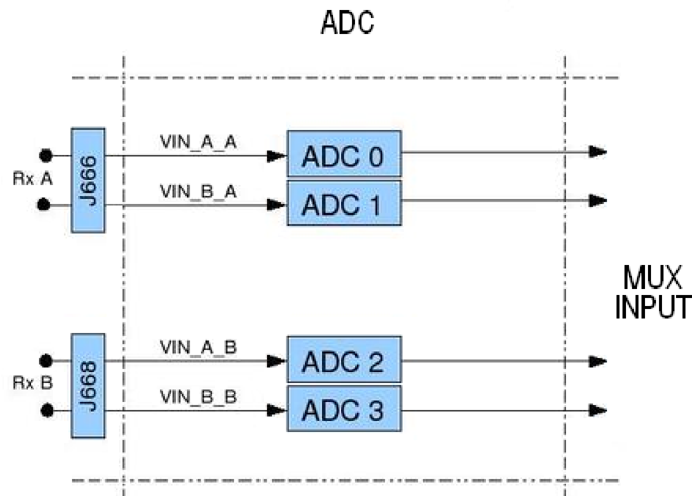

Figura 1.6: Conversor analógico a digital.

1.4.2. DAC

En la etapa de transmisión contamos con 4 conversores de digital a analógico (DAC) con una resolución de 14 bits y una tasa de muestreo de 128MS/seg. Los conversores de digital a analógico son capaces de muestrear señales con un ancho de banda de 64Mhz aunque se recomienda por facilidad trabajar hasta un máximo de 44Mhz. La salida de los DAC entregan 1 voltio pico a una carga diferencial de 50 ohmios y también cuentan con un PGA a la entrada para mejorar la ganancia en 20dB[6], igual que el caso anterior ilustramos en la figura 1.7 como se conectan los DACs a las ranuras de expansión.

1.4.3. Entradas y salidas analógicas auxiliares

Existe también ocho entradas analógicas auxiliares conectadas a un ADC de 10 bits de resolución, capaz de tomar 1.25MS/seg a señales con un ancho de banda aproximado de 200Khz, estas entradas se las puede leer mediante software[6] y se pueden utilizar para realizar medición de la intensidad de señales recibidas RSSI, temperatura y niveles de DC.

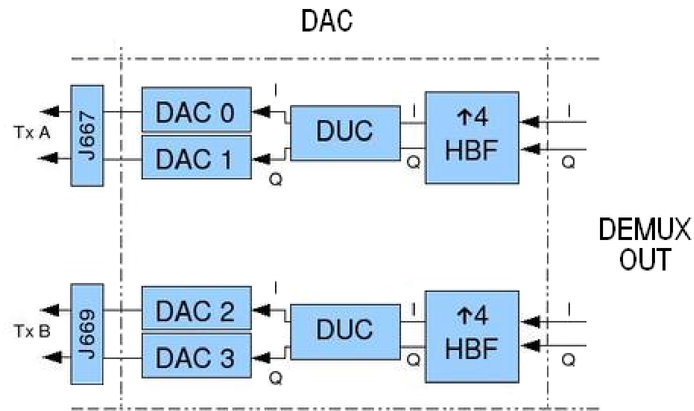

Figura 1.7: Conversor digital a analógico.

TXA	RXA	TXB	RXB
AUX_ADC_A2_A	AUX_ADC_A1_A	AUX_ADC_A2_B	AUX_ADC_A1_B
AUX_ADC_B2_A	AUX_ADC_B1_A	AUX_ADC_B2_B	AUX_ADC_B1_B

Tabla 1.1: Asignación de entradas analógicas auxiliares

La USRP también cuenta con seis salidas analógicas de un DAC con una resolución de 8 bits con las que podemos controlar circuitos independientes como amplificadores o controles automáticos de ganancia, adicionalmente dos salidas forman un modulador sigma-delta con una resolución de 12 bits. Estas salidas se las encuentra en los conectores divididas en grupos de cuatro entre TXA y RXA como se indica en la tabla 1.2.

TXA	RXA	TXB	RXB
AUX_DAC_C_A	AUX_DAC_A_A	AUX_DAC_C_B	AUX_DAC_A_B
AUX_DAC_D_A	AUX_DAC_B_A	AUX_DAC_D_B	AUX_DAC_B_B

Tabla 1.2: Asignación de salidas analógicas auxiliares

1.4.4. FPGA

En el corazón de la USRP tenemos un FPGA (*Field Programmable Gate Array*) Altera Cyclone que opera con un reloj de 64 Mhz y se encarga de realizar operaciones de propósito general como multiplexación, decimación, interpolación, *Digital Down Convertion* (DDC), con el propósito de evitar que se forme un cuello de botella en el puerto USB, todo esto debido a que las 4 entradas para recibir datos de los ADCs y las 4 salidas hacia los DAC generan grandes cantidades de datos. Así la información que se envia al computador puede ser tratada mediante software, obteniendo un arreglo que nos proporciona 4 canales de entrada y 4 canales de salida utilizando muestreo en fase o también se pueden agrupar los canales utilizando muestreo en cuadratura para obtener 2 entradas y 2 salidas complejas[6].

Considerando una señal de radio que se encuentra entre los 39 y 40Mhz con un ancho de banda de 1Mhz el proceso de *Digital Down Convertion* nos permite desplazar la frecuencia de interes hacia la banda báse, de tal manera que disminuyen los requerimientos de muestreo a velocidades elevadas, es decir originalmente para nuestra señal de 40Mhz necesitamos muestrearla a una frecuencia de 100Mhz, pero si desplazamos su espectro diminuyendo su frecuencia hacia banda base es posible seleccionar solo la porción de la señal que nos interesa, en nuestro caso 1Mhz lo que se refleja en una tasa de muestreo disminuida a 2.5Mhz a la salida del FPGA logrando reducir la necesidades en la capacidad de procesamiento y memoria para el tratamiento de las señales recibidas por la USRP[7], por ejemplo si disminuimos a la mitad la tasa de muestreo, el costo de procesamiento se reduce a la cuarta parte, siempre teniendo muy en cuenta el criterio de Nyquist para evitar la pérdida de información por *aliasing*. En la figura 1.8 tenemos el FPGA con los elementos que lleva implementados en su interior.

En el interior del FPGA, también encontramos implementado un multiplexor encargado de determinar como se conectan las salidas del ADC a las entradas de los DDC, en la figura 1.9 vemos la manera en que se obtienen los valores que controlan al multiplexor, existen cuatro DDC y cada uno de ellos tiene 2 entradas, identificamos estas entradas desde I0,Q0 hasta I3,Q3, cada una de las entradas utiliza 4 bits para saber a cual de las cuatro salidas del ADC se encuentra conectadas, entonces, el valor para el multiplexor es un entero de 32 bits para configurar las ocho entradas de los DDC.

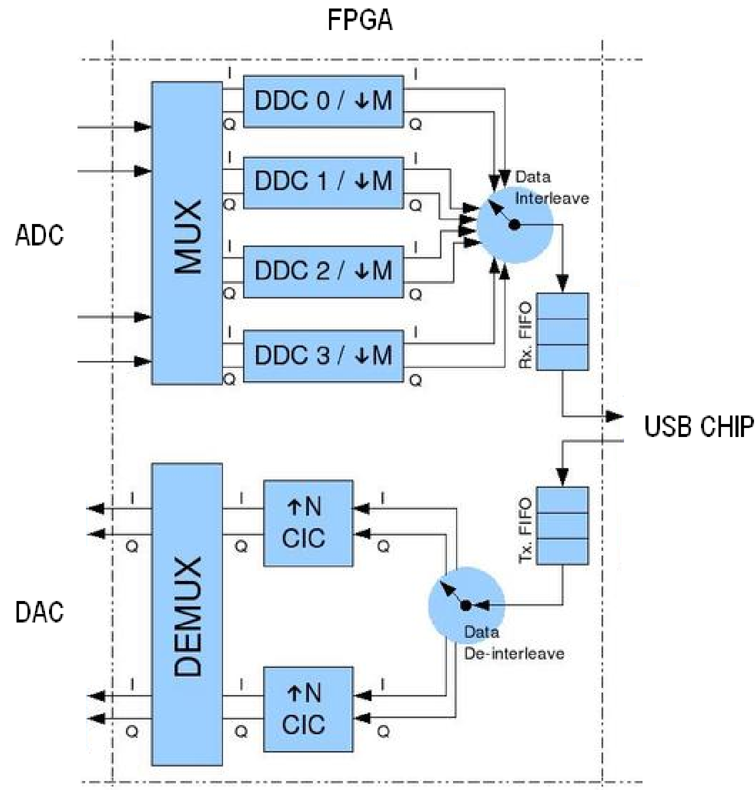

Figura 1.8: FPGA.

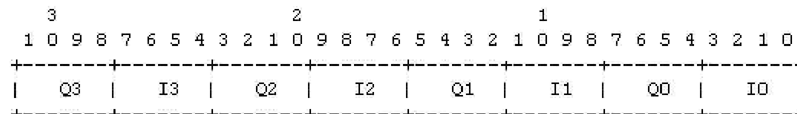

Figura 1.9: Valores del registro de control para el multiplexor.

1.4.5. Puertos digitales auxiliares

En la tarjeta madre tenemos incorporado un puerto digital de 64 bits que pueden ser controlado mediante software para trabajar independientemente como entradas o salidas digitales y su configuración es por lectura o escritura de registros especiales en el FPGA. Los pines de estos puertos son enrutados hacia cada una de las *daughterboards* a través de los conectores TXA, RXA, TXB, RXB con una división de 16 bits por conector. Estos puertos pueden ser utilizados para controlar la fuente de alimentación para las *daughterboards*, im-

plementación de controles automáticos de ganancia (AGC) o para depurar implementaciones con FPGA en conjunto con un analizador lógico[6].

1.4.6. Interfaz USB

En la etapa de comunicación con el computador encontramos un chip Cypress FX2 que contiene un microcontrolador USB 2.0 no compatible con la versión 1.0 y por el lado del FPGA se conecta mediante una GPIF (*General Purpose Interface*) Para separar las diferentes operaciones realizadas en el bus USB se utilizan tan solo tres endpoints, donde las operaciones más comunes son transmisión, recepción y control, una descripción detallada la tenemos en la tabla 1.3. La velocidad máxima soportada por el bus USB es de 32MB/seg y todas las muestras que se envían por este bus son de tipo *signed integer* de 16 bits en formato IQ es decir 16 bits I, 16 bits Q lo que significa un costo de 4 bytes por muestra en cuadratura[6].

endpoint	Descripción
0	Control/Status
2	Host - > FPGA
6	FPGA - > Host

Tabla 1.3: Señales de control en el puerto USB

1.4.7. Alimentación

Para su funcionamiento la USRP utiliza un adaptador universal de corriente alterna con un rango de 90 a 260 voltios y 50 o 60Hz de frecuencia, la tarjeta madre puede funcionar con una fuente de 5V de corriente continua, sin embargo cuando se conectan las *daughterboards* son necesarios 6V y 1.6A. Se puede verificar su buen funcionamiento gracias a un led de color verde que parpadea desde el momento en que se conecta el equipo, en caso de no ser así, se debe reemplazar el fusible (F501) que se encuentra cerca del conector con uno de las siguientes características, tamaño 0603 y 3A[6].

1.5. Módulos adicionales

La USRP puede conectarse a una gran variedad de tarjetas adicionales conocidas como *daughterboards* las mismas que cumplen la tarea de interfaz de RF para la etapa tanto de recepción como transmisión en diversas frecuencias según nuestra necesidad. Cada daughterboard cuenta con una memoria EEPROM (24LC024 ó 24LC025) que le permite identificarse automáticamente el momento que la USRP arranca, en caso de que la memoria no se encuentre grabada, recibiremos un mensaje de advertencia, así mismo cada una de estas tarjetas tiene acceso a dos de los cuatro conversores AD/DA. A continuación damos una breve descripción de algunas de las tarjetas que se encuentran disponibles en el portal de Ettus Research.

1.5.1. Basic TX/RX

Diseñada para operar en el rango de frecuencias de 1 a 250Mhz no posee filtros, mezcladores o amplificadores por lo que para su funcionamiento como etapa de frecuencia intermedia (IF) requiere de un generador de señales externo. También incorporan conectores para acceder a las 16 entradas y salidas auxiliares mientras que las entradas de los ADC y las salidas de los DAC se encuentran acopladas mediante un transformador y conectores SMA de 50 ohmios de impedancia.

1.5.2. LFTX/LFRX

Su diseño es muy parecido a las tarjetas básicas, con la diferencia de que en vez de transformadores de acoplamiento utiliza amplificadores diferenciales y filtros pasa bajos para evitar el aliasing lo que permite expander la frecuencia de trabajo desde DC hasta los 30Mhz de ahi el prefijo LF *low frequency*

1.5.3. TVRX

Como su nombre lo indica esta tarjeta posee los servicios de un receptor de televisión VHF y UHF capaz de sintonizar canales con un ancho de banda de 6Mhz en el rango de frecuencias desde los 50 hasta los 860Mhz, con una figura de ruido de 8dB, este es el único

módulo que no posee capacidades MIMO.

1.5.4. DBSRX

Esta tarjeta de diseño versátil es capaz de trabajar en frecuencias que van desde los 800Mhz hasta los 2.4Ghz con una figura de ruido aproximada de 3 a 5dB puede sintonizar canales controlados por software de un ancho de banda entre 1 y 60Mhz, también cuenta con la posibilidad de alimentar el LNB de una antena de tipo parabólico a traves de su conector SMA. En el rango de frecuencias cubierto por esta tarjeta se incluyen muchas bandas de interes como lo son GPS, Galileo, PCS, DECT, hidrógeno e hidroxil.

1.5.5. Transceivers

El siguiente grupo de tarjetas poseen un conjunto de características comunes, diseñadas para trabajar al mismo tiempo como transmisor y receptor ocupan las dos ranuras de expansión al mismo tiempo, TXA y RXA o TXB y RXB. Funcionan en modo *full duplex* gracias a que poseen osciladores independientes *local oscillators* (LOs) y dos conectores SMA, uno principal para transmisión y recepción más otro auxiliar para recepción con características MIMO. Soportan un canal sintonizable con un ancho de banda igual a 30Mhz.

1.5.5.1. RFX900

Opera en un rango de frecuencias que van desde los 750 hasta los 1050Mhz, aunque posee un filtro para la banda ISM 902-928Mhz que puede ser desactivado para trabajar en las frecuencias de las operadoras celulares entregando una potencia de transmisión de 200mW.

1.5.5.2. RFX2400

Trabaja en un amplio intervalo de frecuencias desde 2.3 hasta 2.9Ghz, como el modelo anterior mediante filtros se puede disminuir el ancho de banda para seleccionar la banda ISM 2400 a 2483Mhz con una potencia de 500mW, ideal para experimentar en las frecuencias de trabajo correspondientes a la familia de estandares 802.11 b/g.

1.5.5.3. XCVR2450

Extiende su funcionamiento a dos rangos de frecuencias, uno cubre los 2.4 hasta los 2.5Ghz y otro desde los 4.9 hasta los 5.9Ghz completando así todas las frecuencias utilizadas por el estándar 802.11a y adicionalmente implementaciones Wimax con frecuencia libre.

CAPÍTULO 2

SISTEMA OPERATIVO LINUX

Linux nace en 1991 como una alternativa a UNIX pero bajo los lineamientos del proyecto GNU (GNU is not Unix) que busca la creación de un sistema operativo y herramientas de licencia libre, que es todo lo contrario a UNIX. Desde entonces Linux poco a poco se ha convertido en el sistema operativo preferido por programadores, administradores de redes e investigadores gracias a los tiempos de operación y fiabilidad que otros sistemas no ofrecen. Aunque posee gran flexibilidad para su configuración, lograr que funcione como necesitamos puede ser una tarea muy laboriosa, esto nos demuestra que aun falta mucho por hacer para que usuarios de escritorio puedan incorporarse al mundo de Linux y no encuentren en él una mala experiencia.

2.1. Distribuciones

En sus comienzos Linux tan solo consistía de su núcleo y unas pocas herramientas GNU incorporadas que con el tiempo fueron aumentando en número y variedad. Con el esfuerzo de diferentes universidades e instituciones crearon sus propias versiones, aunque el núcleo era el mismo los paquetes adicionales incluidos eran distintos, de esta manera se dan a conocer lo que hoy llamamos distribuciones.

Hoy en día entre las distribuciones más populares podemos enumerar Ubuntu, Fedora, Mandriva y openSUSE, en la figura 2.1 observamos los distintos logotipos que identifican a cada una de estas distribuciones.

La pregunta ¿cuál es la mejor distribución de Linux? se puede volver un problema que posee una respuesta relativa, en realidad la pregunta correcta sería ¿cuál es la distribución que mejor se adapta a nuestras necesidades?. Además la compatibilidad con el hardware es

Figura 2.1: Distribuciones de Linux.

otro papel muy importante debido a que los controladores que pudieran estar disponibles en Mandriva no necesariamente los encontraremos en Ubuntu, es por eso que encontraremos defensores de una u otra distribución de acuerdo a la aplicación y equipo sobre el cual fué instalada una distribución.

2.2. Sistema de archivos

Una de las principales diferencias entre Windows y Linux es el sistema de archivos que estos utilizan, es por esto que la primera vez que trabajamos con un sistema operativo Linux nos sentimos desorientados al no poseer NTFS (*New Technology File System*), el mismo sistema de identificación de particiones implementado en Windows, donde a cada partición o unidad de almacenamiento se le asigna una letra C: D: E: etc. Si no conocemos la forma en que se organiza la estructura del sistema de archivos de Linux, no comprenderemos como funciona Linux y encontraremos dificultad el momento de administrar eficientemente los programas instalados.

La jerarquía del sistema de archivos, o FHS (*Filesystem Hierarchy Standard*) obedece a parámetros estandarizados y la mayoría de las distribuciones se apegan a dicho estándar con una u otra modificación [8] actualmente el formato aceptado es *ext3fs*.

- / Es el directorio principal o raíz.
- /home Archivos de los usuarios del sistema[1].
- /root Archivos del administrador del sistema.
- /bin Comandos de control accesible para los usuarios y el administrador.
- /boot Archivos necesarios para el arranque.
- /dev Archivos de configuración de dispositivos.
- /etc Archivos de configuración de programas instalados [2]
- /lib Contiene librerías compartidas de los programas.
- /mnt Punto de montaje de sistema de archivos temporales.
- /opt Paquetes de software.
- /proc Configuración del núcleo.
- /sbin Comandos de control accesibles solo para el administrador.
- /tmp Archivos temporales del sistema.
- /usr Datos compartidos.
- /var Archivos de tamaño variable.

2.3. Particiones

Una partición de disco duro es la división de los sectores de almacenamiento en áreas independientes, el objetivo de crear particiones es tener un mejor control de la información que cada una de estas particiones va a almacenar, en nuestro caso nos apoyamos en su uso para instalar en un mismo disco duro dos sistemas operativos, Microsoft Windows XP y una distribución de Linux. Una forma recomendable de organizar las particiones es la siguiente, sobre una partición primaria ubicamos a Windows y sobre una partición extendida crearemos tres unidades lógicas:

[1] Puede estar en una partición independiente
[2] La ruta puede variar de acuerdo a la distribución

1. C: sobre partición primaria.

2. swap sobre unidad lógica

3. / sobre unidad lógica

4. home sobre unidad lógica

En la figura 2.2 ilustramos en porcentaje cuanto ocupará cada una de las particiones en nuestro disco duro.

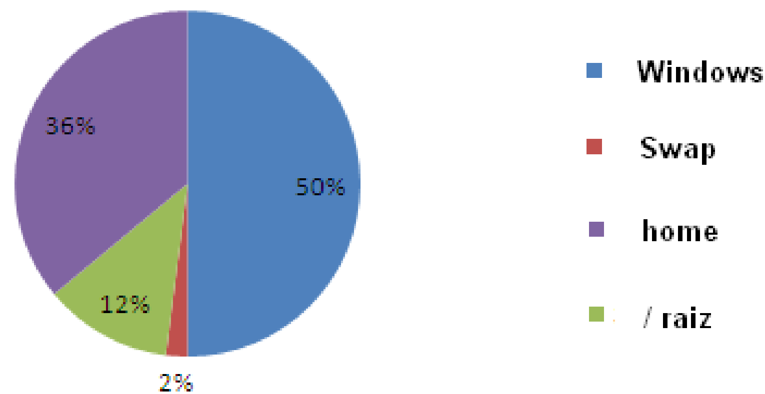

Figura 2.2: Particiones del disco duro.

2.4. Gestores de arranque

Un gestor de arranque es un programa que se encarga de tomar el control de nuestra computadora inmediatamente luego de terminado el proceso de verificación realizado por el BIOS (*Basic In Out System*), por lo general se encuentra instalado en el primer sector del disco duro MBR (*Master Boot Record*), aunque esto no es una regla a seguir. Su principal función es ofrecer al usuario un menú en donde se puede seleccionar uno de los varios sistemas operativos que se encuentren instalados en nuestra máquina con el cual queremos iniciar la sesión.

2.4.1. LILO

Por mucho tiempo LILO (Linux Loader) fue el gestor de arranque con mayor aceptación por casi todas las distribuciones de Linux, se podía decir que era la opción por defecto. LILO esta en capacidad de trabajar hasta con 16 imágenes de distintos sistemas operativos independientemente del sistema de archivos que estos utilicen, normalmente LILO se carga en 4 segundos antes de presentar el menú de sistemas operativos para arrancar y lo más común es instalarlo en el MBR.

2.4.2. Grub

Actualmente Grub (*GNU GRand Unified Bootloader*) es el gestor de arranque más utilizado por Linux, inicialmente desarrollado por Erich Boleyn en 1999 se convierte en un proyecto más de la familia GNU. Al igual que LILO, Grub permite seleccionar el sistema operativo para el inicio de sesión con una gran mejoría al manejar hasta 150 imágenes en un mismo disco duro, inclusive tiene la capacidad de buscar imágenes en un entorno de red. Aunque en realidad la implementacón de una interfaz gráfica no es una prioridad, en Grub podemos encontrar varias opciones personalizables que dan soporte para el uso del ratón.

2.5. Escritorios

Un escritorio es una interfaz gráfica de usuario (GUI), basada en protocolos para aplicaciones que poseen componentes gráficos sean estos botones, íconos o ventanas. La primera versión de escritorio exitosa comercialmente fue diseñada por Apple para sus computadores personales MacOS, recibió gran acogida debido al uso de metáforas entre los archivos y las carpetas así como también la posibilidad de recuperar un archivo borrado desde el tacho de la basura.

2.5.1. KDE

El proyecto *K Desktop Enviroment* (Entorno de escritorio K) curiosamente la K no posee ningún significado, comienza en 1996 con la idea de proporcionar un entorno de trabajo más amigable gracias a que las aplicaciones siguen una guía de estilo coherente, es decir

la interface es intuitiva, por ejemplo la organización del menú superior de los programas siempre comienzan con la opción Archivo y terminan con la opción Ayuda [9]. Actualmente se encuentra en la versión estable 4.1 e incluye más de 250 programas entre las que se incluye juegos, utilidades del sistema y ofimática. En la figura 2.3 podemos ver el escritorio KDE 4.1 implementado en openSUSE 11.1

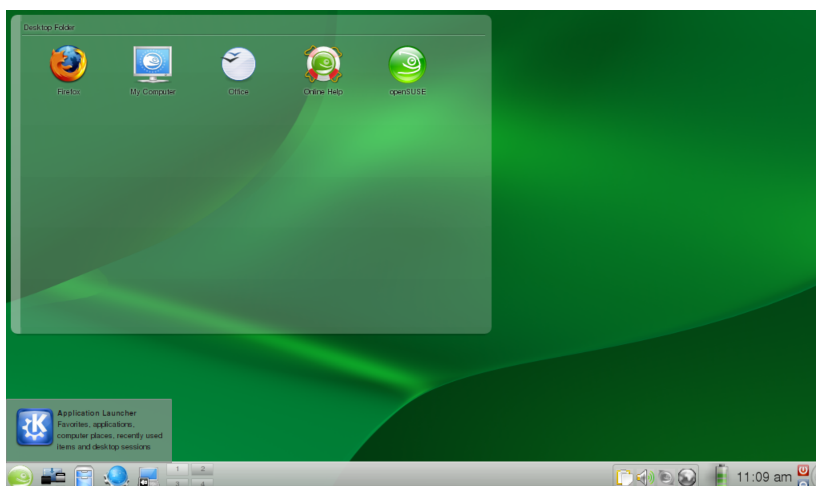

Figura 2.3: Escritorio KDE 4.1 implementado en openSUSE.

2.5.2. Gnome

GNU Network Object Model Environment (Entorno de trabajo en red orientado a objetos GNU) comienza como respuesta al proyecto KDE desarrollado en base a herramientas Qt que no poseían licencia GNU por lo que no eran consideradas de código abierto. Los objetivos que sigue Gnome son los mismos que KDE sin embargo su desarrollo se basa en gtk+, que es un juego de herramientas para dibujar aplicaciones visuales como botones y ventanas, lo que permite a las aplicaciones GNOME presentar una apariencia similar a las de KDE pero con una mayor facilidad para que los desarrolladores puedan modificar las fuentes y redistribuirla.

Una política del proyecto es liberar una nueva versión cada seis meses siendo, Gnome 2.28 la más reciente liberada como versión estable aunque esta solo se encuentra disponible en 25 idiomas, en la figura 2.4 tenemos el escritorio Gnome soportado por Ubuntu Feisty.

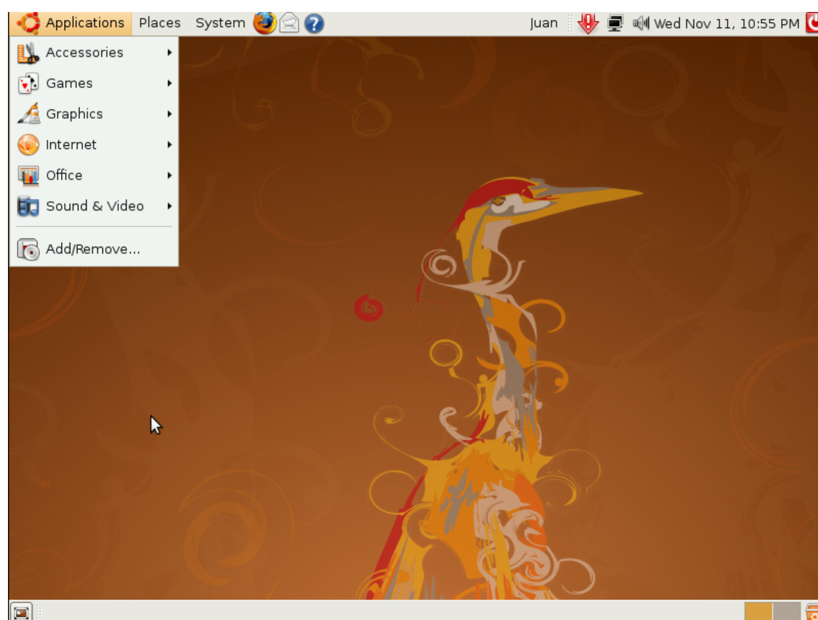

Figura 2.4: Escritorio Gnome implementado en Ubuntu.

2.6. Gestores de paquetes

Hoy en día las distribuciones de linux incluyen una gran variedad de aplicaciones, sin importar cúal utilizemos sus programas se encuentran divididos en paquetes individuales que son administrables mediante gestores, facilitando la manera de actualización de los programas evitando la ejecución de versiones que tengan errores o fallos de seguridad[10].

Otro problema que resuelven los gestores de paquetes es la resolución de dependencias, es decir, verifican cuáles son los requisitos necesarios para que un determinado programa pueda ser instalado en nuestro sistema. En la mayoría de los casos las dependencias son otros programas o librerías que realizan funciones necesarias para que el programa principal funcione correctamente. Los disponibilidad de los gestores de paquetes varía de acuerdo a la distribución siendo los gestores de paquetes RPM y Debian los más utilizados.

2.6.1. Paquetes RPM

El *Red Hat Package Manager* es un sistema de administración de paquetes capaz de instalar, actualizar, verificar y desinstalar programas. Originalmente desarrollado por Red Hat bajo licencia GNU, aunque no solo es utilizado por Red Hat sino también por otras

distribuciones como openSUSE, Mandriva, Fedora y Centos.

Un paquete rpm consiste de archivos que contienen meta-datos con información descriptiva acerca del paquete, donde los paquetes pueden ser de dos tipos, *binary* (ejecutable) o *source* (código fuente). En openSUSE mediante YAST *Yat another Setup Tool* tendremos acceso a una interfaz gráfica que nos permite instalar, desinstalar o actualizar programas. En la figura 2.5 se observa la sencillez de la interfaz gráfica de Yast.

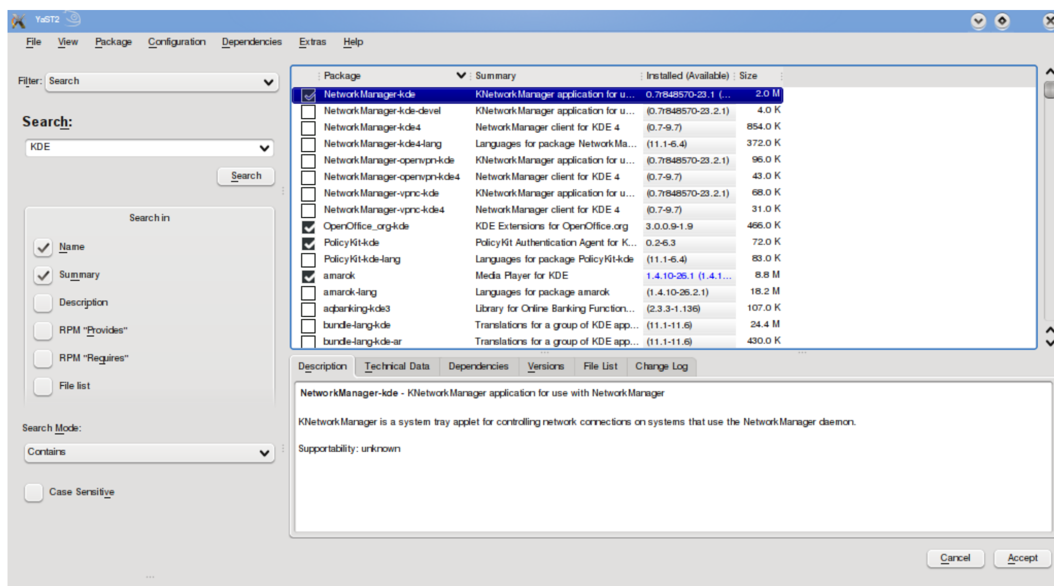

Figura 2.5: Gestor de software en Yast.

2.6.2. DPKG y APT

El *Debian Package System* es un programa utilizado como gestor de paquetes en sistemas basados en Debian, un poco parecido al *Red Hat Package Manager*, sin embargo es una herramienta de bajo nivel que requiere de una interfaz adicional como el apt (*Advanced Package Tool*) En la misma manera que las distribuciones basadas en RPM, las distribuciones Debian también cuentan con aplicaciones gráficas que nos evitan mantener en nuestra memoria los comandos necesarios para la instalación de programas, un ejemplo de estas aplicaciones es Sinaptic. En la figura 2.6 podemos ver que Synaptic presenta una interfaz igual de sencilla en comparación a Yast.

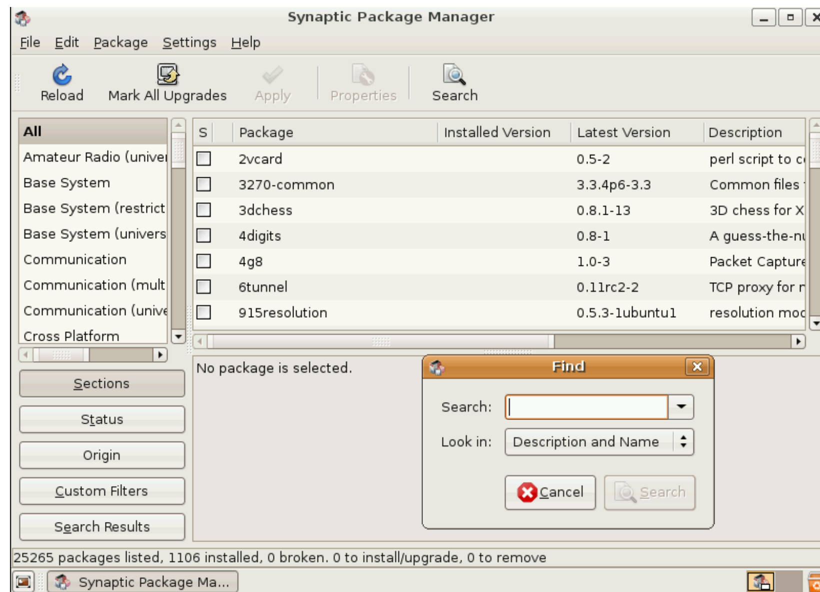

Figura 2.6: Gestor de paquetes Synaptic.

2.7. Archivos TAR

A veces existen programas que no se encuentran disponibles en paquetes RPM o Debian que se puedan instalar fácilmente en nuestro sistema, entonces lo más probable es que tengamos que instalarlo a la manera antigua y complicada descargando los archivos TAR *Tape Archive* (Archivo de cinta), que por lo general contienen el código fuente que debe ser compilado y en el mejor de los casos contienen los archivos binarios del programa que se puede ejecutar directamente.

Es recomendable respetar el orden del sistema de archivos de Linux y colocar los archivos de tipo TAR que contengan código fuente en el directorio */usr/local/src*. Las opciones para trabajar con archivos empaquetados, comprimidos, empaquetados y comprimidos, las describimos en la tabla 2.1.

Archivo	Acción	Comando
tar	Empaquetar	tar cvf archivo.tar
tar	Desempaquetar	tar xvf archivo.tar
tar.gz	Empaquetar y desempaquetar	tar czvf archivo.tar.gz
tar.gz	Desempaquetar y descomprimir	tar xzvf archivo.tar.gz
bz2	Comprimir	bzip2 archivo
bz2	Descomprimir	bunzip2 archivo.bz2
tar.bz2	Comprimir	tar -c archivos bzip2 archivo.tar.bz2
tar.bz2	Descomprimir	tar jvxf archivo.tar.bz2

Tabla 2.1: Comandos para comprimir y descomprimir archivos

Para instalar un programa desde su código fuente el primer consejo a seguir es leer el archivo INSTALL o README que nos proporciona ayuda acerca de cuáles son las dependencias y pasos a seguir durante la instalación, comúnmente se incluye también un *script* conocido como *configure* que es ejecutado por el programa autoconf encargado de recopilar información de nuestro sistema antes de proceder a la compilación del mismo, en caso de que algo falte debemos utilizar el método de ensayo y error, es decir, ejecutamos el *script* y verificamos si nos falta alguna dependencia que cumplir, así hasta que no tengamos ningún error.

Aunque no es una receta universal los pasos a seguir para la instalación de un programa desde su código fuente podrían ser:

```
linux-cb1e:~ # ./configure
linux-cb1e:~ # make
linux-cb1e:~ # make install
```

2.8. Herramientas de programación

Las distribuciones de Linux poseen una gran variedad de herramientas para cada una de las posibles necesidades de un usuario específico, así una persona dedicada a la programación tiene a la mano cientos de programas de desarrollo de código abierto y gratuitos. En la

figura 2.7 nuevamente hacemos referencia a Yast, donde podemos ver que disponemos de las herramientas clasificadas según su utilidad, en este caso observamos todo lo referente a desarrollo.

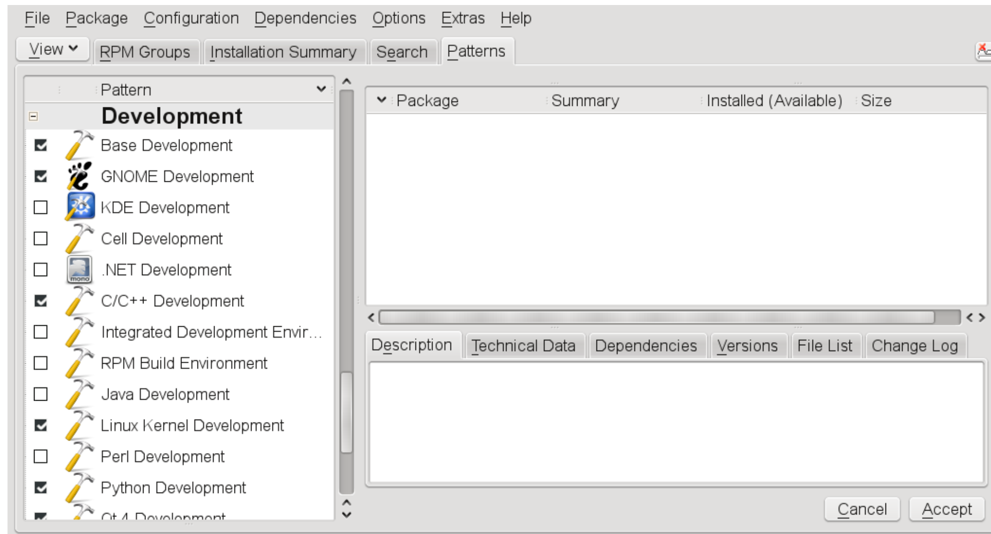

Figura 2.7: Herramientas de programación en Yast.

2.8.1. autoconf

La herramienta *autoconf* lee el archivo configure.ac y produce el script *configure* que contiene las pruebas que se ejecutan automáticamente en el sistema y establece un conjunto de variables que define que se puede utilizar en los Makefiles. Si las funciones que son necesarias no se encuentran, *configure* genera un mensaje de error de salida y de parada.

```
linux-cb1e:~# configure: error: cannot find usable Python headers
```

2.8.2. automake

Se puede decir que *automake* trabaja en conjunto con *configure* para generar archivos *make* con un nivel de descripción mucho más alto que los que figuran en el archivo Makefile.am correspondiente a un instalador. Makefile.am especifica las bibliotecas, programas y los archivos de orígen que se utilizarán para compilar el software a instalar, *automake* lee

Makefile.am y produce Makefile.in mientras que *configure* lee Makefile.in y produce Makefile, el Makefile resultante contiene una gran cantidad de normas para seguir y obtener una correcta instalación[9]. En la figura 2.8 resaltamos estos archivos, que se encuentran contenidos en el directorio de instalación de GNU radio.

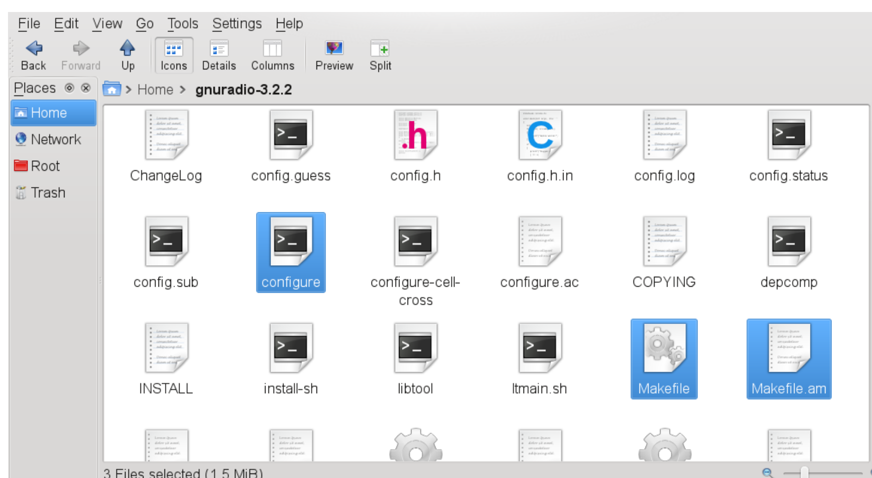

Figura 2.8: Archivos ejecutados durante la instalación.

2.8.3. libtool

Se encarga de administrar y realizar la configuración de librerías compartidas, evitando que un usuario tenga que preocuparse por configuraciones complejas. Cuando un programa se instala verifica la existencia de librerías que pertenecen a otros programas y que puedan ser utilizadas por este, lo que evita volver a instalar dicha librería, manteniendo una organización óptima del sistema de archivos del sistema operativo y posteriormente reduciendo la necesidad de recursos como memoria, ya que verifica si la librería es dinámica para cargarla bajo demanda o si es compartida permite su uso simultáneo por varios programas.

2.8.4. Python

Python es un lenguaje de programación interpretado[3] y orientado a objetos que por su sintaxis muy sencilla y gran cantidad de librerías distribuídas en módulos es ampliamente

[3]No es necesario compilar ni enlazar

utilizado en el desarrollo desde aplicaciones pequeñas como *scripts* hasta extensiones de aplicación, en nuestro caso Python gracias a su facilidad de integración y C++ forman juntos la estructura básica de GNU Radio. En el anexo A incluimos un pequeño tutorial que describe los aspectos más importantes para escribir nuestros propios programas en Python.

2.8.5. Eric

Eric es un entorno de desarrollo integrado (IDE) que consiste de herramientas avanzadas como un editor, compilador, depurador unidos en una misma interfaz gráfica. Eric es desarrollado bajo Python y simplifica la creación de programas en Ruby o Python gracias a funciones básicas pero de mucha ayuda como autocompletado, sangrado automático y remarcación de errores. Existen dos versiones estables, eric4 basado en Qt4 y Python 2 mientras que para Python 3 esta disponible la versión eric5 en su página oficial del proyecto. En la figura 2.9 se observa los diferentes campos del entorno de trabajo presentados por Eric.

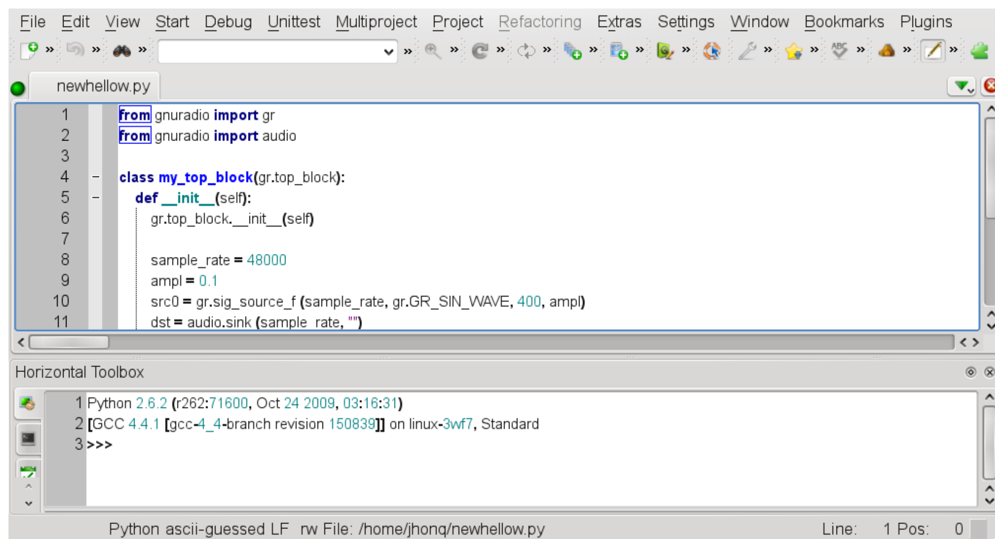

Figura 2.9: Eric, entorno de desarrollo gráfico para Python.

2.8.6. Módulo TUNTAP

Con este módulo estamos en capacidad de crear túneles virtuales, la principal propiedad de un túnel en redes de computadoras es que, para cada puerto de entrada del túnel se le aso-

cia un puerto de salida y cada uno de estos puertos reside en computadoras distintas. Ahora el sufijo virtual nos indica que el túnel no existe en una forma real y permanente, formalmente TUN y TAP permiten la transmisión y recepción de paquetes entre una aplicación y el núcleo de linux.

2.8.6.1. Módulo TUN

El módulo TUN genera un dispositivo de red virtual para crear túneles IP a manera de un enlace Punto-Punto, Una instancia de este módulo provee dos interfaces:

- [/dev/tunX] *character device*[4].
- [tunX] interfaz virtual Punto-Punto.

Una aplicación puede escribir paquetes IP en /dev/tunX y el núcleo recibirá los paquetes en la interfaz tunX, a la inversa ocurre lo mismo, cuando el núcleo escribe paquetes en la interfaz tunX, estos mismos pueden ser leídos por la aplicación en /dev/tunX.

2.8.6.2. Módulo TAP

El módulo TAP genera un dispositivo de red virtual para crear túneles ethernet, su utilización es similar al módulo TUN con la pequeña diferencia que en vez de trabajar con paquetes IP el módulo Tap maneja tramas ethernet. Una instancia de este módulo provee dos interfaces:

- [/dev/tapX] *character device*.
- [tapX] interfaz virtual ethernet.

Una aplicación puede escribir tramas ethernet en /dev/tapXX y el núcleo las recibirá en la interfaz tapX, el proceso inverso es similar, las tramas que el núcleo escriba en la interfaz tapX, la aplicación las recibirá en /dev/tapXX.

[4]Controlador que transfiere datos directamente desde y hacia una aplicación.

CAPÍTULO 3

SOFTWARE DE DESARROLLO

Hoy en día podemos encontrar varias herramientas que nos permiten desarrollar radios definidos por software, GNU Radio bajo linux y simulink-USRP para el sistemas operativo Windows son dos ejemplos, aunque sería ideal trabajar con un ambiente conocido como lo es Windows, esto es un poco más complicado, debido a que la iniciativa SDR ha sido ampliamente apoyada por la comunidad del software libre, misma que utiliza a Linux como su principal caballo de batalla, de ahi que GNU Radio tenga una mayor cantidad de librerías con respecto a simulink-USRP que prácticamente es un proyecto nuevo.

3.1. GNU Radio

Permite el diseño de radios definidos por software mediante el uso de librerías dedicadas que realizan tareas de procesamiento digital de señales, estas librerías son llamadas en forma de módulos integrados en Python, y se conectan entre sí para formar un diagrama lógico a través del cual pasará la información adquirida por la USRP, cada uno de los módulos se orienta a tareas específicas como puede ser filtrado, suma, generación de señales, etc.

GNU Radio se divide en varios componentes autónomos que proveen las herramientas necesarias para iniciarnos en el diseño de radios definidos por software.

- gnuradio-core. Provee las principales librerías que componen GNU radio.
- gnuradio-examples. Contiene código de ejemplo para distintas aplicaciones de GNU-Radio.
- gr-howto-write-a-block. Tutorial para crear bloques para procesamiento de señales para nuestro uso.

- omnithread. Implementa soporte multiplataforma para *threading*[1] para GNU Radio.
- pmt. Nos entrega facilidad de polimorfismo en funciones y datos.
- mblock. Implementación que permite el manejo de mensajes basados en paquetes.
- gr-usrp. Recursos de bajo nivel necesarios para tener acceso a la USRP.
- gr-comedi. En conjunto con libcomedi provee una interfaz en Linux para dispositivos de control y medición, actualmente ya no posee soporte.
- ezdop. Interfaz de bajo nivel para el que permite el funcionamiento de la USRP como hardware de radiolocalización Doppler AE6HO EZ.
- gr-ezdop. Interfaz para EZ Doppler[2].
- gr-audio-alsa. Permite el manejo de la tarjeta de sonido en Linux mediante *Advanced Linux Sound Architecture* (ALSA).
- gr-audio-jack. Conexión con *Jack Audio Connection Kit* (JACK) para compartir audio entre aplicaciones.
- gr-audio-oss. Interfaz para el driver de sonido *Open Sound System Audio* API (OSS).
- gr-audio-osx. Controlador de audio para sistemas operativos Mac OSX.
- gr-audio-portaudio. Interfaz para la API PortAudio para desarrollo de aplicaciones de audio multi-plataforma. cross-platform
- gr-audio-windows. Controlador de audio para sistemas operativos Windows.

3.1.1. Instalación en openSUSE

Para instalar todos los módulos de GNU Radio se debe cumplir con algunas dependencias, en caso de que éstas no se encuentren ya instaladas, tomando en cuenta que las condiciones de instalación existentes son diferentes entre una y otra distribución de Linux, a continuación listamos las dependencias necesarias para realizar la instalación de GNU Radio versión 3.2.2 sobre openSUSE 11.1.

[1] Ejecución de procesos concurrentes duplicados en el procesador
[2] La abreviatura EZ significa *easy*.

- swig.
- fftw3 y fftw3-devel.
- libcppunit y libcppunit-devel.
- libqt4-devel
- boost-devel.
- guile.
- SDL-devel.
- python-wxGTK.
- libjack-devel.
- libxslt-python.
- libportaudio-devel.
- libusb y libusb-devel.
- alsa-devel.
- bison.
- flex.

Mientras que las siguientes dependencias no se encuentran disponibles directamente en YAST pero pueden descargarse de Internet en cada una de las páginas oficiales de los desarrolladores.

- Numpy.
- Cheetah.
- sdcc.
- QwtPlot3d.

La versión de *sdcc* que se encuentra en YAST no esta actualizada, y no cumple con las exigencias de GNU Radio, por lo cual es necesario descargarla directamente de la página del proyecto.

Para instalar *sdcc* seguimos los siguientes pasos, extraemos el archivo que descargamos de Internet, en este ejemplo suponemos que vamos a trabajar con la versión 2.9.

```
linux-cb1e:~# tar xjf path/to/binary/kit/sdcc-2.9.0-i386-unknown-linux2.5.tar.bz2
```

Ingresamos en el directorio que se creó luego de la extracción y copiamos todo su contenido al siguiente directorio /usr/local

```
linux-cb1e:~ # cd sdcc
linux-cb1e:~/sdcc # cp -r * /usr/local
```

Como resultado los archivos binarios se instalan en */usr/local/bin/*, los archivos de cabecera en */usr/local/share/sdcc/include/*, las librerías en /usr/local/share/sdcc/lib/ y la documentación */usr/local/share/sdcc/doc/*

Para comprobar que la instalación fue satisfactoria ingresamos la siguiente línea */usr/local/bin/sdcc -v* que nos muestra en la consola la versión que hemos instalado.

```
linux-cb1e:~ # /usr/local/bin/sdcc -v
SDCC : mcs51/gbz80/z80/avr/ds390/pic16/pic14/TININative/xa51/ds400/hc08 2.9.0
\#5416 (Mar 22 2009) (UNIX)
```

Para instalar el módulo *gr-qtgui* necesitamos de los paquetes que dan soporte a gráficas 3D, *libqwtplot3d-0.2.7-11.1.i586.rpm* y *libqwtplot3d-devel-0.2.7-11.1.i586.rpm*, sin embargo el momento de ejecutar *./configure* estas librerías no serán encontradas por GNU Radio, por lo que es necesario indicarle al compilador la ruta donde se encuentran instaladas.

```
linux-cb1e:~ # ./configure --with-qwtplot3d-libdir=/usr/lib/qwtplot3d\
--with-qwtplot3d-incdir=/usr/include/qwtplot3d
```

También es posible conseguir algunas de las dependencias, e inclusive la versión 3.1.3 de GNU Radio para descargar los paquetes RPM que se pueden instalar mediante un solo click, en la página oficial de openSUSE en la opción *software search*.

Existen dos formas seguras de instalar la versión más actual y estable de GNU Radio en nuestro sistema, la primera es mediante el archivo empaquetado (*tarball*) que lo encontramos en http://gnuradio.org/ y siguiendo los pasos descritos en el capítulo anterior referente a Linux; y la segunda es mediante la utilización del programa *subversion* que se encarga de descargar de la página de GNU Radio todos los módulos de la versión estable más reciente ó la de desarrollo, con la diferencia de que antes de ejecutar el *script ./configure* se debe utilizar el comando ./bootstrap.

Para realizar directamente la descarga desde la consola escribimos la siguiente línea de comandos:

```
linux-cb1e:~ # svn co http://gnuradio.org/svn/gnuradio/branches/releases/3.2 gnuradio
```

Para la instalación podemos seleccionar los módulos que necesitamos instalar mediante los comandos *enable* y *disable* ya que pueden existir módulos que no sean de nuestro interés, o que no sean compatibles con nuestro sistema operativo como es el caso de los módulos *gr-audio-osx* para Mac y *gr-audio-windows* para Microsoft Windows. Si cumplimos con todos los requisitos el resultado será igual al siguiente.

```
gr-wxgui
gr-sounder
gr-utils
gnuradio-examples
grc
docs

You my now run the make command to build these components.

*********************************************************************
The following components were skipped either because you asked not
to build them or they didn't pass configuration checks:

gcell
gr-gcell
gr-audio-osx
gr-audio-windows
```

```
These components will not be built.
```

3.1.2. Instalación en Ubuntu

Para la distribución Ubuntu escogimos la versión Jaunty Jackalope 9.04 debido a que disponemos de un paquete debian no oficial como repositorio en la página de GNU radio, ésta es la manera más sencilla de instalar el programa, aunque debemos asegurarnos de que nuestra conexión sea confiable debido a que el tiempo de instalación es de aproximadamente dos horas con una conexión a Internet de 150Kbps.

Como primer paso para la instalación desde consola debemos agregar los repositorios que contienen la versión estable de GNU radio 3.2.2 con los siguientes comandos.

```
jhonq@jhonq-laptop:~ deb http://gnuradio.org/ubuntu stable main
jhonq@jhonq-laptop:~ deb-src http://gnuradio.org/ubuntu stable main
```

A continuación realizamos una actualización de todos los repositorios mediante:

```
jhonq@jhonq-laptop:~ sudo aptitude update
```

Todo esto nos prepara para la instalación de todas las dependencias necesarias y GNU radio, ahora solo es necesario ejecutar.

```
jhonq@jhonq-laptop:~ sudo aptitude install gnuradio gnuradio-companion
```

Terminado el proceso de instalación podemos verificar que se ha creado un directorio llamado *gnuradio* contenido en el directorio *share*, aquí encontramos los programas de ejemplo que utilizaremos para controlar la USRP, la ruta para navegar por consola hacia el directorio es la siguiente:

```
jhonq@jhonq-laptop:~ cd /usr/share/gnuradio
```

Sin embargo la manera en que Ubuntu accede a la USRP requiere que anexemos nuestro usuario al grupo de trabajo *usrp*, sin embargo los nuevos privilegios solo serán otorgados después de reiniciar el computador.

```
jhonq@jhonq-laptop:~ sudo addgroup <USERNAME> usrp
```

3.1.3. Estructura

El desarrollo de aplicaciones en GNU Radio se basa en bloques que realizan tareas de procesamiento de señales, actualmente cuenta con aproximadamente 100 bloques creados en C++ y en caso de ser necesario podemos generar nuestros propios bloques gracias a que la información y herramientas son libres. Dependiendo de su aplicación, los bloques pueden tener puertos de entrada, puertos de salida o simplemente entradas o salidas que serán las encargadas de recibir y tratar los datos que por estos pasen, teniendo en cuenta que debe mantenerse coherencia entre los puertos que se conectan, es decir si un puerto maneja datos de tipo flotante no podrá conectarse a un puerto de tipo *short* o *complex*.

Aunque los bloques son escritos en C++, las aplicaciones de GNU Radio se basan en diagramas de flujo creados en Python, es decir Python realiza las funciones de lenguaje de alto nivel[11], cada aplicación contiene al menos una fuente y un sumidero, en la figura 3.1 tenemos el diagrama de flujo del programa conocido como hola mundo en GNU radio.

A los bloques que solo tienen una salida se los conoce como fuentes (*source*), mientras que a los bloques que tienen una entrada reciben el nombre de sumidero (*sink*), una fuente puede ser un microfono así como un sumidero lo tenemos en un parlante.

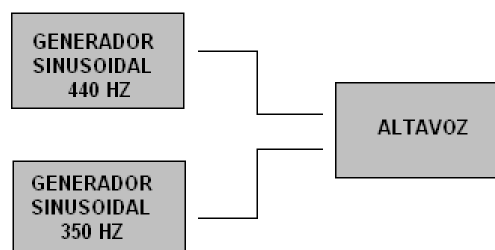

Figura 3.1: Diagrama de bloques.

Para verificar que GNU Radio se encuentra instalado en nuestro sistema creamos un programa mucho más sencillo que el conocido Hola mundo debido a que importa funciones básicas de GNU radio, en caso de no estar bien instalado mostrará el mensaje (Modulo gr no instalado) caso contrario no observaremos ninguna salida debido a que solo creamos una fuente y un sumidero con valores nulos.

```
1  try:
2  # importa los bloques de gnu radio
3      from gnuradio import gr
4  except ImportError:
5      raise ImportError, "Modulo gr no instalado"
6  # crea the top block
7  tb = gr.top_block()
8  # crea una fuente
9  src = gr.null_source(1) # 1 indica el tama\~no de los datos
10 # crea un sumidero
11 sink = gr.null_sink(1)
12 # conecta la fuente con el sumidero
13 tb.connect(src, sink)
14 # ejecuta el diagrama
15 tb.run()
```

Una vez realizada la prueba de que GNU radio se encuentra operativo con los módulos necesarios, podemos ejecutar nuestra primera aplicación con la implementación de un diagrama de flujo sencillo que utiliza al menos dos bloques para generar un tono similar al producido por una central telefónica.

```
1 from gnuradio import gr
2 from gnuradio import audio
3
4 class my_top_block(gr.top_block):
5     def __init__(self):
6         gr.top_block.__init__(self)
7         samp_rate = 44100
8         amp = 0.1
9         src0 = gr.sig_source_f (samp_rate, gr.GR_SIN_WAVE, 350, amp)
```

```
10        src1 = gr.sig_source_f (samp_rate, gr.GR_SIN_WAVE, 440, amp)
11        dst = audio.sink (samp_rate, "")
12        self.connect (src0, (dst, 0))
13        self.connect (src1, (dst, 1))
14
15 if __name__ == '__main__':
16     try:
17         my_top_block().run()
18     except KeyboardInterrupt:
19         pass
```

Las dos primeras líneas son similares a la instrucción *include* utilizada por C++ y mediante su uso tenemos acceso a las principales funciones de GNU radio, normalmente el módulo *gr* siempre debe ser invocado, no así el módulo *audio* que nos ayuda con los bloques necesarios para controlar la tarjeta de sonido. En este ejemplo generamos una clase llamada *my_top_block* que se deriva de *gr.top_block*[3] y es la base principal de nuestro diagrama de flujo que contiene solo una función miembro que es el constructor de la misma clase y está definida con el nombre especial ***__init__***

En las líneas 7 y 8 declaramos las variables *samp_rate* y *amp* que definen la tasa de muestreo y amplitud que enviaremos como parámetros para los bloques generadores de señales, los valores normales para la amplitud están en el rango de 0 a 1 y para la tasa de muestreo es de 44100. Ahora basados en el diagrama de flujo de la figura 3.3 creamos dos fuentes de señales (dos bloques) *src0* y *src1*, para lo cual en las líneas 9 y 10 llamamos a la función *gr.sig_source_f* que tiene algunos atributos como tasa de muestreo, forma de onda, frecuencia y amplitud.[4] Como resultado de esto obtenemos dos ondas senoidales con frecuencias de 350 y 440 hertz, una tasa de muestreo igual a 44100 muestras por segundo.

Continuando con el diagrama de flujo ahora debemos crear una fuente para enviar las señales producidas por *src0* y *src1*, para esto en la línea 11 utilizamos la función *audio.sink* que solo requiere del parametro *samp_rate*, aunque debemos tener en cuenta que esta función acepta datos de tipo flotante con amplitudes entre 1 y -1. Este bloque nos permite reproducir en los parlantes izquierdo y derecho de la computadora las señales *src0* y *src1*.

[3]gr.top_block reemplaza al metodo gr.flow_graph ()
[4]Es necesario consultar la documentación

Ahora para que los bloques que hemos creado trabajen juntos, es necesario conectar sus respectivos puertos, es decir las salidas de un bloque se conectan con las entradas de otro bloque, para lograr este objetivo GNU radio provee el método *self.connect* que tiene la siguiente sintaxis, *self.connect(bloque1, bloque2, bloque3...)* y su resultado es que los datos producidos por el bloque 1 ingresan por la entrada del bloque 2, así mismo la salida del bloque 2 se conectan a la entrada del bloque 3. Sin embargo en las líneas 12 y 13 *connect* se utiliza de una manera especial debido a que *audio.sink* posee dos puertos de entrada es por esto que en los argumentos que enviamos los interpretamos de esta manera, el puerto de salida 1 del bloque *src0* se conecta con el puerto de entrada 1 del bloque *dst* y el puerto de salida 1 del bloque *src1* se conecta con el puerto de entrada 2 del bloque *dst*, con la diferencia de que la numeración de los puertos inicia en cero para el primer puerto, uno para el segundo y así sucesivamente.

Como parte final se necesita una función que controle cuando se debe ejecutar el programa, es así que utilizamos la estructura de control *if-try-except* en conjunto con la función *my_top_block().run()* (linea 17) que arranca el programa, mientras que *KeyboardInterrupt* verifica el ingreso de la secuencia *Ctrl + C* para detener la ejecución del programa. Debemos observar que aunque creamos la clase *my_top_block()* nunca creamos una instancia de la misma, esto no es un problema ya que si lo quisiéramos podemos crear distintas instancias de esta clase, un ejemplo sería *tono = my_top_block()* y cambiamos la forma en que llamamos a sus métodos a una manera más sencilla, *tono.run()*.

3.1.4. Tipos de datos

Como ya hemos mencionado anteriormente a través de los bloques viajan distintos tipos de datos, esto es de acuerdo al bloque que los procesa y en caso de existir alguna inconsistencia se producirá un error en tiempo de ejecución. En general GNU radio maneja escalares y vectores, es así que un escalar de tipo *short* no es más que un vector con dimensión igual a uno[12].

- Byte 8 bits.
- Short entero de 2 bytes.
- Int entero de 4 bytes.

- Float 4 bytes de punto flotante.
- Complex 8 bytes

3.1.5. Nomenclatura de los bloques

La mayoría de las veces podemos identificar el tipo de datos que un bloque maneja gracias a que al final de su nombre se indica con un guión bajo seguido de la primera letra de los distintos tipos de datos que tiene GNU radio, como ejemplo tenemos al bloque generado por la función *gr.sig_source_f*, es claro que produce una salida de tipo flotante. En otras ocaciones también encontraremos bloques con el siguiente sufijo *_vff* que nos indica entrada y salida de tipo vector.

No todos los bloques siguen esta regla, y es así que existen bloques que no nos proveen este tipo de ayuda, *audio.sink* acepta datos de tipo flotante, sin embargo su nombre no termina en *_f*. Mientras que casos especiales como *gr.null_sink()* acepta cualquier tipo de datos pero se debe especificar el tamaño del dato que enviamos en bytes.

3.1.6. Bloques jerárquicos

Cuando el diagrama de flujo se vuelve demasiado complejo, podemos agrupar a ciertos bloques de tal manera que combinados se comporten como una sola entidad (un nuevo gran bloque)[13]. Este nuevo bloque se encontrará disponible para otras aplicaciones si es que nos apoyamos en la modularidad de Python y lo guardamos como archivo distinto *hier_block.py* para luego llamarlo mediante la instrucción *from hier_block import HierBlock*.

```
1  class HierBlock(gr.hier_block2):
2      def __init__(self, audio_rate, if_rate):
3          gr.hier_block2.__init__(self, "HierBlock",
4                          gr.io_signature(1, 1, gr.sizeof_float),
5                          gr.io_signature(1, 2, gr.sizeof_gr_complex))
6
7          B1 = gr.block1(...) # Cambia de acuerdo a nuestra necesidad
8          B2 = gr.block2(...)
9
10         self.connect(self, B1, B2, self)
```

Como podemos observar en las líneas 3, 4 y 5 la declaración del nuevo bloque se asemeja a la creación de un diagrama de flujo común, sino que ahora el constructor de la clase requiere cuatro parámetros:

- self (similar al puntero this en C)
- "HierBlockçadena que dentifica al nuevo bloque
- *input signature*
- *output signature*

Los dos últimos parámetros se pasan a través de la función *gr.io_signature()* mediante la cual indicamos qué tipo de datos maneja nuestro nuevo bloque en sus puertos de entrada y salida. Igualmente esta función requiere los siguientes parámetros:

- mínimo número de puertos.
- máximo número de puertos.
- tamaño de los elementos de entrada y salida.

Mientras que la variable que nos indica el tamaño de los elementos de entrada y salida también puede tomar los siguientes valores:

gr.sizeof_int.

gr.sizeof_short.

gr.sizeof_char.

También debemos notar en la línea 10 que el método *self.connect(self, B1, B2, self)* utiliza *self* como fuente y sumidero.

3.2. Simulink-USRP

No solo existe la iniciativa GNU Radio para trabajar con la USRP desde Linux, sino también podemos encontrar proyectos que buscan proveer una interfaz para Matlab y Simulink en un ambiente Windows ampliamente conocidos por estudiantes y profesores de las ramas de ingeniería electrónica, lo que significaría una reducción de tiempo al menos en el desarrollo de aplicaciones didacticas orientadas a los radios definidos por software. Es por esto que en esta sección explicaremos paso a paso la instalación de todas las herramientas necesarias para que la USRP sea reconocida como un dispositivo USB y pueda ser controlada mediante el *toolbox* de Simulink desarrollado por el Institute fur Nachrichtentechnik.

Debemos tener en cuenta que un toolbox es un conjunto de herramientas previamente desarrollados en Matlab para la simulación de modelos matemáticos que describen un fenómeno físico, es así que mediante el toolbox simulink-USRP podemos manejar la USRP para que funcione como un transmisor o como un receptor de señales de radio.

3.2.1. Controlador

Al igual que cualquier otro dispositivo que se conecta al computador, la USRP requiere de un controlador para su correcto funcionamiento, para lograr este objetivo nos apoyaremos en la librería libusb-win32 que permite a cualquier aplicación acceder a un dispositivo USB de forma genérica al igual que lo haría con dispositivos HID (*human interface device*) manteniendo compatibilidad con los sistemas operativos Win98SE, WinME, Win2k y WinXP.

3.2.2. Instalación del controlador

El controlador se encuentra disponible en http://libusb-win32.sourceforge.net para su descarga gratuita, debemos notar que la USRP no es un dispositivo *plug and play*, sin embargo el proceso de instalación es sencillo, para instalarlo descomprimimos el archivo usrpdriver-1.0.0.zip en un directorio cualquiera, de preferencia localizado la raiz del disco duro C, y a continuación conectamos la USRP a cualquier puerto USB de nuestro computador que inmediatamente reconocerá al nuevo dispositivo preguntándonos la ubicación del controlador. En la figura 3.2 observamos como el sistema operativo detecta el controlador y lo instala para su utilización.

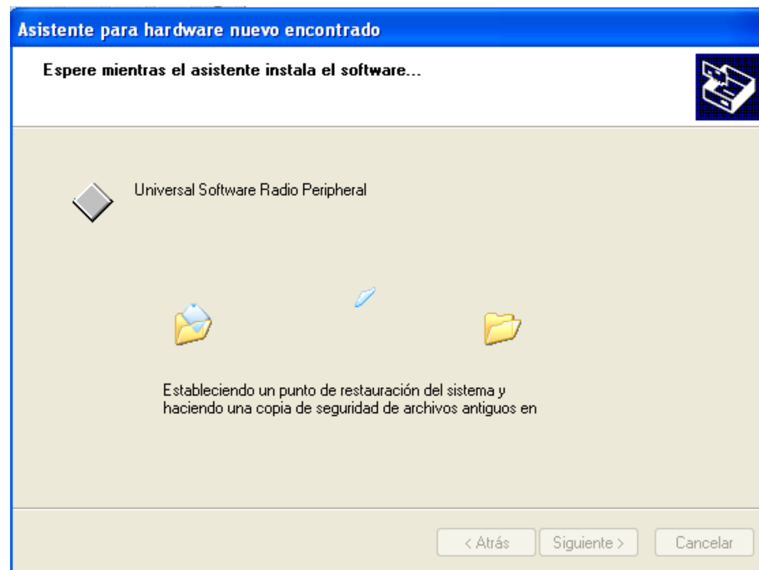

Figura 3.2: Localización del controlador.

Para verificar que controlador se ha instalado y la USRP funciona correctamente tenemos el administrador de dispositivos de Windows, en la figura 3.3 se puede reconocer que el sistema operativo ha detectado sin problemas al nuevo hardware.

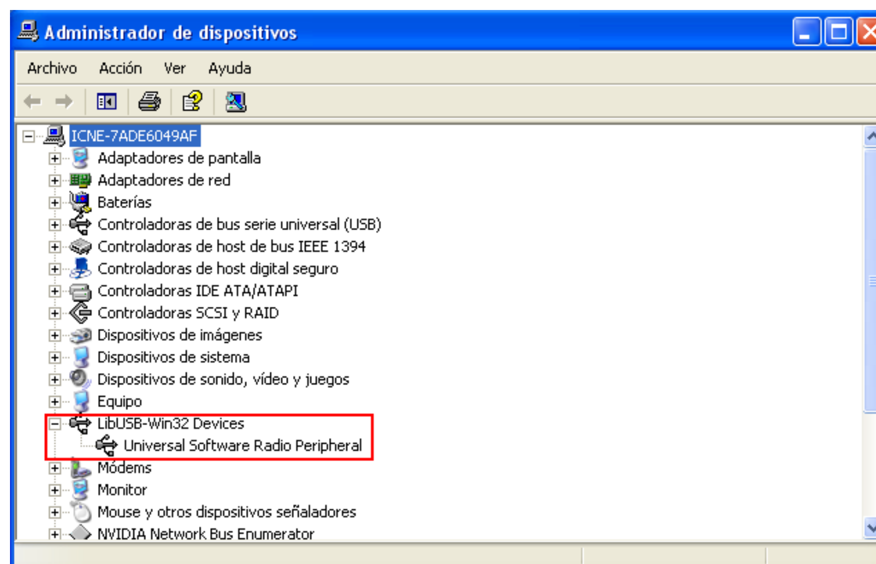

Figura 3.3: Administrador de dispositivos.

3.2.3. Instalación de simulink-USRP

Debido a problemas con licencias y derechos de autor, el *toolbox* simulink-USRP no puede ser incluído directamente con Matlab, es por esto que debe ser instalado de forma separada. Aunque nuestro sistema operativo seleccionado es Windows XP, igual que en Linux también debemos cumplir con ciertas dependencias y realizar unas modificaciones a Matlab para lograr la instalación,

En primer lugar trabajaremos con Matlab 2007a, el mismo que asumimos ya se encuentra instalado en nuestro computador, en teoría el proceso que detallamos a continuación también funciona para la versión de Matlab 2007b y con pequeñas variaciones puede ser implementado en Windows Vista y Windows 7.

Para compilar e instalar el *toolbox* necesitamos de Microsoft Visual C++ 2008 disponible en la versión Express Edition, liberado de forma gratuita aunque limitado en sus funciones, se encuentra orientado para su uso por parte de estudiantes y desarrolladores de aplicaciones que no tengan un propósito comercial.

Otra de las dependencias que debemos cumplir es la instalación las herramientas de Microsoft SDK para Windows Server 2008 y .NET Framework 3.5 que se encuentran en la página de Microsoft para su descarga gratuita. Existe dos versiones disponibles del instalador, la primera es un archivo ejecutable *setup.exe* que se conecta a Internet para descargar todo lo necesario para la instalación y la segunda, contamos con la posibilidad de optar por una imagen ISO para DVD, en nuestro caso nos decidimos por la imagen ISO que nos evita la necesidad de contar con una conexión de Internet para posteriores instalaciones. En la figura 3.4 tenemos la selección de las opciones necesarias para la instalación de las herramientas de Microsoft SDK.

A continuación nuestro siguiente paso es la integración de las herramientas de Microsoft con Matlab, específicamente necesitamos utilizar el compilador de Visual C++ y las librerias de .NET Framework 3.5 para compilar el *toolbox*. Matlab soporta una gran variedad de compiladores[14], no obstante para que el compilador de Visual C++ sea reconocido

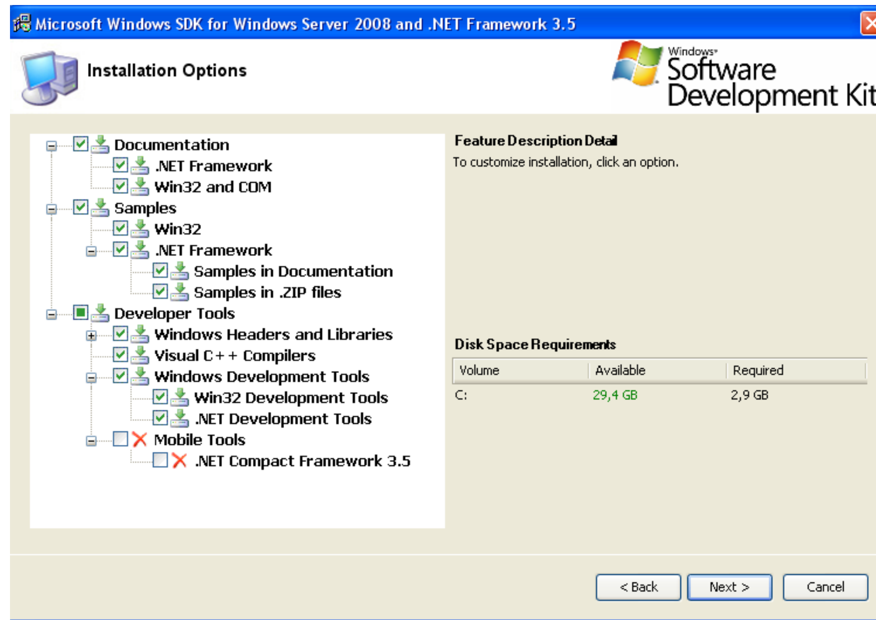

Figura 3.4: Instalación Microsoft SDK.

por Matlab tenemos que instalar los archivos de configuración *MEX-files*[5] compatibles con Matlab 2007a para Windows XP de 32 y 64 bits que se encuentran disponibles en la página de la comunidad de usuarios de Matlab en la sección *file exchange*.

Una vez descargado y descomprimido el archivo copiamos los siguientes archivos en el directorio ubicado en C:\Program Files\MATLAB\R2007a\bin\win32\mexopts:

- msvc90freeengmatopts.bat
- msvc90freematopts.bat
- msvc90freematopts.stp

Reemplazamos los archivos que se encuentran en C:\Program Files\Microsoft Visual Studio 9.0\VC\bin por los siguientes:

- vcvars32.bat
- vcvarsx86_amd64.bat

[5] *Matlab executable files*

Siempre teniendo en cuenta el idioma del sistema operativo[6], definimos las siguientes variables de entorno en el Panel de control.

- MSSdk=C:\Program Files\Microsoft SDKs\Windows\v6.1
- VS90COMNTOOLS=C:\ProgramFiles\MicrosoftVisualStudio9.0\Common7\Tools

Para seleccionar el compilador con el que queremos trabajar ejecutamos el comando *mex-setup* en Matlab, en la figura 3.5 observamos el procedimiento a seguir y una lista de los posibles compiladores con los que Matlab puede trabajar, nuestra elección es el compilador para Visual C++ 2008 Express Edition.

```
Command Window
To get started, select MATLAB Help or Demos from the Help menu.
Done . . .

>> mex -setup
Please choose your compiler for building external interface (MEX) files:

Would you like mex to locate installed compilers [y]/n? n

Select a compiler:
[1] Borland C++ Compiler (free command line tools) 5.5
[2] Borland C++Builder 6.0
[3] Borland C++Builder 5.0
[4] Compaq Visual Fortran 6.1
[5] Compaq Visual Fortran 6.6
[6] Intel C++ 9.1 (with Microsoft Visual C++ 2005 linker)
[7] Intel Visual Fortran 9.1 (with Microsoft Visual C++ 2005 linker)
[8] Intel Visual Fortran 9.0 (with Microsoft Visual C++ 2005 linker)
[9] Intel Visual Fortran 9.0 (with Microsoft Visual C++ .NET 2003 linker)
[10] Lcc-win32 C 2.4.1
[11] Microsoft Visual C++ 6.0
[12] Microsoft Visual C++ .NET 2003
[13] Microsoft Visual C++ 2005
[14] Microsoft Visual C++ 2005 Express Edition
[15] Microsoft Visual C++ 2008 Express Edition
[16] Open WATCOM C++ 1.3

[0] None

Compiler: 15

The default location for Microsoft Visual C++ 2008 Express Edition compilers is C:\Archivos de programa\Microsoft Vi
but that directory does not exist on this machine.

Use C:\Archivos de programa\Microsoft Visual Studio 9.0 anyway [y]/n? y
```

Figura 3.5: Compiladores soportados por Matlab.

Finalmente para compilar el *toolbox* debemos establecer dos nuevos *path*[7] indicando la ruta al directorio que contiene el código descargado de Internet \simulink-usrp\bin y otro a \ simulink-usrp\blockset.

[6]Los nombres de los directorios son distintos si el idioma cambia.
[7]Un *path* en Matlab es un conjunto de directorios que contienen archivos ejecutables de vital importancia.

```
>> usrpBuildBinaries
Running: mex -outdir 'C:\Archivos␣de␣programa\MATLAB\R2007a\usrp\bin'
-output libusrp LINKER=lib.exe LINKFLAGS=/NODEFAULTLIB
\Matlab\R2007a\usrp\src\usrp\libusrp\db_base.cpp
\Matlab\R2007a\usrp\src\usrp\libusrp\db_basic.cpp
\Matlab\R2007a\usrp\src\usrp\libusrp\db_boards.cpp
```

Una vez culminado el proceso de instalación ejecutamos Simulink y comprobamos que el *toolbox* se ha instalado correctamente, de ser así debe aparecer el bloque USRP en *Simulink library browser* tal como lo demuestra la figura 3.6 caso contrario debemos revisar todos los pasos realizados anteriormente.

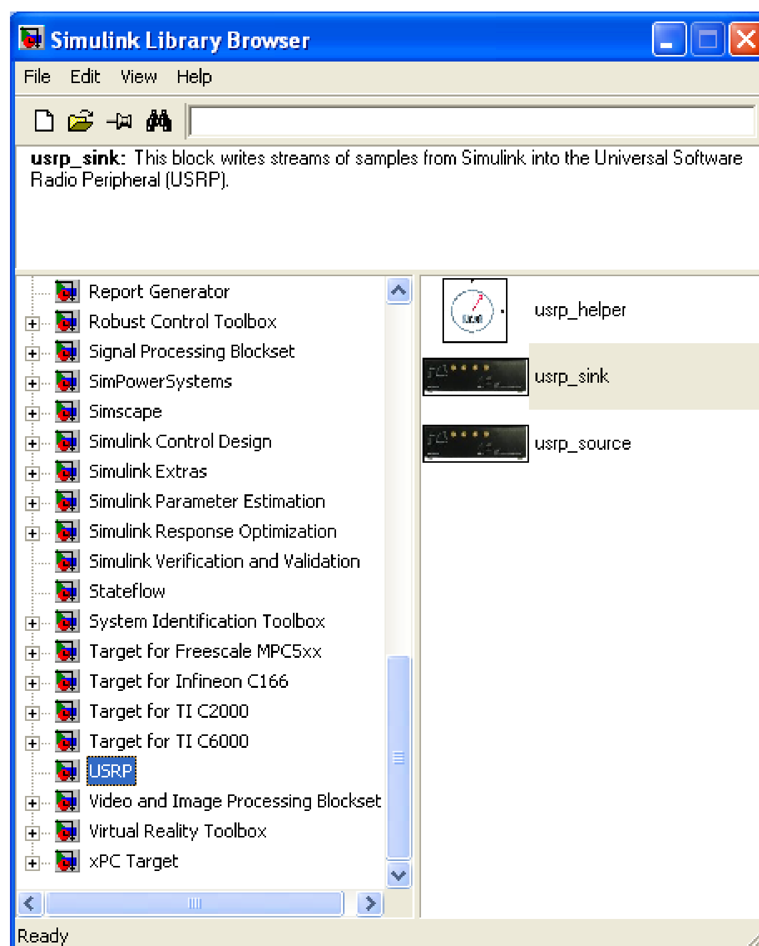

Figura 3.6: Simulink library browser.

CAPÍTULO 4

IMPLEMENTACIÓN Y PRUEBAS DE LA PLATAFORMA

En este capítulo evaluaremos el desempeño de la plataforma de comunicaciones, implementada mediante dos USRP equipadas con los módulos RFX900 y RFX2400 que enlazarán a dos computadoras con dos distribuciones diferentes de Linux, Ubuntu y openSUSE respectivamente; de igual manera en cada una de estas distribuciones hemos instalado la versión 3.2.2 de GNU radio. También damos una explicación de las partes más importantes del programa utilizado para realizar la comunicación entre las dos USRP.

4.1. Pruebas con la USRP

Para comenzar a trabajar con la USRP y GNU radio en conjunto debemos asegurarnos de que el módulo gr-usrp se encuentra correctamente instalado y funcionando, para esto contamos con el programa de ejemplo *usrp_benchmark_usb.py* ubicado en el directorio */gnuradio-3.2.2/gnuradio-examples/python/usrp* que nos permite medir la velocidad a la que se comunica nuestra computadora con la USRP a través del puerto USB. En caso de encontrar algún error obtendremos el siguiente mensaje *Testing 2MB/sec... usrp: failed to find usrp[0]*, caso contrario, el resultado de la ejecución sin errores del programa será la siguiente.

```
linux-gkfh:~usrp # python usrp_benchmark_usb.py
Testing 2MB/sec... usb_throughput = 2M
ntotal    = 1000000
nright    = 998329
runlength = 998329
delta     = 1671
OK
Testing 4MB/sec... usb_throughput = 4M
```

```
ntotal    = 2000000
nright    = 1998165
runlength = 1998165
delta     = 1835
OK
Testing 8MB/sec... usb_throughput = 8M
ntotal    = 4000000
nright    = 3997985
runlength = 3997985
delta     = 2015
OK
Testing 16MB/sec... usb_throughput = 16M
ntotal    = 8000000
nright    = 7999563
runlength = 7999563
delta     = 437
OK
Testing 32MB/sec... usb_throughput = 32M
ntotal    = 16000000
nright    = 15999338
runlength = 15999338
delta     = 662
OK
Max USB/USRP throughput = 32MB/sec
```

4.2. Descripción del programa

El programa *tunnel.py* crea una interfaz en capa dos, MAC, mediante el módulo TAP, cada interfaz en Linux lleva un nombre específico *eth0* y *wlan0* para la tarjeta de red *ethernet* y *wireless lan* respectivamente, mientras que la nueva interfaz virtual creada por el módulo TAP normalmente recibe el nombre *gr0*. Formalmente las distribuciones de Linux con kernel en las versiones 2.6 ya incluyen este módulo por lo que no es necesario realizar ninguna instalación adicional[1].

En la figura 4.1 tenemos un diagrama de bloques, donde podemos observar como se intercambia la información, entre las principales funciones que definen al SDR.

Para ejecutar el programa desde consola transmitiendo en el primer canal WiFi a una frecuencia de 2412Mhz y con modulación dqpsk escribimos:

[1]Posiblemente el módulo necesite ser agregado al núcleo mediante el comando modprobe

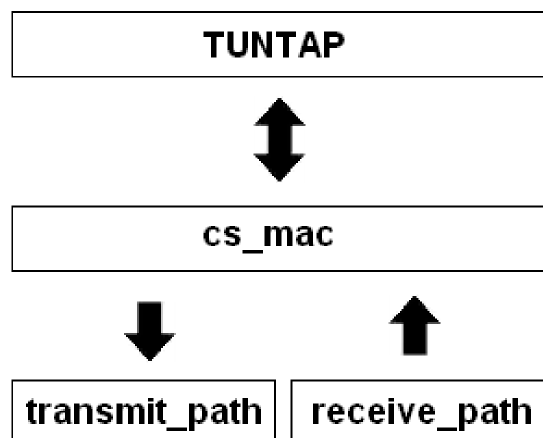

Figura 4.1: Diagrama de bloques del programa.

```
linux-cb1e:~ # sudo python tunnel.py -f 2412e6 -m dqpsk
```

Sin embargo para tener comunicación en niveles superiores de la modelo de referencia OSI debemos asignar una dirección IP a la interfaz virtual.

```
linux-cb1e:~ # sudo ifconfig gr0 192.168.10.1
```

El mismo procedimiento lo repetimos en otro computador, lógicamente la dirección de red debe ser distinta y luego de esto es posible habilitar otros servicios como *Samba Server* para realizar intercambio de archivos entre los computadores.

Para mayor facilidad el programa se encuentra dividido en varios archivos de tal manera que que podemos reutilizar el código mediante la opción *import* de Python.

- tunnel.py
- receive_path.py
- transmit_path.py

La manera en que se realizaron las pruebas se observa en la figura 4.2, la portátil HP con procesador AMD Turion 64x2 1.9Ghz, 2GB de memoria RAM, trabaja con Ubuntu

9.04, mientras que la *netbook* Dell, procesador Intel Atom 1.6Ghz, 1 GB de memoria RAM, trabaja con openSUSE 11.1.

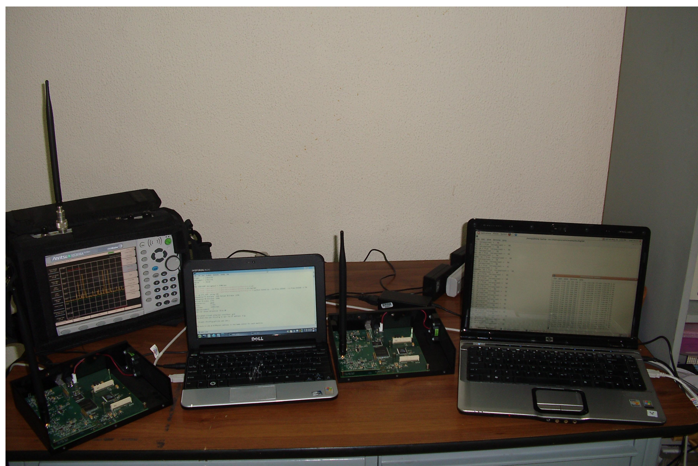

Figura 4.2: Enlace mediante GNU radio.

4.2.1. Módulo tunnel.py

En la cabecera del programa primero llamamos a los módulos propios de GNU radio que nos facilitan las herramientas para la implementación de dos radios capaces de comunicarse uno con el otro, los módulos *gr, gru, modulation_utils y usrp* son imprescindibles, no así los módulos *eng_notation* que su principal función es mejorar la manera de ingresar los argumentos requeridos por el programa mediante números con formato en notación científica e incorpora la utilización de los sufijos del sistema internacional de la forma 10M para 10 Megas o 10K para 10 Kilos; mientras que *eng_option* define el formato de dato *subdev* para identificar mediante tuplas a las tarjetas opcionales (*daghterboards*).

```
1 from gnuradio import gr, gru, modulation_utils
2 from gnuradio import usrp
3 from gnuradio import eng_notation
4 from gnuradio.eng_option import eng_option
```

El siguiente grupo de módulos los describiremos brevemente, *os* es el más complejo de todos debido a que nos permite utilizar funciones de comunicación con en el sistema

operativo; *sys* permite acceder de una manera más completa a los datos que se ingresan y almacenan desde consola; *struct* maneja datos en formato binario usualmente utlizados en conexiones de red; *time* provee acceso a funciones de tiempo, nuestro interés son los temporizadores del sistema; *random* permite generar números pseudo-aleatorios.

```
7   import random
8   import time
9   import struct
10  import sys
11  import os
```

Para finalizar con los archivos de cabecera hacemos el llamado a dos archivos adicionales que contienen la forma en que se configura la USRP para transmisión y recepción, estos serán estudiados en otra sección.

```
14   import usrp_transmit_path
15   import usrp_receive_path
```

La parte que define a este programa es la creación de la interfaz de red virtual mediante el módulo TUNTAP, lo primero que definimos son los posibles modos de funcionamiento de la interfaz, como ya hemos visto anteriormente las principales opciones son *ethernet* o IP, adicionalmente utilizamos el modo *IFF_NO_PI* para indicar que no se agregen bytes adicionales a las tramas que se envian, por lo general se agregan cuatro bytes, dos al inicio y dos al final de la trama.

```
# Linux specific...
# TUNSETIFF ifr flags from <linux/tun_if.h>

IFF_TUN         = 0x0001   # tunnel IP packets
IFF_TAP         = 0x0002   # tunnel ethernet frames
IFF_NO_PI       = 0x1000   # don't pass extra packet info
IFF_ONE_QUEUE   = 0x2000   # beats me ;)
```

La función *open_tun_interface* se encarga de crear y abrir la interfaz virtual, para esto necesitamos tener acceso a ficheros de dispositivos, motivo por el cual importamos el módulo *ioctl*. Como primer paso utilizamos la función *os.open* que nos permitirá comunicarnos con la interfaz virual debido a que retorna un descriptor[2] y sus argumentos son el nombre de la interfaz y los permisos de lectura y escritura para el descriptor; ahora para crear la interfaz se llama la función *ioctl* que requiere tres parámetros, el descriptor del fichero del dispositivo apropiado, el número de ioctl (constante TUNSETIFF), y una estructura con parámetros como el nombre de la interfaz y el modo de operación.

```
def open_tun_interface(tun_device_filename):
    from fcntl import ioctl

    mode = IFF_TAP | IFF_NO_PI
    TUNSETIFF = 0x400454ca

    tun = os.open(tun_device_filename, os.O_RDWR)
    ifs = ioctl(tun, TUNSETIFF, struct.pack("16sH", "gr%d", mode))
    ifname = ifs[:16].strip("\x00")
    return (tun, ifname)
```

Si la llamada al módulo TUNTAP es exitosa, el programa nos guiará para que configuremos la nueva interfaz virtual mediante el comando *ifconfig*, la manera en que Python despliega mensajes en consola mediante *print* se asemeja a la que utilizamos en C++. En las siguientes líneas de código podemos observar que para la salida de texto acompañada de una variable tipo *string* se escribe el texto entre comillas y el momento de hacer referencia a la variable usa el marcador *%s* para luego especificar que variable queremos presentar mediante *% (tun_ifname,)*.

```
print "Allocated virtual ethernet interface: %s" % (tun_ifname,)
print "You must now use ifconfig to set its IP address. E.g.,"
print
print "sudo ifconfig %s 192.168.200.1" % (tun_ifname,)
print
print "Use a different address in the same subnet for each machine."
```

[2]Un descriptor en el núcleo es comparable a un puerto en la capa de transporte

El resultado de la secuencia anterior de comandos *print* es la siguiente:

```
Allocated virtual ethernet interface: gr0
You must now use ifconfig to set its IP address. E.g.,

   sudo ifconfig gr0 192.168.200.1

Use a different address in the same subnet for each machine.
```

La estructura principal del programa es la clase[3] *my_top_block*, el constructor contiene instancias a los módulos *transmit_path* y *receive_path* que son bloques jerárquicos donde se configura la USRP para transmisión y recepción, algo muy particular de estos bloques es que el método *connect* hace referencia solo a *self* y así mismos, esto se debe a que son bloques puramente sumidero y puramente fuente.

```
class my_top_block(gr.top_block):

    def __init__(self, mod_class, demod_class,
                 rx_callback, options):

        gr.top_block.__init__(self)
        self.txpath = transmit_path(mod_class, options)
        self.rxpath = receive_path(demod_class, rx_callback, options)
        self.connect(self.txpath);
        self.connect(self.rxpath);
```

Otra clase relevante es *cs_mac*, que contiene las funciones necesarias para establecer la comunicación entre la interfaz virtual y la capa física, ya sea que los datos se envíen hacia la interfaz virtual o provengan de esta. La función *phy_rx_callback* es la encargada de recibir los datos de la capa física y los pasa a la interfaz virtual; el parámetro *ok* indica que no hay errores en la trama, entonces los datos recibidos en la variable *payload* son pasados al descriptor de la interfaz virtual.

```
def phy_rx_callback(self, ok, payload):

    if self.verbose:
```

[3]En programación orientada a objetos una clase contiene los atributos y métodos que definen a un objeto

```
            print "Rx: ok = %r  len(payload) = %4d" % (ok, len(payload))
        if ok:
            os.write(self.tun_fd, payload)
```

Si el flujo de datos proviene de la interfaz virtual hacia la capa física, estos son tratados por la función *main_loop*, donde en caso de ocurrir algún problema en la lectura de la interfaz virtual, se introduce la variable *delay* para que el transmisor espere un tiempo prudencial hasta el siguiente intento de transmisión, de ser así en pantalla solo observamos una letra B indicando un período de espera *back-off*; caso contrario *tb.send_pkt* transmitirá la trama.

```
def main_loop(self):

        while 1:
            payload = os.read(self.tun_fd, 10*1024)
            if not payload:
                self.tb.send_pkt(eof=True)
                break

            if self.verbose:
                print "Tx: len(payload) = %4d" % (len(payload),)

            delay = min_delay
            while self.tb.carrier_sensed():
                sys.stderr.write('B')
                time.sleep(delay)
                if delay < 0.050:
                    delay = delay * 2    # exponential back-off

            self.tb.send_pkt(payload)
```

En Python podemos verificar los argumentos que recibe nuestro programa desde la línea de comandos utilizando el módulo *OptionParser*[4], esto nos ayuda a que la aplicación sea más amigable debido a que es posible generar un menú que despliega los argumentos que este requiere. Para generar nuestro menú debemos crear una instancia de *OptionParser*, uno de

[4]Interpretamos el nombre del módulo como analizador de opciones

los principales parámetros que recibe esta función es el método de resolución de conflictos aplicable, adicionalmente debido a que vamos a manejar una gran cantidad de opciones es recomendable agruparlas, por este motivo creamos el grupo *Expert*.

```
parser = OptionParser (option_class=eng_option, \
                        conflict_handler="resolve")

expert_grp = parser.add_option_group("Expert")
```

La función *add_option* se encarga de agregar al menú las opciones que necesitemos, donde cada una de estas opciones admite varios parámetros, si analizamos el código los dos primeros parámetros realizan la misma tarea, seleccionar el tipo de modulación con la diferencia de que una presenta un formato más corto que la otra, es decir, luego de escribir en la consola *python tunnel.py* es posible ingresar *-m dbpsk* o *–modulation dbpsk*, ahora el parametro *type="choice"* nos indica que la variable modulación tiene varias opciones predefinidas y su valor se obtiene de la función de GNU radio *mods.keys()* y en caso de no ingresar ningún valor se utiliza por defecto *gmsk*. Por ultimo con *help* ingresamos una pequeña explicación acerca del parámetro a ingresar, esta explicación se utiliza como ayuda y para invocarla ingresamos desde teclado *python tunnel.py -h*

```
parser.add_option("-m", "--modulation", type="choice", \
         choices=mods.keys(),
         default='gmsk',
         help="Select modulation from: %s [default=%%default]"
         % (', '.join(mods.keys()),))

expert_grp.add_option("-c", "--carrier-threshold", \
         type="eng_float",
         default=30,
         help="set carrier detect threshold (dB) [default=%default]")
```

Una parte de la opción de ayuda también se despliega el momento que llamamos el programa desde consola y no ingresamos ningún argumento.

```
if len(args) != 0:
        parser.print_help(sys.stderr)
```

```
        sys.exit(1)

if options.rx_freq is None or options.tx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)
```

Anteriormente se definió la función *open_tun_interface*, ahora para poder hacer uso de la interfaz, hacemos el llamado mediante:

```
# open the TUN/TAP interface
(tun_fd, tun_ifname) = open_tun_interface(options.tun_device_filename)
```

Si bien el control de los valores ingresados por consola para evitar errores es importante, el siguiente grupo de funciones que encontramos en la función *main*, crean las instancias de los objetos definidos por las clases inicialmente definidas, para luego establecer comunicación con la interfaz:

Creamos una instancia de la clase *cs_mac* que contiene todos los métodos necesarios para enviar los datos a la interfaz virtual

```
# instantiate the MAC
mac = cs_mac(tun_fd, verbose=True)
```

Generamos el diagrama de flujo que contiene todos los bloques que definen al radio *tunnel.py*, que debido a su complejidad[5], necesita de varios argumentos:

```
tb = my_top_block(mods[options.modulation],
                  demods[options.modulation],
                  mac.phy_rx_callback,
                  options)
```

Los datos deben viajar desde la capa física, que esta determinada por *tb*, hacia la capa de control de acceso MAC, es por esto que la clase *cs_mac* cuenta con la siguiente función:

[5] Comparado a *dial_tone.py*

```
mac.set_top_block(tb)
```

Ahora solo nos falta agregar las acciones de control para el diagrama de flujo mediante las funciones propias de GNU radio:

```
tb.start()      # Start executing the flow graph
mac.main_loop()     # don't expect this to return
tb.stop()       # but if it does, tell flow graph to stop.
tb.wait()       # wait for it to finish
```

4.2.2. Módulo transmit_path.py

Como ya hemos visto en el capítulo anterior, la USRP debe ser configurada como sumidero (*sink*) para desempeñar las funciones de un transmisor, sin embargo, para evitar que el código fuente se vuelva demasiado complejo, esta configuración ha sido separada en un archivo distinto.

Los módulos que importamos para este programa son casi los mismos que los utilizados en *tunnel.py*, sino que adicionalmente necesitamos de *blks2* que nos provee de herramientas adicionales para modulación, demodulación y diseño de filtros; *gru* entrega herramientas matemáticas y otras utilidades; por último *pick_bitrate* calcula la velocidad de transmisión de los datos, basado en el tipo de modulación que vamos a utilizar[6].

```
from gnuradio import gr, gru, blks2
from gnuradio import usrp
from gnuradio import eng_notation

import copy
import sys

# from current dir
from pick_bitrate import pick_tx_bitrate
```

[6]Los valores de bit por simbolo según la modulacón son BPSK 1, QPSK 2, 8PSK 3

El programa *transmit_path.py* esta estructurado en forma de un bloque jerárquico, el nombre que identifica a este bloque es *transmit_path* y los valores *input* y *output signature* tienen asignado un valor de cero debido a que su función es la de recibir datos, es decir, es un sumidero.

```
class transmit_path(gr.hier_block2):
    def __init__(self, modulator_class, options):

        gr.hier_block2.__init__(self, "transmit_path",
                                gr.io_signature(0, 0, 0),
                                gr.io_signature(0, 0, 0))
```

Los principales atributos de la USRP también se encuentran declarados en esta clase, y como lo veremos más adelante, en su gran mayoría son casi los mismos que los de la clase que define al bloque jerárquico para la recepción, *receive_path*.

```
# make a copy so we can destructively modify
options = copy.copy(options)

    self._verbose            = options.verbose
    # tranmitter's center frequency
    self._tx_freq            = options.tx_freq
    # digital amplitude sent to USRP
    self._tx_amplitude       = options.tx_amplitude
    # daughterboard to use
    self._tx_subdev_spec     = options.tx_subdev_spec
    # desired bit rate
    self._bitrate            = options.bitrate
    # interpolating rate for the USRP (prelim)
    self._interp             = options.interp
    # desired samples/baud
    self._samples_per_symbol = options.samples_per_symbol
    # usb info for USRP
    self._fusb_block_size    = options.fusb_block_size
    # usb info for USRP
```

```
    self._fusb_nblocks        = options.fusb_nblocks
    # increment start of whitener XOR data
    self._use_whitener_offset = options.use_whitener_offset
```

Para el envío de los paquetes tenemos la función *packet_transmitter*, que se encarga de aplicar un modulador a la información que queremos transmitir, los argumentos de esta función son: el tipo de modulador que vamos a aplicar; *access_code* es una secuencia de bits que identifican el inicio de la trama; *msgq_limit* es el número máximo de tramas que pueden esperar en cola; *pad_for_usrp* se utiliza como relleno para completar 128 muestras en los paquetes.

```
self.packet_transmitter = \
    blks2.mod_pkts(self._modulator_class(**mod_kwargs),
                   access_code=None,
                   msgq_limit=4,
                   pad_for_usrp=True,
                   use_whitener_offset=options.use_whitener_offset)
```

Los paquetes que se encuentran en cola para ser enviados son procesados por la subfunción *send_pkt*, estos datos son conocidos como *payload* y finalizan con el caracter EOF. Debemos observar la manera especial muy caracteristica de Python en la que es invocada esta función.

```
def send_pkt(self, payload='', eof=False):

        return self.packet_transmitter.send_pkt(payload, eof)
```

El trabajo de configurar la USRP es realizado por una gran cantidad de funciones, y estas funciones las encontramos reunidas en *_setup_usrp_sink*, que se encarga simplificar esta tarea, debido a que aqui se determina automaticamente los valores más adecuados para la mayoría de los atributos de las clase *transmit_path*.

La primera función definida en *_setup_usrp_sink* es *usrp.sink_c*[7], y es la encargada de configurar la USRP específicamente como un bloque sumidero que aceptará datos para pos-

[7]El sufijo c indica una entrada compleja flotante de 4 bytes

teriormente transmitirlos, los argumentos *fusb* permiten controlar la manera en que enviamos los datos a través del bus USB[8].

```
def _setup_usrp_sink(self):
    self.u = usrp.sink_c(fusb_block_size=self._fusb_block_size,
                         fusb_nblocks=self._fusb_nblocks)
```

El siguiente grupo de datos son de vital importancia, especialmente el valor de interpolación, y los obtenemos de la función *pick_tx_bitrate*, basados en la velocidad del DAC, de estos valores depende el desempeño que tendrá el radio.

```
dac_rate = self.u.dac_rate();
(self._bitrate, self._samples_per_symbol, self._interp) = \
        pick_tx_bitrate(self._bitrate, \
            self._modulator_class.bits_per_symbol(), \
            self._samples_per_symbol, self._interp, dac_rate)

    self.u.set_interp_rate(self._interp)
```

También se simplifica la manera en que determinamos el tipo de *daughterboard* que estamos utilizando y los puertos donde las tarjetas se encuentran conectadas, esto se debe a que el identificador que estas tarjetas tienen grabado, es recuperado en la variable *subdev*. Otro dato importante es el valor del multiplexor, y depende de el tipo de *daugterboard* que estamos utilizando, obtenemos fácilmente el valor del multiplexor mediante la función *usrp.determine_tx_mux_value*.

```
# determine the daughterboard subdevice we're using
if self._tx_subdev_spec is None:
     self._tx_subdev_spec = usrp.pick_tx_subdevice(self.u)
self.u.set_mux(usrp.determine_tx_mux_value(\
              self.u, self._tx_subdev_spec))
self.subdev = usrp.selected_subdev(self.u, self._tx_subdev_spec)
```

[8]Si alteramos el tamaño de los bloques debemos tener en cuenta que el tamaño minimo es 512 bytes

A continuación sólo nos falta configurar la frecuencia a la que queremos transmitir, para lograrlo la función *u.tune* requiere de tres parámetros: el primero de ellos *subdev._which* indica que DUC utilizar; el tipo de tarjeta *daughterboard* que está conectada, identificada por *subdev*; para finalmente sintonizar el dispositivo a *target_freq*.

```
def set_freq(self, target_freq):

    r = self.u.tune(self.subdev._which, self.subdev, target_freq)
    if r:
        return True
```

4.2.3. Módulo receive_path.py

La etapa de recepción es muy similar a la de transmisión, el nombre de la clase principal que define a este bloque cambia a *receive_path*, sin embargo *input* y *output signature* siguen siendo valores nulos, debido a que ahora el bloque se comporta como una fuente. Los nombres de los atributos aunque similares también sufren un cambio, en su mayoría el prefijo TX es reemplazado por RX.

```
class receive_path(gr.hier_block2):
    def __init__(self, demod_class, rx_callback, options):

        gr.hier_block2.__init__(self, "receive_path",
                                gr.io_signature(0, 0, 0),
                                gr.io_signature(0, 0, 0))
```

Para seleccionar la porción del espectro que nos interesa, GNU radio facilita el diseño de filtros por el método de ventanas, los coeficientes del filtro[9] son determinados mediante la herramienta *firdes*, los argumentos necesarios son la ganancia, frecuencia de corte, el ancho de la banda de transición[10] y el tipo de filtro, para esta aplicación se utiliza un filtro pasa bajo basado en una ventana de Hann[11]. Una vez determinados los coeficientes podemos crear un

[9]Aunque no lo indica, los datos entregados por la función son de tipo complejo.
[10]Se debe tomar en cuenta que estos valores son normalizados
[11]Otros valores permitidos son WIN_HAMMING, WIN_BLACKMAN, WIN_RECTANGULAR y WIN_KAISER

filtro mediante la función *fft_filter_ccc*.

```
sw_decim = 1
chan_coeffs = gr.firdes.low_pass (\
            1.0,                   # gain
            sw_decim * self._samples_per_symbol, # sampling rate
            1.0,                   # midpoint of trans. band
            0.5,                   # width of trans. band
            gr.firdes.WIN_HANN)    # filter type

# Decimating channel filter
# complex in and out, float taps
self.chan_filt = gr.fft_filter_ccc(sw_decim, chan_coeffs)
```

Ahora para recibir los datos se aplica un demodulador de la misma clase que utilizamos el momento de la transmisión, los dos primeros argumentos son los mismos, el tipo de demodulador y el código de acceso para verficación de la trama, por último, el valor *threshold* detecta el código de acceso con una tolerancia de error de un bit.

```
self.packet_receiver = \
     blks2.demod_pkts(self._demod_class(**demod_kwargs),
                      access_code=None,
                      callback=self._rx_callback,
                      threshold=-1)
```

Una vez que se tiene los datos necesarios para configurar la USRP, un par de funciones se encargarán de monitorear la presencia de una portadora, es decir se puede configurar la sensibilidad de recepción. En caso de requerirlo, se puede almacenar los resultados de la detección de portadora; este es un buen ejemplo para la creación de otro tipo de sumidero, debido a que los datos son guardados en un archivo llamado *rxpower.dat*, generado por la función *gr.file_sink*[12]. Las funciones encargadas de realizar la medición son *probe_avg_mag_sqrd_cf*, su salida es almacenada en el archivo de datos, mientras que el valor retornado por la función *probe_avg_mag_sqrd_c* se guarda en la variable *probe*, misma

[12]file_sink_c produce datos complejos de punto flotante, cada complejo de 8 bytes se divide en 4 bytes para I, y 4 bytes para Q.

que será verificada por *self.probe.unmuted()* para determinar si se encuentra sobre el nivel requerido *thresh*.

```
alpha = 0.001
  thresh = 30    # in dB, will have to adjust

  if options.log_rx_power == True:
      self.probe = gr.probe_avg_mag_sqrd_cf(thresh ,alpha)
      self.power_sink = gr.file_sink(gr.sizeof_float , "rxpower.dat")
      self.connect(self.chan_filt , self.probe , self.power_sink)
  else:
      self.probe = gr.probe_avg_mag_sqrd_c(thresh ,alpha)
      self.connect(self.chan_filt , self.probe)
```

Las funciones restantes para configurar la USRP como receptor son análogas entre sí, *_setup_usrp_source* es muy similar a la función utilizada en el módulo para la transmisión *_setup_usrp_sink* e inclusive *set_freq* no tiene grandes variaciones, por este motivo no las describiremos.

4.3. Transmisión a 900 Mhz

Seleccionamos la banda de 900Mhz debido a que con el paso del tiempo los equipos de comunicaciones migran a frecuencias más elevadas en busca de un mayor ancho de banda, sin embargo cuando esto sucede, la nueva banda se satura inmediatamente y el ciclo se vuelve a repetir. Vamos a realizar el enlace con la USRP en la frecuencia de 950Mhz utilizando la tarjeta RFX900, para luego comparar su desempeño consigo mismo pero trabajando con frecuencias más elevadas.

En la figura 4.3 podemos observar la medición con un analizador de espectros, de la potencia de una señal transmitida con modulación GMSK y a una frecuencia de 950Mhz, los resultados de las mediciones de potencia para las modulaciones restantes las encontramos en el anexo C.

A continuación en la tabla 4.1 recopilamos los resultados de transmisiones con distintos

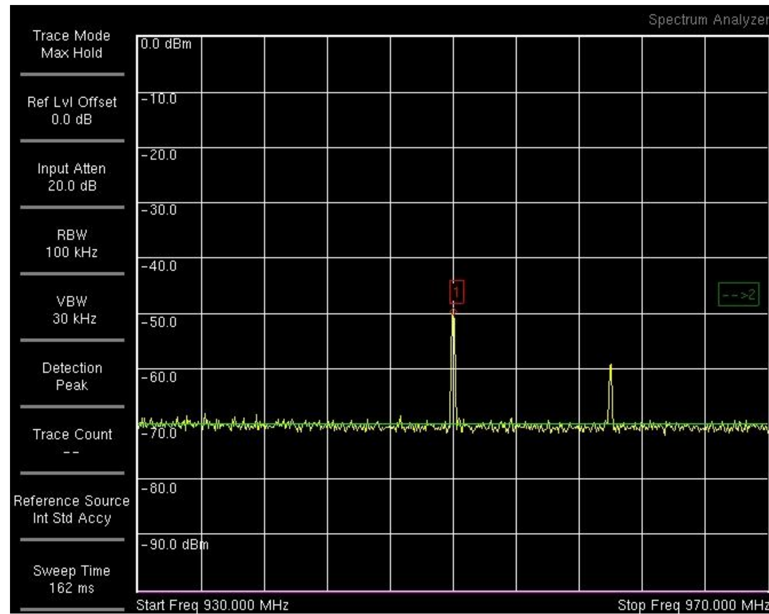

Figura 4.3: GMSK a 950Mhz.

tipos de modulaciones, lo que nos interesa es la tasa de transferencia y el tiempo de respuesta RTT al comando *ping*.

Modulación	Potencia(dBm)	TasaTX(Kbps)	TasaRX(Kbps)	RTT(ms)
GMSK	-47.32	125	125	31 - 58
DBPSK	-45.42	125	125	59 - 86
DQPSK	-45.8	250	250	41 - 58
D8PSK	errror	error	error	error

Tabla 4.1: Resultados de transmisión a 950Mhz

4.4. Transmisión a 2.4 Ghz

Actualmente la banda de los 2.4Ghz es una de las más utilizadas gracias a la popularidad de equipos tanto WiFi como BlueTooth. De la misma manera que lo hicimos en 950Mhz transmitiremos utilizando los mismos tipos de modulación anteriores, la figura 4.4 tenemos la señal modulada con GMSK pero en la frecuencia de 2412Mhz correspondiente al primer canal WiFi, para trabajar en esta frecuencia utilizamos la tarjeta RFX2400. Igualmente los

resultados de las mediciones de potencia para las modulaciones restantes las encontramos en el anexo C

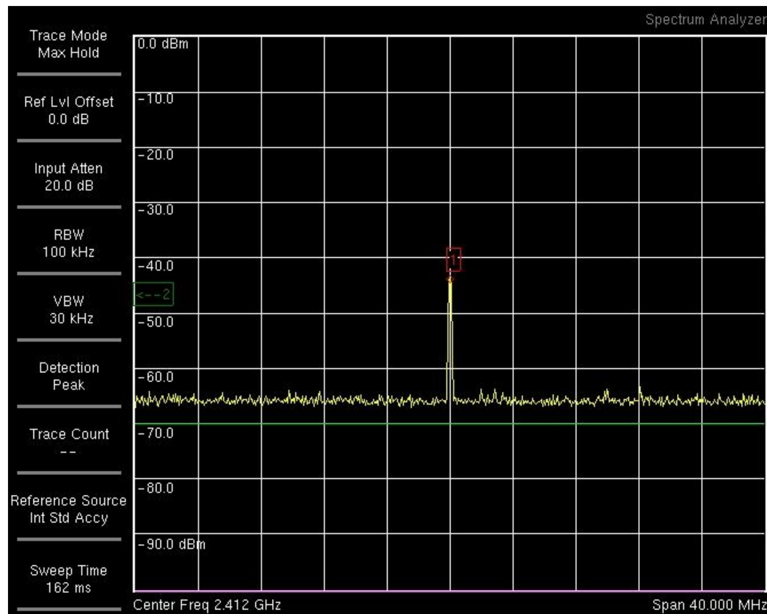

Figura 4.4: GMSK a 2412Mhz.

Los resultados son muy parecidos a los obtenidos en 950Mhz, aunque observamos un ligero aumento en el RTT de los paquetes enviados por el comando *ping*.

Modulación	Potencia(dBm)	TasaTX(Kbps)	TasaRX(Kbps)	RTT(ms)
GMSK	-43.9	125	125	70 - 100
DBPSK	-45.86	125	125	73 - 102
DQPSK	-44.39	250	250	60 - 95
D8PSK	error	error	error	error

Tabla 4.2: Resultados de transmisión a 2412Mhz

4.5. Análisis de resultados

En teoría, una de las causas que provoca elevados tiempos de respuesta RTT en la comunicación de las computadoras, se debe a la latencia que se produce por el intercambio de datos entre la USRP y cada una de las computadoras a través del bus USB. Si calculamos el retardo de transferencia de una trama ethernet generada por el módulo TAP, que es enviada por el puerto USB, suponiendo una MTU igual a 1518 *bytes* y la velocidad máxima de transferencia de 32MB/seg del bus USB[13], encontramos que el tiempo de transferencia es de 0.05 milisegundos.

Sin embargo existen más factores que afectan el desempeño de la USRP, uno de ellos es la pila FIFO que posee el FPGA, donde cada elemento de la pila posee un tamaño maximo de 512 *bytes*, entonces, si enviamos tramas de 512 en vez de 1518 *bytes*, disminuimos el tiempo de transferencia a $t1 = 0{,}02ms$, pero la responsabilidad de fragmentar los paquetes pasa al módulo TAP.

Ahora esta misma trama de 512 *bytes* antes de dirigirse al FPGA sufre un retardo $t2 = 4us$ al pasar por el proceso de *buffering* en el chip FX2[14], luego viaja hacia el FPGA, por un bus paralelo GPIF a una velocidad igual a 96MB/seg[6], demorando $t3 = 5{,}08ms$, entonces el tiempo parcial que viaja la trama es $t1 + t2 + t3 = 29ms$, tomando en cuenta que el paquete realiza un viaje de ida y vuelta, este tiempo pasa a ser $58ms$.

Si quisieramos recibir señales WiFi con el estandar 802.11b, que posee un ancho de canal de 11Mhz, necesitamos una velocidad de muestreo igual a 22MS/sec según Nyquist, aplicando muestreo en fase y cuadratura (*IQ sampling*), implicaría duplicar la cantidad de datos que deben ser enviados por el bus USB a un total de 44MB/s, superando el límite impuesto por el bus USB, a pesar de todo esto existe el proyecto *Implementation of Full-Bandwidth 802.11b Receiver* de la Universidad de Utah, que ha logrado recibir señales WiFi a velocidades de 1Mbps, mediante tecnicas de submuestreo.

[13]La velocidad máxima teórica del bus USB 2.0 es 60MB/seg, sin embargo existen señales de control que ocupan parte de este ancho de banda reduciendo la capacidad del mismo.

[14]Este tiempo puede ser disminuido manipulando la variable *fusb_block_size*

CAPÍTULO 5

CONCLUSIONES Y RECOMENDACIONES

Este capítulo se encuentra dividido en dos partes, en la primera comparamos las fortalezas y debilidades durante la instalación de GNU Radio en cada una de las distribuciones de Linux seleccionadas para esta tesis, Mandriva, Ubuntu, openSUSE y en Windows el *toolbox* simulink-USRP, mientras en la segunda parte evaluamos la estabilidad de la comunicación entre dos computadoras utilizando la USRP.

5.1. Mandriva

Siendo una de las distribuciones con una amplia trayectoria que se extiende desde sus inicios como Mandrake hasta su más reciente versión Mandriva 2010, realizamos pruebas para instalar GNU Radio con Mandriva 2007 y Mandriva 2010, siendo esta ultima versión aun inestable al menos en la configuración de los repositorios que aunque es muy sencilla de realizar, el momento de descargar actualizaciones.se presentan errores de compatibilidad y las herramientas de desarrollo inicialmente incluídas son muy limitadas a tal punto que no disponemos de la utilidad *make*.

Encontramos que el proceso de instalación de GNU Radio en Mandriva 2007 es el más sencillo de realizar comparado con otras distribuciones, prácticamente cumple con la mayoría de las dependencias y en caso de que alguna no se encuentre instalada los paquetes pueden descargarse sin problemas mediante el administrador de software que se encuentra en *Mandriva Control Center*.

5.2. Ubuntu

Es una de las distribuciones de Linux con mayor éxito en los últimos años, posee una gran cantidad de seguidores que mantienen foros y páginas de discusión donde se puede conseguir información que nos ayuda a resolver los diferentes inconvenientes que pudieramos tener, sin embargo la documentación para la instalación en Ubuntu se encuentra un poco desorganizada y por lo general es la última en ser actualizada cada vez que se lanza una nueva versión de GNU Radio.

Aunque en la página del proyecto GNU Radio encontramos una guía de instalación para las versiones[1] Ubuntu Edgy Eft 6.10 y Feisty Fawn 7,04, no recomendamos utilizar ninguna de estas debido a que los repositorios no cuentan con las dependencias actualizadas, lo más recomendable es trabajar con versiones más actuales, como mínimo sugerimos Ubuntu Intrepid Ibex 8.10 en caso de que intentemos compilar GNU Radio desde su código fuente.

El paquete binario debian de la versión 3.2.2 de GNU Radio es de gran ayuda porque evita el problema de lidiar con la resolución de las dependencias aunque solo es compatible con la versión de Ubuntu Jaunty Jackalope 9.04 y no incluye el paquete GRC.

El intento de trabajar con Ubuntu Karmic Koala 9.10 utilizando una computadora portátil HP modelo dv2025nr fracaso por falta de controladores y supuestos problemas con sectores defectuosos en el disco duro,

5.3. openSUSE

Aunque no es una de las distribuciones más conocidas openSUSE es la única distribución que le apuesta a KDE como su escritorio estándar y una de las pocas que se encuentra disponible en DVD con una gran cantidad de herramientras de apoyo, lo que evita en muchas ocasiones la necesidad de contar con una conexión de Internet para descargar actualizaciones.

Si bien el proceso de instalación de GNU Radio en openSUSE no es fácil debido a la gran cantidad de dependencias que tenemos que cumplir, solucionar esto no es un proceso

[1] Curiosamente los nombres de las distintas versiones de Ubuntu siguen un orden alfabetico

que un usuario no pueda enfrentar, en gran parte debido a que la mayoría de dependencias se encuentran disponibles forma de paquetes rpm en YAST o en la página de soporte de la comunidad de openSUSE.

De la misma manera que tenemos un paquete Debian para la instalación en Ubuntu, en openSUSE también disponemos de varios paquetes rpm de GNU Radio en la versión 3.1.3 de autoría de Michael Spraus que nos evitan el problema de recorrer la Internet en busca de las dependencias que necesitamos cumplir.

5.4. Windows

Ponemos énfasis en que el trabajo realizado en Windows fue la instalación del paquete Simulink-USRP, en ningún momento hemos instalado GNU radio sobre Windows mediante Cywin[2].

La mejor manera de evitar inconvenientes y conseguir una instalación limpia es cumplir con los requisitos siguiendo este orden:

1. Visual C++ 2008
2. Microsoft Windows SDK for Windows Server 2008
3. Matlab 7.4.0 (R2007a)
4. USRP driver
5. simulink-USRP

En teoría manejar cualquier trabajo sobre un ambiente Windows no tiene por qué ser complicado, sin embargo debemos tener mucho cuidado debido a que Visual C++ tiene la tendencia de ejecutar el *debugger* para intentar corregir cualquier aplicación que presente errores aún cuando esto no sea del todo verdadero.

En caso de no haber logrado instalar correctamente el *toolbox* y se tome la decisión de desinstalar del sistema a Visual C++, tenemos que estar seguros sobre cada una de las librerías dinámicas (archivos dll) que eliminamos, debido a que estas pueden estar compartidas

[2]Emulador del núcleo de Linux

por más de un programa y su eliminación provocaría daños irreparables en el sistema operativo.

5.5. Enlace

Los cálculos indican que el bus USB limita la velocidad de transferencia de datos, formando un cuello de botella entre la computadora y la USRP, para superar este problema se desarrolló la nueva versión de la USRP, conocida como USRP 2, que reemplaza el bus USB por una interfaz Gigabit Ethernet, que aparte de solucionar la baja tasa de transferencia, nos provee de una interfaz física, evitando tratar con una interfaz virtual.

Existen tablas teóricas que relacionan la intensidad de la señal recibida con la velocidad de transmisión de datos, por ejemplo, a una distancia de 109 metros, con una intensidad de señal de -71dBm se espera una velocidad de 54Mbps[3], sin embargo, utilizando la USRP no se encontró una relación semejante.

Sin importar la distribución de Linux que utilicemos, los resultados obtenidos fueron los mismos, por lo tanto es necesario realizar un análisis más exhaustivo de la función *pick_bitrate* para buscar la manera de optimizar la tasa de transferencia. En lo posible, se debe probar con otros esquemas de modulación más eficientes como lo son QAM y OFDM.

Por lo expuesto anteriormente, podemos reducir a dos las variables que determinan la eficiencia de la plataforma, el hardware en sí mismo, y el código del programa, hacemos énfasis en la optimización del programa. Aún así, sin importar las limitaciones, la USRP puede ser utilizada para proyectos con fines didácticos.

5.6. Trabajo futuro

Este trabajo es una primera aproximación al estudio de los radios definidos por software, nos proporciona una guía para comenzar a desarrollar múltiples aplicaciones orientadas a las comunicaciones inalámbricas. Aunque existen varios proyectos relacionados con GNU Radio, de los cuales se puede obtener nuevas experiencias y más fuentes de conocimiento, recomendamos continuar con el estudio de los siguientes proyectos.

[3]Estas tablas generalmente son aplicables a equipos WiFi

openBTS es uno de los proyectos más prometedores en el campo de las telecomunicaciones, mediante openBTS es posible implementar la infraestructura para una operadora GSM de telefonía celular, el hardware necesario se basa en la USRP y la generación de las llamadas esta a cargo de Asterisk.

GRC (*GNU Radio Companion*) es una herramienta que permite el desarrollo, mediante una interfaz gráfica, de radios definidos por software, los diagramas de flujo anteriormente escritos en GNU Radio se generan en una manera análoga a Simulink.

simulink-USRP es la aplicación que integra la USRP con Simulink, agregando una gran ventaja respecto a los demás proyectos, debido a que se apoya en Simulink, que es un programa conocido, estable y con soporte, eliminando en lo posible las complicaciones debido a la gran cantidad de dependencias de terceros.

ANEXOS

ANEXO A

PYTHON

La siguiente es una guía rápida muy básica para familiarizarse con Python, sin embargo para obtener información más avanzada recomendamos revisar la documentación oficial o cualquier libro especializado[1].

Para la declaración de variables se utiliza el operador “ = ” formalmente las variables no requieren definirse como un tipo específico de dato, es decir de acuerdo al valor que se les asigna asumen la característica de ser entero, caracter o cadena de caracteres[15].

```
q = 14
x,y,z = 1,2,3
primero, segundo = segundo, primero
a = b = 123
```

A diferencia de otros lenguajes de programación como C, Python define los bloques de código mediante sangrado o identación (tabulaciones), no existe la necesidad de colocar llaves ni nada por el estilo y todo lo que se escribe después del simbolo “ # ” se interpreta como un comentario.

```
if x < 5 or (x > 10 and x < 20):
        print "El valor es correcto."
# este codigo es equivalente al anterior
if x < 5 or 10 < x < 20:
        print "El valor es correcto."
#el siguiente es un lazo for
```

[1]Este anexo esta basado en los ejemplos presentados por Magnus Lie Hetland en su tutorial *Instant Python*.

```
for i in [1,2,3,4,5]:
        print "Pasada n = ", i
# y esto un bucle while
x = 10
while x >= 0:
        print "x aun no es negativo."
        x = x-1
```

Otra manera de realizar un lazo *for* es con la ayuda del comando *range()*.

```
# Mostrar en pantalla los valores de 0 a 99
for valor in range(100):
print valor
```

Para ingresar un valor entero por teclado

```
x = input("Introduce un numero:")
print "acabas de ingresar el numero:", x
```

En el caso de que necesitemos ingresar una cadena de texto utilizamos *raw_input()*

```
x = raw_input("Dime tu nombre: ")
print "tu nombre es:", x
```

Sin embargo ni el comando *input()* ni *raw_input()* verifican lo que el usuario ingresa por teclado, para evitar esto podemos utilizar la función de control *parser* que estudiaremos más adelante.

En Python existen estructuras de datos más avanzadas como las listas y los diccionarios. Primero nos enfocamos en las listas, que se declaran entre corchetes y sus valores se ingresan mediante índices que van desde cero hasta $n-1$ elementos que posee la lista, otra característica de las listas es que se pueden anidar.

```
elementos = ["diodos", "resistencias", "capacitores"]
```

```
#esta es una lista anidada
x = [[1,2,3],[a,b,c]]
#para visualizar elementos individuales
print elementos[1], elementos[0]
#para ingresar elementos individuales
elementos[0] = "transistores"
```

Los diccionarios son como las listas sino que los elementos se declaran entre llaves y cada uno de los elementos tiene una clave o “nombre” que se utiliza para buscar el elemento de la misma forma que en un diccionario de verdad.

```
empleado = {'nombre':"Kevin", 'apellido':"Shields",\
            'Cargo':"musico"}
#para cambiar un valor individual
persona['apellido']="Corgan"
```

Para la declaración de funciones utilizamos la palabra reservada “def” mientras que todos los parámetros se pasan por referencia.

```
def suma(num1, num2):
        """este texto entre comillas es la ayuda de la funcion"""
        return num1+num2
# para llamar a la funcion
suma(4, 2)
```

Algo muy útil de Python es que todo se considera un objeto, aunque el siguiente método parezca algo irrelevante cuando trabajamos con funciones más complejas encontraremos que en realidad es de gran ayuda, es así que podemos realizar la siguiente asignación:

```
#suma es la misma funcion del ejemplo anterior
a=suma
a(2, 4)
```

Para verificar que los datos ingresados por el usuario son correctos, y no provocarán

errores en el programa, utilizamos la módulo *optparse*, mediante el cual podemos especificar la sintaxis correcta para el ingreso de los datos, más una pequeña ayuda que sirve de guía para la utilización del programa. Ek siguiente código es un ejemplo de como podemos verificar el ingreso de sos números de tipo entero, que posteriormente serán usados en la función suma.

```
from optparse import OptionParser
#para mostrar ayuda
guia= "usage: %prog [opciones] num1 num2"
#creamos la instancia
parser = OptionParser(usage=guia)
#aqui creamos las opciones
parser.add_option("-x", "--x1", action="store", type="int", \
dest="x", help="primer numero")
parser.add_option("-y", "--y1", action="store", type="int", \
dest="y", help="segundo numero")
(options, args) = parser.parse_args()
```

ANEXO B

CÓDIGO FUENTE

B.1. tunnel.py

```
1  # Copyright 2005,2006,2009 Free Software Foundation, Inc.
2  #
3  # This file is part of GNU Radio
4
5  from gnuradio import gr, gru, modulation_utils
6  from gnuradio import usrp
7  from gnuradio import eng_notation
8  from gnuradio.eng_option import eng_option
9  from optparse import OptionParser
10
11 import random
12 import time
13 import struct
14 import sys
15 import os
16
17 # from current dir
18 import usrp_transmit_path
19 import usrp_receive_path
20 # Linux specific...
21 # TUNSETIFF ifr flags from <linux/tun_if.h>
22
23 IFF_TUN        = 0x0001   # tunnel IP packets
24 IFF_TAP        = 0x0002   # tunnel ethernet frames
25 IFF_NO_PI      = 0x1000   # don't pass extra packet info
26 IFF_ONE_QUEUE  = 0x2000   # beats me ;)
27
28 def open_tun_interface(tun_device_filename):
```

```
29      from fcntl import ioctl
30      mode = IFF_TAP | IFF_NO_PI
31      TUNSETIFF = 0x400454ca
32      tun = os.open(tun_device_filename, os.O_RDWR)
33      ifs = ioctl(tun, TUNSETIFF, struct.pack("16sH", "gr%d", mode))
34      ifname = ifs[:16].strip("\x00")
35      return (tun, ifname)
36
37  # /////////////////////////////////////////////////////////////////
38  #                        the flow graph
39  # /////////////////////////////////////////////////////////////////
40
41  class my_top_block(gr.top_block):
42
43      def __init__(self, mod_class, demod_class,
44                   rx_callback, options):
45
46          gr.top_block.__init__(self)
47          self.txpath = usrp_transmit_path.usrp_transmit_path\
48                  (mod_class, options)
49          self.rxpath = usrp_receive_path.usrp_receive_path\
50                  (demod_class, rx_callback, options)
51          self.connect(self.txpath)
52          self.connect(self.rxpath)
53
54      def send_pkt(self, payload='', eof=False):
55          return self.txpath.send_pkt(payload, eof)
56
57      def carrier_sensed(self):
58          """Return True if receive path thinks there's carrier"""
59          return self.rxpath.carrier_sensed()
60
61  # /////////////////////////////////////////////////////////////
62  #                     Carrier Sense MAC
63  # /////////////////////////////////////////////////////////////
64
65  class cs_mac(object):
66      """Prototype carrier sense MAC
67
68      Reads packets from the TUN/TAP interface, and sends them
```

```
69        to the PHY.
70        Receives packets from the PHY via phy_rx_callback, and
71        sends them into the TUN/TAP interface.
72
73        Of course, we're not restricted to getting packets via
74        TUN/TAP, this is just an example."""
75        def __init__(self, tun_fd, verbose=False):
76            self.tun_fd = tun_fd # file descriptor for TUNTAP interface
77            self.verbose = verbose
78            self.tb = None        # top block (access to PHY)
79
80        def set_top_block(self, tb):
81            self.tb = tb
82
83        def phy_rx_callback(self, ok, payload):
84            """Invoked by thread associated with PHY to pass received
85            packet up.
86            @param ok: bool indicating whether payload CRC was OK
87            @param payload: contents of the packet (string)"""
88            if self.verbose:
89                print "Rx: ok = %r  len(payload) = %4d"\
90                % (ok, len(payload))
91            if ok:
92                os.write(self.tun_fd, payload)
93
94        def main_loop(self):
95            """Main loop for MAC.
96            Only returns if we get an error reading from TUN.
97
98            FIXME: may want to check for EINTR and EAGAIN and reissue
99            """
100           min_delay = 0.001               # seconds
101
102           while 1:
103               payload = os.read(self.tun_fd, 10*1024)
104               if not payload:
105                   self.tb.send_pkt(eof=True)
106                   break
107
108               if self.verbose:
```

```
109                     print "Tx: len(payload) = %4d" % (len(payload),)
110
111                 delay = min_delay
112                 while self.tb.carrier_sensed():
113                     sys.stderr.write('B')
114                     time.sleep(delay)
115                     if delay < 0.050:
116                         delay = delay * 2    # exponential back-off
117
118                 self.tb.send_pkt(payload)
119
120 # /////////////////////////////////////////////////////////
121 #                          main
122 # /////////////////////////////////////////////////////////
123
124 def main():
125
126     mods = modulation_utils.type_1_mods()
127     demods = modulation_utils.type_1_demods()
128
129     parser = OptionParser (option_class=eng_option, \
130                            conflict_handler="resolve")
131     expert_grp = parser.add_option_group("Expert")
132     expert_grp.add_option("", "--rx-freq", type="eng_float", \
133             default=None,
134             help="set Rx frequency to FREQ [default=%default]",
135             metavar="FREQ")
136     expert_grp.add_option("", "--tx-freq", type="eng_float", \
137             default=None,
138             help="set transmit frequency to FREQ [default=%default]",
139             metavar="FREQ")
140     parser.add_option("-m", "--modulation", type="choice", \
141             choices=mods.keys(), default='gmsk',
142             help="Select modulation from: %s [default=%%default]"
143             % (', '.join(mods.keys()),))
144
145     parser.add_option("-v","--verbose", action="store_true", \
146                       default=False)
147     expert_grp.add_option("-c", "--carrier-threshold", \
148             type="eng_float", default=30,
```

```
149             help="set carrier detect threshold dB [default=%default]")
150     expert_grp.add_option("","--tun-device-filename", \
151             default="/dev/net/tun",
152             help="path to tun device file [default=%default]")
153 
154     usrp_transmit_path.add_options(parser, expert_grp)
155     usrp_receive_path.add_options(parser, expert_grp)
156 
157     for mod in mods.values():
158         mod.add_options(expert_grp)
159 
160     for demod in demods.values():
161         demod.add_options(expert_grp)
162 
163     (options, args) = parser.parse_args ()
164     if len(args) != 0:
165         parser.print_help(sys.stderr)
166         sys.exit(1)
167 
168     # open the TUN/TAP interface
169     (tun_fd, tun_ifname) = open_tun_interface\
170                              (options.tun_device_filename)
171 
172     # Attempt to enable realtime scheduling
173     r = gr.enable_realtime_scheduling()
174     if r == gr.RT_OK:
175         realtime = True
176     else:
177         realtime = False
178         print "Note: failed to enable realtime scheduling"
179 
180 # If the user hasn't set the fusb_* parameters on the command line,
181 # pick some values that will reduce latency.
182 
183     if options.fusb_block_size == 0 and options.fusb_nblocks == 0:
184         if realtime:                        # be more aggressive
185             options.fusb_block_size = gr.prefs().get_long\
186                                       ('fusb', 'rt_block_size', 1024)
187             options.fusb_nblocks    = gr.prefs().get_long\
188                                       ('fusb', 'rt_nblocks', 16)
```

```
189         else:
190             options.fusb_block_size = gr.prefs().get_long\
191                                    ('fusb', 'block_size', 4096)
192             options.fusb_nblocks    = gr.prefs().get_long\
193                                    ('fusb', 'nblocks', 16)
194 
195     #print "fusb_block_size =", options.fusb_block_size
196     #print "fusb_nblocks    =", options.fusb_nblocks
197 
198     # instantiate the MAC
199     mac = cs_mac(tun_fd, verbose=True)
200 
201     # build the graph (PHY)
202     tb = my_top_block(mods[options.modulation],
203                       demods[options.modulation],
204                       mac.phy_rx_callback,
205                       options)
206 
207     # give the MAC a handle for the PHY
208     mac.set_top_block(tb)
209 
210     if tb.txpath.bitrate() != tb.rxpath.bitrate():
211       print "WARNING: Transmit bitrate = %sb/sec, Receive bitrate = \
212         %sb/sec" % (
213             eng_notation.num_to_str(tb.txpath.bitrate()),
214             eng_notation.num_to_str(tb.rxpath.bitrate()))
215 
216     print "modulation:      %s"    % (options.modulation,)
217     print "freq:            %s"    \
218     % (eng_notation.num_to_str(options.tx_freq))
219     print "bitrate:         %sb/sec" \
220     % (eng_notation.num_to_str(tb.txpath.bitrate()),)
221     print "samples/symbol:  %3d" % (tb.txpath.samples_per_symbol(),)
222     #print "interp:         %3d" % (tb.txpath.interp(),)
223     #print "decim:          %3d" % (tb.rxpath.decim(),)
224 
225     tb.rxpath.set_carrier_threshold(options.carrier_threshold)
226     print "Carrier sense threshold:", options.carrier_threshold, "dB"
227 
228     print
```

```
229        print "Allocated virtual ethernet interface: %s" % (tun_ifname,)
230        print "You must now use ifconfig to set its IP address. E.g.,"
231        print
232        print "sudo ifconfig %s 192.168.200.1" % (tun_ifname,)
233        print
234        print "Use a different address same subnet for each machine."
235        print
236        # Start executing the flow graph (runs in separate threads)
237        tb.start()
238
239        mac.main_loop()    # don't expect this to return...
240
241        tb.stop()     # but if it does, tell flow graph to stop.
242        tb.wait()     # wait for it to finish
243
244    if __name__ == '__main__':
245        try:
246            main()
247        except KeyboardInterrupt:
248            pass
```

B.2. transmit_path.py

```
1  # Copyright 2005,2006,2009 Free Software Foundation, Inc.
2  #
3  # This file is part of GNU Radio
4
5  from gnuradio import gr, gru, blks2
6  from gnuradio import usrp
7  from gnuradio import eng_notation
8
9  import copy
10 import sys
11
12 # from current dir
13 from pick_bitrate import pick_tx_bitrate
14
15 # /////////////////////////////////////////////////////////////
16 #                          transmit path
17 # /////////////////////////////////////////////////////////////
```

```
18
19 class transmit_path(gr.hier_block2):
20     def __init__(self, modulator_class, options):
21         '''See below for what options should hold'''
22         gr.hier_block2.__init__(self, "transmit_path",
23                                 gr.io_signature(0, 0, 0),
24                                 gr.io_signature(0, 0, 0))
25         # make a copy so we can destructively modify
26         options = copy.copy(options)
27
28         self._verbose          = options.verbose
29         # tranmitter's center frequency
30         self._tx_freq          = options.tx_freq
31         # digital amplitude sent to USRP
32         self._tx_amplitude     = options.tx_amplitude
33         # daughterboard to use
34         self._tx_subdev_spec   = options.tx_subdev_spec
35         # desired bit rate
36         self._bitrate          = options.bitrate
37         # interpolating rate for the USRP (prelim)
38         self._interp           = options.interp
39         # desired samples/baud
40         self._samples_per_symbol = options.samples_per_symbol
41         # usb info for USRP
42         self._fusb_block_size    = options.fusb_block_size
43         # usb info for USRP
44         self._fusb_nblocks       = options.fusb_nblocks
45         # increment start of whitener XOR data
46         self._use_whitener_offset = options.use_whitener_offset
47         # the modulator_class we are using
48         self._modulator_class = modulator_class
49
50         if self._tx_freq is None:
51             sys.stderr.write\
52            ("-f FREQ or --freq FREQ or --tx-freq FREQ \
53             must be specified\n")
54             raise SystemExit
55
56         # Set up USRP sink; also adjusts interp,
57         # samples_per_symbol, and bitrate
```

```
58          self._setup_usrp_sink()
59
60      # copy the final answers back into options for use by modulator
61          options.samples_per_symbol = self._samples_per_symbol
62          options.bitrate = self._bitrate
63          options.interp = self._interp
64
65          # Get mod_kwargs
66          mod_kwargs = \
67              self._modulator_class.extract_kwargs_from_options(options)
68
69          # Set center frequency of USRP
70          ok = self.set_freq(self._tx_freq)
71          if not ok:
72              print "Failed to set Tx frequency to %s" \
73              % (eng_notation.num_to_str(self._tx_freq),)
74              raise ValueError
75
76          # transmitter
77          self.packet_transmitter = \
78              blks2.mod_pkts(self._modulator_class(**mod_kwargs),
79                      access_code=None,
80                      msgq_limit=4,
81                      pad_for_usrp=True,
82                      use_whitener_offset=options.use_whitener_offset)
83
84
85          # Set the USRP for maximum transmit gain
86          # (Note that on the RFX cards this is a nop.)
87          self.set_gain(self.subdev.gain_range()[1])
88
89          self.amp = gr.multiply_const_cc(1)
90          self.set_tx_amplitude(self._tx_amplitude)
91
92          # enable Auto Transmit/Receive switching
93          self.set_auto_tr(True)
94
95          # Display some information about the setup
96          if self._verbose:
97              self._print_verbage()
```

```
98
99          # Create and setup transmit path flow graph
100         self.connect(self.packet_transmitter, self.amp, self.u)
101
102     def _setup_usrp_sink(self):
103         """Creates a USRP sink, determines settings for best bitrate
104         and attaches to the transmitter's subdevice."""
105
106         self.u = usrp.sink_c(fusb_block_size=self._fusb_block_size,
107                              fusb_nblocks=self._fusb_nblocks)
108         dac_rate = self.u.dac_rate();
109
110         # derive values of bitrate, samples_per_symbol
111         # and interp from desired info
112         (self._bitrate, self._samples_per_symbol, self._interp) = \
113             pick_tx_bitrate(self._bitrate,
114                             self._modulator_class.bits_per_symbol(),
115                             self._samples_per_symbol, self._interp,
116                             dac_rate)
117
118         self.u.set_interp_rate(self._interp)
119
120         # determine the daughterboard subdevice we're using
121         if self._tx_subdev_spec is None:
122             self._tx_subdev_spec = usrp.pick_tx_subdevice(self.u)
123         self.u.set_mux(usrp.determine_tx_mux_value\
124                        (self.u, self._tx_subdev_spec))
125         self.subdev = usrp.selected_subdev\
126                      (self.u, self._tx_subdev_spec)
127
128
129     def set_freq(self, target_freq):
130         """Set the center frequency we're interested in.
131         @param target_freq: frequency in Hz
132         @rypte: bool
133         Tuning is a two step process. First we ask the front-end to
134         tune as close to the desired frequency as it can. Then we use
135         the result of that operation and our target_frequency to
136         determine the value for the digital up converter."""
137         r = self.u.tune(self.subdev._which, self.subdev, target_freq)
```

```
138            if r:
139                return True
140
141            return False
142
143        def set_gain(self, gain):
144            """Sets the analog gain in the USRP"""
145            self.gain = gain
146            self.subdev.set_gain(gain)
147
148        def set_tx_amplitude(self, ampl):
149            """Sets the transmit amplitude sent to the USRP
150            @param: ampl 0 <= ampl < 32768.  Try 8000"""
151
152            self._tx_amplitude = max(0.0, min(ampl, 32767.0))
153            self.amp.set_k(self._tx_amplitude)
154
155        def set_auto_tr(self, enable):
156            """Turns on auto transmit/receive of USRP daughterboard
157            (if exits; else ignored)"""
158            return self.subdev.set_auto_tr(enable)
159
160        def send_pkt(self, payload='', eof=False):
161            """Calls the transmitter method to send a packet"""
162            return self.packet_transmitter.send_pkt(payload, eof)
163
164        def bitrate(self):
165            return self._bitrate
166
167        def samples_per_symbol(self):
168            return self._samples_per_symbol
169
170        def interp(self):
171            return self._interp
172
173        def add_options(normal, expert):
174            """Adds transmitter-specific options to the Options Parser"""
175            add_freq_option(normal)
176            if not normal.has_option('--bitrate'):
177                normal.add_option("-r", "--bitrate", type="eng_float",\
```

```
178                 default=None,
179                 help="specify bitrate. samples-per-symbol and \
180                     interp/decim will be derived.")
181             normal.add_option("-T", "--tx-subdev-spec", type="subdev", \
182                 default=None,
183                 help="select USRP Tx side A or B")
184             normal.add_option("", "--tx-amplitude", type="eng_float", \
185                 default=12000, metavar="AMPL",
186                 help="set transmitter digital amplitude: \
187                     0 <= AMPL < 32768 [default=%default]")
188             normal.add_option("-v", "--verbose", action="store_true", \
189                   default=False)
190             expert.add_option("-S", "--samples-per-symbol", type="int", \
191                   default=None,
192                   help="set samples/symbol [default=%default]")
193             expert.add_option("", "--tx-freq", type="eng_float", \
194                   default=None,
195                   help="set transmit frequency to FREQ \
196                   [default=%default]", metavar="FREQ")
197             expert.add_option("-i", "--interp", type="intx", \
198                 default=None,
199                 help="set fpga interpolation rate to INTERP \
200                 [default=%default]")
201             expert.add_option("", "--log", action="store_true", \
202                 default=False,
203                 help="Log all parts of flow graph to file \
204                 (CAUTION: lots of data)")
205             expert.add_option("","--use-whitener-offset", \
206                 action="store_true", \
207                 default=False,
208                 help="make sequential packets use different whitening")
209
210         # Make a static method to call before instantiation
211         add_options = staticmethod(add_options)
212
213         def _print_verbage(self):
214             """Prints information about the transmit path"""
215             print "Using TX d'board %s" \
216             % (self.subdev.side_and_name(),)
217             print "Tx amplitude     %s" \
```

```
218             % (self._tx_amplitude)
219             print "modulation:        %s" \
220             % (self._modulator_class.__name__)
221             print "bitrate:           %sb/s" \
222             % (eng_notation.num_to_str(self._bitrate))
223             print "samples/symbol:   %3d" \
224             % (self._samples_per_symbol)
225             print "interp:            %3d"    % (self._interp)
226             print "Tx Frequency:      %s" \
227             % (eng_notation.num_to_str(self._tx_freq))
228 
229 def add_freq_option(parser):
230     """Hackery that has the -f/--freq option set
231     both tx_freq and rx_freq"""
232     def freq_callback(option, opt_str, value, parser):
233         parser.values.rx_freq = value
234         parser.values.tx_freq = value
235 
236     if not parser.has_option('--freq'):
237        parser.add_option('-f', '--freq', type="eng_float",
238         action="callback", callback=freq_callback,
239         help="set Tx and/or Rx frequency to FREQ [default=%default]",
240         metavar="FREQ")
```

B.3. receive_path.py

```
1  # Copyright 2005,2006,2009 Free Software Foundation, Inc.
2  #
3  # This file is part of GNU Radio
4  from gnuradio import gr, gru, blks2
5  from gnuradio import usrp
6  from gnuradio import eng_notation
7  import copy
8  import sys
9 
10 # from current dir
11 from pick_bitrate import pick_rx_bitrate
12 
13 # ////////////////////////////////////////////////////////////
14 #                      receive path
```

```
15 # //////////////////////////////////////////////////////////////
16
17 class receive_path(gr.hier_block2):
18     def __init__(self, demod_class, rx_callback, options):
19
20         gr.hier_block2.__init__(self, "receive_path",
21                                 gr.io_signature(0, 0, 0),
22                                 gr.io_signature(0, 0, 0))
23         # make a copy so we can destructively modify
24         options = copy.copy(options)
25
26         self._verbose          = options.verbose
27         # receiver's center frequency
28         self._rx_freq          = options.rx_freq
29         # receiver's gain
30         self._rx_gain          = options.rx_gain
31         # daughterboard to use
32         self._rx_subdev_spec = options.rx_subdev_spec
33         # desired bit rate
34         self._bitrate          = options.bitrate
35         # Decimating rate for the USRP (prelim)
36         self._decim            = options.decim
37         # desired samples/symbol
38         self._samples_per_symbol = options.samples_per_symbol
39         # usb info for USRP
40         self._fusb_block_size    = options.fusb_block_size
41         # usb info for USRP
42         self._fusb_nblocks       = options.fusb_nblocks
43         # this callback is fired when there's a packet available
44         self._rx_callback = rx_callback
45         # the demodulator_class we're using
46         self._demod_class = demod_class
47
48         if self._rx_freq is None:
49             sys.stderr.write\
50             ("-f FREQ or --freq FREQ or --rx-freq FREQ \
51              must be specified\n")
52             raise SystemExit
53
54         # Set up USRP source; also adjusts decim, samples_per_symbol,
```

```
55          # and bitrate
56          self._setup_usrp_source()
57
58          g = self.subdev.gain_range()
59          if options.show_rx_gain_range:
60              print "Rx Gain Range: minimum = %g, maximum = %g, \
61                    step size = %g"  % (g[0], g[1], g[2])
62
63          self.set_gain(options.rx_gain)
64          # enable Auto Transmit/Receive switching
65          self.set_auto_tr(True)
66
67          # Set RF frequency
68          ok = self.set_freq(self._rx_freq)
69          if not ok:
70              print "Failed to set Rx frequency to %s" \
71              % (eng_notation.num_to_str(self._rx_freq))
72              raise ValueError, eng_notation.num_to_str(self._rx_freq)
73
74     # copy the final answers back into options for use by demodulator
75          options.samples_per_symbol = self._samples_per_symbol
76          options.bitrate = self._bitrate
77          options.decim = self._decim
78
79          # Get demod_kwargs
80          demod_kwargs = \
81          self._demod_class.extract_kwargs_from_options(options)
82
83          # Design filter to get actual channel we want
84          sw_decim = 1
85          chan_coeffs = gr.firdes.low_pass\
86                 (1.0,                   # gain
87                  sw_decim * self._samples_per_symbol, # sampling rate
88                  1.0,                   # midpoint of trans. band
89                  0.5,                   # width of trans. band
90                  gr.firdes.WIN_HANN)# filter type
91
92          # Decimating channel filter
93          # complex in and out, float taps
94          self.chan_filt = gr.fft_filter_ccc(sw_decim, chan_coeffs)
```

```
95          #self.chan_filt = gr.fir_filter_ccf(sw_decim, chan_coeffs)
96
97          # receiver
98          self.packet_receiver = \
99              blks2.demod_pkts(self._demod_class(**demod_kwargs),
100                                 access_code=None,
101                                 callback=self._rx_callback,
102                                 threshold=-1)
103
104         # Carrier Sensing Blocks
105         alpha = 0.001
106         thresh = 30   # in dB, will have to adjust
107
108         if options.log_rx_power == True:
109             self.probe = gr.probe_avg_mag_sqrd_cf(thresh,alpha)
110             self.power_sink = \
111                  gr.file_sink(gr.sizeof_float, "rxpower.dat")
112             self.connect(self.chan_filt, self.probe, self.power_sink)
113         else:
114             self.probe = gr.probe_avg_mag_sqrd_c(thresh,alpha)
115             self.connect(self.chan_filt, self.probe)
116
117         # Display some information about the setup
118         if self._verbose:
119             self._print_verbage()
120
121         self.connect(self.u, self.chan_filt, self.packet_receiver)
122
123     def _setup_usrp_source(self):
124         self.u = usrp.source_c(fusb_block_size=self._fusb_block_size,
125                                   fusb_nblocks=self._fusb_nblocks)
126         adc_rate = self.u.adc_rate()
127
128         # derive values of bitrate, samples_per_symbol,
129         # and decim from desired info
130         (self._bitrate, self._samples_per_symbol, self._decim) = \
131             pick_rx_bitrate(self._bitrate,
132                               self._demod_class.bits_per_symbol(),
133                               self._samples_per_symbol, self._decim,
134                               adc_rate)
```

```
135
136         self.u.set_decim_rate(self._decim)
137
138         # determine the daughterboard subdevice we're using
139         if self._rx_subdev_spec is None:
140             self._rx_subdev_spec = usrp.pick_rx_subdevice(self.u)
141         self.subdev = usrp.selected_subdev(self.u,
142                                           self._rx_subdev_spec)
143
144         self.u.set_mux(usrp.determine_rx_mux_value\
145                                  (self.u, self._rx_subdev_spec))
146
147     def set_freq(self, target_freq):
148         """Set the center frequency we're interested in.
149
150         @param target_freq: frequency in Hz
151         @rypte: bool
152
153         Tuning is a two step process. First we ask the front-end to
154         tune as close to the desired frequency as it can. We use
155         the result of that operation and our target_frequency to
156         determine the value for the digital up converter."""
157         r = self.u.tune(0, self.subdev, target_freq)
158         if r:
159             return True
160
161         return False
162
163     def set_gain(self, gain):
164         """Sets the analog gain in the USRP"""
165         if gain is None:
166             r = self.subdev.gain_range()
167             gain = (r[0] + r[1])/2        # set gain to midpoint
168         self.gain = gain
169         return self.subdev.set_gain(gain)
170
171     def set_auto_tr(self, enable):
172         return self.subdev.set_auto_tr(enable)
173
174     def bitrate(self):
```

```
175             return self._bitrate
176 
177         def samples_per_symbol(self):
178             return self._samples_per_symbol
179 
180         def decim(self):
181             return self._decim
182 
183         def carrier_sensed(self):
184             """Return True if we think carrier is present."""
185             #return self.probe.level() > X
186             return self.probe.unmuted()
187 
188         def carrier_threshold(self):
189             """Return current setting in dB."""
190             return self.probe.threshold()
191 
192         def set_carrier_threshold(self, threshold_in_db):
193             """Set carrier threshold.
194 
195             @param threshold_in_db: set detection threshold
196             @type threshold_in_db:  float (dB)"""
197             self.probe.set_threshold(threshold_in_db)
198 
199         def add_options(normal, expert):
200             """Adds receiver-specific options to the Options Parser"""
201             add_freq_option(normal)
202             if not normal.has_option("--bitrate"):
203                 normal.add_option("-r", "--bitrate", type="eng_float",\
204                     default=None,
205                     help="specify bitrate. samples-per-symbol and \
206                         interp/decim will be derived.")
207             normal.add_option("-R", "--rx-subdev-spec", type="subdev", \
208                     default=None,
209                     help="select USRP Rx side A or B")
210             normal.add_option("", "--rx-gain", type="eng_float", \
211                     default=None, metavar="GAIN",
212                     help="set receiver gain in dB [default=midpoint]. \
213                         See also --show-rx-gain-range")
214             normal.add_option("", "--show-rx-gain-range", \
```

```
215                 action="store_true", \
216                 default=False,
217                 help="print min and max Rx gain available on \
218                     selected daughterboard")
219         normal.add_option("-v", "--verbose", action="store_true", \
220                 default=False)
221         expert.add_option("-S", "--samples-per-symbol", \
222                 type="int", \
223                 default=None,
224                 help="set samples/symbol [default=%default]")
225         expert.add_option("", "--rx-freq", type="eng_float", \
226                 default=None,
227                 help="set Rx frequency to FREQ [default=%default]",
228                     metavar="FREQ")
229         expert.add_option("-d", "--decim", type="intx", \
230                 default=None,
231                 help="set fpga decimation rate to DECIM \
232                     [default=%default]")
233         expert.add_option("", "--log", action="store_true", \
234                 default=False,
235                 help="Log all parts of flow graph to files \
236                     (CAUTION: lots of data)")
237         expert.add_option("", "--log-rx-power", \
238                 action="store_true", \
239                 default=False,
240                 help="Log receive signal power to file \
241                     (CAUTION: lots of data)")
242 
243     # Make a static method to call before instantiation
244     add_options = staticmethod(add_options)
245 
246     def _print_verbage(self):
247         """Prints information about the receive path"""
248         print "\nReceive Path:"
249         print "Using RX d'board %s"\
250         % (self.subdev.side_and_name(),)
251         print "Rx gain:         %g"    % (self.gain,)
252         print "modulation:      %s"\
253         % (self._demod_class.__name__)
254         print "bitrate:         %sb/s"\
```

```
255             % (eng_notation.num_to_str(self._bitrate))
256             print "samples/symbol:   %3d"\
257             % (self._samples_per_symbol)
258             print "decim:               %3d"    % (self._decim)
259             print "Rx Frequency:      %s"\
260             % (eng_notation.num_to_str(self._rx_freq))
261             # print "Rx Frequency:      %f"      % (self._rx_freq)
262 
263         def __del__(self):
264             # Avoid weak reference error
265             del self.subdev
266 
267 def add_freq_option(parser):
268     """Hackery that has the -f / --freq option set both
269        tx_freq and rx_freq"""
270     def freq_callback(option, opt_str, value, parser):
271         parser.values.rx_freq = value
272         parser.values.tx_freq = value
273 
274     if not parser.has_option('--freq'):
275         parser.add_option('-f', '--freq', type="eng_float",
276                 action="callback", callback=freq_callback,
277                 help="set Tx and/or Rx frequency to FREQ \
278                 [default=%default]",
279                 metavar="FREQ")
```

ANEXO C

MEDICIONES DEL ESPECTRO

C.1. 900Mhz

C.1.1. DBPSK

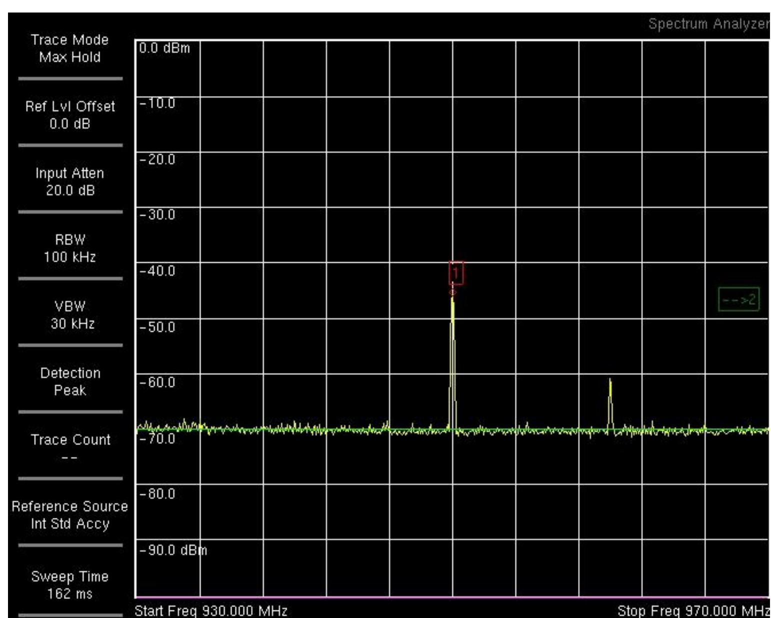

Figura C.1: DBPSK a 950Mhz.

C.1.2. DQPSK

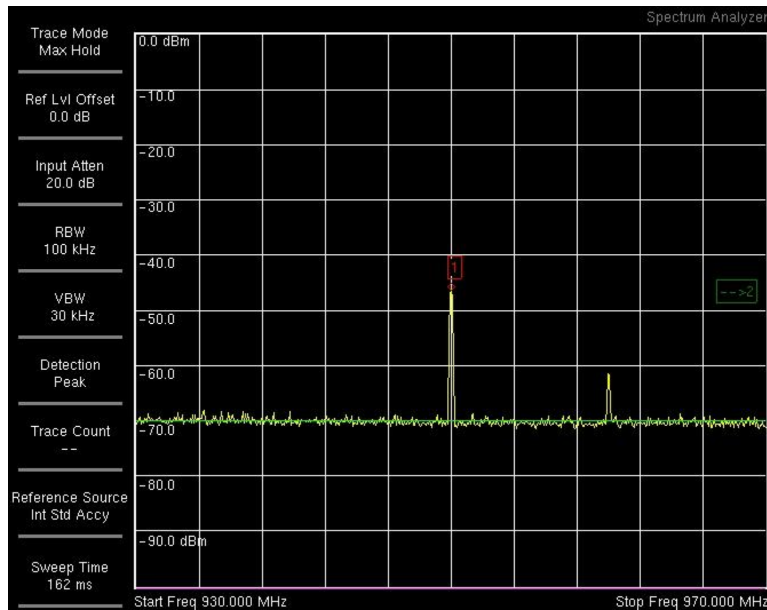

Figura C.2: DQPSK a 950Mhz.

C.1.3. D8PSK

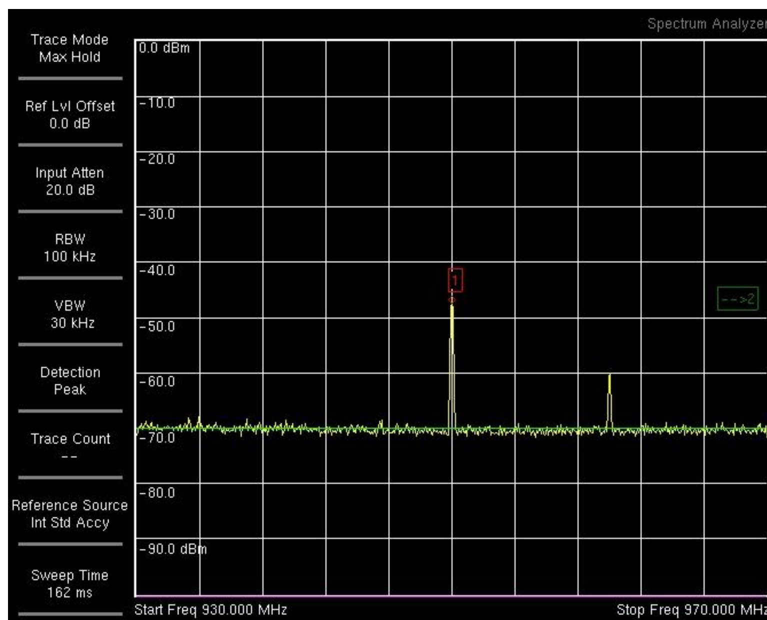

Figura C.3: D8PSK a 950Mhz.

C.2. 2400Mhz

C.2.1. DBPSK

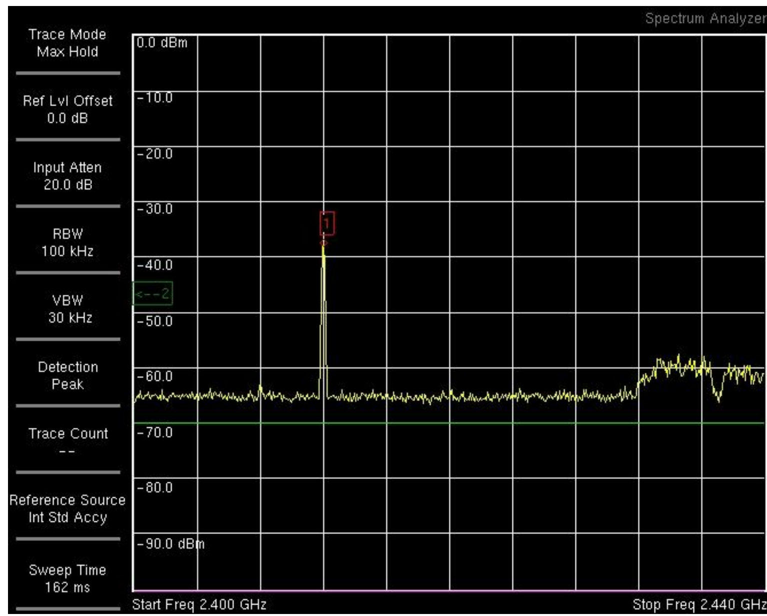

Figura C.4: DBPSK a 2412Mhz.

C.2.2. DQPSK

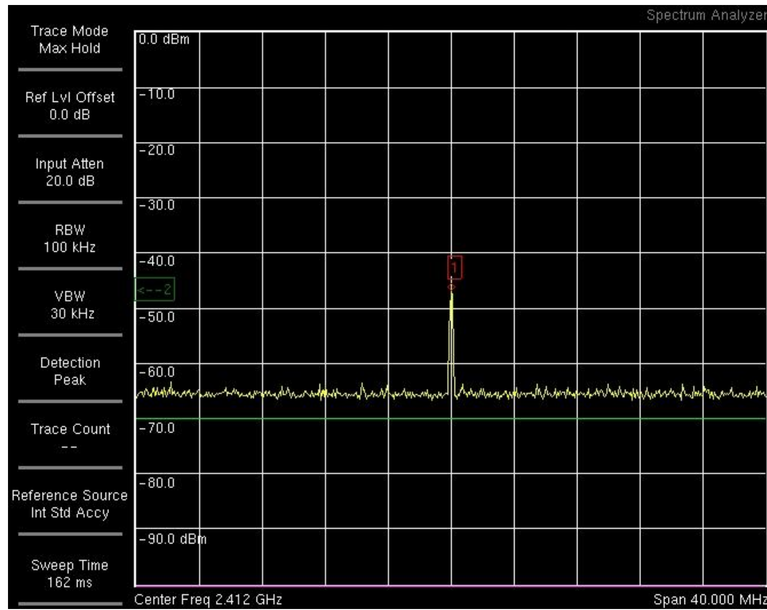

Figura C.5: DQPSK a 2412Mhz.

C.2.3. D8PSK

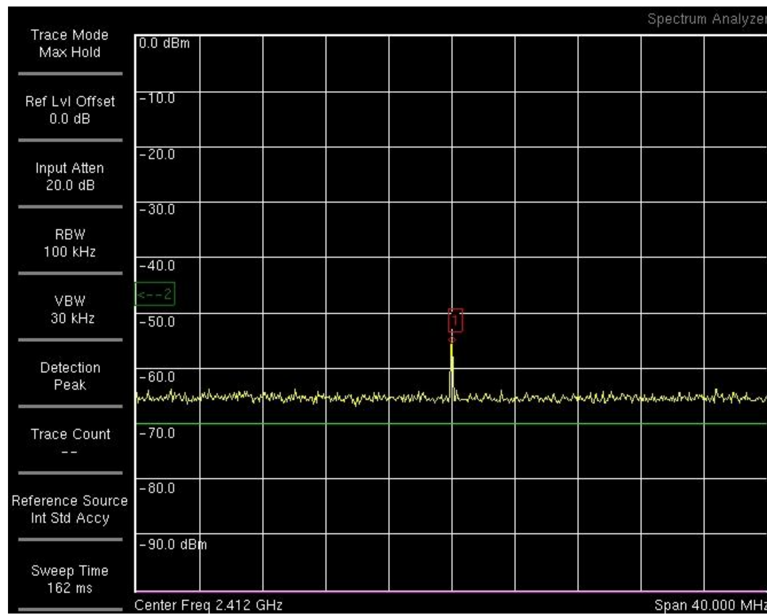

Figura C.6: D8PSK a 2412Mhz.

BIBLIOGRAFÍA

[1] Air Force Research Laboratory. *SPEAKeasy Military Software Defined Radio*, Julio 1998.

[2] Kiichi Niitsu. Reconfigurable RF system for software defined radio. Technical report, Keio University, 2006.

[3] FlexRadio Systems. The history of flexradio systems. http://www.flex-radio.com/About.aspx, Enero 2010.

[4] Andreas Viklund. Project description. http://openhpsdr.org/, Enero 2010.

[5] L. Koonts. Codecon 2.0 presentations. http://www.linuxjournal.com/node/6643, Febrero 2003.

[6] Firas Abbas Hamza. *The USRP under 1.5x magnifying lens*. GNU radio project, Junio 2008.

[7] T. Hollis y R. Weir. The theory of digital down conversion. Technical report, Hunt Engineering, Junio 2003.

[8] Carla Schroeder. *Curso de Linux*. Anaya Oreilly, España, segunda edition, 2005.

[9] David Bandel y Robert Napier. *Edición especial Linux*. Prentice Hall, España, sexta edition, 2001.

[10] George Kurtz Brian Hatch, James Lee. *Hackers en Linux*. McGraw-Hill, España, primera edition, 2001.

[11] Dawei Shen. Before diving into gnu radio, you should. Technical report, University of Notre Dame, Mayo 2005.

[12] Josh Blum. Gnuradio. http://www.joshknows.com/gnuradio, Febrero 2010.

[13] Josh Blum. Gnuradio. http://gnuradio.org/redmine/wiki/gnuradio/Tutorials-WritePythonApplications, Febrero 2010.

[14] Larry Baker. Microsoft 32/64-bit visual c++ 2008 express support files. http://www.mathworks.de/matlabcentral/fileexchange/22689, Enero 2009.

[15] Magnus Lie Hetland. Instant python. http://hetland.org/writing/instant-python.html, Septiembre 2009.

FECHA DE ENTREGA

El proyecto fue entregado al Departamento de Eléctrica y Electrónica y reposa en la Escuela Politécnica del Ejército desde:

Sangolqui, a ____del 2010

ELABORADO POR:

____

Juan Francisco Quiroz Terreros
1715446744

AUTORIDAD

____

Ing. Carlos Romero
Coordinador de la Carrera de Ingeniería en Electrónica
Redes y Comunicación de Datos