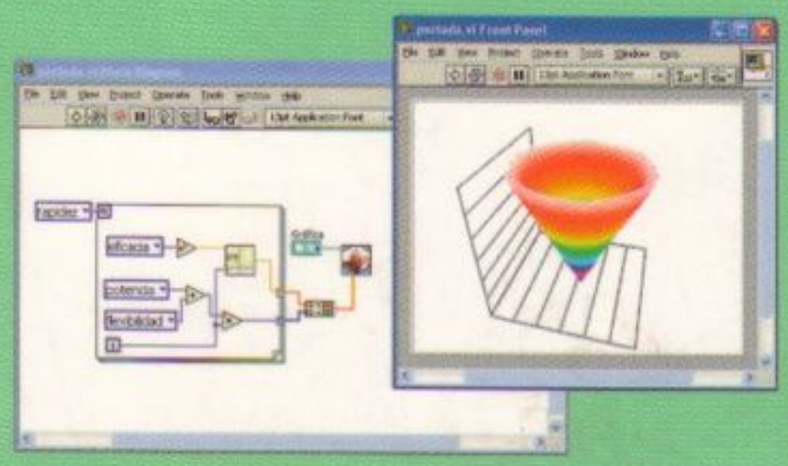
LabVIEW

Entorno gráfico de programación

José Rafael Lajara Vizcaíno
José Pelegrí Sebastiá

LabVIEW 8.20

y versiones anteriores



Incluye CD

Alfaomega

marcombo

Título: **LabVIEW. Entorno gráfico de programación**

Autores:
José Rafael Lajara Vizcaíno
José Pelegrí Sebastiá

Reservados todos los derechos de publicación, reproducción, préstamo, alquiler o cualquier otra forma de cesión del uso de este ejemplar de la presente edición española por
© MARCOMBO S. A. 2007
Gran Via de les Corts Catalanes, 594
08007 Barcelona

© ALFAOMEGA GRUPO EDITOR, S.A. 2007
C/ Pitágoras 1139
Colonia del Valle - 03100
México D.F. (México)

Quedan rigurosamente prohibidas, sin la autorización escrita de los titulares del *copyright*, bajo las sanciones establecidas en las leyes, la reproducción total o parcial de esta obra por cualquier medio o procedimiento, incluidos la reprografía y el tratamiento informático, así como la distribución de ejemplares mediante alquiler o préstamo públicos.

Consulte nuestro fondo editorial
http://www.marcombo.com

Pueden existir materiales adicionales a este libro atojados en la URL:
http://www.marcombo.com/descargas

El autor de la obra alojará en esta dirección materiales adicionales que considere de interés para sus lectores, y los irá actualizando periódicamente.

ISBN (por MARCOMBO): 84-267-1426-9
ISBN-13 (por MARCOMBO): 978-84-267-1426-8
ISBN (por ALFAOMEGA GRUPO EDITOR): 970-15-1133-6
ISBN-13 (por ALFAOMEGA GRUPO EDITOR): 978-970-15-1133-6

Depósito legal: BI-3379-06
Impreso en Gráficas Díaz Tuduri, S.L.

Agradecimientos

Queremos agradecer a una serie de personas su colaboración directa o indirectamente, algunos aportando medios y otros aportando información que hemos empleado en la elaboración de este texto.

Parte del presente trabajo está incluido dentro de los trabajos de la asignatura "Sistemas de Adquisición de Datos" en la titulación de Ingeniería Técnica de Telecomunicaciones en la Escuela Politécnica Superior de Gandia. Por tanto, de algún modo también ha colaborado el anterior profesor de la asignatura y compañero José Francisco Toledo Alarcón.

También nombrar a Pablo Soto, por iniciarnos en la impartición de cursos de postgrado, que posteriormente han dado lugar a este texto.

Por supuesto hemos de agradecer también la confianza que *National Instruments* ha puesto en nosotros a través de Javier Gutiérrez y Guillermo Prados y la ayuda prestada con la utilización de un equipo PXI. También agradecer a Jeroni Boixareu por creer en nosotros con este proyecto editorial.

Agradecer también el buen quehacer a todos los miembros del "Grupo de Sensores y Magnetismo" (GSYM) al cual pertenecemos y en especial a Jorge Alberola.

Finalmente, nos gustaría agradecer a nuestras familias por su cariño, paciencia y comprensión.

Introducción

Tras 20 años en el mercado LabVIEW se ha convertido en un estándar en el desarrollo de aplicaciones de test y medida, control de instrumentación y sistemas de adquisición de datos. Su flexibilidad y potencia, le ha hecho expandirse a otras áreas tales como visión artificial, PACs, control de movimiento, HMI y SCADAs para automatización industrial, análisis de ruido y vibraciones, gestión de información y generación de informes, etc. Desde hace 5 años, LabVIEW está creciendo en nuevas areas estratégicas donde reside nuestra visión desde National Instruments, relacionada con nuevos campos de trabajo como Simulación, Diseño de Control, sistemas embebidos en tiempo real (FPGAs, DSPs, microprocesadores), algoritmos matemáticos avanzados, etc.

Durante estos 20 años, LabVIEW ha tenido una gran expansión en la comunidad educativa y científica, tanto en universidades y centros de enseñanza secundaria, como en centros de investigación, en la elaboración de prácticas de laboratorio, clases teóricas en las áreas de control, instrumentación, tratamiento digital de la señal, electrónica, etc, en el desarrollo de proyectos fin de carrera, tesinas y tesis, y siendo un puente entre la comunidad educativa y la industria a nivel de I+D+i.

Es un placer, como director de *National Instruments España*, realizar el prólogo de esta 1 edición de su libro LabVIEW: Entorno de Programación Gráfico*f*, por la colaboración profesional durante 8 años con el grupo de trabajo de José Pelegrí y José Rafael Lajara, por su labor docente con LabVIEW durante tantos años y también por la confianza puesta en la programación gráfica como lenguaje de desarrollo.

Estamos convencidos de que esta edición totalmente adaptada a la versión de LabVIEW 8.20 será una referencia en español para el aprendizaje del lenguaje G, por su interpretación de los principios de funcionamiento de LabVIEW así como su carácter práctico en el desarrollo de ejercicios.

José Pelegrí y su grupo en la Universidad Politécnica de Valencia en Gandía han decidido dar un paso adelante en la divulgación científica y tecnológica con este libro, fruto de su experiencia como profesores e investigadores en ámbitos de la instrumentación virtual, empleando estándares industriales como PXI, sistemas de adquisición de datos y GPIB, así como cultivando una estrecha e intensa colaboración con National Instruments en estas áreas. Gracias por la confianza demostrada y °mucha suerte!

Luis Fco. Bussión Fernández
Director de National Instruments España.

Guillermo Prados Gimeno
Responsable de área de Valencia

Madrid. Noviembre de 2006.

Índice

Sección I. INTRODUCCIÓN

Capítulo 1. Introducción a LabVIEW. Entorno. 3

Introducción a la herramienta de programación LabVIEW. Presentación del entorno de trabajo y los menús de herramientas, controles y funciones. Se indica cómo crear un programa y cómo se ejecuta éste. Se introducen los conceptos de VI, subVI y jerarquía. Cómo generar un proyecto y depurar el de código. Termina con algunos ejercicios propuestos.

Capítulo 2. Estructuras. 37

Las instrucciones de control permiten a un programa ejecutar un código de forma condicional o repetirlo cierto número de veces. En LabVIEW estas instrucciones son estructuras que encierran en su interior el código al que afectan. Cabe destacar las nuevas estructuras introducidas en las últimas versiones como EVENT y TIMED LOOP, así como las tradicionales FOR y WHILE. Se indican algunos ejemplos prácticos y se proponen algunos ejercicios.

Capítulo 3. Tipos de datos. 63

Estudio en profundidad de los tipos de datos disponibles en LabVIEW y de su manipulación. También se explican conceptos como variables, referencias y propiedades que se pueden utilizar. Se finaliza el capítulo con ejemplos prácticos y ejercicios propuestos.

Sección II. ADQUISICIÓN Y COMUNICACIONES

Capítulo 4. Manejo de Ficheros. 91

En este capítulo se tratan las diferentes funciones para manejar ficheros mediante ejemplos. A parte de su utilidad también se asentarán los conceptos expuestos en la sección I. Se finaliza con ejercicios propuestos.

Capítulo 5. Comunicación serie. 115

La comunicación serie no se debe olvidar ya que está presente en multitud de dispositivos. En este capítulo se presentan dos ejemplos prácticos en los

que se puede ver la facilidad de implementar dicho tipo de comunicación en el entorno de LabVIEW.

Capítulo 6. Bus de comunicaciones GPIB. 131

En sus orígenes, LabVIEW se aplicaba fundamentalmente en el control de instrumentación y para tal fin se utilizaban diferentes buses de instrumentación, en este caso cabe tratar el bus GPIB. El capítulo comienza con una introducción teórica al estándar IEEE 488.1, al IEEE 488.2 y a la norma SCPI. A continuación se describen las funciones para utilizar dicho estándar. De manera sencilla se ve su uso y los comandos básicos con unos ejemplos prácticos. Para finalizar el capítulo se proponen unos ejercicios.

Capítulo 7. Adquisición de datos. 161

En este capítulo se trata uno de los principales usos de LabVIEW: la adquisición y generación de señales eléctricas a través de tarjetas de adquisición de datos. Hay varios tipos de sistemas de adquisición de datos: los Data Loggers, las tarjetas DAQ con interfaces PCI, PCI Express, PCI..., y externas como USB o RS-232. Se describen las capacidades comunes a todas ellas así como las funciones que emplea LabVIEW para manejarlas. Además se presentará el programa Measurement & Automation (MAX).
A continuación se muestran ejemplos prácticos para asentar los principales conceptos y se proponen ejercicios.

Capítulo 8. Protocolos de comunicación: TCP y UDP. 187

La importancia que tiene hoy día internet justifica que se dedique un tema completo a estudiar sus principales protocolos y la forma en que se pueden manejar desde LabVIEW.

Capítulo 9. Acceso remoto: VI Server
y Comunicaciones Avanzadas. 205

Este capítulo es una continuación del anterior, en él se explica cómo usar el servidor web que incluye LabVIEW, cómo manejar un programa residente en un ordenador desde otro y otros protocolos de nivel superior a los del capítulo 8. En la segunda parte del capítulo se hablará sobre VI Server y su utilidad.

Sección III. PROGRAMACIÓN AVANZADA

Capítulo 10. Sincronización y Multihilo. 225

Cuando se realiza una programación más compleja aparecen términos como pueden ser multitarea, procesos y sincronización. Se dice que los sistemas

operativos son multitarea porque pueden ejecutar más de un programa a la vez; este concepto puede extenderse también a una sola aplicación y hacer que la aplicación pueda tener varios 'hilos' ejecutándose en paralelo.

Este capítulo comienza explicando los conceptos teóricos y cómo se aplican en LabVIEW, a continuación se describen las técnicas de sincronización y los posibles problemas y soluciones que pueden aparecer cuando se trabaja con múltiples hilos.

Capítulo 11. Modelos de programación. 251

Continuando con conceptos avanzados, en este capítulo se verán técnicas y modelos para distintos tipos de programas: tratamiento de errores, sistemas de comunicaciones y principalmente máquinas de estados y programación orientada a objetos.

Capítulo 12. Código externo. 285

Este capítulo abre el código de LabVIEW a otros lenguajes de programación, en él se verá cómo hacer que el código creado con lenguajes como C, C++, C# o Visual Basic se puede usar en LabVIEW y viceversa. Además también se estudiará la forma de usar tecnologías como ActiveX o la reciente .NET en LabVIEW. Una vez más, las explicaciones se basan en ejemplos prácticos.

Capítulo 13. Optimización del interfaz. 313

Los dos últimos capítulos tratan de optimizaciones, en este caso sobre el interfaz de usuario, para ello se muestran algunos elementos nuevos o no habituales que pueden usarse en los interfaces. El capítulo finaliza con una serie de consejos para dar al programa un aspecto profesional.

Capítulo 14. Optimización del código. 337

Este capítulo comienza con una breve introducción a la ingeniería del software y a los sistemas de control de versiones para más tarde centrarse en cómo mejorar el rendimiento de los programas, comparando el rendimiento de ejemplos de programas 'buenos' y 'malos' (o mejor dicho, 'no tan buenos').

Capítulo 15. Otras plataformas. 357

En el último capítulo se verán sencillas aplicaciones del uso de LabVIEW, con algunos módulos extra, para programar sobre distintas plataformas: en ordenadores de mano (PDA), en dispositivos reconfigurables (FPGA) y en sistemas de alto rendimiento (PXI).

Sección I

INTRODUCCIÓN

Capítulo 1. Introducción a LabVIEW. Entorno.

Introducción a la herramienta de programación LabVIEW. Presentación del entorno de trabajo y los menús de herramientas, controles y funciones. Se indica cómo crear un programa y cómo se ejecuta éste. Se introducen los conceptos de VI, subVI y jerarquía. Cómo generar un proyecto y depurar el de código. Termina con algunos ejercicios propuestos.

Capítulo 2. Estructuras.

Las instrucciones de control permiten a un programa ejecutar un código de forma condicional o repetirlo cierto número de veces. En LabVIEW estas instrucciones son estructuras que encierran en su interior el código al que afectan. Cabe destacar las nuevas estructuras introducidas en las últimas versiones como EVENT y TIMED LOOP, así como las tradicionales FOR y WHILE. Se indican algunos ejemplos prácticos y se proponen algunos ejercicios.

Capítulo 3. Tipos de datos.

Estudio en profundidad de los tipos de datos disponibles en LabVIEW y de su manipulación. También se explican conceptos como variables, referencias y propiedades que se pueden utilizar. Se finaliza el capítulo con ejemplos prácticos y ejercicios propuestos.

Capítulo 1

Introducción a LabVIEW. Entorno

LabVIEW es el acrónimo de *Laboratory Virtual Instrument Engineering Workbech*. Es un lenguaje y a la vez un entorno de programación gráfica en el que se pueden crear aplicaciones de una forma rápida y sencilla.

National Instruments es la empresa desarrolladora y propietaria de LabVIEW, comenzó en 1976 en Austin, Texas y sus primeros productos eran dispositivos para el bus de instrumentación GPIB. En abril de 1983 comenzó el desarrollo de lo que sería su producto estrella: LabVIEW, que vería la luz en octubre de 1986 con el lanzamiento de LabVIEW 1.0 para Macintosh (los ordenadores más populares en aquella época que ya disponían de interfaz gráfica) y en 1990 la versión 2. Para Windows habría que esperar a septiembre de 1992. Los principales hitos de LabVIEW pueden verse en la Tabla 1.

Tabla 1 - Evolución de las diferentes versiones de LabVIEW

Fecha	Hito
Abril de 1983	Comienza el desarrollo de LabVIEW
Octubre de 1986	LabVIEW 1.0 para Macintosh
Enero de 1990	LabVIEW 2.0
Septiembre de 1992	LabVIEW para Windows
Octubre de 1992	LabVIEW para Sun
Octubre de 1993	LabVIEW 3.0 multiplataforma
Abril de 1994	LabVIEW para Windows NT
Octubre de 1994	LabVIEW para Power Macintosh
Octubre de 1995	LabVIEW para Windows 95
Mayo de 1997	LabVIEW 4.0
Marzo de 1998	LabVIEW 5.0
Febrero de 1999	LabVIEW 5.1, LV para Linux y LV Real-Time
Agosto de 2000	LabVIEW 6i
Enero de 2002	LabVIEW 6.1
Mayo de 2003	LabVIEW 7 Express, LabVIEW PDA y FPGA
Mayo de 2004	LabVIEW 7.1

Fecha	Hito
Mayo de 2005	LabVIEW DSP
Junio de 2005	LabVIEW Embedded
Octubre de 2005	LabVIEW 8
Agosto de 2006	LabVIEW 8.20 (Edición especial por el 20º aniversario)

1.1 Entorno

LabVIEW es una herramienta de programación gráfica. Originalmente este programa estaba orientado a aplicaciones de control de instrumentos electrónicos usadas en el desarrollo de sistemas de instrumentación, lo que se conoce como instrumentación virtual. Por este motivo los programas creados en LabVIEW se guardarán en ficheros llamados VI y con la misma extensión, que significa instrumento virtual (*Virtual Instruments*). También relacionado con este concepto se da nombre a sus dos ventanas principales: un instrumento real tendrá un Panel Frontal donde estarán sus botones, pantallas, etc. y una circuitería interna. En LabVIEW estas partes reciben el nombre de Panel Frontal y Diagrama de Bloques respectivamente.

- **Panel Frontal**, es la parte que verá el usuario, suele tener fondo gris.
- **Diagrama de Bloques**, es donde se realizará la programación y suele tener fondo blanco.

El Panel Frontal y el Diagrama de Bloques están conectados a través de los terminales (elementos que sirven como entradas o salidas de datos). De la misma forma que un indicador luminoso de la carátula de un instrumento está representado como un diodo en la circuitería interna, en un programa en LabVIEW ese mismo indicador luminoso estará representado en el Diagrama de Bloques como una salida de tipo booleano sobre el que escribir un valor.

En la figura 1-1 pueden verse las dos ventanas mencionadas antes. En primer término está el Panel Frontal, con fondo gris y dos terminales: uno numérico llamado *Meter* y otro que simboliza un interruptor llamado *Boolean*. En el Diagrama de Bloques también aparecen estos dos elementos bajo los mismos nombres y representados con dos iconos. Al programar en el Diagrama de Bloques se leerá y escribirá de estos iconos.

En la parte superior de estas ventanas se sitúa una barra con varias herramientas. En el Diagrama de Bloques esta barra tiene algunas opciones más, figura 1-2.

- El primer grupo de herramientas sirve para controlar la ejecución de un programa en LabVIEW. El primer botón indica si hay errores en el programa (fecha rota) y, cuando no los hay (flecha completa como en la imagen), ejecuta una vez el programa. El segundo botón ejecuta de forma continua el programa, como regla general este botón no debe usarse, en su lugar se empleará un bucle en el programa. El tercer botón aborta la ejecución y el cuarto permite realizar una pausa.

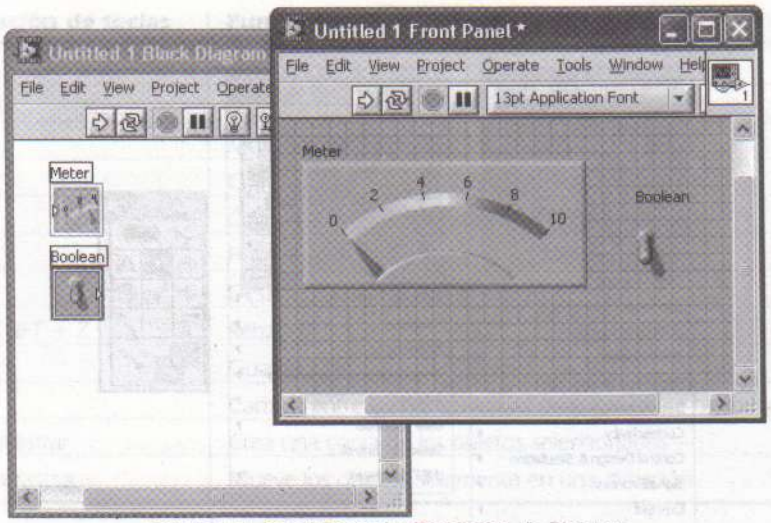


Figura 1-1. Panel Frontal y Diagrama de Bloques

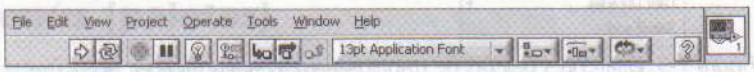


Figura 1-2. Barra de herramientas

- El segundo grupo de botones sirve para ayudar a su depuración. El primer botón es *Highlight Execution*, una de las herramientas más útiles para depurar, ralentiza la ejecución permitiendo ver el camino que siguen los datos en el programa. El siguiente, *Retain Wire Values* permite que al colocar un *probe* se obtenga el valor anterior. Los tres siguientes se utilizan para ejecutar el programa paso a paso.
- 13pt Application Font El menú desplegable permite formatear textos, es recomendable usar los formatos predefinidos como *Application Font* o *System Font*.
- El siguiente grupo se usa para alinear, distribuir, controlar el tamaño, agrupar y ordenar objetos.
- Abre la ayuda.

En el lateral derecho tanto del Panel Frontal como del Diagrama de Bloques aparece el icono que representa al VI, . Tiene un menú contextual que permite acceder a diversas características que serán tratadas más adelante.

Para colocar funciones en el Diagrama de Bloques y terminales en el Panel Frontal se tienen las paletas, o menús flotantes, llamadas paleta de funciones y de controles respectivamente. Además también se tiene la paleta de herramientas que consta de diferentes útiles. En la figura 1-3 puede verse el aspecto de estas tres paletas.

Las paletas de funciones y controles se despliegan haciendo clic con el botón secundario del ratón sobre una zona vacía del Diagrama de Bloques o del Panel Frontal. También pueden dejarse fijas presionando el clip ( ) de la esquina superior izquierda.

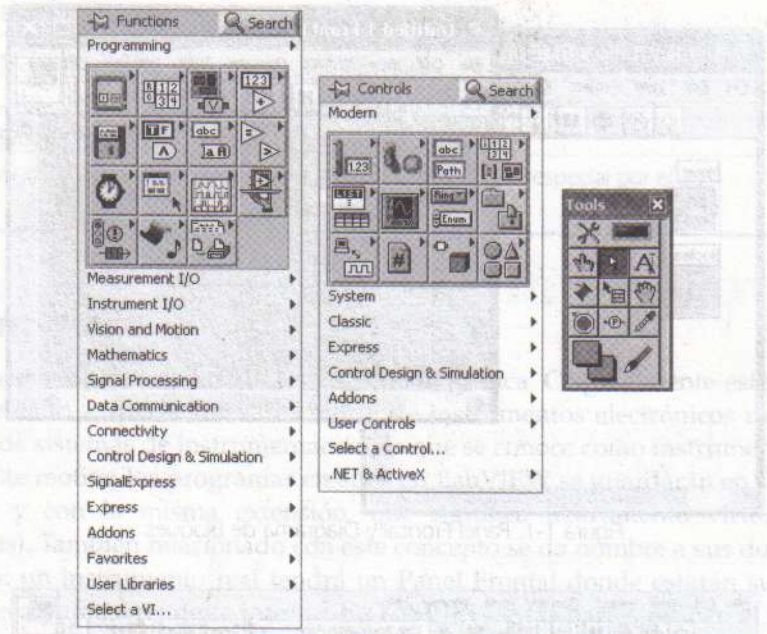


Figura 1-3. Diferentes paletas de trabajo: funciones, controles y herramientas

Otra ventana muy útil es la de ayuda contextual, puede abrirse en *Help > Show Context Help*. Esta ventana muestra información del objeto que esté situado bajo el cursor, ya sea una función, un VI, un control o un indicador. Además puede aparecer un enlace a la página de la ayuda relacionada con ese objeto.

Figura 1-4. Ventana de ayuda contextual

A continuación se muestra la Tabla 2 con los atajos de teclado más útiles de LabVIEW. Pueden personalizarse en *Tools > Options > Menu Shortcuts*.

Tabla 2 - Atajos de teclado más útiles

Combinación de teclas	Función
CTRL + R	Ejecuta el programa
CTRL + .	Aborta la ejecución
CTRL + E	Conmuta entre la dos ventanas principales

Combinación de teclas	Función
CTRL + B	Elimina los hilos rotos
CTRL + H	Muestra u oculta la ayuda contextual
CTRL + ?	Muestra la ayuda
CTRL + C	Copia los objetos seleccionados al portapapeles
CTRL + X	Corta los objetos seleccionados al portapapeles
CTRL + V	Pega los objetos desde el portapapeles
CTRL + Z	Deshace la última acción
CTRL + SHIFT + Z	Rehacer
CTRL + S	Guarda el VI
TAB	Cambia entre las herramientas de la paleta de herramientas
CTRL + arrastrar	Crea una copia de los objetos seleccionados
SHIFT + arrastrar	Mueve los objetos solamente en una dirección

1.2 Menús (paletas)

1.2.1 Menú de herramientas

Mediante la paleta de herramientas se puede seleccionar el instrumento que se desea usar, esto hace cambiar el tipo de cursor del ratón. En la figura 1-5 se puede ver la paleta de herramientas con 11 botones.

Figura 1-5. Paleta de herramientas

Automatic Tool Selection. Si está activado sirve para que se seleccione la herramienta de forma automática dependiendo del elemento sobre el que sitúe el cursor.

Operate Value. Sería el cursor disponible cuando el VI está ejecutándose, sirve principalmente para cambiar valores en los controles del Panel Frontal.

Position/Size/Select. Como su nombre indica sirve para posicionar objetos, tanto en el Diagrama de Bloques como en el Panel Frontal. También sirve para cambiar el tamaño y seleccionar elementos.

Edit Text. Cambia el cursor para poder escribir texto. Se puede usar, entre otras cosas, para escribir comentarios en el Diagrama de Bloques.

Connect wire. Es la herramienta que sirve para cablear. Uniendo unos elementos con otros por cables, los datos fluirán a través de ellos. Si los cables aparecen en color gris y de forma discontinua significa que el cable está roto (hay un error): el tipo de datos es incompatible o los terminales no son los adecuados.

Object Shortcut Menu. Despliega el menú contextual asociado a un elemento. Es equivalente a hacer clic con el botón secundario del ratón.

Scroll Window. Sirve para mover el contenido de las ventanas del mismo modo que si se usarán las barras de desplazamiento laterales.

Set/Clear Breakpoint. Crea o borra un punto de ruptura en un determinado elemento (función, VI, estructura o cable). Cuando la ejecución llega a ese punto se detiene.

Probe Data. Crea un *Probe* en un cable. Un *probe* es una ventana flotante que muestra el valor que circula por el cable.

Get Color. Obtiene el valor del color de un elemento.

Set Color. Colorea un elemento. Tiene dos posibles colores, el principal y el de fondo, ambos pueden ser asignados de forma independiente. Uno de los colores disponibles está marcado mediante una T, se trata del color transparente.

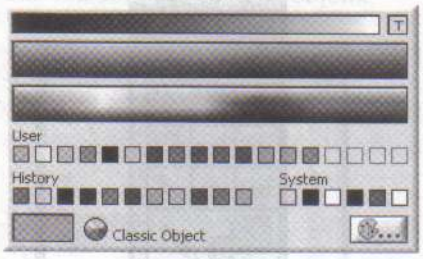


Figura 1-6. Colores

1.2.2 Controles

El menú que aparece en el Panel Frontal es el menú de controles, en él se pueden seleccionar los terminales que servirán para interactuar con el usuario.

Los terminales se dividen en controles e indicadores, aunque a veces se les suele llamar a todos genéricamente controles. Los controles propiamente dichos son las entradas de datos y los indicadores las salidas. Están clasificados según su estilo en varios submenús: *Modern, System* y *Classic.* Dentro de cada submenú hay otros menús que clasifican los controles por el tipo de datos. Los más usados son los que sirven para dibujar gráficas y los de la primera fila del submenú *Modern,* éstos de izquierda a derecha, corresponden a datos de tipo numérico, booleano, textos y tipos compuestos.

Para usar los controles hay que seleccionar el terminal deseado y llevarlo al lugar deseado del Panel Frontal.

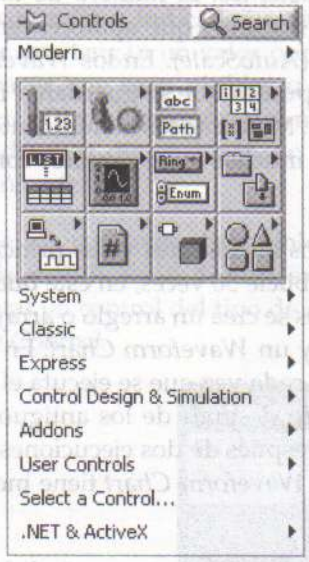


Figura 1-7. Paleta de controles

Figura 1-8. Paleta de gráficos

Los indicadores para gráficas merecen un trato aparte debido a su importancia. En la paleta de gráficas hay de varios tipos, siendo los más importantes *Waveform Chart* y *Waveform Graph*.

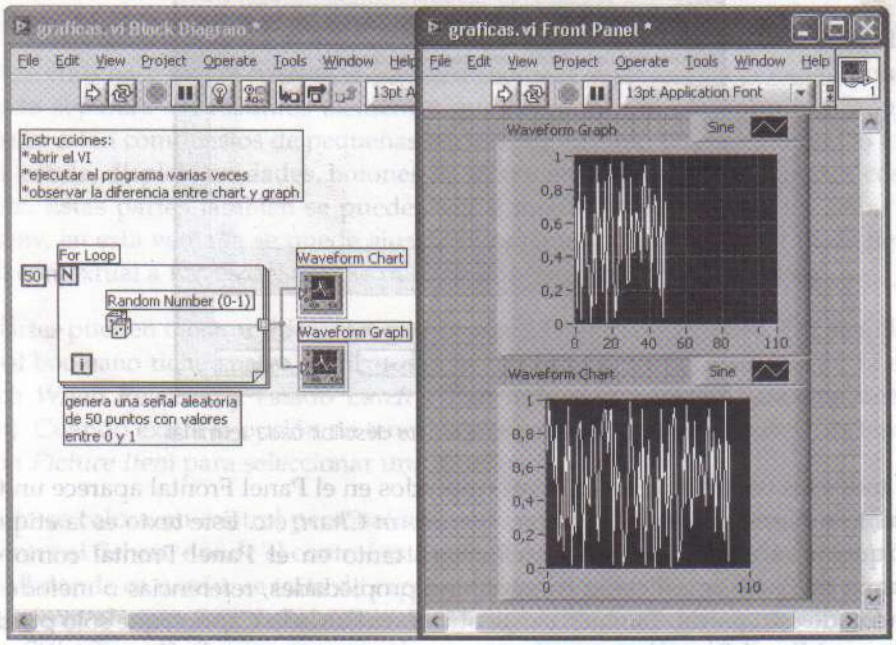


Figura 1-9. Comparación entre un *Waveform Graph* y un *Waveform Chart*

Los indicadores *Waveform Chart* se diferencian de los *Waveform Graph* en que éstos últimos dibujan totalmente la gráfica cuando llegan nuevos datos, sin embargo los *Waveform Chart* anexan el nuevo dato a los existentes. El número de datos que se muestran puede ajustarse escribiendo sobre los números que aparecen en los ejes o dejar que LabVIEW los ajuste automáticamente (*AutoScale*). En los *Waveform Chart* también se puede ajustar el número máximo de puntos que se guardan a través de la opción *Chart History Length* del menú contextual. Mediante los menús contextuales de los elementos de las gráficas se pueden modificar muchos otros aspectos como el color, interpolación, elementos a mostrar, etc.

En la imagen de la figura 1-9 pueden verse ambos indicadores y sus principales diferencias. El código de la parte izquierda repite un bucle 50 veces, en este bucle se genera un número aleatorio por cada iteración, después se crea un arreglo o *array* con los 50 números y se muestran en un *Waveform Graph* y un *Waveform Chart*. En el primero sólo se dibujan 50 puntos, y éstos se sobrescriben cada vez que se ejecuta el programa. En el segundo los datos nuevos se van colocando después de los antiguos (véase la numeración del eje X), el resultado mostrado es después de dos ejecuciones (100 datos generados). En conclusión se puede decir que el *Waveform Chart* tiene memoria y el *Waveform Graph* no.

Una práctica recomendable en todos los controles e indicadores es añadirles un texto que indique la función que realizan en la ventana que aparece al seleccionar en el menú contextual *Description and Tip*. Este texto se mostrará en la ventana de ayuda contextual cuando el usuario coloque el ratón sobre el terminal.

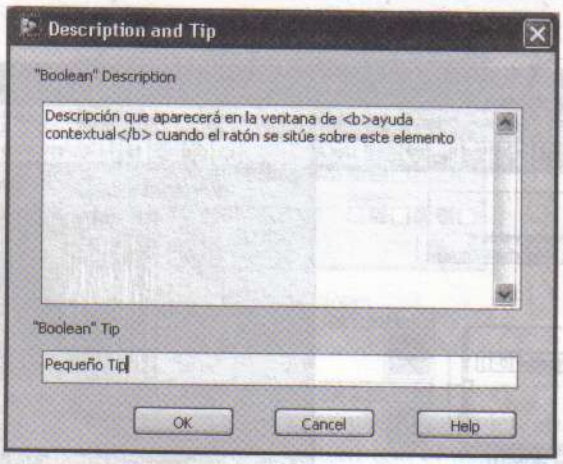


Figura 1-10. Ventana que permite describir cada terminal

Sobre todos los controles e indicadores colocados en el Panel Frontal aparece un texto, por ejemplo *Numeric, Boolean, Meter, Waveform Chart*, etc. Este texto es la etiqueta o *Label* que sirve para identificar al elemento tanto en el Panel Frontal como en el Diagrama de Bloques, así como en variables, propiedades, referencias o métodos que dependan de ese control. También existe otro texto llamado *Caption* que sólo puede ser visible en el Panel Frontal. Aunque no es necesario, es conveniente que cada control

tenga un *Label* diferente que describa su función, si es necesario en el Panel Frontal se puede usar *Caption* y *tip* para presentar una descripción más extensa.

Todos los controles e indicadores tienen elementos comunes en su menú contextual, como hacer que tengan un valor por defecto (*Data Operations > Make Current Value Default*), hacer que sean visibles o no (desde el Diagrama de Bloques *Hide/Show Control*), etc. Otros elementos del menú dependen del tipo de control, como en los *arrays* añadir más dimensiones, en los gráficos modificar colores, fuentes, tipos de interpolación, etc.

Además de los controles disponibles en este menú, se pueden crear otros nuevos mediante la modificación de uno existente, para ello primero hay que colocar en el Panel Frontal un control del tipo de datos deseado y después elegir en el menú *Edit > Customize Control.*

Figura 1-11. Opción para modificar o crear un control

En la parte superior aparece el botón *Edit Mode* y *Customize Mode* . El primero de ellos hace que el control se comporte como si estuviera en el Panel Frontal. El segundo separara los distintos elementos que componen el control o indicador. Los controles están compuestos de pequeñas partes, por ejemplo un *slide* numérico tendría *Label*, escala, display, unidades, botones de incremento y decremento, partes cosméticas, etc. Estas partes también se pueden ver y modificar en *Windows > Show Parts Window*, en esta ventana se puede ajustar su tamaño y posición. Cada parte tiene un menú contextual a través del cual se pueden importar/exportar imágenes.

Las partes pueden mostrar diferentes gráficos en diferentes momentos. Por ejemplo, un control booleano tiene cuatro gráficos diferentes: estado FALSE, estado TRUE, estado *Switch When Released* y estado *Latch When Released* (mientras está presionado el botón). Cuando existe la opción de tener varios gráficos, el menú contextual tendrá la opción *Picture Item* para seleccionar uno de ellos.

Cuando se coloca un control personalizado en un Panel Frontal no existe ningún vínculo entre el fichero donde el control está definido (*.ctl) y sus instancias empleadas en cada VI donde es usado, se trata de copias independientes. Si una instancia se modifica no afecta al resto. Las definiciones de tipos y definiciones de tipos estrictos (*Type Def.* y *Strict Type Def.*) se usan para enlazar todas las instancias de un control o indica-

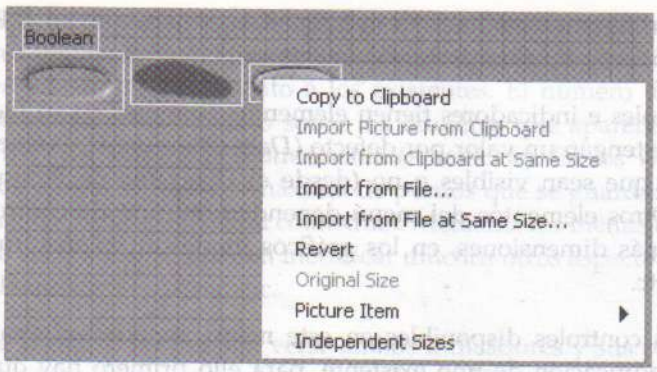


Figura 1-12. Manejo de los elementos que componen un control

dor a un mismo fichero *.ctl. De esta forma al modificar el fichero se modifican todas las instancias. Se puede elegir cómo se desea crear el control mediante una lista desplegable en la barra de herramientas junto al botón *Edit/Customize Mode*.

Las *Type Definitions* definen el tipo de datos de un control. Cuando el tipo de datos cambia, todos los controles asociados a él también cambian. En el menú contextual de una instancia se puede seleccionar *Auto-Update from Type Def.* para desactivar la actualización automática.

Un *Strict Type Definition* hace que todas las instancias sean iguales, no sólo el tipo de datos, también en características como el rango, tamaño, color, etc. De esta forma si se modifica la definición se actualizarían todas las instancias. Desde el menú contextual se puede eliminar el enlace entre la instancia y la definición.

1.2.3 Funciones

Esta paleta es la que se muestra al trabajar sobre el Diagrama de Bloques, en ella se puede acceder a las diferentes funciones, subVIs y estructuras disponibles.

Al igual que en el menú de controles, en este también hay varios submenús que se dividen dependiendo de la aplicación. Las funciones más usadas son las del submenú *Programming*.

El primero de los submenús de *Programming* es *Structures*. Contiene elementos que son equivalentes a las instrucciones de control de los lenguajes convencionales, es decir, son los bucles como WHILE o FOR y la estructura condicional CASE además de otras. Más adelante se dedica un tema completo a las estructuras.

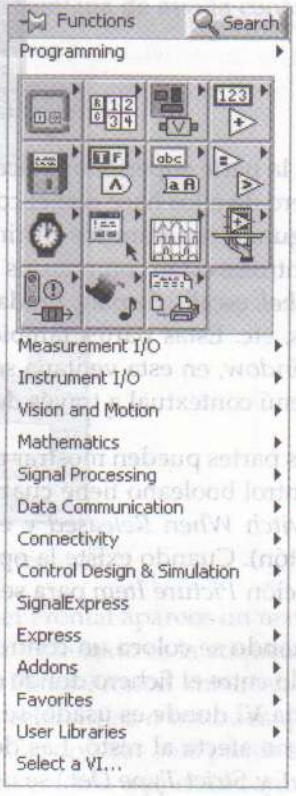


Figura 1-13. Paleta de funciones

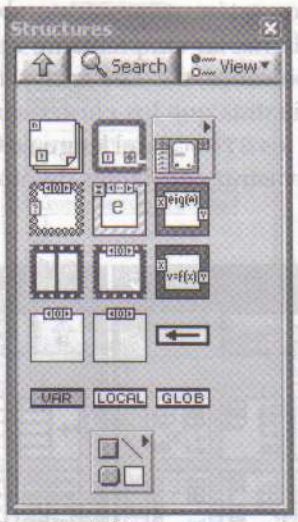


Figura 1-14. Paleta de estructuras

Los siguientes menús de *Programming* se dividen según el tipo de datos. Están los datos simples como los numéricos, booleanos y *strings* (texto), además de los compuestos como los *clusters* y *arrays*. Cada uno de estos menús tiene funciones para trabajar con esos datos.

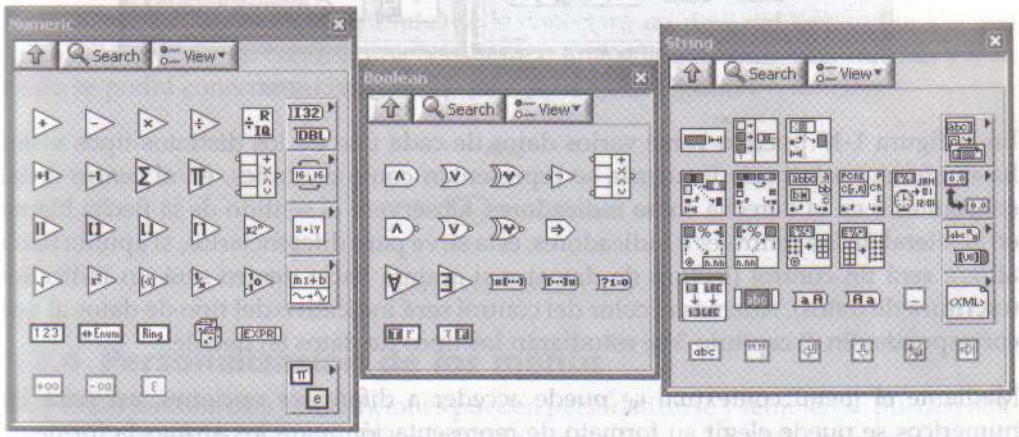


Figura 1-15. Paletas de tipos de datos numéricos, booleanos y texto

Los datos numéricos se dividen en enteros y de coma flotante y dentro de cada uno puede haber distintos tamaños, se puede cambiar de unos a otros mediante el *menú contextual > Representation*, si se aplican dos números, por ejemplo, un entero y otro flotante a una función, ésta cambiará automáticamente el tipo de los datos (*coercion*) para que así se pueda operar con ellos. Los booleanos únicamente pueden tener dos valores: Verdadero (TRUE) o Falso (FALSE), por esto son los apropiados para crear botones.

Los tipos de datos compuestos están, como su nombre indica, formados por otros datos, por lo tanto no se puede hablar simplemente de *arrays* sino que se debe decir *array* de números, *array* de booleanos, etc. Los *arrays*, también llamados arreglos, son listas ordenadas de valores mientras que los *cluster* son un conjunto desordenado de otros datos, son equivalentes a los STRUCT del lenguaje C. Para indicar los datos de que están compuestos basta con arrastrar constantes de otros tipos de datos en el interior de los *arrays* o *clusters*.

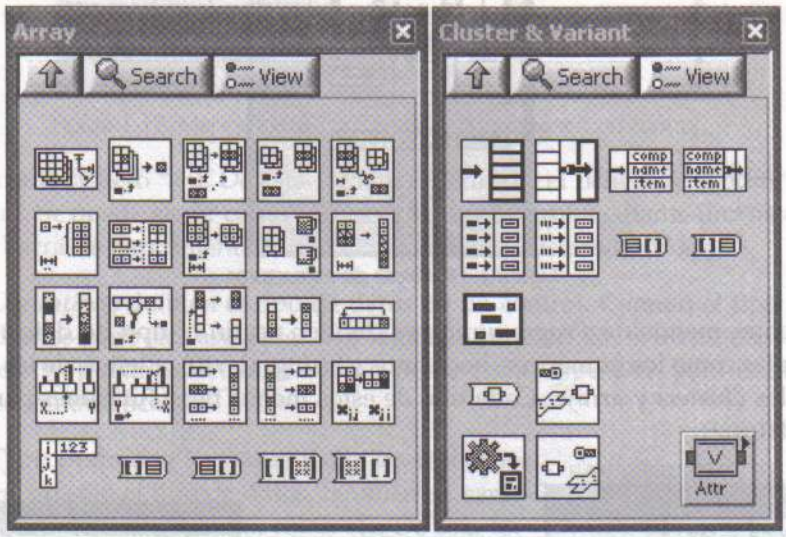


Figura 1-16. Paletas de datos tipo arrays y clusters

En la figura 1-17 pueden verse varios datos de cada uno de los distintos tipos vistos hasta ahora, en la parte izquierda se representan como controles, en el centro como constantes y en la derecha como indicadores. Obsérvese el sentido de la flecha blanca en el lateral de los controles e indicadores, ésta sirve para diferenciarlos, si apunta hacia afuera será un control (lectura de datos) y si apunta hacia dentro será un indicador (escritura de datos). Además el color del control será indicativo del tipo de datos al que corresponde. En el capítulo 3 se estudiarán los tipos de datos en detalle.

Mediante el menú contextual se puede acceder a diferentes opciones, así para los numéricos se puede elegir su formato de representación, para los *strings* la forma en que se visualizarán (normal, contraseña, etc.). A los *arrays* se les pueden añadir dimensiones, etc.

Al igual que con los controles, el método para usar las funciones y los VIs de la paleta de funciones es "arrastrar y colocar". La diferencia entre una función y un VI es que las funciones son objetos con una funcionalidad fija y sin Panel Frontal mientras que los VIs son programas hechos en LabVIEW por National Instruments u otros programadores que sí tienen Panel Frontal y se puede acceder a él haciendo doble clic sobre el VI. Los VIs disponibles están en librerías dentro del directorio *%directorio de instalación de LabVIEW%\vi.lib*.

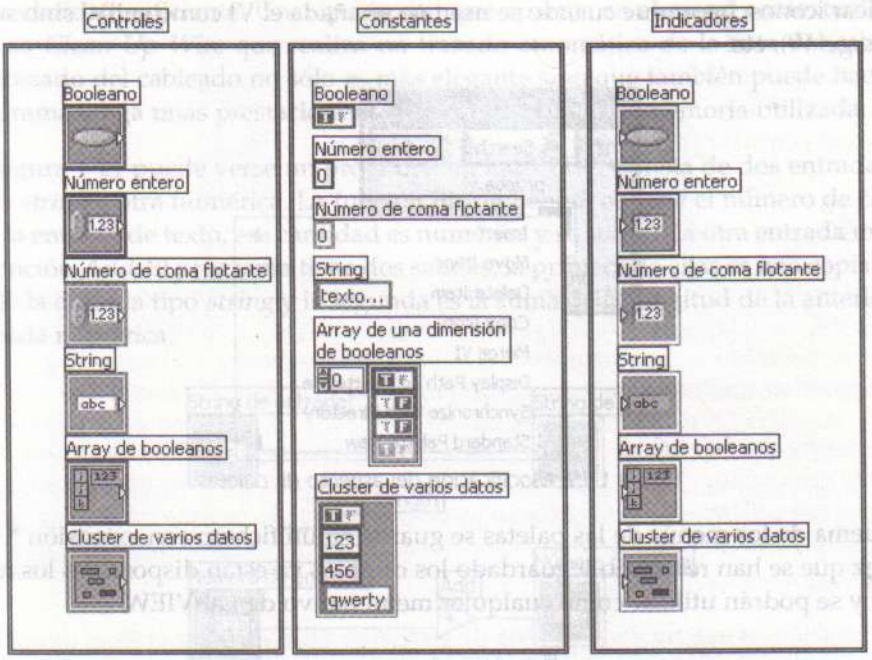


Figura 1-17. Diferentes tipos de datos

Las funciones y VIs generalmente operarán sobre una o varias entradas y proporcionarán una o más salidas. A las entradas se le conectará un dato del tipo requerido que puede provenir de un control, una constante o una salida de otro VI o función y a su salida se podrá conectar un indicador o una entrada de algún VI o función.

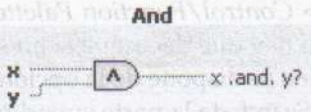


Figura 1-18. Función *And*

1.2.4 Personalización de los menús

Las paletas de controles y de funciones pueden personalizarse y también se pueden añadir nuevos menús basados en VIs creados por uno mismo o por otros programadores.

Cuando se crean menús nuevos, por convención, los VIs y controles se guardan en *%directorio de instalación de LabVIEW%\user.lib* y los nuevos menús deben crearse dentro del submenú *User Libraries* y *User Controls*.

Para modificar el aspecto de las paletas hay que ir a *Tools > Advanced > Edit Palette Set*, figura 1-19. Cuando se hace se abren las dos paletas principales y se puede navegar por ellas de forma normal, pero no se pueden arrastrar los VIs y controles al Diagrama de Bloques y al Panel Frontal sino que en cada VI y submenú aparece un menú desplegable con una serie de opciones que permiten añadir nuevos menús y VIs,

modificar iconos, hacer que cuando se usen no se añada el VI como subVI sino su código (*Merge VI*), etc.

Figura 1-19. Modificación del aspecto de paletas

El esquema de los menús de las paletas se guarda en un fichero con extensión *.MNU. Una vez que se han realizado y guardado los cambios ya están disponibles los nuevos menús y se podrán utilizar como cualquier menú nativo de LabVIEW.

Figura 1-20. Submenú de la librería de usuario

También en *Tools > Options > Control/Function Palettes* hay varios parámetros para cambiar su aspecto. Por último hay que decir que si no se encuentra un VI entre todos esos menús o submenús también se dispone de la opción de buscar un VI o control por su nombre mediante el botón *Search* de la parte superior de las paletas.

1.3 Creación de programas

En LabVIEW la programación se realiza en el Diagrama de Bloques. Un programa habitualmente está formado por:

1. Controles: sirven de entrada para los datos.
2. Funciones, VIs y estructuras: realizan una o varias operaciones con esos datos.
3. Indicadores: sirven de salida para los datos.

Los datos "circulan" por el programa mediante cables, que sirven para unir unos elementos con otros. Para realizar la conexión se utiliza la herramienta *Connect Wire* de la paleta de herramientas. Un cable tendrá una única fuente (control, constante o salida de otro elemento) y uno o varios destinos (indicador o entradas de otros elementos) de tipos compatibles. El cableado, en general, debe ser lo más corto posible mantenien-

dio una claridad suficiente. Una opción útil que aparece en el menú contextual de los cables es *Clean Up Wire* que realiza un trazado automático de la ruta del cable. Un buen trazado del cableado no sólo es más elegante sino que también puede hacer que el programa tenga unas prestaciones superiores en cuanto a memoria utilizada.

En la figura 1-21 puede verse un programa en LabVIEW. Consta de dos entradas, una de tipo *string* y otra numérica. La función *String Length* obtiene el número de caracteres de la entrada de texto, esa cantidad es numérica y se suma a la otra entrada mediante la función *Add*. El programa tiene dos salidas, la primera de ellas es una copia duplicada de la entrada tipo *string* y la segunda es la suma de la longitud de la anterior más la entrada numérica.

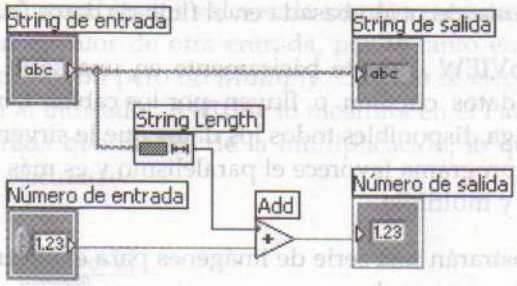


Figura 1-21. Programa sencillo en LabVIEW

Una vez creado el programa se deben introducir los datos iniciales en los controles del Panel Frontal, ejecutarlo presionando el botón *Run* (CTRL+R) y cuando acabe ver los resultados en los indicadores. La figura 1-22 muestra la ejecución del programa anterior.

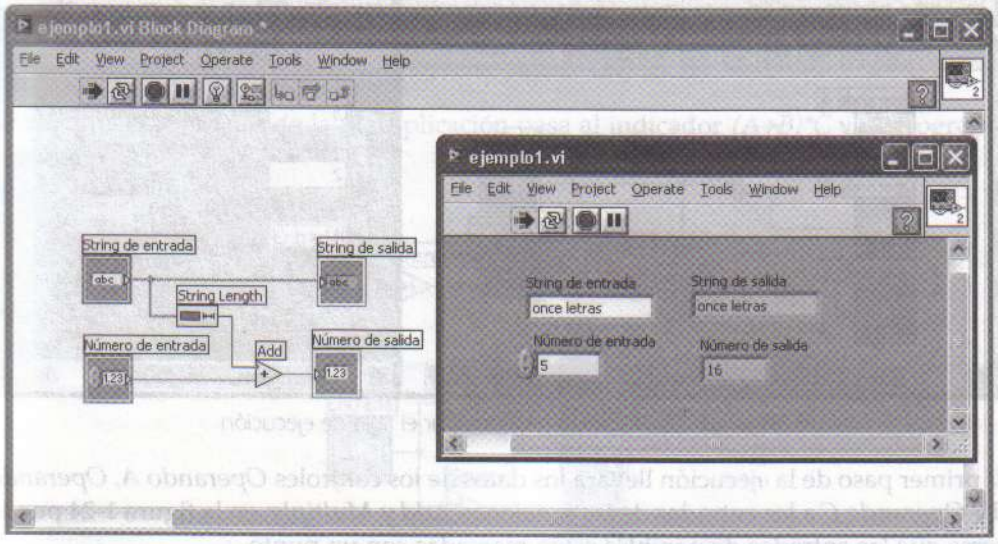


Figura 1-22. Ejecución del programa anterior

Para obtener un alto rendimiento LabVIEW es un lenguaje compilado. Cuando un programa ha sido modificado y se va a guardar o ejecutar generalmente se recompila. Al compilar el código del Diagrama de Bloques pasa a código máquina. El código compilado hará llamadas a otras funciones de librerías externas (*LabVIEW Run-Time Engine*) para tareas como dibujar gráficos o acceso a ficheros.

1.4 Flujo de ejecución

Al lenguaje de programación que usa LabVIEW también se le llama lenguaje G. La mayoría de los lenguajes se basan en una programación imperativa, la cual es simplemente una sucesión de operaciones. Sin embargo el lenguaje G no usa una programación imperativa sino una ejecución basada en el flujo de datos (*dataflow*).

Un programa en LabVIEW consiste básicamente en una serie de funciones unidas mediante cables. Los datos circulan o fluyen por los cables. Una función sólo podrá ejecutarse cuando tenga disponibles todos los datos que le sirven como entradas. Esta forma de ejecutar un programa favorece el paralelismo y es más apropiada para sistemas multiprocesador y multihilo.

A continuación se mostrarán una serie de imágenes para explicar el flujo de ejecución de LabVIEW mediante un ejemplo.

El programa consiste en dos operaciones matemáticas: la suma de dos números y la multiplicación del resultado de ésta por otro número. En la figura 1-23 se puede ver el programa y los datos de entrada en el Panel Frontal.

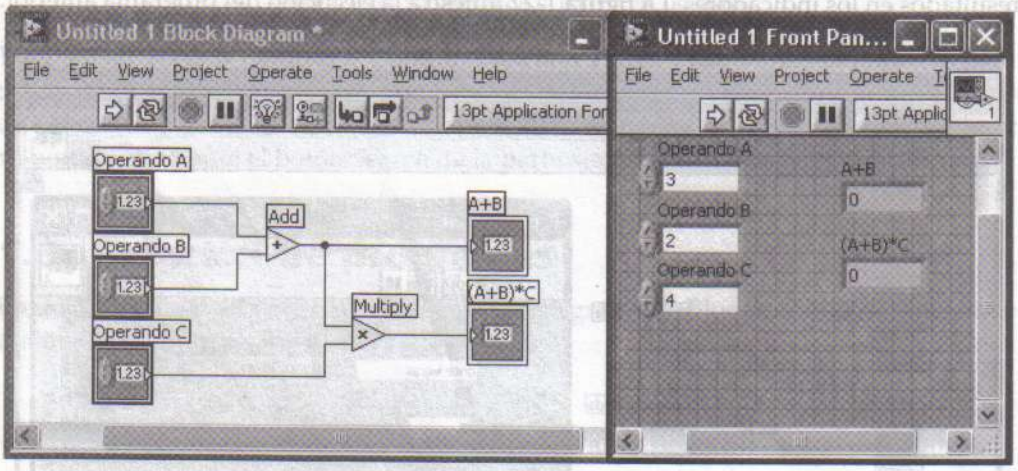


Figura 1-23. Programa para analizar el flujo de ejecución

El primer paso de la ejecución llevará los datos de los controles *Operando A, Operando B* y *Operando C* a las entradas de las funciones *Add* y *Multiply*, en la figura 1-24 puede verse que las entradas disponibles están marcadas con un punto.

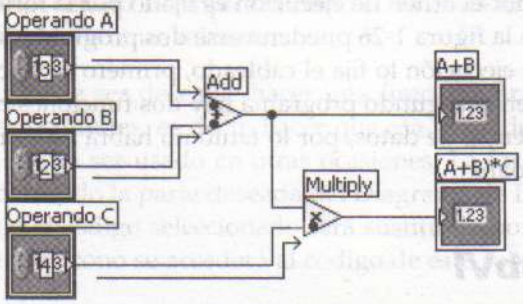


Figura 1-24. Valores disponibles en las entradas de la función Add

La función *Add* tiene disponibles los valores de sus dos entradas, sin embargo la función *Multiply* necesita el valor de otra entrada, por lo tanto en el siguiente paso se podrá ejecutar la función *Add* pero no *Multiply*. Cuando se ejecute el resultado de la suma, su valor pasará al indicador *A+B* que lo mostrará en el Panel Frontal y también circulará hasta la entrada que faltaba de la multiplicación, lo que permite que en el siguiente paso pueda ejecutarse.

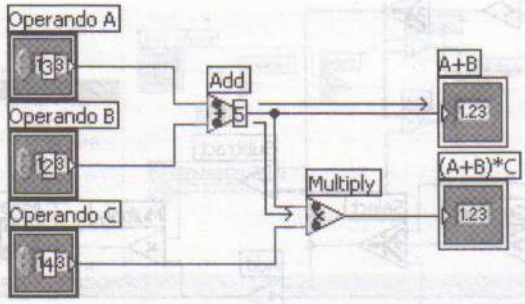


Figura 1-25. Siguiente paso de la ejecución

Finalmente el resultado de la multiplicación pasa al indicador *(A+B)*C* y el programa finaliza.

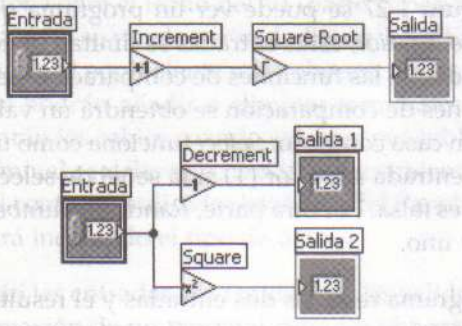


Figura 1-26. Flujo de ejecución

En el ejemplo anterior el orden de ejecución es fijado por la forma de cablear unas funciones con otras. En la figura 1-26 pueden verse dos programas, en el primero ocurre lo mismo: el orden de ejecución lo fija el cableado, primero se ejecutará *Increment* y después *Square Root*; en el segundo programa hay dos funciones pero entre ellas no existe ninguna dependencia de datos, por lo tanto no habrá ningún orden en la ejecución de *Decrement* y *Square*.

1.5 VI y subVI

Los ficheros con los programas creados en LabVIEW se llaman VIs (*Virtual Instrument*). En muchas ocasiones un programa será de un tamaño tal que habrá que separarlo en varios ficheros o habrá alguna sección de código que convenga reutilizarla varias veces. Un VI puede contener a otro de forma que el segundo sería un subVI del primero, el concepto es equivalente a las funciones de un lenguaje tradicional.

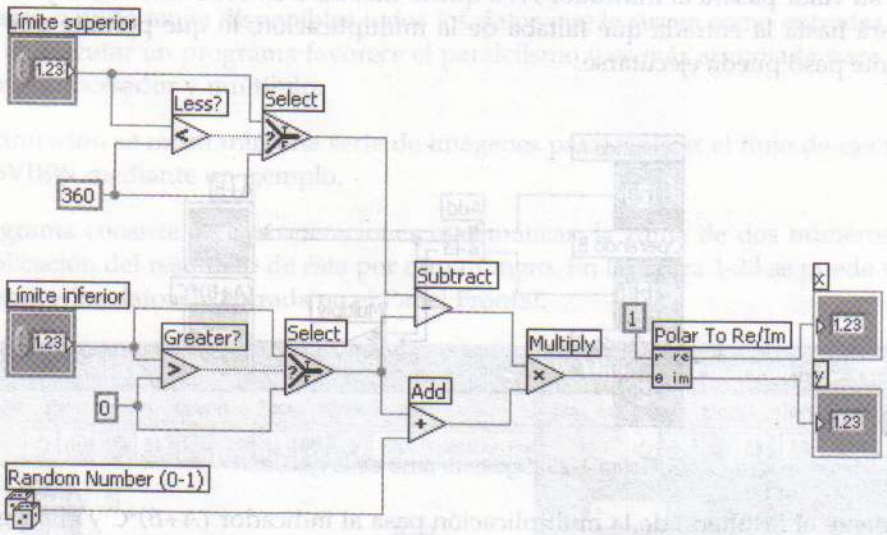


Figura 1-27. Programa que utiliza diferentes funciones

En el ejemplo de la figura 1-27 se puede ver un programa que tiene como entradas *Límite superior* y *Límite inferior*, estas entradas se limitan de forma programada a 360 y 0 respectivamente mediante las funciones de comparación *Less?*, *Greater?* y *Select*. A las salidas de las funciones de comparación se obtendrá un valor TRUE si la comparación es cierta y FALSE en caso contrario. *Select* funciona como un multiplexor: a su salida estará el valor de la entrada superior (T) si la señal de selección (?) es cierta y el de la entrada inferior (F) si es falsa. Por otra parte, *Random Number (0-1)* genera un número aleatorio entre cero y uno.

La parte central del programa resta las dos entradas y el resultado lo multiplica por la suma del límite inferior y el número aleatorio. Con esto se consigue generar un número aleatorio comprendido entre los dos límites indicados, que nunca serán mayores de 360 y menores de 0.

Finalmente este número generado se empleará como el valor de un ángulo en coordenadas polares, y de él se obtendrá un número.

En el ejemplo anterior puede ser deseable hacer una función para la generación del número aleatorio entre dos límites, es decir, hacer que esa parte del código sea un VI distinto, de esta forma podrá ser usado en otras ocasiones. La forma más sencilla de conseguir esto es seleccionando la parte deseada del Diagrama de Bloques e ir a *Edit > Create SubVI*. Al hacerlo el código seleccionado será sustituido por el icono de un VI, con un doble clic sobre este icono se accederá al código de este subVI.

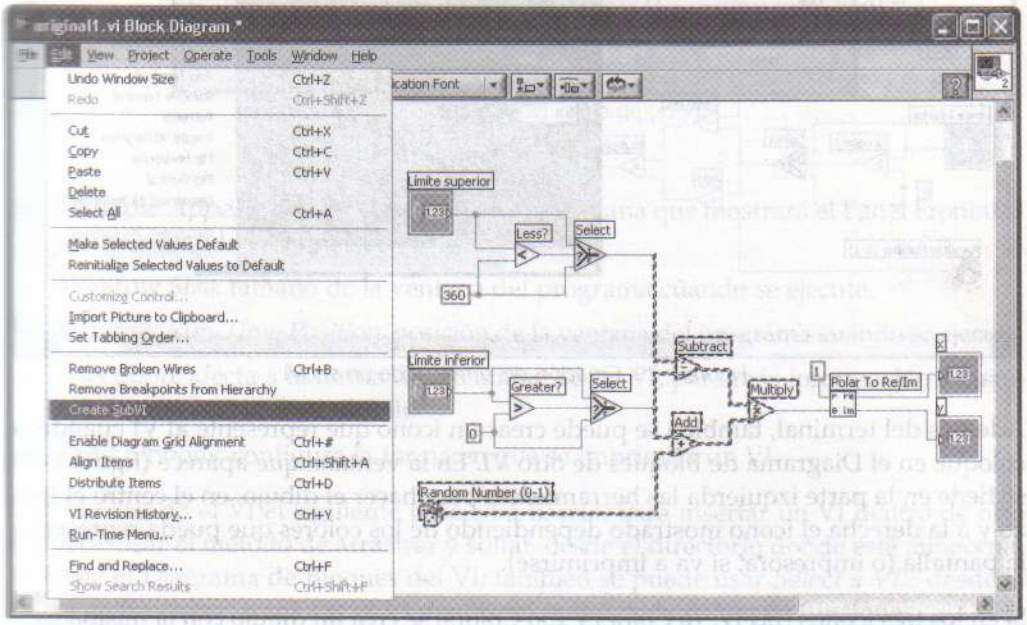


Figura 1-28. Crear un subVI seleccionando una parte del programa

Otra forma de crear un VI es definiendo de forma manual su interfaz, es decir, la forma en que se realizarán las conexiones cuando se use como subVI dentro de otro VI. El primer paso será guardar el VI, después situarse en su Panel Frontal y hacer clic con el botón secundario del ratón sobre el icono del VI (parte superior derecha) para desplegar su menú contextual, como se puede ver en la figura 1-29. En el menú contextual se mostrará el conector, que es el equivalente a la cabecera de las funciones en otros lenguajes. En el conector se podrán añadir o eliminar terminales, los terminales son los lugares donde se conectarán los cables cuando se use como subVI. Para asignar un control o indicador a un terminal se debe seleccionar la herramienta *Connect Wire* y hacer clic en el terminal y en el control o indicador asociado del Panel Frontal, en ese momento el terminal se coloreará indicando el tipo de datos.

Por claridad se conectarán las entradas a la izquierda y las salidas a la derecha. También se puede indicar si la conexión de un terminal debe ser obligatoria (*Required*), normal (*Recommended*) u opcional (*Optional*), en la ventana de ayuda contextual los terminales se mostrarán en negrita, de forma normal u ocultos respectivamente. Si en el Panel

Frontal se había indicado algún valor por defecto en algún control, éste será el valor que tenga el control si no se le conecta un cable cuando se use como subVI.

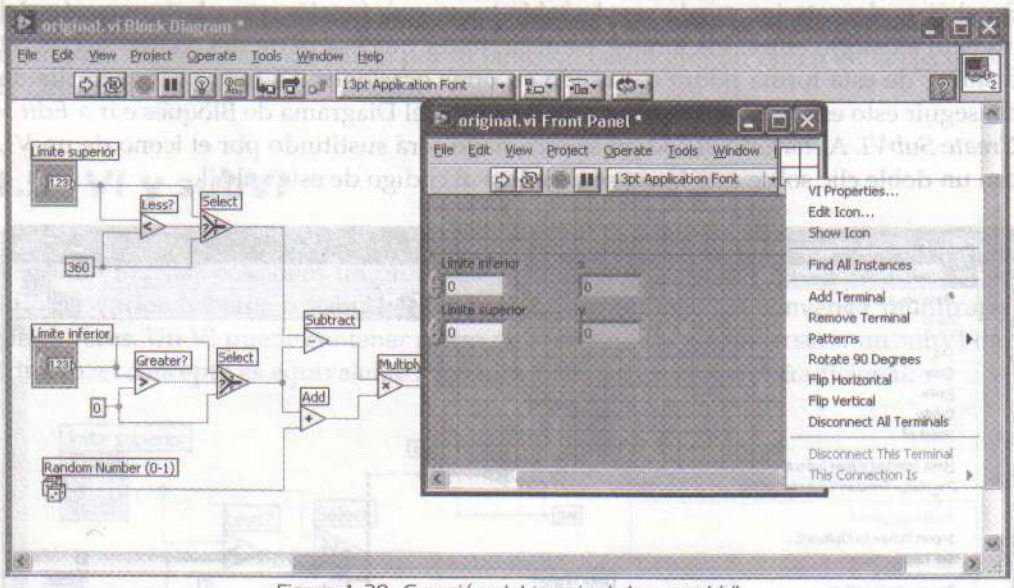


Figura 1-29. Creación del terminal de un subVI

Además del terminal, también se puede crear un icono que represente al VI cuando se coloque en el Diagrama de Bloques de otro VI. En la ventana que aparece (figura 1-30) se tiene en la parte izquierda las herramientas para hacer el dibujo, en el centro el lienzo y a la derecha el icono mostrado dependiendo de los colores que pueda representar la pantalla (o impresora, si va a imprimirse).

Si en los tres iconos (*B&W, 16 Colors* y *256 Colors*) se crea un dibujo con la misma figura, por ejemplo un triángulo, el VI aparecerá como un triángulo en vez de cuadrado. Si se está creando una librería o familia de subVIs es conveniente dar a todos un estilo parecido. Por último hay que decir que se puede acceder a opciones especiales haciendo doble clic sobre algunas herramientas, como la de texto.

Desde el menú contextual del icono o desde *File > VI Properties* se accederá a las propiedades del VI, en ellas se podrá ver información y modificar parámetros del VI como:

- *General*: información sobre la versión, cambios sin salvar, etc.
- *Memory Usage*: espacio que ocupan las distintas partes que componen el VI.
- *Documentation*: información sobre el VI, al usarlo como subVI aparecerá en la ventana de ayuda contextual y un enlace a un fichero de ayuda en el que se podría ampliar la información.
- *Revision History*: configuración e información sobre el historial del VI.
- *Editor Options*: algunos parámetros que afectan a la hora de crear el VI.
- *Security*: permite bloquear y/o proteger con contraseña el código del fichero.

Figura 1-30. Editor del icono de un VI

- *Window Appearance*: configuración de la ventana que mostrará el Panel Frontal al usuario cuando el VI se ejecute.
- *Window Size*: tamaño de la ventana del programa cuando se ejecute.
- *Window Run-Time Position*: posición de la ventana del programa cuando se ejecute.
- *Execution*: afecta a la forma de ejecutarse de un VI, para más información consultar el tema acerca de multihilo.
- *Print Options*: configura la forma en que se imprimirá un VI.

Una vez creado el VI el siguiente paso será usarlo. Para insertar un VI dentro de otro se puede usar el método de arrastrar y soltar, desde el directorio donde esté almacenado hasta el Diagrama de Bloques del VI; también se puede usar *Select a VI...* desde la paleta de funciones o colocar el VI dentro un menú de la paleta de funciones.

Como se puede ver en las propiedades de un VI (*File > VI properties > Memory Usage*) internamente se compone de un espacio de datos, el código compilado, el Panel Frontal y el Diagrama de Bloques. Al cargar un VI se llevará a memoria el espacio de datos y el código compilado; el Panel Frontal y el Diagrama de Bloques se cargarán cuando LabVIEW lo considere necesario.

Cada VI se guardará como un fichero. Para agrupar varios VIs se puede emplear una librería, la cual es otro fichero con extensión *.LLB. Hay varios métodos para crear una librería:

- En un fichero nuevo, *File > Save as > New LLB*.
- En *Tools > LLB Manager > File > New LLB*.

También se puede convertir un directorio que contenga VIs a una librería y viceversa desde *Tools > LLB Manager*, se selecciona el directorio o librería deseados y se elige *Edit > Convert*.

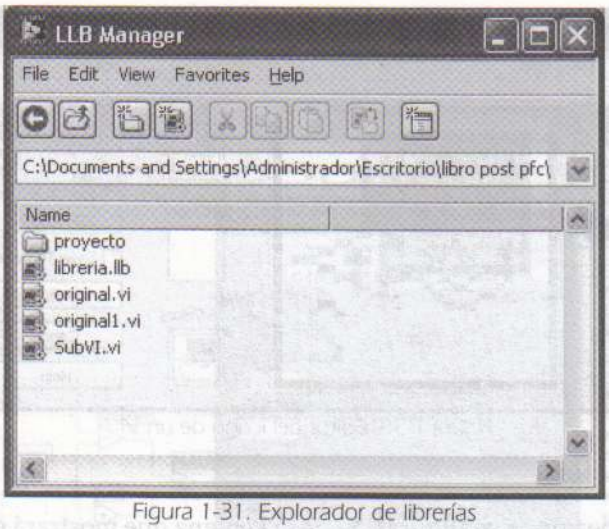


Figura 1-31. Explorador de librerías

Si está activada la casilla de *Tools > Options > Environment > Enable Windows Explorer for LLB Files* se podrá tratar el fichero LLB como si fuera un directorio más (en Windows 2000 o XP) y añadir o borrar ficheros. Tanto desde el *LLB Manager* como desde el explorador se podrá marcar un VI como *Top Level*, esto provoca que, cuando la librería se abra desde la línea de comandos o sin la opción de explorarlo, se abra directamente el VI principal. Una vez creada la librería se podrá trabajar con ella desde LabVIEW como si fuera un directorio más, pero para el sistema operativo será sólo un fichero.

Otra alternativa a la hora de guardar VIs es hacerlo en forma de plantillas (*templates*). Estos ficheros tienen por extensión *.VIT (o *.CTT para controles). Sirven para guardar en ellos códigos o componentes muy habituales. Son exactamente igual que los VIs excepto que en lugar de abrirse el fichero de la plantilla, LabVIEW abrirá una copia como un VI.

Al igual que en otros lenguajes, LabVIEW también admite el polimorfismo o sobrecarga de funciones, lo que significa que puede haber funciones distintas bajo el mismo nombre que, en principio, pueden tener distinto tipo de datos en las entradas; la función concreta a usar se puede adaptar a las entradas o seleccionarse de forma manual. Muchas de las funciones y VIs disponibles en la paleta de funciones son polimórficos.

Para crear un VI polimórfico se debe partir de los VIs que lo compondrán por separado; cada una de las instancias del VI polimórfico debe tener la misma estructura en el conector, aunque el tipo de datos obviamente puede ser diferente. En la figura 1-32 se muestra el código fuente de tres VIs con los que se creará un VI polimórfico.

Una vez creados los VIs con sus respectivos conectores se debe ir a *File > New...> Polymorphic VI*. En esta ventana se pueden unir todos los VIs en un único fichero, como se ve en la figura 1-33.

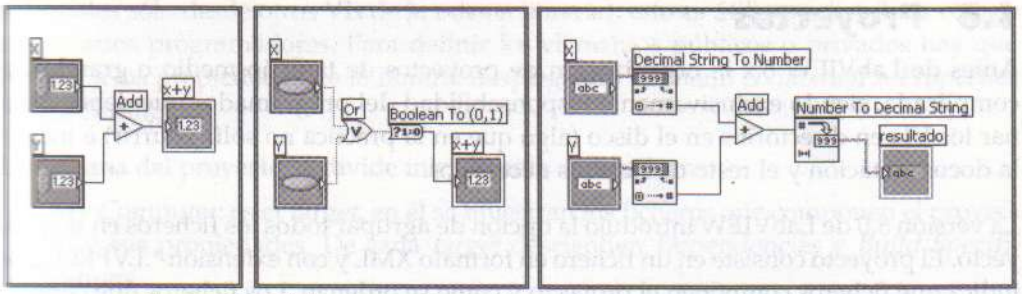


Figura 1-32. Código fuente de tres VIs con los que crear un VI polimórfico

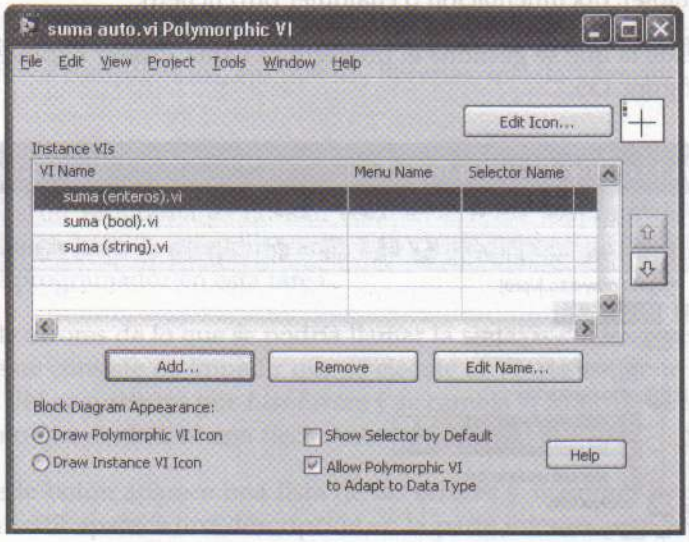


Figura 1-33. Creación de VI polimórfico

En la figura 1-34 está el resultado. En la parte izquierda se selecciona automáticamente la instancia concreta del VI a usar de acuerdo al tipo de datos que se conecta y en la derecha se muestra un VI que en su parte inferior tiene un selector para elegir de forma manual la instancia.

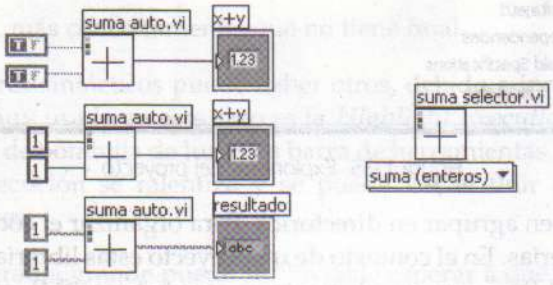


Figura 1-34. Utilización del VI polimórfico

1.6 Proyectos

Antes de LabVIEW 8.x la organización de proyectos de tamaño medio o grande era complicada, siendo exclusivamente responsabilidad del programador que debía ordenar los VIs en directorios en el disco (algo que en la práctica no solía ocurrir) e incluir la documentación y el resto de ficheros necesarios.

La versión 8.0 de LabVIEW introdujo la opción de agrupar todos los ficheros en un proyecto. El proyecto consiste en un fichero en formato XML y con extensión *.LVPROJ que indica qué ficheros componen el proyecto y cómo se ordenan. Los ficheros que componen el proyecto pueden ser VIs, controles, ficheros de configuración (por ejemplo para crear un ejecutable), documentación o cualquier otro fichero.

La ventana desde la que se gestiona el proyecto se llama *Project Explorer* y muestra el aspecto de la figura 135.

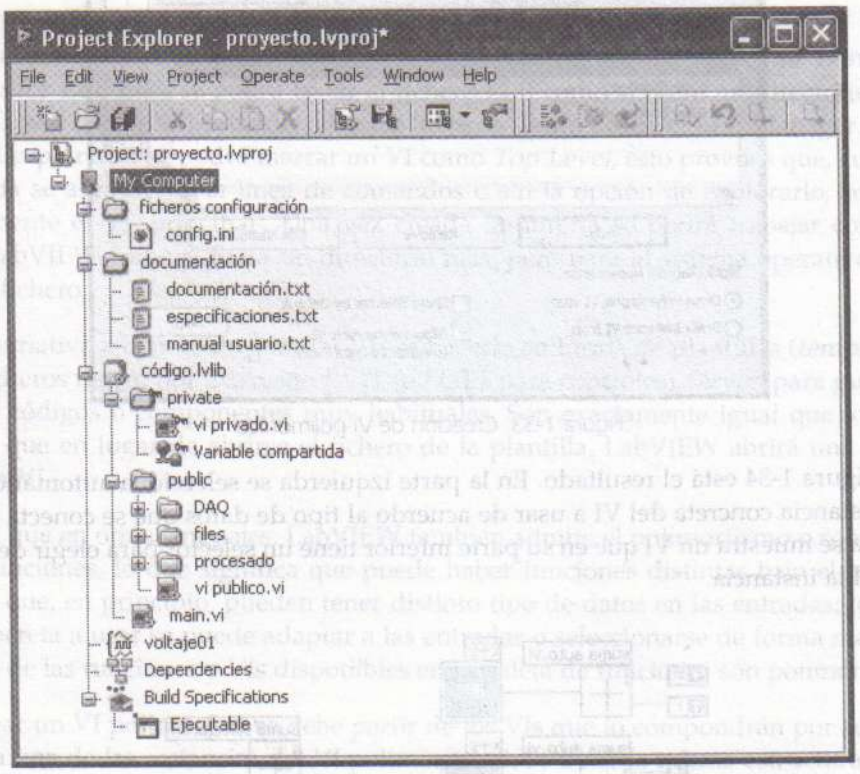


Figura 1-35. Explorador del proyecto

Los ficheros se pueden agrupar en directorios. Para organizar el código también puede ser útil emplear librerías. En el contexto de un proyecto estas librerías no deben confundirse con ficheros LLB ya que simplemente es otro fichero XML con referencias a los ficheros contenidos y la descripción de sus propiedades. Dentro de las librerías los elementos pueden ser públicos (accesibles desde otras partes del proyecto) o privados

(accesibles sólo desde otros VIs de la misma librería), esto es útil para distribuir código entre varios programadores. Para definir los elementos públicos o privados hay que acceder a las propiedades de la librería desplegando su *menú contextual* > *Properties* > *Item Settings*.

La ventana del proyecto se divide inicialmente en tres ítems:

- *My Computer*: es el *target*, en él se muestran los ficheros que componen el proyecto y sus propiedades. De cada *target* descienden *Dependencies* y *Build Specifications*.
- *Dependencies*: muestra las dependencias del proyecto, por ejemplo librerías compartidas.
- *Build Specifications*: guarda la configuración para las distribuciones del proyecto como archivos comprimidos, ejecutables, librerías, instaladores, etc.

1.7 Depuración de código

Cuando se hace un programa es normal cometer errores. Depurar es el proceso de encontrar y corregir los fallos del programa. LabVIEW dispone de varias herramientas para ayudar al programador en esta tarea.

La detección de errores de lo que se podría llamar la sintaxis del lenguaje es automática y simplemente consiste en impedir que se ejecute el VI que contiene un error, para ello el botón RUN se sustituye por *List Errors* y se cambia el icono de la barra de herramientas por uno que simboliza una flecha rota .

Al presionar este botón aparece una lista de los errores y *warnings* del programa. Al seleccionar un error aparece una descripción en la parte inferior, y haciendo doble clic o presionando el botón *Show Error* se traslada el foco del programa a la parte donde se ha producido el fallo.

En la figura 1-36 se indica que hay tres errores en el código:

- Una estructura CASE cuyo selector no tiene nada cableado.
- Una función de suma que presenta errores en sus terminales.
- Un cable roto , más concretamente, que no tiene final.

A parte de los errores sintácticos puede haber otros, debido a innumerables motivos. Una herramienta muy usada en este caso es la *Highlight Execution*, representada por un icono con forma de bombilla de luz en la barra de herramientas. Cuando esta opción está activada la ejecución se ralentiza y se puede ver el fluir de los datos por el Diagrama de Bloques.

A veces en un programa grande puede ser inviable esperar a que el flujo de ejecución llegue a la parte deseada con la *Highlight Execution*, para ello se pueden usar los *breakpoints* o puntos de ruptura. Éstos se seleccionan en la paleta de herramientas y después presionando sobre el objeto en el que se desea establecer el punto de ruptura. El objeto

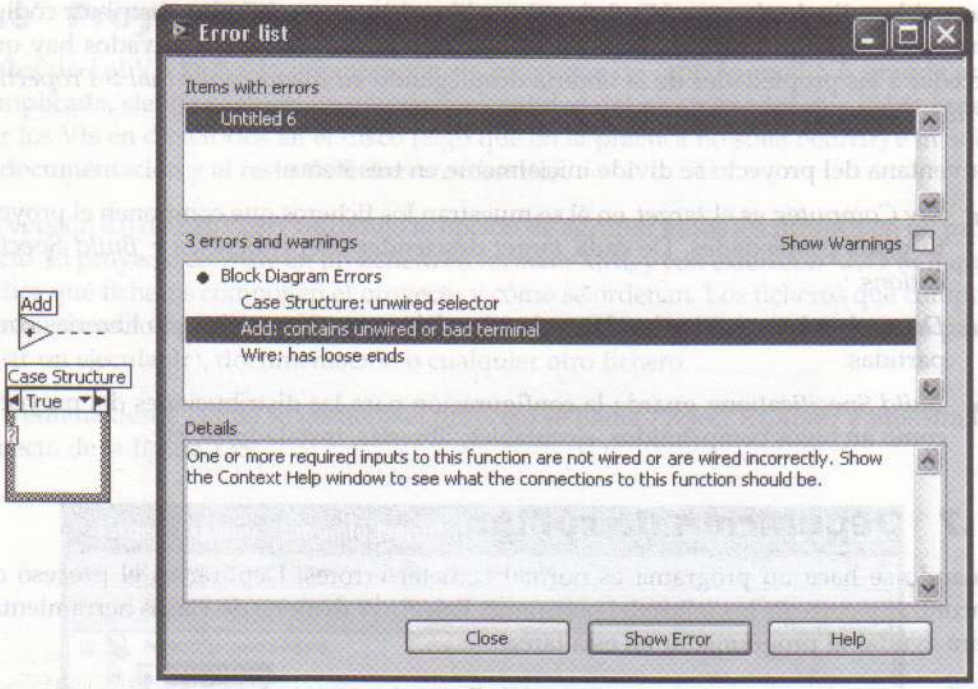


Figura 1-36. Lista de errores de un VI

puede ser un cable o una estructura o una función, ver figura 1-37. Al crear un *breakpoint* aparece una marca roja indicando su situación. En la siguiente imagen se pueden apreciar tres puntos de ruptura: en el interior de la estructura WHILE, en el cable entre el control de *x* y la función suma y en la propia función.

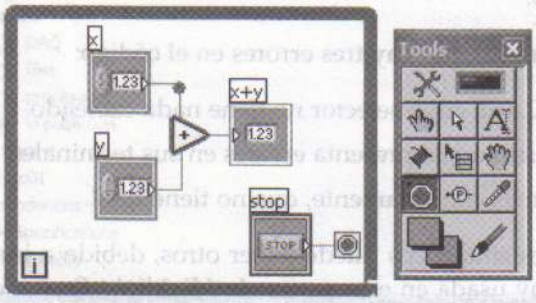


Figura 1-37. Utilización de puntos de ruptura para depuración de código

Una vez seleccionados los puntos de ruptura se puede ejecutar el programa de forma normal y al llegar la ejecución a uno de ellos ésta se detendrá automáticamente, igual que si se hubiera pulsado el botón *Pause* de la barra de herramientas (el cual pasa a tener fondo rojo y se transforma en *Continue*).

Si lo que se desea es obtener el valor de un dato cualquiera se puede usar la herramienta *Probe*, ver figura 1-38. Se puede activar sobre un control, indicador o cable usando la paleta de herramientas o el menú contextual del cable; cuando se coloque aparecerá una ventana flotante en la que puede verse el valor del dato seleccionado en tiempo real.

Otra utilidad de los *Probe* es crear un *Custom Probe* con una condición, en ese caso cuando se cumpla la condición el VI se detendrá igual que si hubiera un *breakpoint*. A esta opción se puede acceder desde el menú contextual de los cables *Custom Probe > Conditional [Tipo] Probe*. Por último también hay que decir que un probe puede personalizarse como un VI con *Custom Probe > New*.

Otro método más para detectar errores es revisar el valor de ciertos datos, como los clusters de error que tienen muchas funciones a la entrada y salida; para esto hay VIs especializados en la paleta Programming > Dialog & User Interface.

- Los *cluster* de error son un dato especial. Se componen de tres parámetros:
- Un booleano que indica si hay error o no.
- Un número identificativo del error.
- Un *string* que contiene una descripción del error.

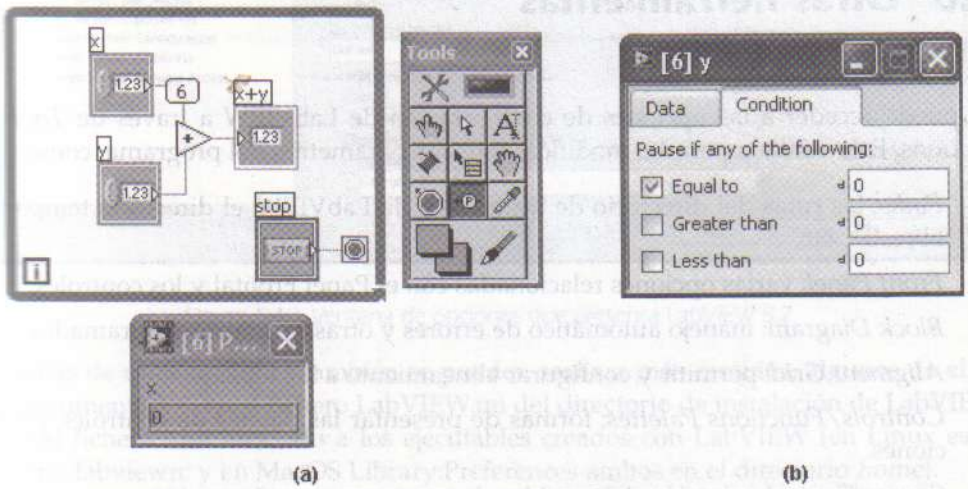


Figura 1-38. (a) Utilización del Probe. (b) Probe condicional

Existe una herramienta que amplía la información sobre el error, se puede acceder directamente a ella desde el indicador del *cluster* presionando con el botón derecho sobre él y eligiendo *Explain Error*. También se puede ir a *Help > Explain Error*.

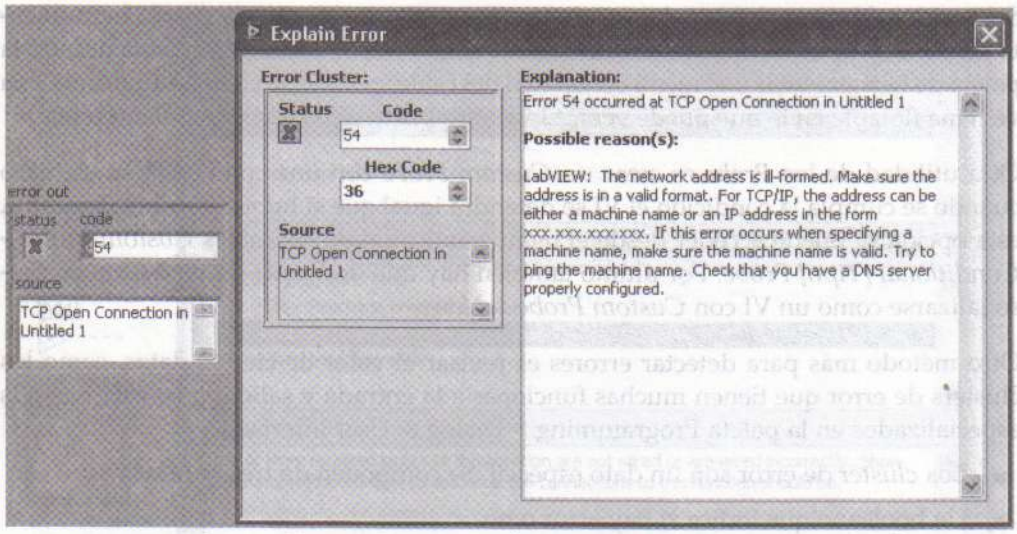


Figura 1-39. Información del error detectado

1.8 Otras herramientas

1.8.1 Opciones

Se puede acceder a las opciones de configuración de LabVIEW a través de *Tools > Options*. Esta ventana permite modificar muchos parámetros del programa, como:

- *Paths*: las rutas del directorio de instalación de LabVIEW, el directorio temporal, librerías, etc.
- *Front Panel*: varias opciones relacionadas con el Panel Frontal y los controles.
- *Block Diagram*: manejo automático de errores y otras ayudas al programador.
- *Aligment Grid*: permitir y configurar alineamiento a las rejillas.
- *Controls/Functions Palettes*: formas de presentar las paletas de controles y funciones.
- *Source Control*: selección del servidor de versiones y configuración.
- *Debugging*: opciones para depurar errores.
- *Colors*: cambiar el color de varios elementos de LabVIEW.
- *Fonts*: define las fuentes de aplicación, diálogo y sistema.
- *Printing*: opciones a la hora de imprimir.
- *Revision History*: permite guardar información cada vez que se guarda un VI.
- *Menu Shortcuts*: modifica y crea atajos de teclado para las opciones de los menús.
- *Environment*: opciones de configuración misceláneas.
- *Security*: permite restringir el acceso a LabVIEW

- *Shared Variable Engine*: ordenadores que ofrecen servicios para manejar variables compartidas.
- *VI Server*: configuración del acceso a VI Server.
- *Web* Server: configuración del servidor web que incorpora LabVIEW.

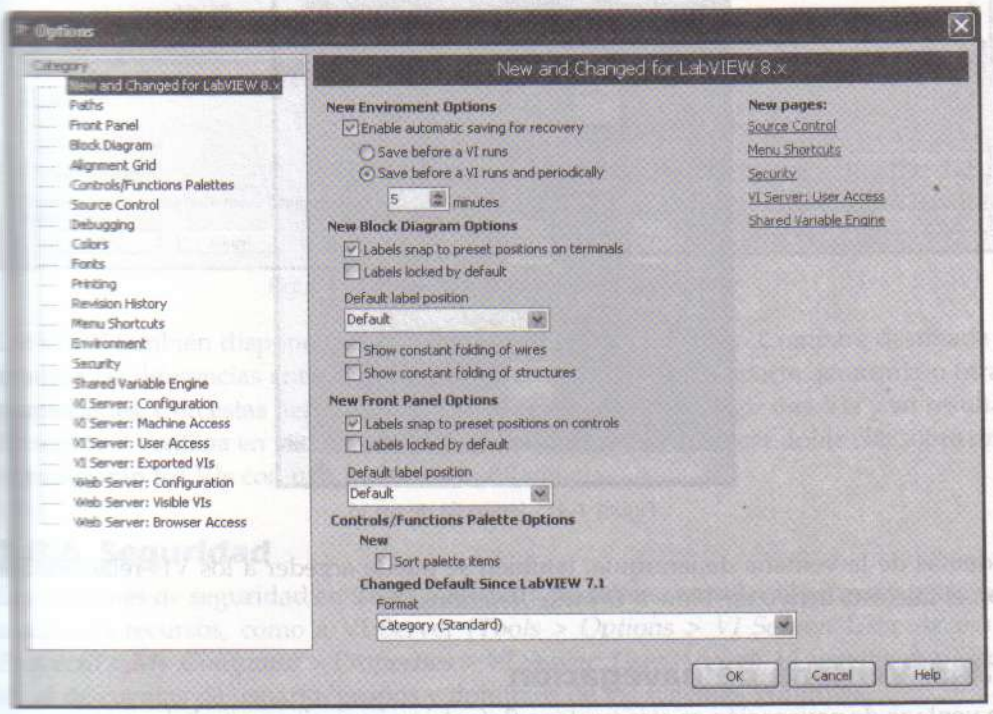


Figura 1-40. Ventana de opciones que presenta LabVIEW 8.2

Además de estas opciones también se pueden realizar más cambios, algunos de ellos no documentados, en el fichero LabVIEW.ini del directorio de instalación de LabVIEW o en el fichero *.ini asociado a los ejecutables creados con LabVIEW (en Linux es el fichero .labviewrc y en MacOS Library:Preferences ambos en el directorio *home*).

1.8.2 Jerarquía

Cuando se usan unos VIs dentro de otros se va creando una jerarquía de VIs. En *View > VI Hierarchy* se puede abrir una herramienta para ver la estructura jerárquica de un programa, ver el ejemplo de la figura 141.

Además de ver la estructura de un proyecto de forma gráfica también puede ser de utilidad en grandes proyectos para buscar VIs.

Una buena jerarquía es la que está organizada por niveles bien definidos, los VIs de un nivel se apoyan en los del nivel inmediatamente inferior y no hay saltos de niveles .

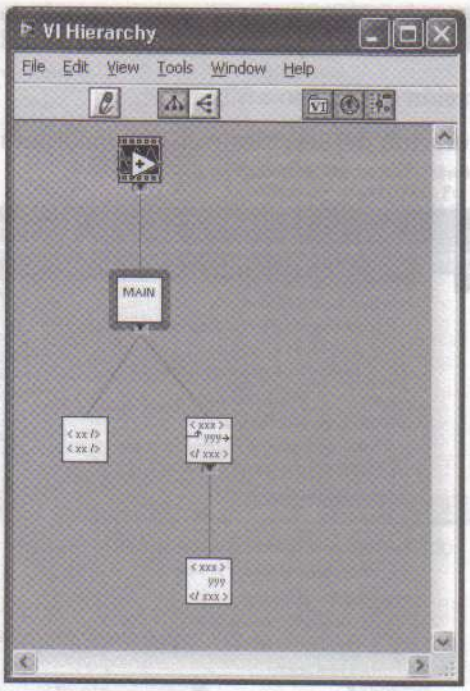


Figura 1-41. Jerarquía de un VI

Además de la ventana de jerarquía, también se puede acceder a los VIs relacionados con el que está activo en *View > Browse Relationship*.

1.8.3 Ventana de navegación

La ventana de navegación es otra ventana flotante que permite ver toda la ventana activa de LabVIEW mediante un zoom automático. Se puede acceder a ella desde *View > Navigation Window* y puede ser útil cuando el Diagrama de Bloques tiene un tamaño superior a la pantalla (aunque esto es algo que se debe evitar haciendo el programa más modular).

1.8.4 Compilación masiva

La compilación masiva es un método que se usa sobre todo al actualizar la versión de LabVIEW. Su función básicamente es abrir todos los VIs de un directorio, relinkar y recompilar todos los VIs y subVIs que contiene. Se puede acceder a esta herramienta en *Tools > Advanced > Mass Compile*.

1.8.5 Búsqueda y comparación

En el menú *Edit > Find and Replace* se halla una herramienta que permite buscar tanto texto como objetos en los VIs cargados en memoria. Los objetos a buscar pueden ser VIs, funciones, *breakpoints*, terminales, etc. En la ventana de resultados se pueden sustituir todas las instancias del objeto buscado por otra función.

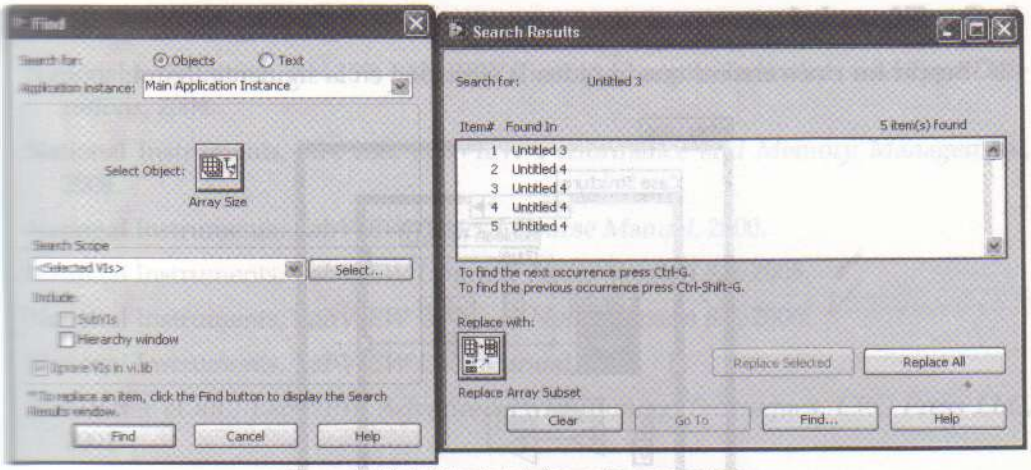


Figura 1-42. Utilidad de búsqueda y reemplazo

LabVIEW también dispone de otra utilidad en el menú *Tools > Compare* destinada a mostrar las diferencias entre dos VIs, tanto en su código, su aspecto, jerarquía u otras propiedades. Con estas herramientas simplemente hay que elegir dos VIs y su resultado será una ventana en la que se muestran las diferencias, al hacer doble clic sobre una se mostrarán los VIs con una marca en la diferencia.

1.8.6 Seguridad

Las opciones de seguridad en *Tools > Security* permiten controlar el acceso y permisos a algunos recursos, como a VI Server (*Tools > Options > VI Server: User Access* o *Proyecto > My Computer > Properties > VI Server: User Access*). El sistema es parecido al de cuentas de usuario, grupos y dominios de los sistemas Windows.

Para que un usuario se registre como tal debe dirigirse a *Tools > Security > Login*, además se puede activar la opción *Tools > Options > Security > Show the login prompt at LabVIEW startup time* para hacer que se pida un usuario cada vez que se inicie LabVIEW.

Algunos módulos de LabVIEW pueden extender las funciones de seguridad a elementos como variables compartidas o librerías de proyectos.

1.8.7 Ejemplos

Una de las cosas más apreciadas de *National Instruments* es su soporte, tanto en línea como en sus programas. Dentro de la ayuda de LabVIEW ya se ha visto antes la ventana de ayuda contextual, también dispone de muchos documentos explicando detalladamente cada aspecto del programa. Ahora se hablará sobre su colección de ejemplos, a la que se puede acceder desde *Help > Find Examples*.

Se puede modificar la ventana que muestra los ejemplos en *Tools > Prepare Example VIs for NI Example Finder*.

1.9 Ejercicios

1. Buscar los cinco errores que impiden la ejecución en la siguiente figura:

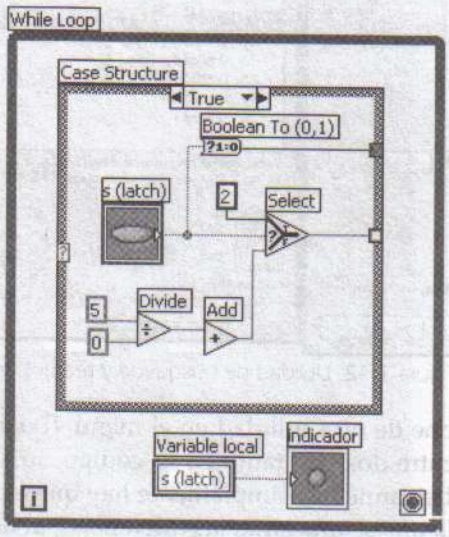


Figura 1-43. Ejercicio 1

2. Obtener mediante las funciones del menú *Programming > Numeric* el número áureo y su inverso: $\Phi = \frac{\sqrt{5}+1}{2}$, $1/\Phi = \frac{\sqrt{5}-1}{2}$. Ejecutar el programa con *Highlight Execution* activado.
3. Usar el programa del ejercicio anterior como un subVI de uno nuevo para verificar si $\Phi - 1 = \frac{1}{\Phi}$ y $\Phi^3 = \frac{\Phi+1}{\Phi-1}$.
4. Depura el siguiente programa para encontrar y solucionar el motivo por el que no funciona correctamente.

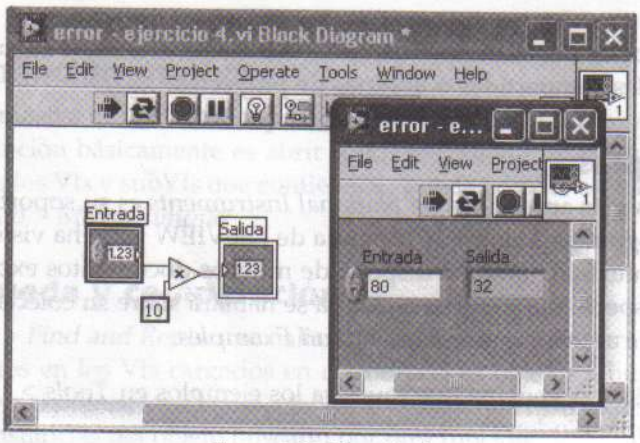


Figura 1-44. Ejercicio 4

1.10 Bibliografía

National Instruments, AN 159: *LabVIEW Custom Controls, Indicators, and Type Definitions*, 2004.

National Instruments, AN 168: *LabVIEW Performance and Memory Management*, 2004.

National Instruments, *LabVIEW Basics I Course Manual*, 2000.

National Instruments, *LabVIEW Upgrade Notes Version 8.0*, 2005.

National Instruments, *LabVIEW Upgrade Notes Version 8.2*, 2006.

National Instruments, *LabVIEW User Manual*, 2001.

Rick Bitter et alt., *LabVIEW Advanced Programming Techniques, CRC Press LLC*, 2001

Capítulo 2

Estructuras

Las instrucciones de control permiten a un programa ejecutar un código de forma condicional o repetirlo cierto número de veces.

En LabVIEW estas instrucciones son estructuras que encierran en su interior el código al que afectan. Se encuentran en el menú *Programming > Structures*.

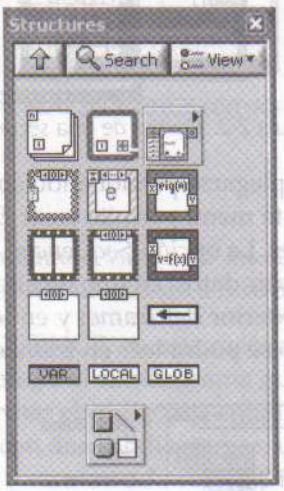


Figura 2-1. Estructuras

A continuación se va a tratar cada una de las estructuras que aparecen en la Figura 2-1.

2.1 SEQUENCE

En los lenguajes tradicionales basados en texto, el orden de ejecución se corresponde con el orden en que las instrucciones están escritas. Ya se ha visto que el sistema de ejecución de LabVIEW sigue el modelo de flujo de datos (*dataflow*), un nodo necesita tener disponibles los datos en todas sus entradas para ejecutarse, pero si hay dos nodos en condición de ejecutarse no se podrá determinar, en principio, el orden de ejecución; esto en la mayoría de los casos no será un problema, es más, será incluso beneficioso. No obstante puede haber ocasiones en que haya nodos independientes, ambos en situación de ejecutarse, pero se necesita fijar el orden de los mismos.

Las estructuras de tipo SEQUENCE sirven precisamente para esto: ordenan el orden de ejecución del código que está en su interior.

Su diseño recuerda a los fotogramas de una película. En una película los fotogramas colocados al principio se visualizarán antes que los siguientes, con un orden secuencial. Las estructuras SEQUENCE también tienen fotogramas o *frames* ordenados, en el interior de cada *frame* se situará una sección de código. La ejecución comenzará por el primer *frame* y cuando acabe se pasará a ejecutar el siguiente, y así sucesivamente.

Hay dos tipos de SEQUENCE: STACKED SEQUENCE y FLAT SEQUENCE.

La primera era la única disponible en versiones más antiguas de LabVIEW, tiene un menú en la parte superior donde se indica la numeración del *frame* que se muestra, el número total de *frames* que contiene y además da la opción de situarse en otro. En la figura 2-2 se muestran superpuestos los dos *frames* de la misma estructura.

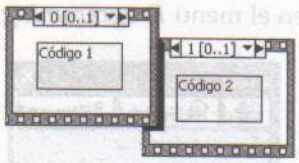


Figura 2-2. *Frames* de una secuencia

El menú contextual puede desplegarse presionando con el botón secundario del ratón en el borde de la estructura, este menú permite crear *frames* antes y después del mostrado, además de otras opciones. La opción *Sequence Local* crea unos túneles entre un *frame* y los demás para compartir datos, en uno de ellos se escribirá un valor (simbolizado con una flecha hacia el exterior del *frame*) y en los posteriores podrá leerse (con una flecha hacia el interior), no se podrá leer en *frames* anteriores al de escritura.

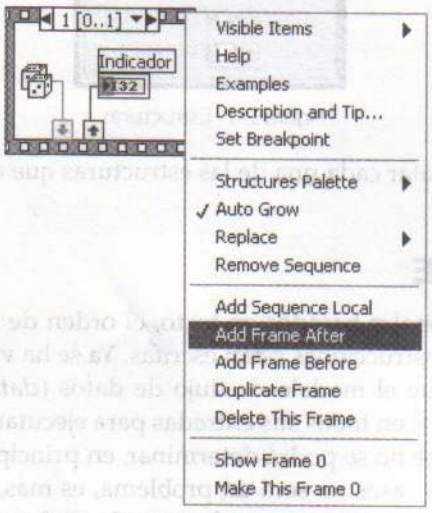


Figura 2-3. *Sequence Local*: permite compartir datos entre *frames*

También pueden pasarse y recibirse datos desde una estructura SEQUENCE al exterior a través de túneles, representados mediante un pequeño cuadrado en el borde de la estructura. Cuando hay un dato de salida, sólo un *frame* de la estructura puede escribir valores en él.

FLAT SEQUENCE funciona de igual forma, sólo que es más visual, los *frames* se ven uno a continuación del siguiente, el orden de ejecución será de izquierda a derecha. En este caso no hay *Sequence Local* y los datos podrán cablearse directamente desde un *frame* a otro a través de túneles. El menú contextual también será el que permita añadir y eliminar *frames*, también se puede cambiar de un tipo de SEQUENCE a otro de forma automática.

En la figura 2-4 puede verse una estructura FLAT SEQUENCE con dos *frames*, en el primero se genera un número aleatorio y se pasa al segundo *frame* a través de un túnel.

Figura 2-4. FLAT SEQUENCE con dos *frames*

2.2 CASE

La estructura CASE es el equivalente a varias de los lenguajes basados en texto: `IF`, `SWITCH` y `TRY`. Su utilidad es ejecutar un código u otro dependiendo de una condición.

Al igual que una estructura SEQUENCE, en este caso también se tiene un menú en la parte superior donde se puede elegir el subdiagrama que se muestra. En este menú se puede ver la condición para ejecutar el código del subdiagrama correspondiente. En la Figura 2-5 se muestra un CASE con dos subdiagramas, uno se ejecutará cuando la condición que se evalúa sea FALSE y otro cuando sea TRUE.

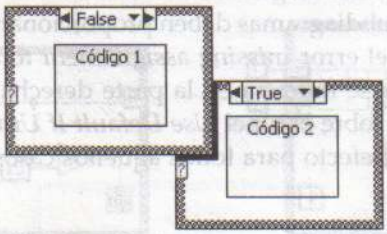


Figura 2-5. Estructura CASE

El terminal que aparece en el lado izquierdo marcado con el símbolo «?» es llamado selector. El valor que llega a este selector es la condición que se evalúa para seleccionar el subdiagrama a ejecutar. Si el tipo de datos que se conecta al selector del CASE es booleano, éste actuará como una sentencia `IF...THEN...ELSE` de un lenguaje de texto tradicional. También pueden conectarse otros tipos de datos, en este caso actuará como un

SWITCH...CASE. Pueden conectarse al selector datos booleanos, numéricos (incluidos *enum* y *ring*), *strings* y *clusters* de error. En la Figura 2-6 se pueden ver todos estos casos.

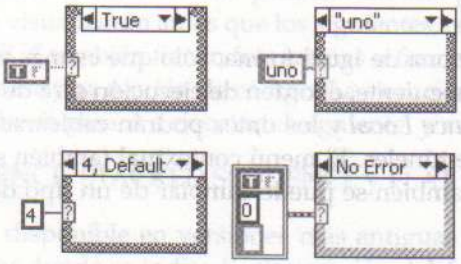


Figura 2-6. Tipos de datos que son válidos en un CASE

Para un selector booleano sólo se tendrán dos casos: verdadero o falso. Para numéricos la condición será que el dato del selector sea igual al mostrado en el menú del CASE, para datos *enum* o *ring* se puede escribir el nombre del ítem en lugar del valor numérico. Con los *strings* sucede lo mismo que con los numéricos *enum* o *ring*, el valor mostrado en el menú aparecerá como un texto encerrado en comillas dobles. En el caso de conectar al selector un *cluster* de error cambiará el color del borde de la estructura, los subdiagramas se ejecutarán dependiendo de si el selector marca un error o no.

Cuando el selector se conecta a un *string* o a un dato numérico es obligatorio tener algún caso que se ejecute por defecto, es decir, debe haber un caso que se ejecute cuando en el selector haya un valor que no esté asignado explícitamente a algún subdiagrama. Para hacer que un subdiagrama sea ejecutado por defecto debe seleccionarse *Make This The Default Case* en el menú contextual.

En los numéricos además se puede asignar una lista de valores escribiéndolos separados por comas o también se puede especificar un rango de valores, por ejemplo si se desea ejecutar el mismo código cuando la entrada tiene el valor 4, 5, 6, 7 y 8, en el menú del CASE se escribirá "4..8".

Se pueden pasar datos a los subdiagramas del CASE a través de túneles. En el caso de datos de salida, todos los subdiagramas deben proporcionar un valor, hasta que esto no ocurra LabVIEW indicará el error *missing assignament to tunnel* y aparecerá el túnel con el interior vacío, como se muestra en la parte derecha de la Figura 2-7. También existe la opción de marcar sobre el túnel *Use Default If Unwired*, con esto se consigue que se asigne el valor por defecto para todos aquellos casos que no se ha cableado un valor en el túnel de salida.

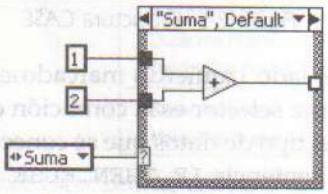


Figura 2-7. Error en un CASE: túnel vacío

Por último también hay que decir que se puede cambiar el orden de los subdiagramas desde la opción *Rearrange Cases...* del menú contextual.

2.3 WHILE

El bucle WHILE repetirá el código de su interior hasta que se cumpla una condición, la cual es evaluada en cada iteración.

En la figura 2-8 puede verse el aspecto de este bucle, en él se aprecian dos terminales:

- El terminal de iteración es el cuadrado azul con el símbolo «i». El valor de este terminal es un número entero que irá aumentando en una unidad por cada iteración del bucle, empezando a contar desde cero.
- La condición de stop es el terminal verde de la esquina inferior derecha de la imagen. A este terminal se podrá conectar bien un valor booleano, bien un *cluster* de error. A través del menú contextual podrá elegirse para los booleanos que el bucle se detenga cuando el valor sea *True* (*Stop if True*) o *False* (*Continue if True*), en el caso de los *cluster* de error sucede algo parecido con *Stop on Error* y *Continue while Error*.

Figura 2-8. Estructura WHILE

Otra de las opciones que muestra el menú contextual es *Add Shift Register*. Esta herramienta añade dos terminales a cada lado de la estructura, estos terminales sirven para transferir un valor desde una iteración del bucle a la siguiente. Los valores se pasarán a la siguiente iteración en el terminal de la derecha y se leerán en el de la izquierda. Si se conecta un valor al terminal de la izquierda en el exterior de la estructura, éste será el valor inicial que circulará por ese cable en la primera iteración.

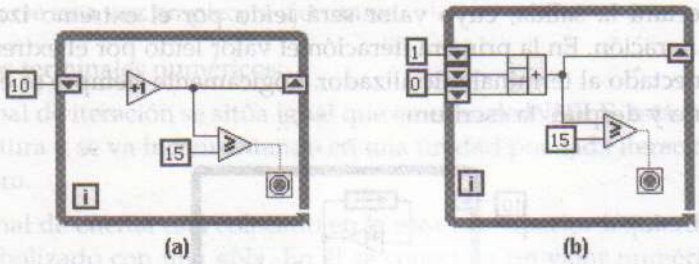


Figura 2-9. (a) *Shift register* en un bucle WHILE. (b) Varios *shift registers*

En la figura 2-9 (a) se muestra el uso de un *Shift register*. En la primera iteración se leerá el valor 10 del terminal de la izquierda, se le sumará 1 y se escribirá en el terminal de la derecha el valor 11. Este valor será el leído en la siguiente iteración en el terminal de la

izquierda, al que se le volverá a sumar 1 y así sucesivamente hasta que se cumpla la condición de parada del bucle, que en este caso es que el número sea mayor o igual a quince. Este proceso puede comprobarse ejecutando el código de la figura con la opción de *Highlight Execution* activada.

El *shift register* de la izquierda puede extenderse para mostrar más terminales, el terminal superior tendrá el valor que se escribió en la derecha en la iteración anterior, el siguiente terminal tendrá el valor que se escribió en la derecha dos iteraciones antes y así sucesivamente.

En el ejemplo de la figura 2-9 (b) se puede ver un ejemplo del uso de varios *shift register*. La Tabla 1 muestra los valores que tendrán cada uno de los terminales en cada iteración.

Tabla 1 - Valores de los *shift* register en cada iteración

Iteración	Shift izq arriba	Shift izq centro	Shift izq abajo	Shift derecha
1	1	0	0	1+0+0=1
2	1	1	0	1+1+0=2
3	2	1	1	2+1+1=4
4	4	2	1	4+2+1=7
5	7	4	2	7+4+2=13
6	13	7	4	13+7+4=24

Hay una utilidad que funciona de igual forma que un *shift register*, es el *Feedback Node*. Éste consta de dos terminales:

- El terminal inicializador permite dar un valor inicial al nodo, es equivalente a conectar un valor al terminal izquierdo de *shift register*. Este terminal se coloca siempre en el borde izquierdo de la estructura a la misma altura que el *Feedback Node*.
- El *Feedback Node* es el otro terminal, tiene forma de flecha. En el extremo derecho se le conectará la salida, cuyo valor será leído por el extremo izquierdo en la siguiente iteración. En la primera iteración el valor leído por el extremo izquierdo será el conectado al terminal inicializador. Lógicamente siempre se ejecuta primero la lectura y después la escritura.

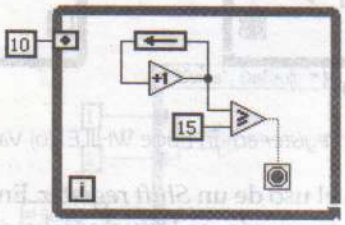


Figura 2-10. *Feedback Node*

El código de la figura 2-10 ilustra el uso del *Feedback Node*. El programa es igual al de la figura 2-9.

En el menú *Tools > Options > Block Diagram* se tiene la opción de insertar automáticamente *Feedback Nodes* dentro de los bucles cuando sea necesario.

Al igual que en otras estructuras, los datos que entren y salgan de una estructura WHILE lo harán a través de túneles. Una opción muy interesante de los túneles en los bucles es el *Autoindexing*, que se puede habilitar a través del menú contextual del túnel cuando se quiera trabajar con *arrays*. Cuando se cablea un *array* desde el exterior al interior de un bucle y se habilita el *autoindexing*, los valores leídos en ese terminal en el interior del bucle serán los elementos que componen el *array*, uno por cada iteración. Cuando el *autoindexing* está en una salida del bucle ocurre lo contrario: se construirá un *array* cuyos elementos serán los generados en cada iteración.

En la figura 2-11 puede verse un programa parecido al anterior. En el borde derecho pueden verse tres terminales, el superior es un túnel, el siguiente un *shift register* y el inferior una salida indexada. El valor de los dos primeros será el generado por el programa en su última iteración (15), mientras que el valor en el último terminal será una lista ordenada de todos los números generados en cada iteración (11, 12, 13, 14 y 15).

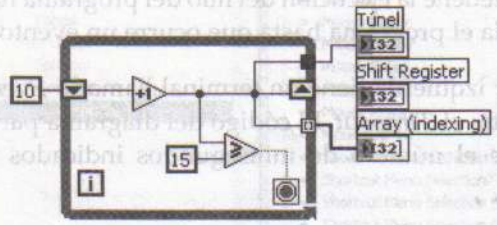


Figura 2-11. Ejemplo de túnel, *shift register* y *autoindexing*

2.4 FOR

El bucle FOR es muy parecido al WHILE, también repite el código de su interior un número de veces, pero a diferencia del anterior este número es fijado a priori y no puede cambiarse una vez empiece a ejecutarse.

Consta de dos terminales numéricos:

- El terminal de iteración se sitúa igual que en el bucle WHILE, está en el interior de la estructura y se va incrementando en una unidad por cada iteración empezando desde cero.
- El terminal de cuenta está colocado en la esquina superior izquierda de la estructura simbolizado con una «N». En él se conectará un valor numérico que será el que fije el número de repeticiones del bucle.

Todo lo dicho en la explicación del bucle WHILE respecto a los *shift register*, los *Feedback Nodes* y la salida indexada también es válido para el FOR. Tanto en el menú contextual del WHILE como del FOR se tiene la opción de sustituir uno por el otro.

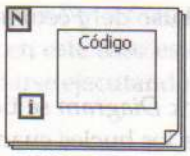


Figura 2-12. Estructura FOR

Los túneles indexados son la opción por defecto en los bucles FOR. Cuando se cablea un *array* de forma indexada como entrada puede obviarse el terminal de cuenta porque se toma el tamaño del *array* como el número de veces que se ha de repetir el bucle.

2.5 EVENT

La estructura EVENT fue introducida por primera vez en la versión 6.1 de LabVIEW. Es una estructura muy útil en VIs con los que interactúa el usuario porque mejora la eficiencia del programa.

Al igual que la estructura CASE tienen varios subdiagramas y un menú en la parte superior para cambiar el que se muestra. En este menú también se tiene una condición que hace que el código del subdiagrama correspondiente se ejecute. La diferencia con CASE es que EVENT detiene la ejecución del hilo del programa hasta que se da esa condición, es decir, congela el programa hasta que ocurre un evento.

En la esquina superior izquierda tiene un terminal llamado *Event Timeout* que se usa en el evento por defecto: el *Timeout*. El código del diagrama para el evento *timeout* se ejecutará cuando pase el número de milisegundos indicados en el terminal *Event Timeout*.

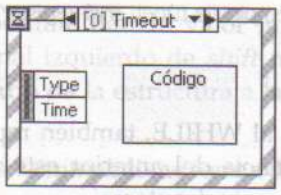


Figura 2-13. Estructura EVENT

Para añadir más subdiagramas hay que proceder de igual manera que con CASE, es decir, a través del menú contextual. Cada diagrama debe tener asociados uno o varios eventos, éstos se configuran desde la ventana *Edit Events*, como se puede ver en la figura 2-14.

En esta ventana primero se muestra el número del diagrama (*Events Handled for Case*) y a continuación la lista de los eventos que pueden dar lugar a la ejecución del diagrama (*Event Specifiers*), para añadir o eliminar eventos se usan los botones de la izquierda. Para definir un evento primero hay que especificar su fuente en *Event Sources*, los eventos de las secciones *Application* y *This VI* están predefinidos y son acciones típicas, como cerrar la ventana, presionar una tecla, etc. La fuente de eventos *Dynamic* sólo

está disponible cuando se activa *Show Dynamic Event Terminals* y se cablean convenientemente, para más información se puede consultar el tema dedicado a programación multihilo; *Panes* se activa cuando el Panel Frontal se divide en varias partes, *Splitters* sólo se activa cuando hay uno de estos elementos en el Panel Frontal y finalmente *Controls* muestra una lista de todos los controles que hay en el VI. En *Events* se podrá elegir el evento concreto asociado a la fuente seleccionada.

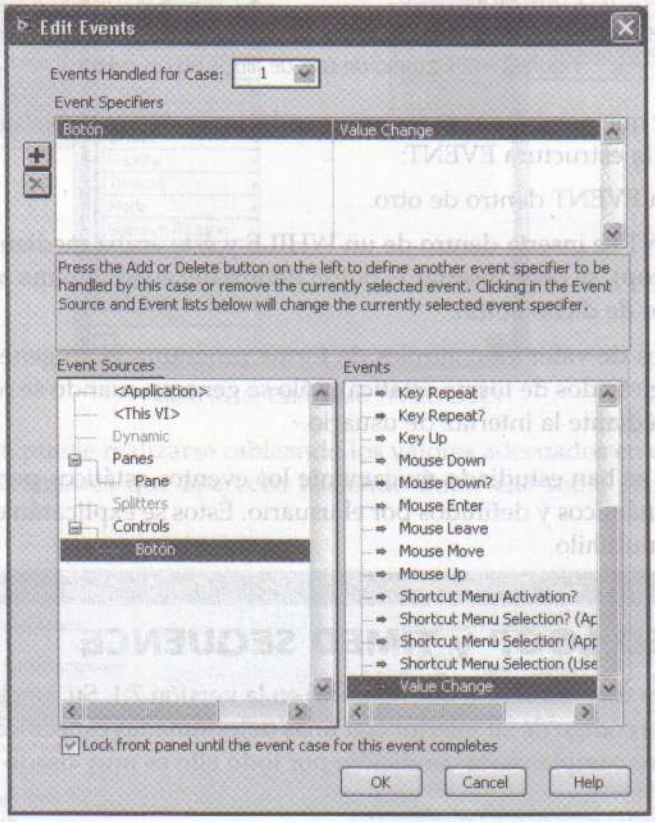


Figura 2-14. Ventana de configuración de eventos

En la parte izquierda de cada subdiagrama de la estructura EVENT se sitúa *Event Data Node* que aporta información sobre la fuente del evento, por ejemplo, una referencia al control, su valor actual y el anterior al evento, etc.

Si se observa con detalle la figura 2-14 puede verse que hay eventos marcados con una flecha roja que acaban en una interrogación y otros con el mismo nombre pero con una flecha verde y sin interrogación. Los que tienen la flecha roja son llamados filtros. Los filtros se ejecutan antes que la acción asociada al evento, pudiendo, entre otras cosas, desactivarlo. Cuando se selecciona un *filter event* aparecerá un nodo en la parte derecha del diagrama llamado *Event Filter Node*.

El ejemplo de la figura 2-15 muestra el uso de los *filter events* para descartar un evento, en este caso sirve para impedir que el usuario cierre el Panel Frontal del VI.

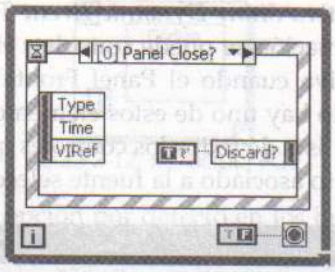


Figura 2-15. Ejemplo de uso de filtros en un evento

Finalmente hay que nombrar algunas recomendaciones que hace National Instruments sobre el uso de la estructura EVENT:

- No usar un EVENT dentro de otro.
- Si un EVENT se inserta dentro de un WHILE y éste acaba mediante un botón (un botón de stop), se debe insertar el botón dentro del subdiagrama asociado al cambio de valor de dicho botón.
- Los cambios de valor en controles de forma programada no generan eventos (en eventos registrados de forma estática), sólo se generan cuando se realiza el cambio de valor mediante la interfaz de usuario.

En esta sección se han estudiado únicamente los eventos estáticos pero hay dos tipos más: eventos dinámicos y definidos por el usuario. Éstos se explicarán en el tema sobre programación multihilo.

2.6 TIMED LOOP y TIMED SEQUENCE

TIMED LOOP es una estructura que apareció en la versión 7.1. Su funcionamiento consiste en repetir el código de su interior con unas determinadas especificaciones de tiempo o hasta que se cumpla cierta condición, por todo ello es muy usado en aplicaciones de tiempo real.

Su dibujo recuerda a un WHILE circundado por un halo azul, aunque a diferencia de éste, no es necesario establecer una condición de parada o continuación. También presenta varios nodos, éstos son (de izquierda a derecha):

- *Input*: permite configurar el funcionamiento del bucle por medio de un asistente o cableando los datos. Este nodo puede extenderse para cablear más datos de los mostrados.
- *Left Data*: proporciona información sobre la iteración anterior, por ejemplo el tiempo que ha tardado (*Iteration Duration*), si le ha dado tiempo a acabar la tarea antes de que empiece una nueva ejecución (*Finished Late? [i-1]*) y otras.
- *Right Data*: se trata de una configuración dinámica, permite modificar los parámetros de configuración de la estructura para la siguiente iteración. Los parámetros son prácticamente los mismos que en el *Input Node*, excepto el nombre del bucle y el origen del reloj.

- *Output*: al igual que el *Left Data Node*, el *Output Node* proporciona información, pero en este caso la información se genera después de que el bucle se haya detenido.

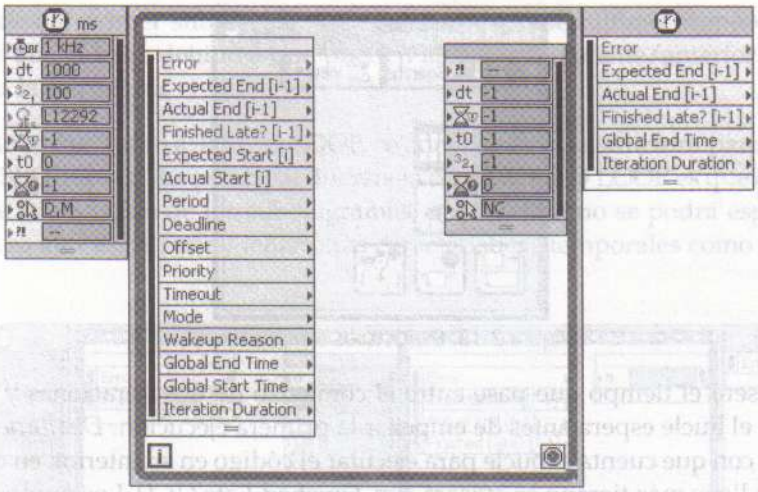


Figura 2-16. Estructura TIMED LOOP

La configuración puede realizarse cableando los valores adecuados en el *Input Node* o mediante los parámetros que aparecen haciendo doble clic sobre el primer y tercer nodo.

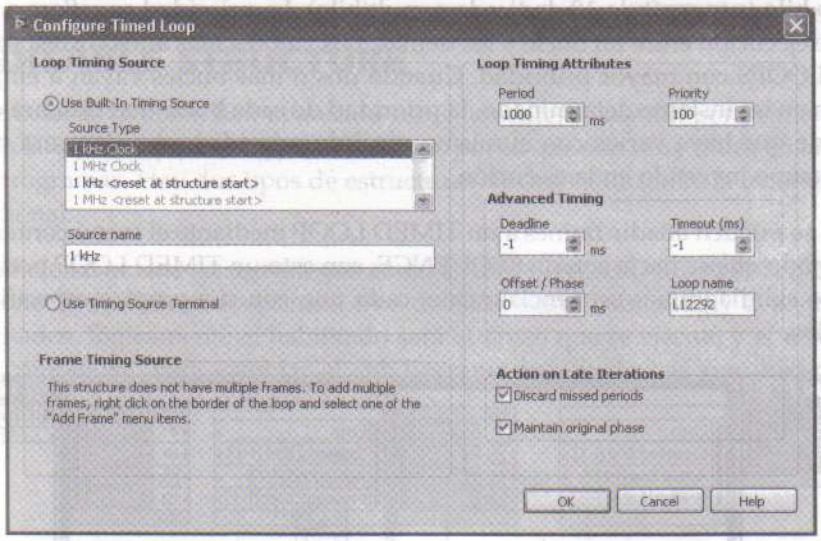


Figura 2-17. Ventana de configuración del TIMED LOOP

Timing Source es la fuente de reloj, cuando el programa se ejecute sobre un PC la fuente será un reloj de 1 kHz del sistema operativo; si la ejecución se realiza en otros dispositivos puede haber otras fuentes disponibles. Para seleccionar la fuente desde el programa puede usarse el VI *Create Timing Source*, en la paleta *Programming > Structures*

> *Timed Structures*. Los otros VIs de esta paleta permiten abortar programadamente un TIMED LOOP (*Stop Timed Structure*), sincronizarlos (*Syncronize Timed Structure Starts*) o controlar el comienzo de cada uno (*Building Timing Source Hierarchy*).

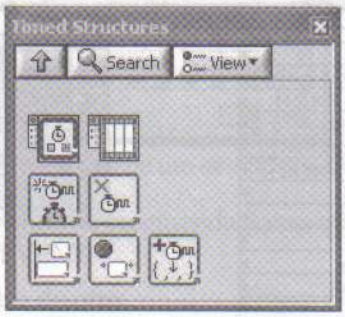


Figura 2-18. Estructuras temporizadas

El periodo será el tiempo que pase entre el comienzo de dos iteraciones y el *offset* el tiempo que el bucle espera antes de empezar la primera ejecución. *Deadline* es el tiempo máximo con que cuenta el bucle para ejecutar el código en su interior, en caso de que la ejecución lleve más tiempo se avisará con *Finished Late? [i-1]*. Los modos de *Action on Late Iterations* configuran la forma en que el bucle responde cuando la ejecución lleva más tiempo del especificado, las opciones son autoexplicativas, básicamente sirven para alinear el comienzo de las ejecuciones y saltar iteraciones.

Para ejecutarse, cada TIMED LOOP crea su propio sistema de ejecución que contiene un único hilo (ver capítulo 10 dedicado a multihilo). La prioridad se refiere a la preferencia de ejecución entre un bucle y los demás. Los valores más altos corresponderán a TIMED LOOPS con mayor prioridad. Cuando dos o más bucles vayan a empezar su ejecución en un instante determinado, la prioridad de cada bucle determinará el orden, en caso de que haya varios con el mismo nivel de prioridad, el orden será por el de menor tiempo invertido en la ejecución.

También se pueden añadir *frames* a un TIMED LOOP mediante el menú contextual del mismo modo que se hacía con un SEQUENCE, con esto un TIMED LOOP podrá ejecutar varios subdiagramas secuencialmente, cada uno con sus propias especificaciones temporales.

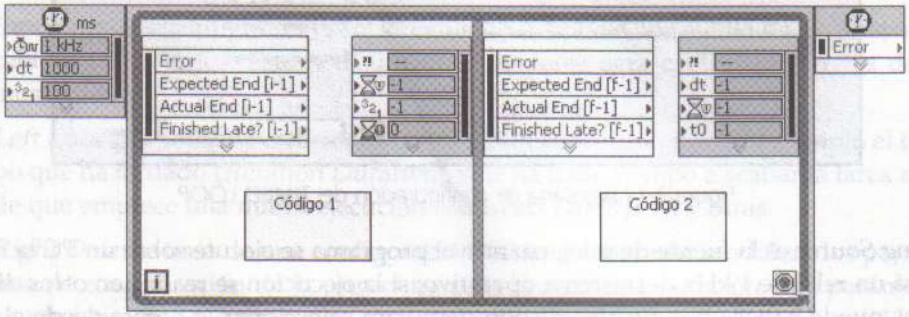


Figura 2-19. TIMED LOOP con *frames*

En este caso el *Right Data Node* de un *frame* cambiará los parámetros del siguiente subdiagrama a ejecutarse en lugar de la siguiente iteración y el *Left Data Node* proporcionará información sobre el subdiagrama anterior.

El *Left Node* del primer subdiagrama y el *Right Node* del último proporcionan más parámetros que el resto debido a que pueden afectar al siguiente/anterior subdiagrama o a toda la estructura.

En el mismo menú, junto a TIMED LOOP, se encuentra una estructura parecida llamada TIMED SEQUENCE. La principal diferencia con el TIMED LOOP es que en este caso no se repite la ejecución de los subdiagramas, por lo tanto no se podrá especificar un periodo, pero aún así sí puede tener otras características temporales como *offset*, *deadline*, etc.

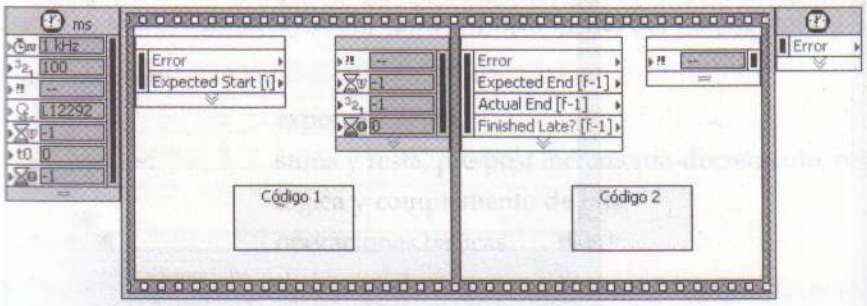


Figura 2-20. Estructura TIMED SEQUENCE

2.7 DISABLE STRUCTURE

Estas estructuras también han aparecido recientemente, se usaron por primera vez en la versión 8.0. Sirven para comentar el código, por lo que son muy útiles en la depuración de programas. Hay dos tipos de estructuras de deshabilitación: la incondicional y la condicional.

La incondicional es una estructura que como CASE, STACKED SEQUENCE o EVENT se compone de varios subdiagramas. Uno de ellos estará habilitado y el resto estarán deshabilitados, lógicamente el habilitado será el único que se ejecute y el resto no llegarán a compilarse. Para cambiar el subdiagrama habilitado hay que hacer uso del menú contextual de la estructura.

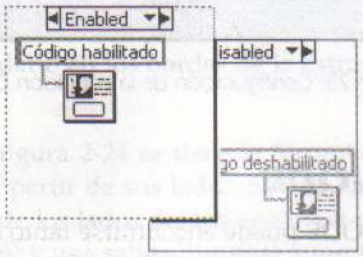


Figura 2-21. Estructura DISABLE

La condicional funciona de la misma forma que la anterior, sólo que el diagrama habilitado se selecciona de forma automática dependiendo del valor de unos símbolos asociados al proyecto. Algunos símbolos predefinidos son OS y CPU.

También se pueden definir nuevos símbolos en la ventana de propiedades del proyecto correspondiente, como muestra la figura 2-22.

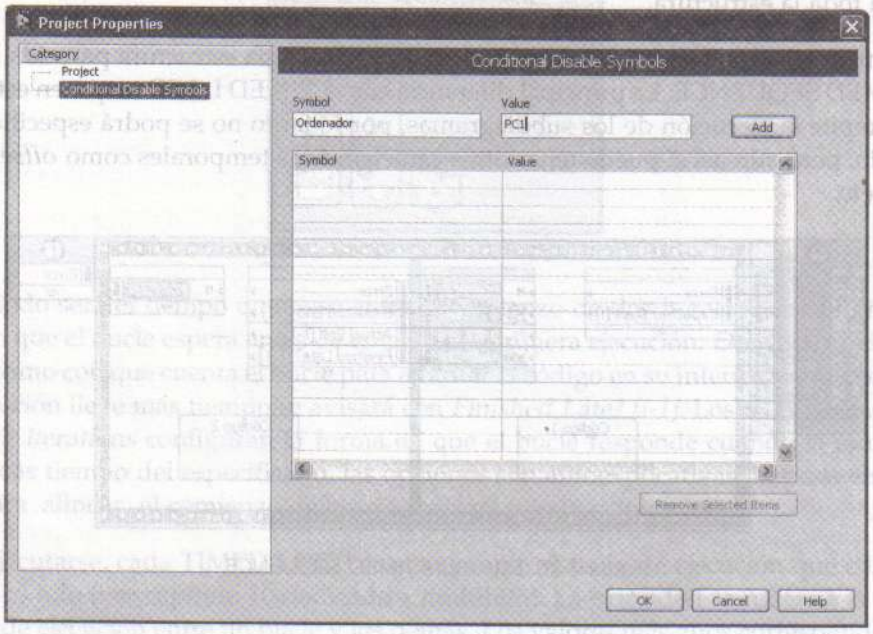


Figura 2-22. Ventana para crear símbolos

Una vez creados los símbolos pertinentes hay que editar la condición de la estructura con *Edit Condition For This Subdiagram...* del menú contextual. Se puede añadir más de una condición con el botón «+».

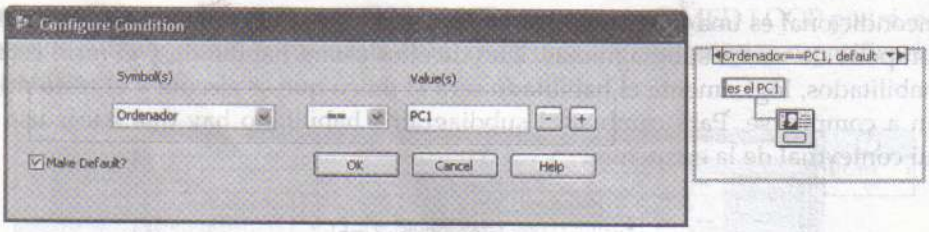


Figura 2-23. Configuración de la condición DISABLE

2.8 FORMULA NODE

La estructura FORMULA NODE puede encontrarse tanto en el menú *Programming > Structures* como en *Mathematics > Scripts & Formulas*.

A diferencia de las anteriores, FORMULA NODE no controla el flujo de ejecución, sino que evalúa una expresión matemática escrita como texto con una sintaxis parecida al lenguaje C.

El texto consistirá en una serie de sentencias finalizadas por el símbolo «;». Las sentencias normalmente son asignaciones que usan operadores o funciones, aunque también pueden ser declaraciones de variables, bucles o sentencias de condición. También pueden insertarse comentarios de la misma manera que en C.

Casi todas las funciones que se pueden usar dentro de un FORMULA NODE tienen su equivalente como VI. Éstas son: abs(*x*), acos(*x*), acosh(*x*), asin(*x*), asinh(*x*), atan(*x*), atan2(*x,y*), atanh(*x*), ceil(*x*), ci(*x*), cos(*x*), cosh(*x*), cot(*x*), csc(*x*), exp(*x*), expm1(*x*), floor(*x*), getexp(*x*), gamma(*x*), getman(*x*), int(*x*), intrz(*x*), ln(*x*), lnp1(*x*), log(*x*), log2(*x*), max(*x,y*), min(*x,y*), mod(*x,y*), pow(*x,y*), rand( ), rem(*x,y*), si(*x*), sec(*x*), sign(*x*), sin(*x*), sinc(*x*), sinh(*x*), sizeOfDim(*array,dim*), spike(*x*), sqrt(*x*), step(*x*), tan(*x*), tanh(*x*).

Los operadores son:

**:	exponenciación.
+, -, ++, - -, !, ~:	suma y resta, pre-post incremento-decremento, negación lógica y complemento de bits.
*, /, %, +, -:	operaciones básicas.
>>, <<:	desplazamientos.
<, >, >=, <=,!=, = =:	comparaciones.
&, ^, |, &&, ||:	operaciones lógicas.
...?... :... :	evaluación condicional.
=:	asignación.

Las estructuras son:

`If` (condición) sentencias1 `else` sentencias2

`do` sentencias `while` (condición)

`while` (condición) sentencias

`for` (asignación; condición; sentencia) sentencias

`switch` (condición) lista de casos

Otras palabras reservadas son: *break, case, continue, default, pi.* Para más información puede consultarse la ayuda.

Presionando el botón secundario en los bordes de la estructura pueden crearse variables de entrada o de salida.

En el ejemplo dado por la figura 2-24 se tiene la fórmula de Herón para calcular la superficie de un triángulo a partir de sus lados, en ella hay tres entradas (a, b, c) que corresponden a la longitud de los lados del triángulo, un resultado intermedio que se trata como una salida más (p) y una salida que es la superficie (S).

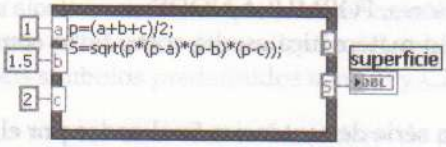


Figura 2-24. Fórmula de Herón

2.9 Scripts

Al igual que el nodo FORMULA NODE hay otras estructuras que también aceptan textos, de esta forma se puede combinar la programación propia de LabVIEW con la más tradicional programación textual.

El MATHSCRIPT NODE es otra de las nuevas estructuras de LabVIEW 8.0 y mejorada en 8.20. A diferencia de las otras dos que se verán en esta sección, MATHSCRIPT NODE no llama a otros programas y su código es compilado junto con el resto del VI.

En *Tools > MathScript Window* puede abrirse una ventana, figura 2-25, para ayudar a depurar el código. En el campo *Command Window* pueden escribirse los comandos, también puede ejecutarse un *script* completo desde la pestaña *Script* y los resultados pueden verse en la pestaña *Variables*.

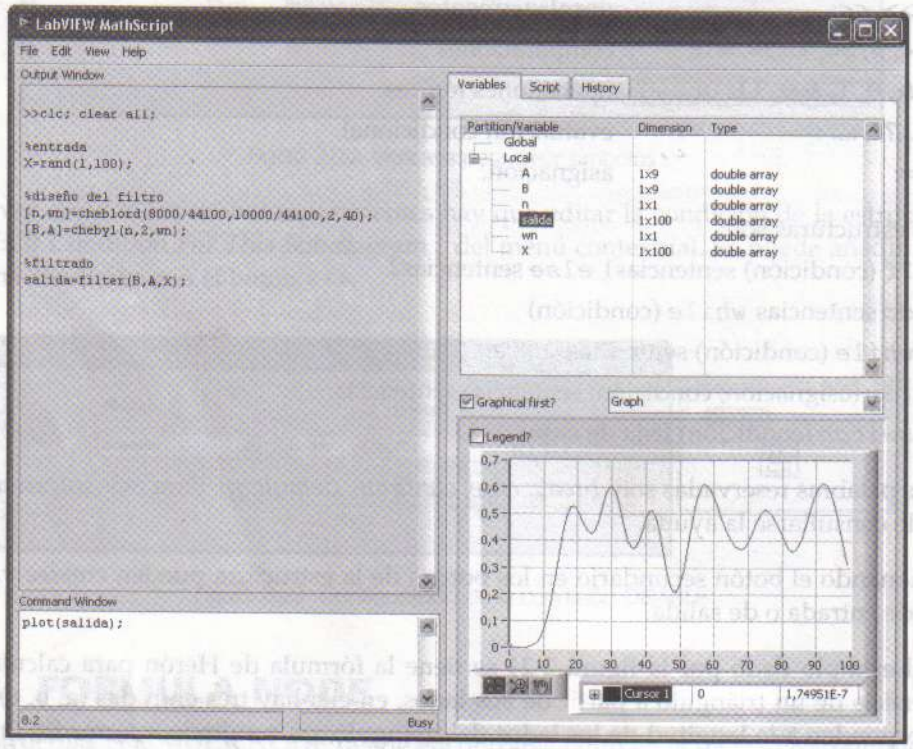


Figura 2-25. Depuración de código con MATH SCRIPT

La sintaxis para usar en la estructura MATHSCRIPT NODE es en gran medida compatible con la de MATLAB. Dispone de cientos de funciones que pueden consultarse en la ayuda.

Por otra parte, MATLAB SCRIPT NODE y XMATH SCRIPT NODE se encuentran únicamente en el menú *Mathematics > Scripts & Formulas > Script Nodes*. Ambos llaman a programas externos. En el caso de MATLAB, el nodo llama al servicio «Matlab Server» a través de ActiveX para ejecutar los comandos (sólo en Windows). XMath es parte del programa MATRIXx, también de National Instruments, es un software especializado en análisis numérico y su sintaxis es un poco diferente a la de MATLAB aunque tienen muchas funciones equivalentes.

Al igual que con FORMULA NODE se deben crear variables de entrada y salida pero en este caso hay que asignar explícitamente el tipo de datos, esto puede verse en la figura 2-26.

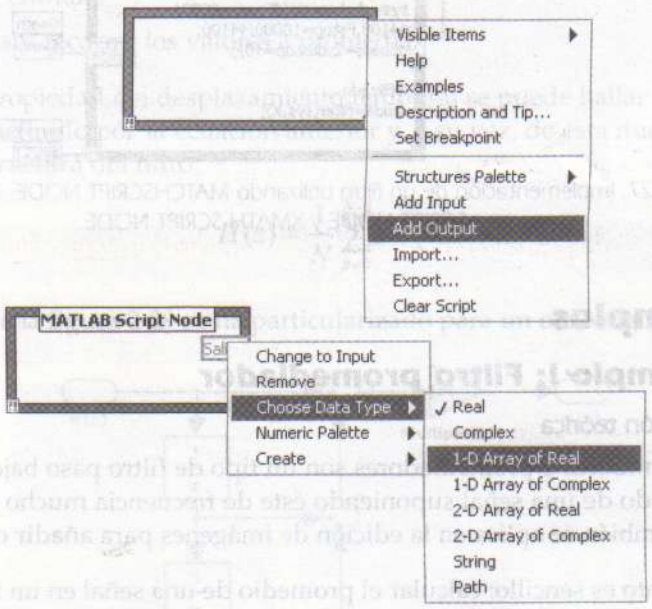


Figura 2-26. MATLAB SCRIPT NODE y XMATH SCRIPT NODE

En la figura 2-27 pueden verse estas tres estructuras. El código de las tres es equivalente, se trata del diseño de un filtro y su aplicación a una señal aleatoria.

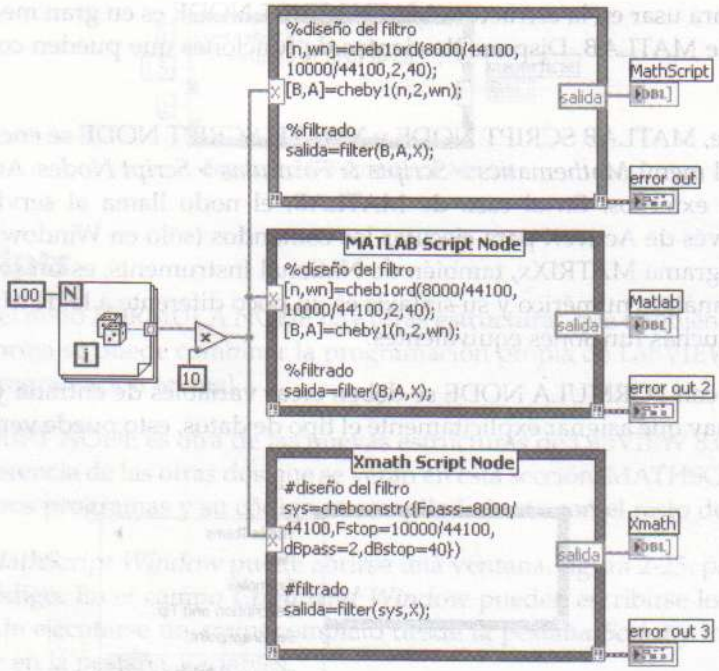


Figura 2-27. Implementación de un filtro utilizando MATCHSCRIPT NODE, MATLAB SCRIPT NODE y XMATH SCRIPT NODE.

2.10 Ejemplos

2.10.1 Ejemplo I: Filtro promediador

2.10.1 Explicación teórica

Los filtros integradores o promediadores son un tipo de filtro paso bajo. Su aplicación es reducir el ruido de una señal suponiendo éste de frecuencia mucho más alta que la propia señal, también se aplica en la edición de imágenes para añadir difuminados.

El funcionamiento es sencillo: calcular el promedio de una señal en un intervalo determinado. Puede aplicarse tanto a sistemas continuos como discretos.

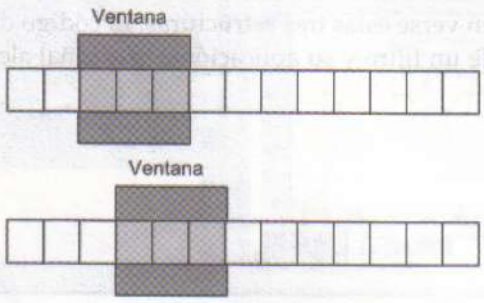


Figura 2-28. Ventana móvil para realizar el promediado

En un promediador móvil existe lo que se llama ventana que indica el tamaño del intervalo a promediar, en un sistema discreto este tamaño serán las muestras de la secuencia de entrada que se promediarán. Una ventana móvil avanza en cada paso del algoritmo una posición en la secuencia de entrada para realizar el promediado como se muestra en la figura 2-28.

Matemáticamente puede expresarse la ecuación en diferencias del sistema como:

$$y(n) = \frac{1}{N}\left(\sum_{k=0}^{N-1} x(n-k)\right)$$

Donde:

- *y(n)*: valor calculado.
- *N*: tamaño de la ventana.
- *x*: señal de entrada.
- *k*: índice para recorrer los valores a promediar.

Aplicando la propiedad del desplazamiento temporal se puede hallar la transformada Z del sistema definido por la ecuación anterior y, a su vez, de esta nueva expresión se obtendrá la estructura del filtro.

$$H(z) = \frac{1}{N}\sum_{k=0}^{N-1} z^{-k}$$

La estructura de la figura 2-29 se ha particularizado para un orden igual a cuatro.

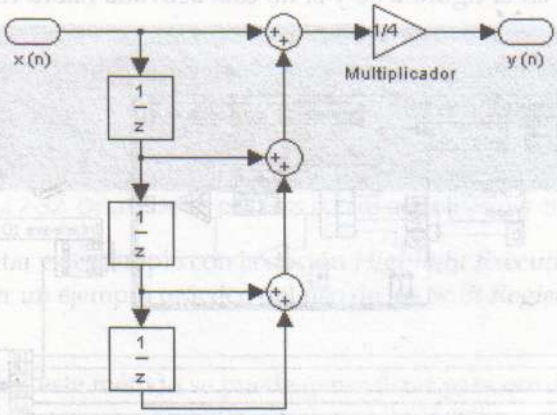


Figura 2-29. Filtro de orden cuatro

2.10.1.2 Código

Para realizar este ejemplo se implementará un filtro promediador móvil de orden cuatro que añada un difuminado a una fotografía.

La entrada será una imagen en blanco y negro con formato BMP que consiste en una lista (por simplicidad de un *array* de una dimensión) de valores de cada uno de los componentes de la imagen, los componentes son RGB (*Red, Green, Blue*), por lo tanto para cada punto habrá tres valores. La siguiente lista representa como sería este *array* para una imagen con cuatro puntos:

R1, G1, B1, R2, G2, B2, R3, G3, B3, R4, G4, B4...

Como la imagen sólo contiene grises, los valores de los tres componentes son iguales, por lo tanto bastará con aplicar el filtro a uno solo de los componentes o lo que es lo mismo, a uno de cada tres valores.

El programa empezará leyendo el fichero y desglosando su información; entre esta información se obtendrá un *array* RGB como el anterior (por simplicidad se trabajará en una dimensión). La función *Decimate 1D Array*, con un tamaño de tres elementos, obtendrá a la primera de sus salidas un *array* cuyos elementos sean los de las posiciones 1, 4, 7, 10, 13... del *array* de entrada, es decir, obtendrá todos los valores del componente R de cada punto.

Los valores R de cada punto son promediados, el promediador sumará el último valor leído y los tres anteriores, el resultado se dividirá por cuatro. La ventana se implementará mediante *Shift Registers* y el resultado se irá indexando en el lateral del FOR.

Para reconstruir la imagen basta con hacer el proceso inverso al de decimar: interpolar. Finalmente se dibujan las imágenes antes y después de aplicar el filtro.

Mediante la señal *selector* se podrá elegir el tipo de estructura implementada, si está activada será como en la figura 2-30 y si no está activada habrá realimentación de las salidas a la entrada.

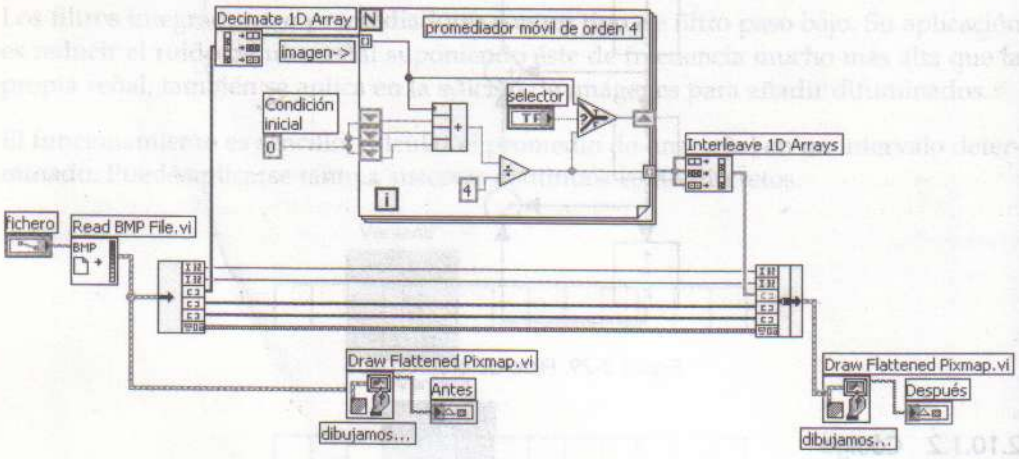


Figura 2-30. VI que implementa un filtro promediador móvil de orden 4

2.10.1.3 Resultado

Para explicar el efecto del filtro se puede decir que proporciona cierta resistencia al cambio en la imagen. En la figura 2-31 puede verse el resultado del filtro sobre una fotografía.

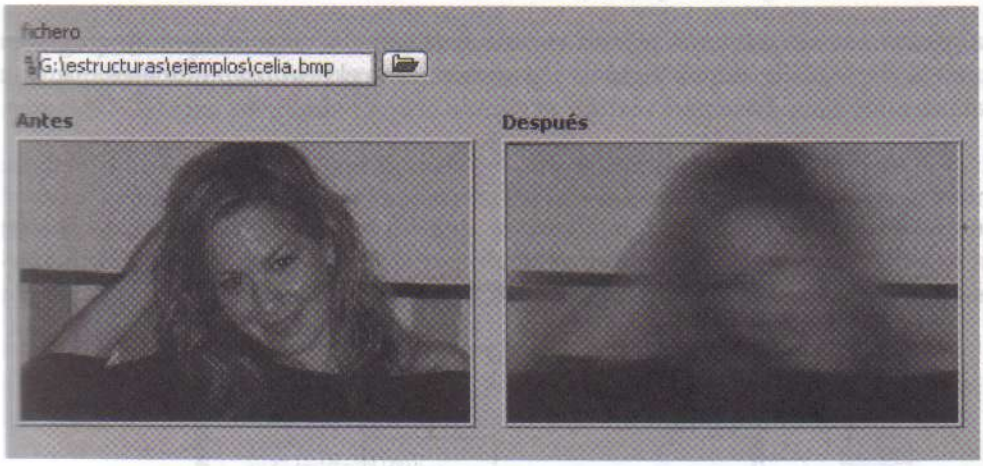


Figura 2-31. Resultado del filtro sobre una fotografía

La figura 2-32 es un detalle del principio de la fotografía después de aplicar el filtro (usando realimentación de salidas a la entrada para magnificar este efecto), en ella se ve como hay una serie de puntos negros que en la original no estaban, esto se debe a la condición inicial del filtro. Para paliar este problema hay varias formas: reducir el orden del filtro en las primeras muestras, usar el primer valor como condición inicial, etc.

Figura 2-32. Detalle de los primeros puntos que pasan por el filtro

Es aconsejable ejecutar este ejemplo con la opción *Highlight Execution* activada, de esta manera se puede ver un ejemplo práctico del uso de los *Shift Registers* y la salida indexada en los bucles.

Como se puede intuir, este método se puede generalizar para otros tipos de filtros FIR e IIR.

2.10.2 Ejemplo II: Generación de números primos

2.10.2.1 Explicación teórica

Como todo el mundo sabe, un número primo es aquel número natural que sólo es divisible por él mismo y por la unidad.

Desde los tiempos de Euclides (300 adC) se sabe que existen infinitos números primos. Los primeros algoritmos para encontrar números primos también proceden de la época de los antiguos griegos, como la «criba de Eratóstenes». Desde entonces ha pasado mucho tiempo pero aún se sigue investigando en este campo, por ejemplo en el año 2004 se creó otro algoritmo que mejora el anterior llamado «criba de Atkin».

La generación de números primos es una herramienta muy interesante en campos como la criptografía, donde algunos algoritmos como el RSA usan números primos de valores altos como base para realizar el cifrado.

2.10.2.2 Código

En este ejemplo se optará por un algoritmo que sea lo más sencillo posible, consistirá en hacer un barrido de números empezando por el dos hasta el límite indicado por el usuario, es decir, se recorrerán los números 2, 3, 4, 5, 6, 7, 8... Para cada número se vol-

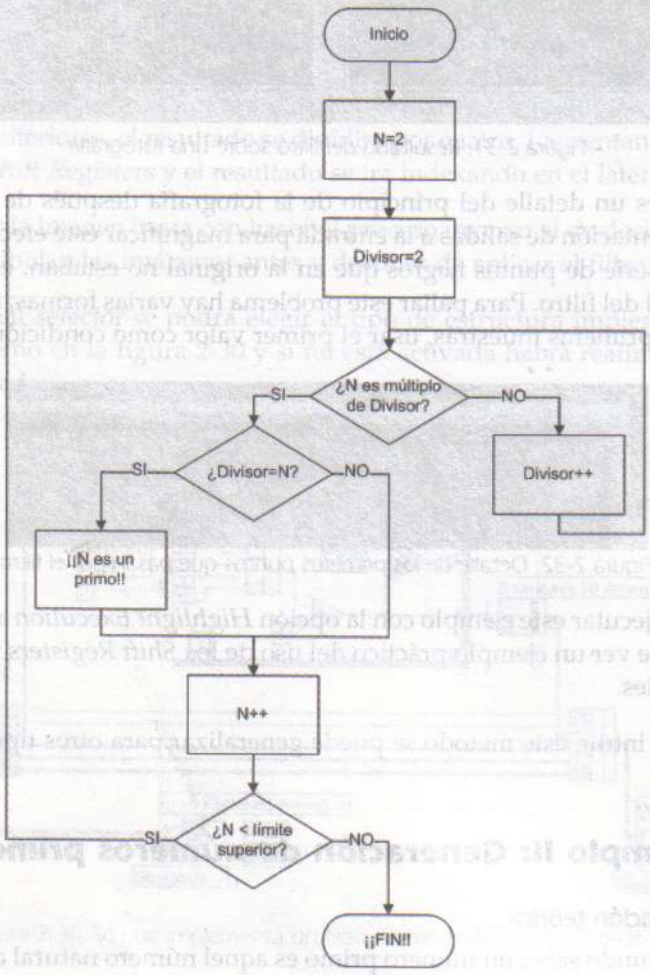


Figura 2-33. Algoritmo para la obtención de números primos

será a hacer un barrido hasta encontrar un número que sea divisor del primero, si estos dos números son iguales significa que el único divisor de este número es él mismo, por lo tanto es un número primo. En la figura 2-33 puede verse un diagrama de flujo de este algoritmo. Nótese que en este algoritmo, al igual que en muchos otros, se obvia el número uno.

El código para implementar este algoritmo con el nodo fórmula es exactamente igual que en lenguaje C. En primer lugar se declararán las variables y a continuación se usan dos bucles FOR, uno para cada barrido.

Para determinar si un número es divisor del otro se comprobará si el resto de la división entre ambos es igual a cero. Finalmente los números primos encontrados se almacenan en un *array*.

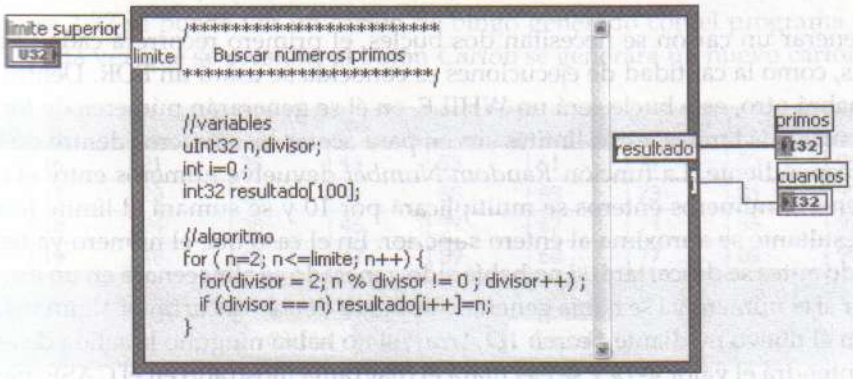


Figura 2-34. Implementación del algoritmo en un VI

2.10.2.3 Resultado

La figura 2-35 muestra un *array* con los números primos que hay entre dos y quince como resultado de la ejecución del programa anterior.

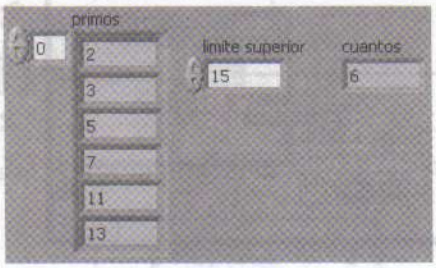


Figura 2-35. Números primos obtenidos

En este ejemplo se han aprendido dos cosas: en primer lugar a usar el FORMULA NODE y en segundo lugar a darse cuenta de que, a pesar de su nombre, esta estructura no sólo sirve para introducir fórmulas matemáticas sino que también se pueden emplear en ella elementos de un lenguaje de programación como bucles, funciones, etc.

2.10.3 Ejemplo III: Bingo

2.10.3.1 Explicación teórica

En esta ocasión se desea realizar un programa que genere un cartón para jugar al bingo. Para esto se necesita crear una matriz o *array* de dos dimensiones que contenga valores aleatorios entre 1 y 100. Habrá diez columnas (una para cada decena) y cuatro filas.

2.10.3.2 Código

El programa principal tiene la típica estructura de un WHILE y un EVENT.

El evento que se muestra en la figura 2-36 es el correspondiente al cambio de valor de un botón llamado *cartón*. Este botón tiene por acción mecánica *Latch When Released* (se verá en el próximo capítulo). Cuando se presiona el botón se ejecutará el subdiagrama.

Para generar un cartón se necesitan dos bucles, el primero recorrerá cada una de las decenas, como la cantidad de ejecuciones es conocida se usará un FOR. Dentro de este bucle habrá otro, este bucle será un WHILE, en él se generarán números de forma aleatoria entre dos límites, estos límites sirven para acotar los números dentro de la decena correspondiente. La función *Random Number* devuelve números entre el 0 y el 1, para generar números enteros se multiplicará por 10 y se sumará el límite inferior, el valor resultante se aproxima al entero superior. En el caso que el número ya haya sido generado antes se descartará; si no había sido generado se almacenará en un *array*. Para detectar si el número ya se había generado se busca dentro del *array* si algún valor coincide con el nuevo mediante *Search 1D Array*, si no había ninguno la salida de esta función contendrá el valor «-1» y se ejecutará el diagrama mostrado en el CASE, para cualquier otro número se ejecutará el caso por defecto que simplemente deja pasar el *array* de la entrada a la salida.

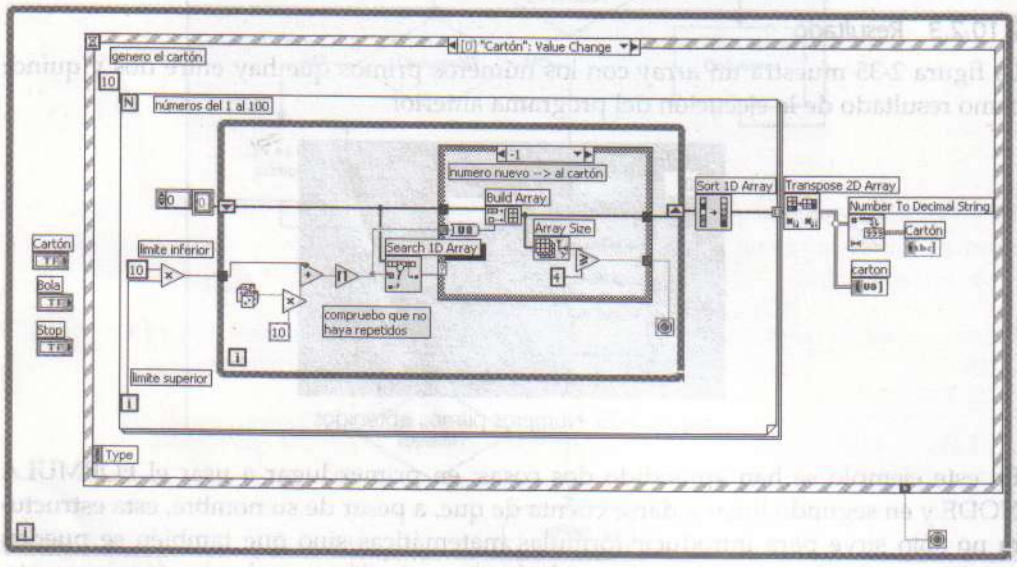


Figura 2-36. VI para la generación de cartones para jugar al bingo

Cuando se han generado cuatro números se detiene el WHILE, se ordenan los números del *array* y se pasa a la siguiente decena. Cuando se recorran todas las decenas, en la salida indexada del FOR se habrá generado un *array* de dos dimensiones con los números de un nuevo cartón.

La estructura EVENT contiene otro subdiagrama para cuando se presione el botón Stop. Este subdiagrama únicamente contiene una constante booleana con el valor *True* cableada a la condición del WHILE principal. Se puede pensar que sería más sencillo dejar vacío este caso y cablear directamente el botón de *Stop* a la condición, pero no es recomendable porque debido al flujo de ejecución de LabVIEW no se podría determinar si la condición de parada del WHILE se evalúa antes o después del evento.

2.10.3.3 Resultado

En la figura 2-37 se puede ver un cartón de bingo generado con el programa de este ejemplo. Cada vez que se presione el botón *Cartón* se generará un nuevo cartón.

Cartón

2	16	21	35	43	53	61	71	83	94
4	17	23	36	44	55	65	73	84	95
6	18	24	38	46	57	67	74	85	97
10	19	28	40	47	59	68	77	89	99

Cartón Cancel

Figura 2-37. Cartón obtenido de la ejecución del VI

Este ejemplo muestra el uso de las estructuras EVENT, WHILE, FOR y CASE. Por una parte se ha visto cuando se debe usar FOR (con un número de ejecuciones conocido) y cuando WHILE (cuando no hay un número determinado de ejecuciones); también se ha visto el uso de la estructura EVENT y el típico problema del botón de *Stop* y por último se ha utilizado un CASE dejando un caso por defecto. En el próximo capítulo se explicarán más detalladamente los *arrays*.

2.11 Ejercicios

1. ¿A qué es equivalente el siguiente programa?

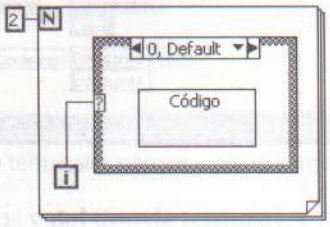


Figura 2-38. Ejercicio 1

2. Realizar un programa que calcule el factorial de un número. Hacerlo primero con estructuras de repetición y luego con FORMULA NODE.
3. Realizar un programa que genere un *array* con el primer millón de números enteros mediante un bucle WHILE y un FOR. ¿Cuál es más eficiente? ¿Por qué?
4. Modificar el programa del ejemplo I para que también acepte imágenes en color.
5. Realizar mediante programación visual el mismo algoritmo que en el ejemplo II. Comparar ambos programas en términos de velocidad y espacio.
6. Añadir al evento del ejemplo III otro caso cuando se presione un botón llamado *bola*. El subdiagrama debe simular el funcionamiento de un bombo para jugar al bingo, también se debe tener en cuenta no repetir las bolas que ya habían sido sacadas. Opcionalmente también puede añadirse algún mecanismo para detectar si la bola que ha salido estaba en el cartón (Pista: ver en el siguiente capítulo las propiedades de los controles).

2.12 Bibliografía

J. Kodosky, E. Pérez, *AN 039: Linear System in LabVIEW*, National Instruments, 1993.

National Instruments, *AN 200: Using the Timed Loop to Write Multirate Applications in LabVIEW*, 2004.

National Instruments, *LabVIEW Basics I Course Manual*, 2000.

National Instruments, *LabVIEW User Manual*, 2001.

Rick Bitter et alt., *LabVIEW Advanced Programming Techniques*, CRC Press LLC, 2001.

Capítulo 3

Tipos de datos

3.1 Tipos de terminales

En el Panel Frontal se pueden colocar controles e indicadores. Como se ha dicho en capítulos anteriores, los controles son el punto de entrada de información al VI y los indicadores de salida. Los controles e indicadores tendrán su representación en el Diagrama de Bloques, donde podrán ser cableados para leer o escribir su valor. En el Diagrama de Bloques además se podrán tener también constantes. En el menú contextual de estos elementos hay opciones para cambiar de un tipo a otro, en la figura 3-1 se ha marcado la opción para cambiar un control de tipo *string* a un indicador del mismo tipo.

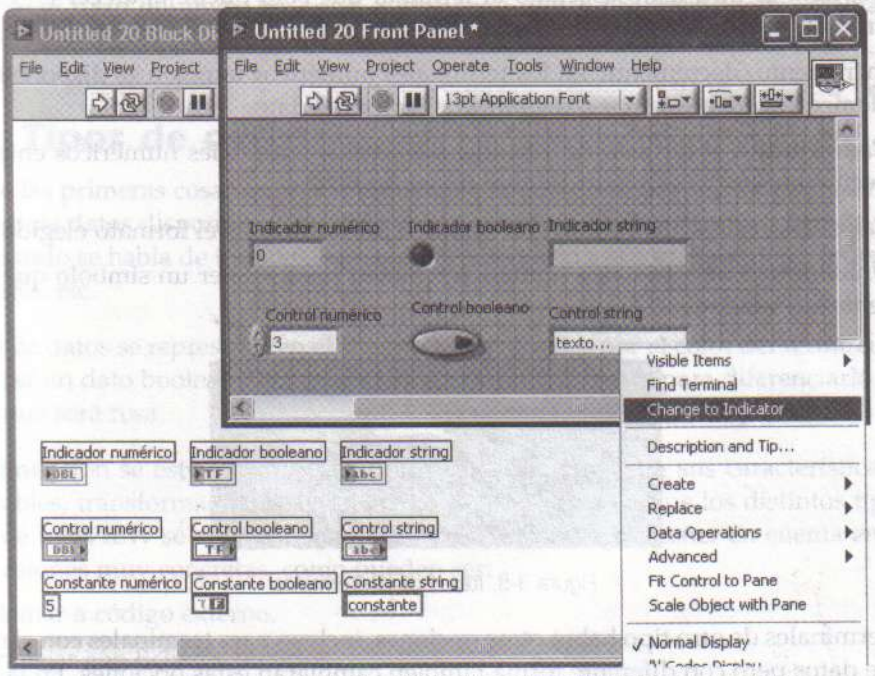


Figura 3-1. Tipos de terminales básicos y como cambiar de un tipo a otro

Dependiendo del tipo de datos y del tipo de terminal se tendrán unas opciones u otras, las más importantes para cada tipo de datos son:

- En los controles de tipo booleano se tiene la opción de *Mechanical Action* que les permite actuar como pulsadores (*latch*) o interruptores (*switch*), ver figura 32.

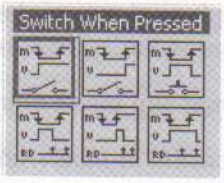


Figura 3-2. Interruptores y pulsadores

- Los numéricos pueden acotar el rango de entrada con *Data Range* y modificar varias opciones de visualización con *Format and Precision*.
- En los controles e indicadores de tipo *string* se puede ver el contenido de forma normal, representado por unos códigos (ver Tabla 1), como asteriscos o por su valor hexadecimal.

Junto a un terminal pueden aparecer varios ítems. En la figura 3-3 se muestra un control numérico digital con los siguientes ítems:

- *Label*: es un texto que da un nombre al terminal en el Panel Frontal, esta etiqueta será usada para identificar al elemento en variables, propiedades, etc.
- *Caption*: es otro texto asociado al terminal que sólo puede aparecer en el Panel Frontal.
- Incremento/decremento: en los terminales numéricos también se dispone de este elemento para aumentar o disminuir el valor del dato.
- *Radix*: indica el formato de visualización en los terminales numéricos enteros en decimal (d), octal (o), hexadecimal (x) y binario (b).
- Valor: es el valor que hay en el terminal representado en el formato elegido.
- Unidades: el tipo de datos numérico también puede tener un símbolo que represente sus unidades.

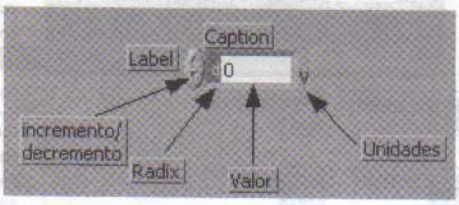


Figura 3-3. Ítems de un terminal

Para terminales de otro tipo habrá otras opciones, incluso para terminales con el mismo tipo de datos pero con diferente forma también cambiarán estas opciones. En la figura 34 pueden apreciarse tres controles numéricos con formas diferentes.

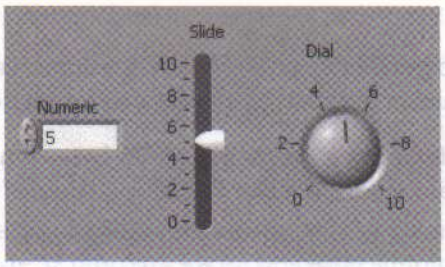


Figura 3-4. Controles numéricos con diferente formato

En el Diagrama de Bloques se representan los indicadores con una flecha en el lado izquierdo que apunta hacia el terminal simbolizando una entrada, y los controles con una flecha a la derecha apuntando hacia afuera. Estos terminales pueden verse como icono o con su aspecto clásico, como en se aprecia en la figura 35.

Figura 3-5. Símbolos de los terminales

La visualización con forma de icono permite ver la forma del terminal, lo que puede ayudar a identificarlo; por otra parte la vista clásica ocupa menos espacio y será la utilizada en el resto del libro. En *Tools > Options > Block Diagram* puede cambiarse la forma en que LabVIEW dibuja los terminales por defecto.

3.2 Tipos de datos

Una de las primeras cosas que se aprende en cualquier lenguaje de programación son los tipos de datos disponibles. No debe confundirse el tipo de datos con tipo de terminal. Cuando se habla de tipo de datos se hace referencia a si son numéricos, cadenas de caracteres, etc.

El tipo de datos se representa en el Diagrama de Bloques por el color del terminal y del cable, así un dato booleano tendrá terminales y cables verdes para diferenciarlo de un *string* que será rosa.

A continuación se estudiarán los distintos tipos de datos con sus características más destacables, transformaciones de unos a otros y la forma en que los distintos tipos de datos de LabVIEW se almacenan en memoria. Esto es algo a tener en cuenta en algunas ocasiones muy concretas, como pueden ser:

- Llamar a código externo.
- Trabajar con ficheros binarios.
- Otras funciones I/O.

3.2.1 Boolean

Los datos de este tipo sólo pueden tener dos posibles valores: verdadero (TRUE) o falso (FALSE). Debido a esto suelen usarse en controles con forma de botón o pulsador.

Cada dato booleano se almacena en memoria en un byte completo, si este byte tiene todos sus bits a cero, el dato tendrá el valor FALSE y cualquier otro valor del byte hará que el dato pase a TRUE.

Figura 3-6. Dato Booleano

En los controles booleanos se puede seleccionar su *Mechanical Action*, como mostraba la figura 3-2. Ya se dijo que los *switch* son equivalentes a interruptores y los *latch* a pulsadores. En los dibujos que representn cada acción hay unas letras a la izquierda: «m», «v» y «RD». La primera de ellas («m») simboliza la acción mecánica que hace el usuario sobre el control. La segunda («v») representa el cambio de estado que sufre el control. Finalmente la tercera («RD») sólo se puede ver en los de tipo *latch* y alude al momento en que, desde el Diagrama de Bloques, el valor del control es leído. Como se puede ver en los iconos tipo *latch*, el control tiene un "estado estable" y uno "aestable", cuando se activa el control se pasa del estado estable al aestable durante un tiempo, y vuelve al estable inmediatamente después de que el valor del control sea leído.

3.2.2 Numeric

Al trabajar con datos numéricos hay que distinguir entre números enteros, números racionales y complejos.

Los números enteros tienen asociado el color azul y puede elegirse su tamaño (8, 16, 32 o 64 bits), si se emplea un bit de signo (*signed* o *unsigned*) y su representación (binario, octal, decimal, hexadecimal).

Los controles de tipo *Enum* y *Ring* también son numéricos. Se trata de listas que asocian una cadena de caracteres a un valor numérico. Mediante su menú contextual se puede acceder a *Edit Item* para definir la lista de valores. La principal diferencia entre estos dos tipos es que en los *enum* el texto se considera también parte del control, por lo que para conectar un control o constate *enum* a un indicador *enum* se necesita que ambos tengan ítems con los mismos nombres, mientras que con los *ring* no es necesario.

En la figura 3-7 pueden verse dos constantes, el símbolo que aparece a su izquierda (una letra «d» o «x») es su representación, este ítem puede activarse con la opción *radix* en el menú contextual *Visible Items*; la primera constante es decimal y la segunda hexadecimal. La primera está conectada a un indicador que visualiza datos con signo de 32 bits (I32) mientras que la segunda a un indicador sin signo de 16 bits (U16).

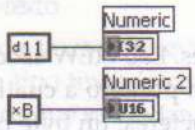


Figura 3-7. Constantes con diferente representación

Los números racionales y complejos tienen asociado el color naranja. Siguen el estándar IEEE; el tamaño es de 32 bits para los de precisión simple, 64 bits para los de doble precisión y el tamaño de los extendidos depende de la plataforma: 80 bits para Windows y Linux y para 64 MacOS. El primer bit siempre es el signo, los siguientes el exponente, que es de tamaño 7, 10 y 14 bits respectivamente y finalmente la mantisa. Los complejos simplemente son dos números racionales de uno de los tamaños anteriores.

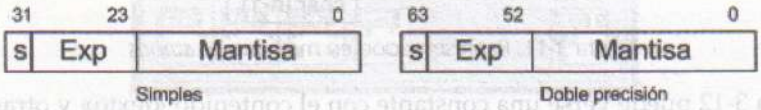


Figura 3-8. Representación en memoria de números en coma flotante

En la figura 3-9 puede verse un número complejo, dos constantes con los valores p y *e* y tres constantes que se conectan a tres indicadores, cada uno de la precisión indicada.

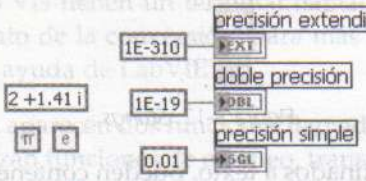


Figura 3-9. Diferentes tipos de valores numéricos

Las conversiones entre tipos de datos numéricos suelen ser automáticas, cuando se hace, se representa mediante un pequeño punto de color rojo (*coercion dot*) justo donde el cable se conecta al indicador, como se puede ver en el indicador de precisión simple de la figura anterior (en versiones anteriores a la 8.20 el color por defecto es gris, pero puede cambiarse en *Tools > Options > Colors*). También es interesante la opción de usar unidades junto con los números, así en la figura 3-10 se puede ver la resta de 1 Voltio y 1 mV, el resultado mostrado en el indicador será 0,999 V.

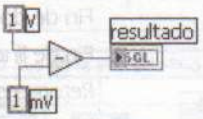


Figura 3-10. Resta de dos números utilizando unidades

Otro tipo de datos es el de tiempo (*Time Stamp*) que es un conjunto de cuatro enteros, los dos primeros indican el número de segundos desde la 1:00:00 del 1 de enero de 1904 y los otros dos las fracciones de segundo.

3.2.3 Strings

Los *strings* son cadenas de caracteres. LabVIEW asocia el color rosa a este tipo de datos. En memoria se almacenan como un puntero a cuatro bytes que indican el tamaño y a continuación los valores de los caracteres, un byte por cada uno, así el texto «hola» se almacenará como «0000 0004 686F 6C61», siendo 4 el tamaño y 68h, 6Fh, 6Ch y 61h los códigos ASCII de cada una de las letras del texto anterior.

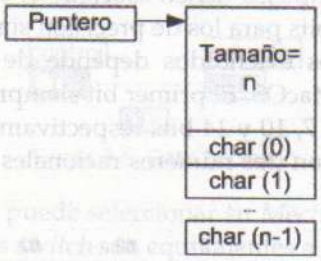


Figura 3-11. Representación en memoria de *strings*

En la figura 3-12 puede verse una constante con el contenido «texto» y otras dos constantes que representan un *string* vacío y un retorno de carro.

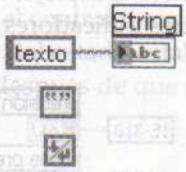


Figura 3-12. *Strings*

Aunque los *strings* están destinados a texto, pueden contener valores que no tienen un carácter de representación, esto puede comprobarse cambiando la forma de representación a hexadecimal. Al visualizar los códigos se verán los caracteres precedidos de una barra inversa, desde /00 a /FF; algunos códigos representan caracteres especiales, éstos se escriben siempre en minúsculas, son:

Tabla - Códigos especiales

Código	Descripción
\b	Retroceso (BS)
\f	Fin de documento (FF)
\n	Fin de línea (LF)
\r	Retorno de carro (CR)
\t	Tabulador
\s	Espacio
\\	Barra invertida
%%	Porcentaje

3.2.3.1 Transformaciones número-texto

Las transformaciones de números a texto y viceversa son muy usadas en LabVIEW. Muchos VIs trabajan con datos de tipo texto (*strings*) como por ejemplo los de GPIB, sin embargo puede ser que los datos que se envían o reciben sean números y necesiten convertirse para realizar algún cálculo con ellos.

En la paleta *Programming > String > String/Number Conversion* se encuentran varios VIs para realizar estas transformaciones.

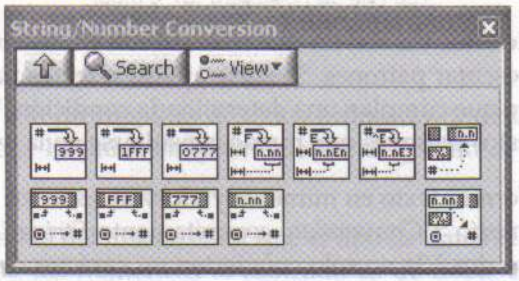


Figura 3-13. Paleta de conversión número/texto

En la primera fila se encuentran las funciones para pasar de número a texto, la segunda fila es para pasar de texto a número. El formato de los números puede ser entero o de coma flotante, decimal, hexadecimal u octal. Los dos VIs de la derecha son *Format Value* y *Scan Value*, estos VIs tienen un terminal llamado *format string* en el que se puede especificar el formato de la conversión; para más información sobre la sintaxis de este *string* consultar la ayuda de LabVIEW.

En el menú *string* también aparecen dos funciones llamadas *Scan From String* y *Format Into String*. Estos VIs realizan funciones de escaneo, transformaciones y concatenación. Básicamente son una herramienta configurable con la que transformar números y textos a otro formato. Todos los VIs de esta "familia" trabajan de forma parecida a la función *sprintf* del lenguaje C; tienen un terminal común, el *format string* que sirve para configurar esa transformación mediante unos códigos precedidos del símbolo «%», los códigos más habituales son «%s» para *strings*, «%f» para números en coma flotante, «%.xf» para indicar el número de decimales de los números flotantes, «%d» para enteros decimales, etc.

Por ejemplo, si se desea construir un texto que diga «He medido 3.64 voltios en la entrada 2.» Siendo 3,64 y 2 entradas numéricas se podría usar el código de la figura 3-14.

Figura 3-14. Ejemplo VI para el manejo de cadenas y texto

En el ejemplo anterior se usa un *string* inicial que contiene «He medido », también hay tres entradas más: un número en coma flotante, un *string* y un número entero. El *format string* se muestra en la parte superior, indica el formato de cada entrada mediante `%XY`, donde `X` es la longitud e `Y` el tipo de datos, además en este *string* también se puede incluir texto.

Para configurar este tipo de VIs se puede recurrir a la documentación para conocer la sintaxis del terminal *format string*. También se puede usar un asistente pinchando dos veces sobre el icono y rellenando los campos de forma correcta, ver figura 3-15.

Una generalización de estos VIs son los que trabajan con expresiones regulares, que son un método para representar patrones. Se suele aplicar a textos para encontrar y manipular cadenas de texto que cumplan una determinada condición. Usan un complejo sistema para codificar estos patrones que puede consultarse en la ayuda de LabVIEW.

Otros VIs para transformar texto en números y viceversa son los que se encuentran en *Programming > Numeric > Conversion*, concretamente *String To Byte Array* y *Byte Array To String*. Estos dos VIs no sólo convierten un carácter de texto a un número o viceversa, lo hacen con cadenas de caracteres, por lo tanto son idóneos para transmitir y recibir señales a través de interfaces como TCP o el puerto serie.

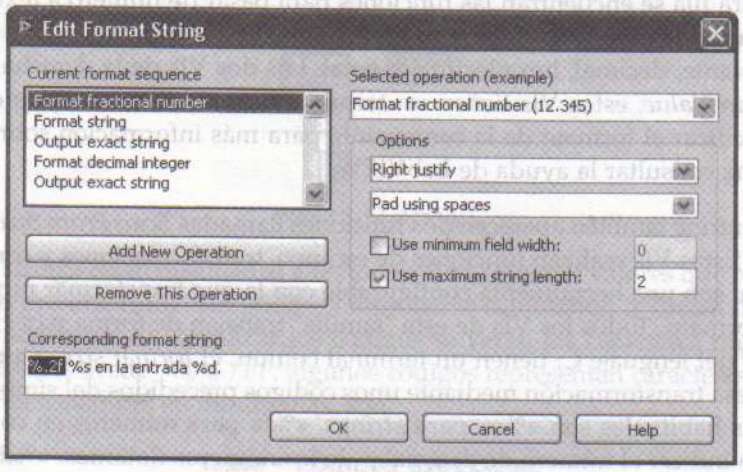


Figura 3-15. Asistente del *Format String*

3.2.4 Path

Los *path* sirven para indicar rutas relativas o absolutas a directorios o ficheros tanto de la máquina local como de otra en red.

Al guardarse en memoria se almacena un puntero a una estructura que se forma del número de partes que contiene la ruta (subdirectorios), después cada parte de la ruta se guarda como un *string*, de esta forma se consigue que la ruta sea independiente de la plataforma, es decir, la misma ruta podría ser entendida por LabVIEW funcionando sobre Windows, Unix o MacOS.

En la figura 3-16 se puede ver una constante que se conecta a un indicador y abajo dos nodos que representan a la ruta del VI que lo contiene y la ruta del directorio de datos de LabVIEW.

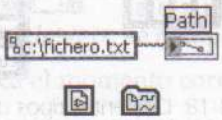


Figura 3-16. Utilización de un *Path*

3.2.5 Arrays

Los *arrays* o arreglos son un conjunto de datos ordenados y de un tipo determinado; de esta forma no puede hablarse simplemente de *array* sino de *array* de booleanos, *array* de *strings*, etc.

El cableado de un *array* es del mismo color que el tipo de datos que contiene, pero más grueso y en el caso de los numéricos el cableado es de doble línea para dos o más dimensiones. En el terminal del Diagrama de Bloques aparecerá el tipo de datos entre corchetes.

Junto a los terminales (control, indicador o constante) de cualquier tipo de *array* aparecen uno o varios índices numéricos que indican el primer elemento que se muestra (nótese que no afecta al contenido, sólo a la visualización, como un *scrollbar*). Estos índices también indican la dimensión del *array* de manera que si sólo hay uno será de una dimensión, si hay dos índices será de dos dimensiones y así sucesivamente.

Al guardarse en memoria se guarda un puntero a una estructura que consta, en primer lugar, del tamaño de cada dimensión en cuatro bytes por cada uno y a continuación van los datos guardados en el formato que se corresponda al tipo de datos.

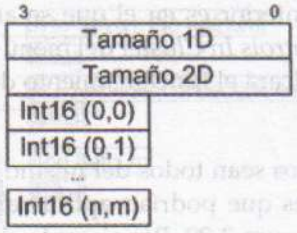


Figura 3-17. Representación en memoria de un *array* de dos dimensiones

En la figura 3-18 hay cuatro *arrays*, el primero es uno vacío, para asignarle un tipo hay que arrastrar una constante a su interior, debajo de éste hay un *array* numérico de una dimensión, el de la parte superior a la derecha es otro *array* numérico pero éste de dos dimensiones, el último es un *array* de *strings* que muestra los elementos a partir del segundo.

Al usar un *array* no hay que declarar el tamaño de cada dimensión como en otros lenguajes como C, el *array* puede re-dimensionarse cuando sea necesario.

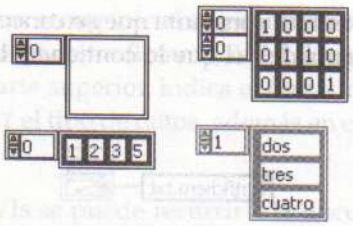


Figura 3-18. Diferentes tipos de *arrays*

3.2.6 Clusters

Al igual que los *arrays*, los *clusters* son un tipo de datos estructurado, también son un conjunto de datos, pero a diferencia de los anteriores, los *cluster* pueden contener datos de varios tipos en su interior, incluidos otros *clusters*. El concepto es equivalente a los STRUCT en C o a los RECORD en Pascal.

La figura 3-19 muestra un *cluster* vacío, otro compuesto de un entero y un *array* de *strings* y otro con un booleano, un entero y un *string*, este último es un tipo especial de *cluster* porque es el usado para describir errores (desde la versión 8.20 se muestra por defecto con un tono de color marrón).

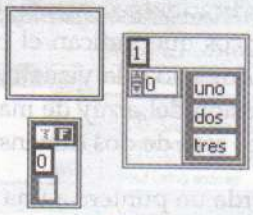


Figura 3-19. *Clusters*

Para crear un *cluster* se coloca uno vacío y se arrastran a su interior otros tipos de datos. El orden de los datos en su interior es en el que se añaden, aunque puede variarse yendo a la opción *Reorder Controls In Cluster* del menú contextual, en el capítulo sobre la creación del interfaz se explicará el funcionamiento de esta utilidad cuando se hable sobre el orden de tabulación.

Si se crea un *cluster* cuyos datos sean todos del mismo tipo, podrán aplicarse sobre el *cluster* las mismas operaciones que podrían aplicarse sobre los elementos de forma individual, como muestra la figura 3-20. Por ejemplo si se crea un *cluster* cuyos datos sean numéricos se podrá sumar a otro número.

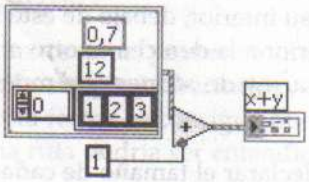


Figura 3-20. Suma entre un entero y *Cluster* con los datos numéricos

Los *clusters* son recomendables cuando en un subVI hay una gran cantidad de conexiones, empleándolos se pueden agrupar y hacer la conexión más sencilla.

3.2.7 Waveforms

Los *waveforms* son parecidos a los *clusters*, se componen de otros datos, éstos son:

- t0: es un *time stamp* que indica el momento correspondiente al primer valor.
- dt: es la diferencia de tiempo entre muestras, se supone que ésta no varía.
- Y: un *array* que contiene los valores del eje vertical.
- *attributes*: otra información que puede ser añadida.

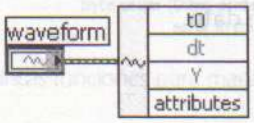


Figura 3-21. *Waveforms*

3.2.8 Referencias

Las referencias (*refnum*) o manejadores se usan para simbolizar un objeto externo al programa, éste puede ser el contenido de un fichero (no confundir con *path*), una conexión TCP, otro VI, etc.

Normalmente lo primero que se hace con las referencias es abrir una conexión y luego se irá pasando secuencialmente la referencia entre los VIs, nodos de propiedades o métodos que trabajan con ese objeto.

En memoria se almacenan como números de 32 bits.

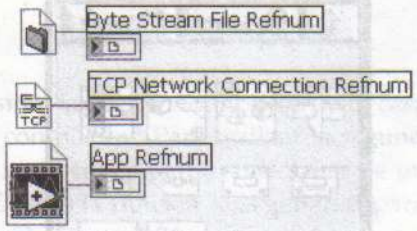


Figura 3-22. Algunos tipos de referencias

3.2.9 Variant

Los *variant* son un tanto especiales, no se trata de un tipo de datos concreto y además pueden incluir propiedades. Se usan cuando los datos pueden cambiar de tipo, por ejemplo, un dato que unas veces pueda ser *numeric* y otras *string*. También se usan al trabajar con objetos ActiveX.

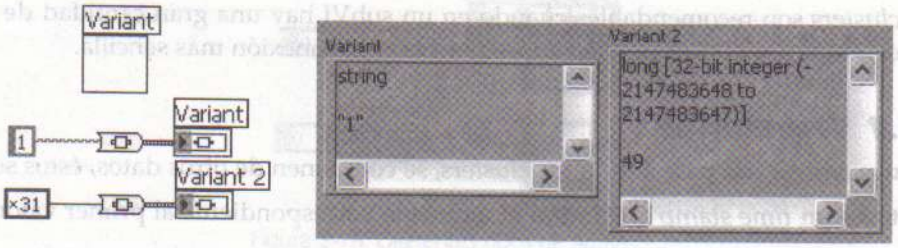


Figura 3-22. *Variant*

Además del dato, contienen el tipo, como puede verse en la figura anterior, para esto hay que elegir la opción *Show Type* en el menú contextual del indicador.

En memoria se almacena como un puntero a una estructura. La estructura reserva cuatro bytes para codificar el tipo de dato.

3.2.10 Dynamic

Este tipo de datos se usa con los VIs Express. Incluye datos y atributos asociados con una señal, como el nombre, fecha, etc. Los tipos de datos *Dynamic*, al igual que los *waveform*, enfatizan el hecho que los datos son medidas y no sólo números en una columna.

3.3 Manipulación de datos

3.3.1 Manipulación de bytes y bits

Las funciones más sencillas para manipular datos son las que trabajan con bytes y bits. Éstas se encuentran en la paleta *Programming > Numeric > Data Manipulation*.

Figura 3-24. Paleta para manipulación de datos

En la segunda fila de este menú están los VIs para realizar rotaciones y desplazamientos. La tercera fila tiene VIs para dividir y unir bytes en words y viceversa además de intercambiar la parte alta y baja de un entero. Mientas que los de la primera fila se verán a continuación.

3.3.2 Otras transformaciones

En la figura 3-25 pueden verse las funciones *Flatten To String* y *Unflatten From String*. El primero, *Flatten To String* (traducido como aplanar) es una función que convierte todo el bloque de memoria que ocupa el dato a un *string*. Se puede elegir que la información estructural también se tenga en cuenta con el terminal *prepend array or string size?* y el formato de los datos con *byte order* (*big-endian, little-endian* y el nativo de la plataforma). La entrada *anything* puede ser cualquier tipo de datos. *Unflatten From String* realiza el proceso inverso. En el menú contextual se puede indicar que manejen los datos como lo hacía LabVIEW en otras versiones.

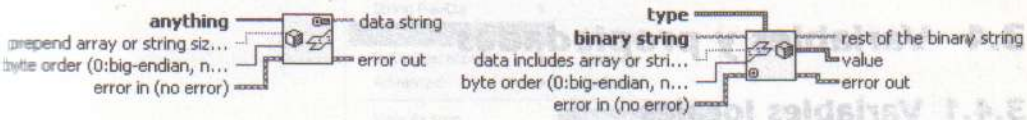


Figura 3-25. Distintas funciones para manipulación de datos

A diferencia de los *variant*, cuando se aplana un dato se pierde información sobre el tipo de datos origen, no tienen formato, por lo tanto al des-aplanar es necesario indicar el tipo de datos destino.

Por ejemplo, un *array* de números de 8 bits se almacena como una cabecera consistente en un número de 32 bits que indica su longitud y tantos bytes como elementos tenga el *array*. Al aplanar este *array*, si no se indica nada en el terminal *prepend array or string size?* se considera todo como parte de los datos y resultando un *string* cuyo valor es el resultado de anexar la cabecera y los datos e interpretarlo como texto. Si este *array* fuese de dos elementos con los valores 1 y 2, el resultado es un *string* cuyo valor hexadecimal es «0000 0002 0102».

Algunas funciones realizan el aplanado de datos automáticamente. También puede ser necesario al trabajar con librerías externas.

3.3.3 Type Cast

Hay dos tipos de conversión: la *coercion* (implícita) y la *cast* (explícita). La primera la hace automáticamente el compilador. Para realizar la segunda hay que utilizar alguna instrucción, antes se han visto varias para transformar de un tipo concreto a otro tipo también concreto. *Type Cast* es la función más genérica para transformar datos de un tipo a otro, viene dada por la figura 3-26.

El terminal *x* es el dato a convertir y *type* el tipo al que será convertido, por defecto lo considera *string*, pero se pueden conectar otros. Esta función no realiza ningún tipo de comprobación, así que al usarla hay que tener en cuenta que las entradas *type* y *x* sean compatibles, sino el resultado puede no ser el esperado.

Por ejemplo, este VI podría sustituir a *String To Byte Array* si en el terminal *type* se conecta un *array* de enteros de ocho bits.

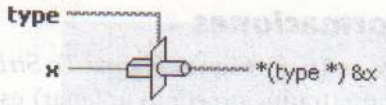


Figura 3-26. Función *Type*

En el menú contextual también tiene la opción de manejar los datos como se hacía en la versión 4 de LabVIEW, esto afecta únicamente a los datos booleanos porque en esta versión los guardaba ocupando únicamente un bit y desde la versión 5 ocupan un byte completo.

3.4 Variables y propiedades

3.4.1 Variables locales

Para llevar un dato desde/hacia un control/indicador se han usado cables. A veces no se puede realizar el cableado, como en bucles funcionando en paralelo y otras veces sencillamente se quiere escribir en un control o leer de un indicador. Para todas estas situaciones sirven las variables locales.

Las variables locales están asociadas a los controles e indicadores presentes en el Panel Frontal de un VI. Mediante ellas se puede leer y escribir el valor de ese control o indicador.

Para crear una variable local se puede ir a la paleta *Programming > Structures > Local Variable*, después asociar esa variable a uno de los controles o indicadores que existen mediante el menú contextual > *Select Item*, ver figura 3-27.

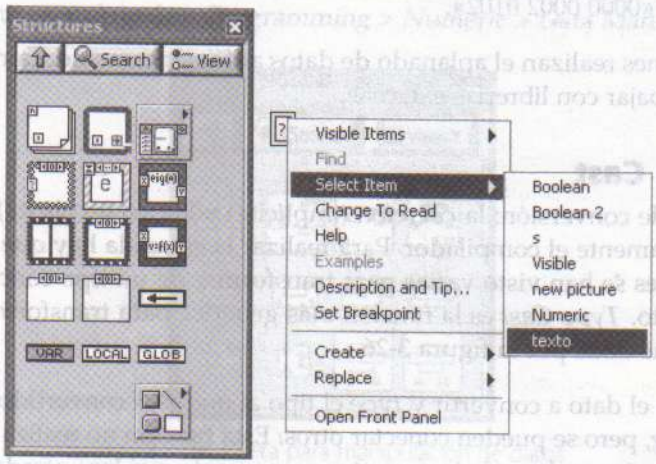


Figura 3-27. Variables locales

También pueden crearse directamente a partir del terminal, para ello en el Diagrama de Bloques se despliega el menú contextual y se selecciona *Create > Local Variable*, de la forma que indica la figura 3-28.

Una vez creada la variable se puede indicar si se usará como lectura o escritura (por defecto es escritura) con *Change To Read* o *Change To Write*. Que la variable local sea de lectura o escritura es independiente de si el terminal es un control o un indicador.

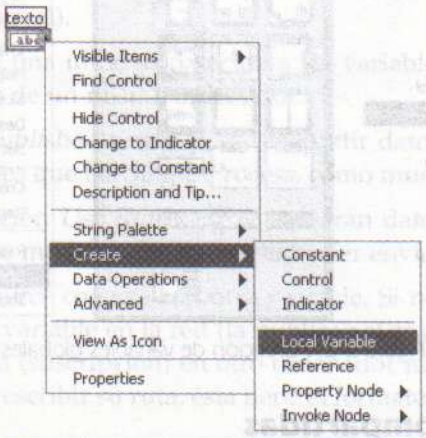


Figura 3-28. Creación de variables locales desde el propio terminal

Las variables locales tienen un alcance que comprende al VI en el que está colocado el control o indicador al que hacen referencia. No pueden usarse fuera de ese VI.

Como conclusión hay que decir que en general es preferible cablear las señales porque sigue el modelo de *Dataflow*, las variables locales sólo deben usarse cuando sean necesarias porque pueden aparecer problemas de no inicialización de datos, condiciones de carrera, etc.

3.4.2 Variables globales

Las variables globales se usan para compartir información entre distintos VIs ejecutándose en la misma máquina.

Pueden crearse también desde la paleta de estructuras, arrastrándolo al Diagrama de Bloques y haciendo doble clic o también desde *File > New... > Global Variable*.

Conceptualmente son como un VI normal pero sin código. Una vez abierta la variable global hay que colocar en su Panel Frontal un control o indicador del tipo que se desee. La información se almacenaría en este nuevo «VI» y podría ser usada en aquellos otros VIs que hagan referencia a la variable, para usar la variable global se aplica el mismo método que para cualquier subVI.

Si se añaden varios controles o indicadores a la variable global podrá elegirse en concreto cual quiere usar mediante el menú contextual eligiendo la opción *Select Item*. Al igual que las variables locales, también debe indicarse si se va a leer o escribir.

Como se ha dicho antes, el alcance de una variable global llega a cualquier VI. Cuando se use en varios VIs a la vez hay que prestar atención porque pueden aparecer problemas de condición de carrera, para más información se puede consultar el capítulo 10.

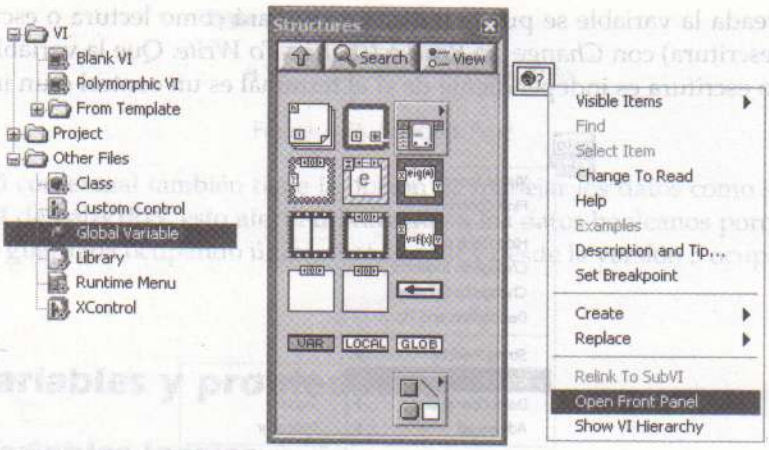


Figura 3-29. Creación de variables globales

3.4.3 Variables compartidas

Las variables compartidas (*Shared Variable*) son un nuevo elemento introducido en la versión 8.0 de LabVIEW. Son otro mecanismo para compartir información entre VIs de un mismo ordenador, entre ordenadores diferentes conectados en red o entre programas ejecutándose en hardware diferente (por ejemplo un dispositivo PAC (*Programmable Automation Controller*) y un PC); junto con las estructuras TIMED son la base de la programación en tiempo real.

Las variables compartidas se crean dentro de una librería de un proyecto como muestra la figura 3-30. Para configurar este tipo de variables hay que entrar en sus propiedades.

Figura 3-30. Creación de variables compartidas

Para usarlas sólo hay que arrastrarlas sobre un Diagrama de Bloques y configurarlas como lectura o escritura. Tienen tres terminales: el valor de la variable, un terminal de error y, opcionalmente, un *time stamp*. También puede asociarse una variable a un terminal del Panel Frontal, para esto se puede o bien arrastrar sobre él la variable compartida o bien entrar en las propiedades del terminal e ir a la pestaña *data binding*.

En la configuración de la variable se le debe asignar un nombre, tipo de datos y tipo de variable.

Hay tres tipos de variables compartidas: *Single-Process, Network-Published* y *Time Trigger* (sólo para tiempo real).

Las *Single-Process* tienen una utilidad parecida a las variables globales, se usan para compartir datos entre VIs de un mismo ordenador.

Las variables *Network-Published* sirven para compartir datos entre distintos ordenadores. Tienen más opciones que las *Single-Process*, como muestra la figura 331.

- Si no se activa la opción *Use Buffering* se enviarán datos únicamente cuando el valor de la variable se modifica, activando el buffer envía continuamente datos.
- La casilla *Bind to Source* conecta con otra variable. Si no se activa, el ordenador local da a conocer la variable en la red (la publica); si se activa podrá elegirse una variable ya publicada (suscripción) en otro ordenador mediante *Browse*, como se ve en la figura 3-32 o escribir su ruta, ésta tiene el formato `\\ordenador\libre-ría\variable`.

Si se instalan los módulos DSC o Real Time se tendrán muchas más opciones, como por ejemplo, alarmas, escalado, seguridad, etc.

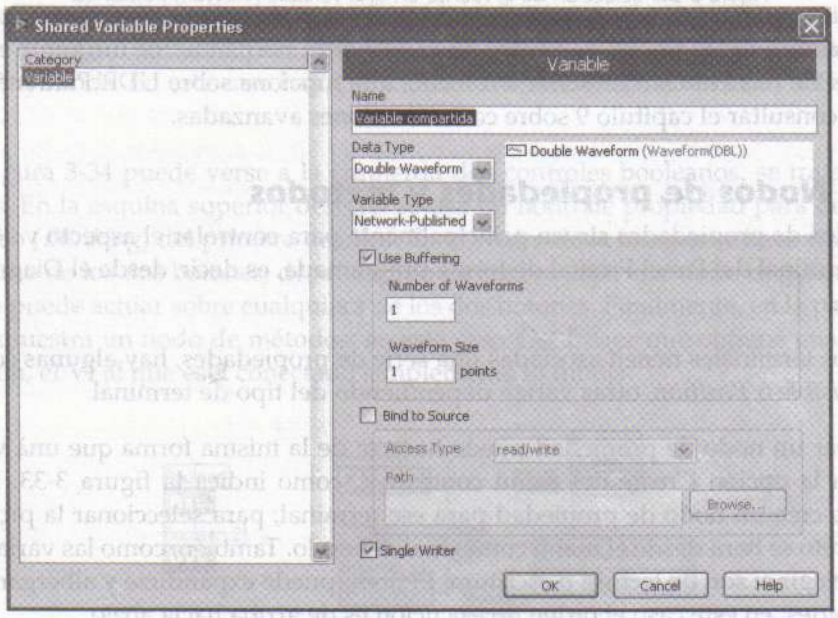


Figura 3-31. Propiedades de una variable compartida

El funcionamiento de las variables compartidas en red es el siguiente: LabVIEW configura el *shared variable engine* (SVE) como un servicio (sólo en Windows). SVE es un *framework* que permite publicar variables en la red. Se encarga de enviar los valores de estas variables a todos los suscriptores que necesiten leerla y recibe los datos de los suscriptores que escriben sobre la variable. Debe haber por lo menos un SVE en la red.

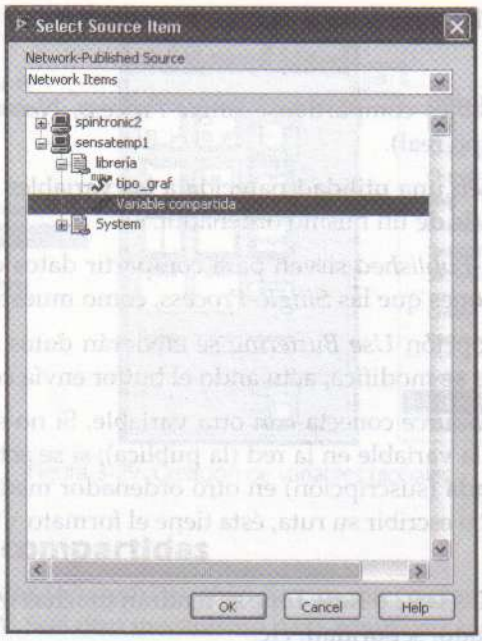


Figura 3-32. Selección de la fuente en una variable Network-Published

La transferencia de información entre *hosts* se realiza normalmente mediante el protocolo NI-PSP (*NI Publish-Subscribe Protocol*), que funciona sobre UDP. Para más información consultar el capítulo 9 sobre comunicaciones avanzadas.

3.4.4 Nodos de propiedades y métodos

Los nodos de propiedades sirven principalmente para controlar el aspecto y el estado de un terminal del Panel Frontal de forma programada, es decir, desde el Diagrama de Bloques.

Todos los terminales tienen asociadas una serie de propiedades, hay algunas comunes como *Visible* o *Position*, otras varían dependiendo del tipo de terminal.

Para crear un nodo de propiedad puede hacerse de la misma forma que una variable local: en la opción *Create* del menú contextual, como indica la figura 3-33. De esta forma se crea un nodo de propiedad para ese terminal; para seleccionar la propiedad en concreto se hará desde el menú contextual del nodo. También, como las variables, se deberá elegir si son de lectura o escritura. El nodo puede expandirse y albergar a otras propiedades, en este caso el orden de ejecución es de arriba hacia abajo.

Hay cientos de propiedades, así que no se pueden explicar todas en este libro, pero en la ayuda contextual se ofrece una breve descripción de cada propiedad. Algunas de las más usadas son: *Visible, Position, Disabled* y *Bounds*. La propiedad *Value* debe evitarse y sustituirla por una variable local (o mejor aún, un cable) cuando sea posible, pues los nodos de propiedad no son tan eficientes como una simple variable.

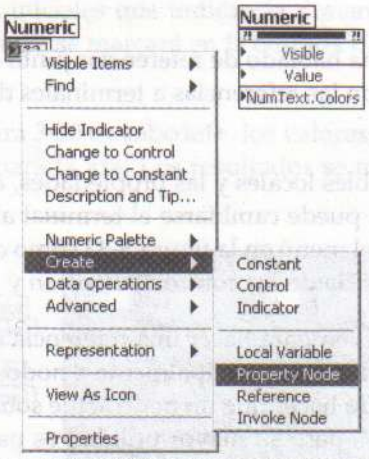


Figura 3-33. Creación de un nodo de propiedad

En el menú de la figura anterior también puede verse *Invoke Node*, esto sirve para crear un nodo muy parecido al de propiedades, en este caso de métodos. Estos nodos no están relacionados con el aspecto sino que efectúan alguna acción, como por ejemplo reiniciar a los valores por defecto.

Estos nodos también se pueden usar de forma genérica, es decir, que a priori no esté asociado a ningún terminal, para ello debe usarse *Programming > Application Control > Property Node*. En este caso hay que cablear una referencia al terminal sobre el que actuará.

En la figura 3-34 puede verse a la izquierda dos controles booleanos, se trata de dos botones. En la esquina superior derecha se usa un nodo de propiedad para detener el parpadeo (*Blinking*) del primero de ellos. En el centro de la imagen se hace visible o se oculta uno de los dos botones, dependiendo de cual se elija, este nodo es genérico, por lo tanto puede actuar sobre cualquiera de los dos botones. Finalmente, en la parte inferior se muestra un nodo de métodos, en este caso *Get Image* que obtiene una imagen del botón, el VI al que está conectado simplemente lo dibuja.

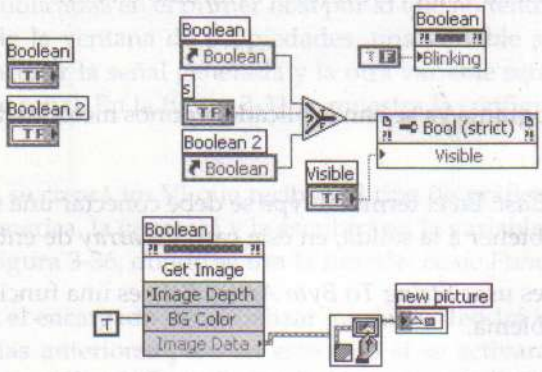


Figura 3-34. Ejemplo del uso de nodos de propiedades y métodos

3.4.5 Referencias

En una sección anterior se ha hablado de referencias junto con el tipo de datos *refnum*, esto no debe confundirse con las referencias a terminales del Panel Frontal, que será lo que estudie a continuación.

Se crean igual que las variables locales y las propiedades, en el menú *Create* del menú contextual. Una vez creada puede cambiarse el terminal al que está asociada desde el menú *Link to*. Puede verse el menú en la figura 3-33 como crear referencias y en la figura 3-34 dos nodos de referencia de los controles *Boolean* y *Boolean 2*.

Como su nombre indica, sirven para hacer una referencia al terminal al que están asociados. Estas referencias se aplican principalmente a nodos de métodos o propiedades genéricos. Pueden servir para hacer que un nodo actúe sobre uno u otro terminal como se muestra en la figura 3-34, pero su mayor utilidad es para poder usar los nodos de propiedades y métodos asociados a un terminal desde un VI distinto al que están colocados, las referencias no tienen alcance como las variables locales, pueden pasarse a otro VI como un parámetro más.

3.5 Ejemplos

3.5.1 Ejemplo I: Varios métodos para convertir la representación ASCII a su valor

3.5.1.1 Explicación teórica

Este ejemplo se explica perfectamente con el título, no necesita más descripción, aquí simplemente se justifica el motivo de incluirlo: a muchos usuarios noveles que trabajan con interfaces como el bus serie RS-232 se les presenta el problema de que lo enviado y transmitido es un *string*, sin embargo deben interpretarlo como números; en principio se puede pensar en usar las funciones de *Programming > String > String/Number Conversion*, pero éstas no valen porque lo que convierten es un carácter numérico al número que representa o viceversa, pero no pueden convertir un carácter a su valor ASCII, por ejemplo en un *string* el carácter «1» (cuyo valor ASCII en hexadecimal es el 31) no se convertiría a su valor ASCII (49 = 31h), sino en el 1.

3.5.1.2 Código

Para realizar este programa ya se han explicado algunos métodos, aquí se recopilan tres de ellos.

El primero es *Type Cast*. En el terminal *type* se debe conectar una señal del mismo tipo de la que se desea obtener a la salida, en este caso un *array* de enteros de ocho bits.

El segundo método es usar *String To Byte Array*. Esta es una función pensada expresamente para este problema.

El último método recopilado es usar *Unflatten From String*. Este método aprovecha que el dato de entrada es un *string*, sin embargo para transformarse al *array* de enteros de

salida necesitaría los 32 bits iniciales que indican la dimensión de esa dimensión del *array*. Como el *string* no lo tiene se marcará en la entrada *data includes array or string size?* como FALSE.

El *string* mostrado en la figura 3-35 es «abcdef», los valores de estos caracteres van del 97 para la «a» hasta el 102 para la «f». Los resultados se muestran a la derecha de la imagen.

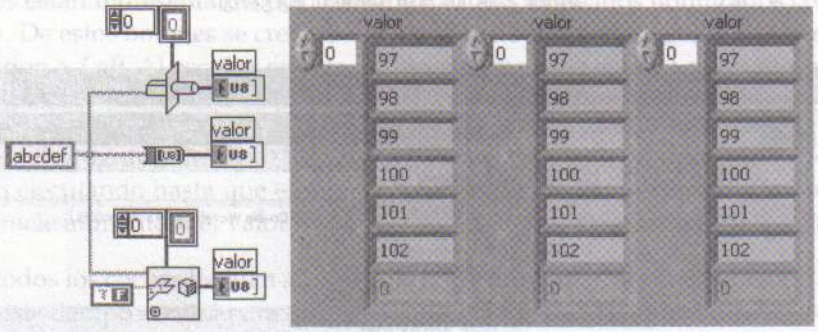


Figura 3-35. VI que pasa un ASCII a su valor

3.5.2 Ejemplo II: Sistema distribuido con variables compartidas

3.5.2.1 Explicación teórica

En este ejemplo se mostrará el uso de las variables compartidas en red, ya que es su mayor utilidad. El objetivo es crear un sistema distribuido, habrá un *host* que genera los datos y otro que los visualiza y configura al primero.

3.5.2.2 Código

En primer lugar se deben crear sendos proyectos en los dos *hosts* y una librería con dos variables compartidas.

Las variables serán publicadas en el primer *host* por lo que no tendrán activada la casilla *Bind To Source* de la ventana de propiedades, una variable será de tipo *Double Waveform* para almacenar la señal generada y la otra variable será un número entero para definir el tipo de señal. En la figura 3-31 se muestra la configuración de la segunda variable.

En este *host* también se creará un VI que recibirá el tipo de gráfica que se desea a través de la variable numérica, la generará y la escribirá en la variable *waveform*. El código se muestra en la figura 3-36, donde se usa la función *Basic Function Generator*.

El segundo *host* será el encargado de visualizar los datos. Tendrá un proyecto con dos variables igual que las anteriores pero en este caso sí se activará la casilla *Bind To Source* para poder seleccionar las variables del primer *host*.

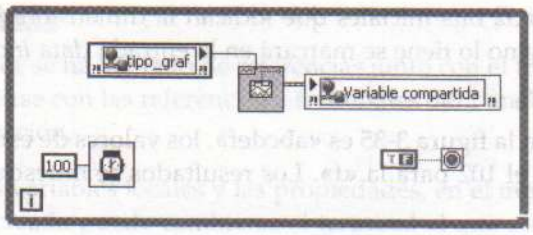


Figura 3-36. VI que genera una señal

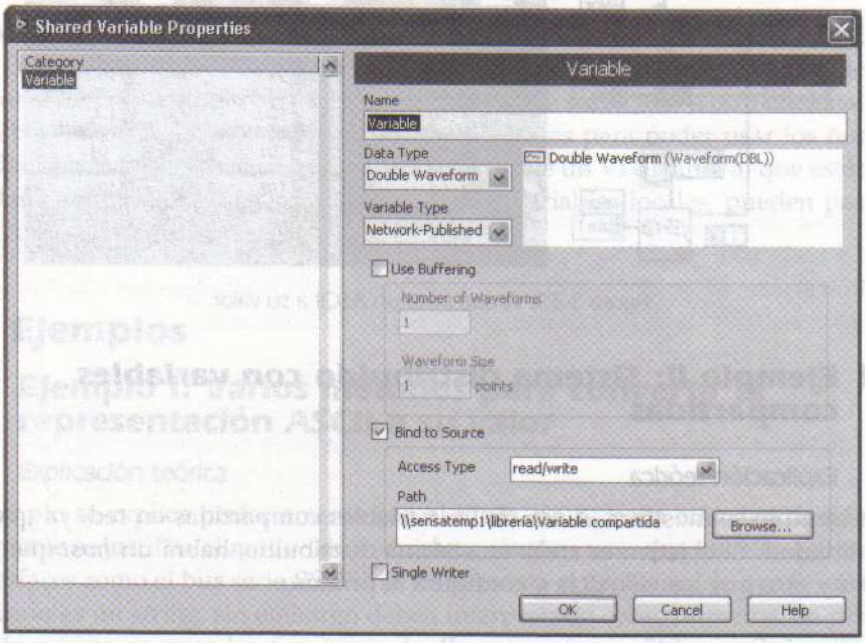


Figura 3-37. Configuración de la variable

El código del VI de este *host* puede ser tan simple como el mostrado en la figura 3-38.

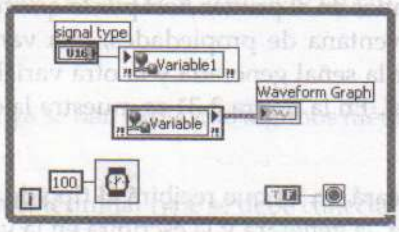


Figura 3-38. VI que controla y muestra la señal generada por un VI en otro ordenador

3.5.3 Ejemplo III: Carrera de fórmula 1

3.5.3.1 Explicación teórica

El objetivo de este ejemplo será crear un sencillo simulador de carreras con tres coches. Los coches avanzarán en línea recta desde la salida hasta la meta. Cuando lleguen, el primero obtendrá una medalla de oro, el segundo de plata y el tercero de bronce.

Los coches están representados por unos indicadores booleanos nombrados como «1», «2» y «3». De estos botones se crearán unos nodos de propiedad en los que se seleccionará *Position > Left*. Al escribir en este nodo un número, el control se desplazará hasta esa coordenada.

El núcleo del programa consistirá en tres bucles en paralelo, uno por cada coche, que se estarán ejecutando hasta que el coche correspondiente llegue a meta. En cada iteración del bucle aumentará el valor de la posición un valor aleatorio entre el 1 y el 4.

Cuando todos los coches lleguen a la meta se construirá un *array* con el tiempo que han tardado, este tiempo servirá para que el color del fondo de la etiqueta cambie a uno que represente una de las medallas. En esta ocasión se usará un nodo de propiedad genérico y se elegirá el control a colorear seleccionando su referencia.

3.5.3.2 Código

El código para este programa se puede ver en la figura 3-39.

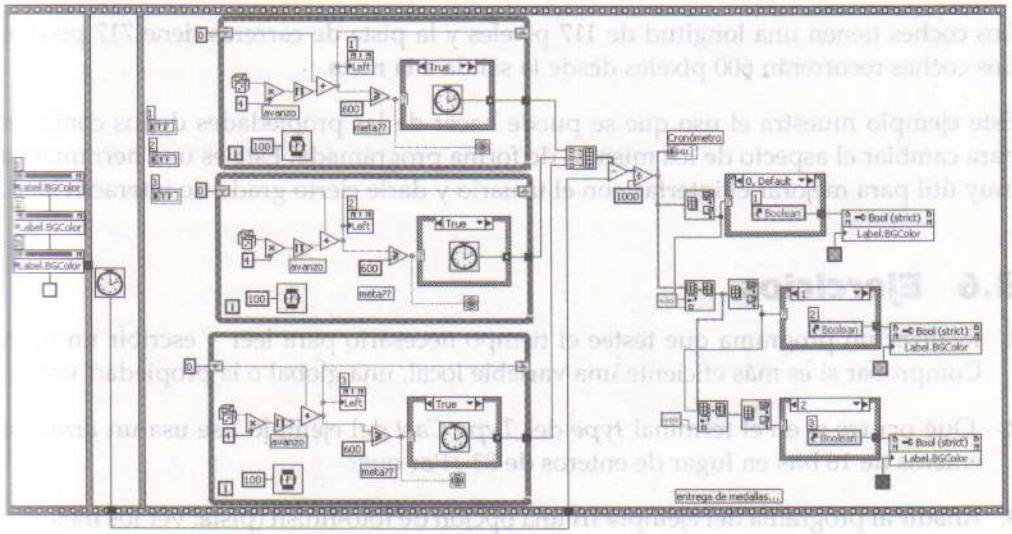


Figura 3-39. Código del simulador de carreras

Consiste en un SEQUENCE, en el primer *frame* se resetean los colores de las etiquetas y en el segundo se obtiene el tiempo en la salida, el tercer *frame* contiene tres bucles WHILE que hacen avanzar los controles, cuando estos bucles acaban también se obtie-

ne el tiempo, que se restará con el anterior para calcular el tiempo invertido en el recorrido. Con los tiempos de los tres controles se crea un *array* y en él se busca el menor valor que corresponderá con el primer control en acabar el recorrido, la referencia de ese control se pasará al nodo que cambia el color de la etiqueta a dorado. El valor en el *array* se sustituirá por infinito y se procederá de igual forma para obtener el segundo y el tercer clasificado.

3.5.3.3 Resultados

En la figura 3-40 se muestra una captura de una ejecución del programa. Los controles booleanos han sido modificados para que tengan el aspecto de un coche. El número que aparece sobre el capó es su etiqueta.

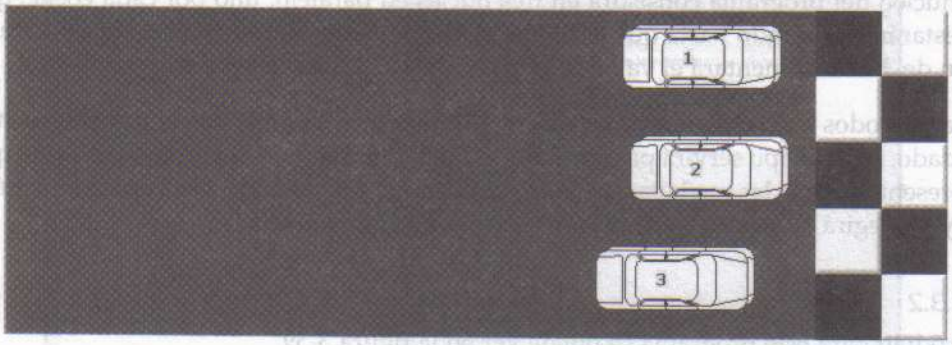


Figura 3-40. El coche nº 2 va en cabeza

Los coches tienen una longitud de 117 píxeles y la pista de carreras tiene 717 píxeles. Los coches recorrerán 600 píxeles desde la salida a la meta.

Este ejemplo muestra el uso que se puede hacer de las propiedades de los controles para cambiar el aspecto de los mismos de forma programada. Ésta es una herramienta muy útil para mejorar el interfaz con el usuario y darle cierto grado de interactividad.

3.6 Ejercicios

1. Realizar un programa que testee el tiempo necesario para leer y escribir un dato. Comprobar si es más eficiente una variable local, una global o la propiedad *value*.

2. Qué ocurre si en el terminal *type* del *Type Cast* del ejemplo I se usa un *array* de enteros de 16 bits en lugar de enteros de 8? ¿Por qué?

3. Añadir al programa del ejemplo III una opción de foto-finish (pista: ver los métodos asociados al VI).

4. Construye un VI que acepte como parámetro de entrada un dato que pueda ser de tipo numérico, booleano, *array* o *cluster* y cuya salida sea un *array* de *strings* en el que cada elemento tenga el valor de cada componente del dato de entrada. Por ejemplo, si la entrada es un *array* de booleanos, la salida será un *array* de *strings* con

los valores 0 o 1; si la entrada es un *cluster* de un dato numérico y un *string* la salida será un *array* en el que el primer elemento sea el valor del dato numérico y el segundo el del *string*; si la entrada es un booleano o un numérico, el *array* de salida lógicamente sólo tendrá un elemento del mismo valor que la entrada (Ayuda: ver en el tema 4 las funciones sobre XML).

3.7 Bibliografía

Archana Shrotriya, *AN 087: Writing Win32 Dynamic Link Libraries (DLL) and Calling Them from LabVIEW.*

Jean Pierre Drolet y Jim Kring, *LabVIEW Data, Data Types and Storage Formats*, 2002.

National Instruments, *AN 154: LabVIEW Data Storage.*

National Instruments, *AN 168: LabVIEW Performance and Memory Management*, 2004.

Sección II

ADQUISICIÓN Y COMUNICACIONES

Capítulo 4. Manejo de Ficheros.

En este capítulo se tratan las diferentes funciones para manejar ficheros mediante ejemplos. A parte de su utilidad también se asentarán los conceptos expuestos en la sección I. Se finaliza con ejercicios propuestos.

Capítulo 5. Comunicación serie.

La comunicación serie no se debe olvidar ya que está presente en multitud de dispositivos. En este capítulo se presentan dos ejemplos prácticos en los que se puede ver la facilidad de implementar dicho tipo de comunicación en el entorno de LabVIEW.

Capítulo 6. Bus de comunicaciones GPIB.

En sus orígenes, LabVIEW se aplicaba fundamentalmente en el control de instrumentación y para tal fin se utilizaban diferentes buses de instrumentación, en este caso cabe tratar el bus GPIB. El capítulo comienza con una introducción teórica al estándar IEEE 488.1, al IEEE 488.2 y a la norma SCPI. A continuación se describen las funciones para utilizar dicho estándar. De manera sencilla se ve su uso y los comandos básicos con unos ejemplos prácticos. Para finalizar el capítulo se proponen unos ejercicios.

Capítulo 7. Adquisición de datos.

En este capítulo se trata uno de los principales usos de LabVIEW: la adquisición y generación de señales eléctricas a través de tarjetas de adquisición de datos. Hay varios tipos de sistemas de adquisición de datos: los Data Loggers, las tarjetas DAQ con interfaces PCI, PCI Express, PCI..., y externas como USB o RS-232. Se describen las capacidades comunes a todas ellas así como las funciones que emplea LabVIEW para manejarlas.

Además se presentará el programa Measurement & Automation (MAX). A continuación se muestran ejemplos prácticos para asentar los principales conceptos y se proponen ejercicios.

Capítulo 8. Protocolos de comunicación: TCP y UDP.

La importancia que tiene hoy día internet justifica que se dedique un tema completo a estudiar sus principales protocolos y la forma en que se pueden manejar desde LabVIEW.

Capítulo 9. Acceso remoto: VI Server y Comunicaciones Avanzadas.

Este capítulo es una continuación del anterior, en él se explica cómo usar el servidor web que incluye LabVIEW, cómo manejar un programa residente en un ordenador desde otro y otros protocolos de nivel superior a los del capítulo 8. En la segunda parte del capítulo se hablará sobre VI Server y su utilidad.

Capítulo 4

Manejo de ficheros

4.1 Ficheros de texto

Las funciones de manejo de ficheros tradicionalmente suelen ser unas de las que primero se explican cuando se enseña LabVIEW, esto se debe a que emplean un método que siguen muchas otras funciones: abrir, leer-escribir y cerrar. Además de su utilidad pedagógica también tienen utilidad práctica, y mucha.

Hay gran cantidad de funciones para manejar ficheros, agrupadas en varios menús, en este capítulo se tratará de reunir a la mayoría. En primer lugar se estudiarán las más genéricas que permiten manipular ficheros de texto, después se verán, en su sección correspondiente, el resto de funciones para un tipo de fichero concreto, un tipo de dato específico o simplemente para ayudar a depurar un programa.

El menú principal se encuentra en *Programming > File I/O*.

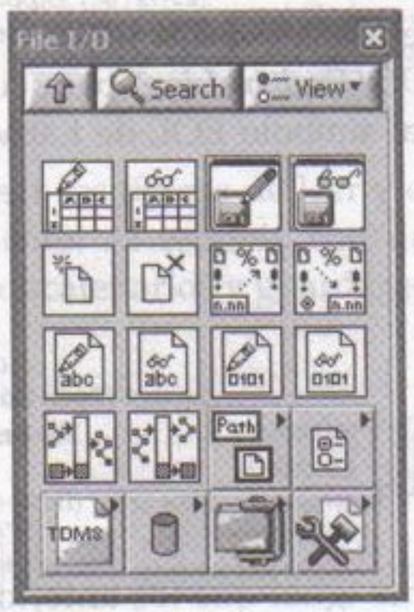
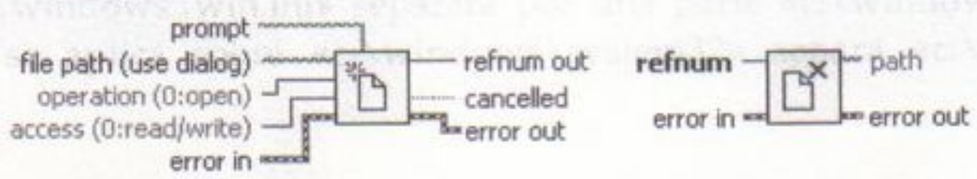


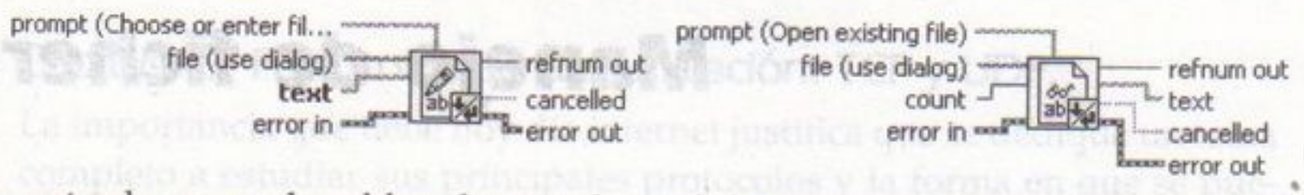
Figura 4-1. Paleta de ficheros

Abrir y cerrar



Al abrir un fichero se devuelve una referencia o manejador que sirve para representarlo y trabajar con él. Si no se indica de forma explícita el fichero aparecerá una ventana pidiendo su ruta. Al abrir el fichero también se puede indicar la operación (abrir, crear o reemplazar) y el modo de acceso (lectura, escritura). Cerrar es la operación contraria, libera el fichero.

Leer y escribir

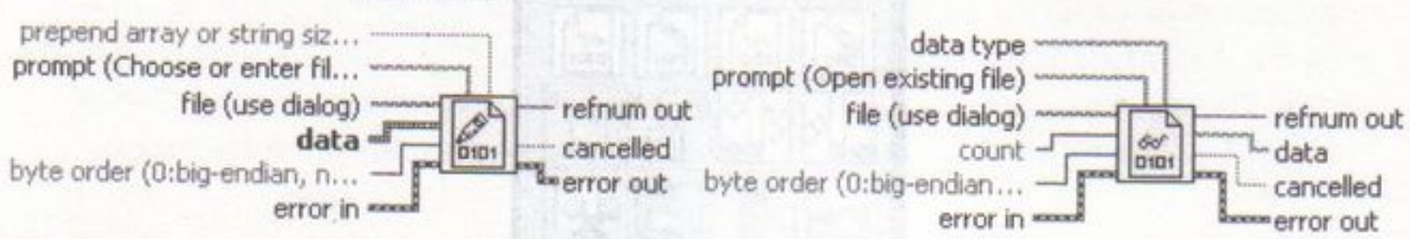


Como entrada se puede cablear la ruta de un fichero o una referencia, dependiendo de cual sea, estos VIs actuarán de forma diferente. Si en la entrada hay una ruta (*path*), estas funciones internamente abren el fichero; al escribir se hace sobre el fichero completo, por lo tanto sustituirá el contenido anterior del fichero. Si no se indica la ruta también la pide, como los VIs anteriores. En cambio si se emplea una referencia (*refnum*) la escritura se realizará a partir de una posición determinada, esta posición es un puntero al fichero que irá variando dependiendo de las operaciones que se hagan, por ejemplo, si se abre el fichero la posición será 0 (el principio del fichero), si se escriben cuatro caracteres la posición aumentará en cuatro; en el submenú avanzado hay funciones para variar la posición.

Cuando se lee se hará a partir de la posición que hubiera antes si se conecta una referencia, o desde el principio si es un *path*. Se leerán los bytes indicados en *count* o hasta el final del fichero si no se indica nada.

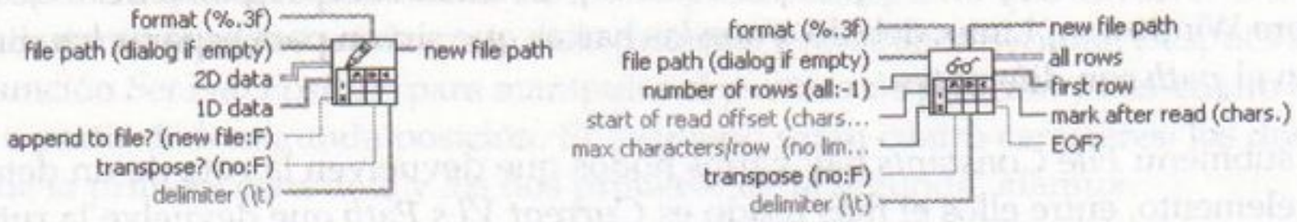
El cuadrado que puede aparecer en la esquina inferior izquierda significa que se tiene activada la opción *Convert EOL* (*End of Line*) para convertir los símbolos de fin de línea de cualquier plataforma al símbolo EOL de LabVIEW, puede desactivarse en el menú contextual.

Ficheros binarios



Estos VIs funcionan exactamente igual que los anteriores en cuanto al tipo de datos que se cablea en el terminal *file*. La diferencia es que admiten cualquier tipo de datos y lo almacenan una vez aplanado, mientras que los anteriores trabajan con cadenas de caracteres en ASCII. Sería equivalente a usar un *Flatten To String* antes de las anteriores.

Spreadsheet

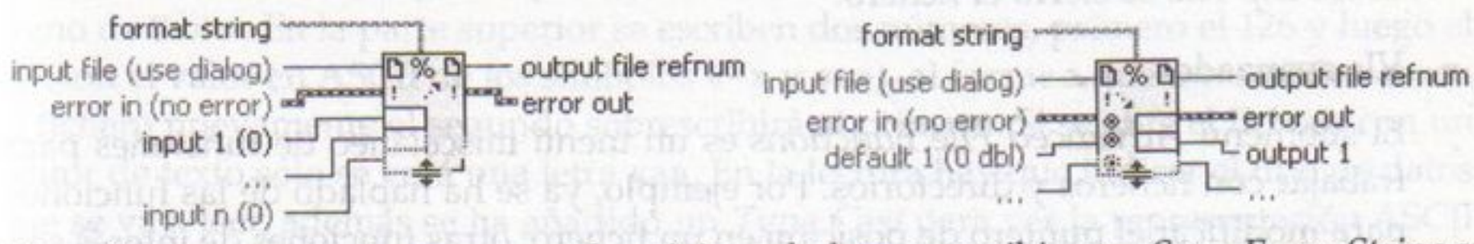


Los ficheros *Spreadsheet* suelen tener extensión *.csv y son ficheros de texto que representan tablas, cada línea del fichero sería una fila de la tabla y para indicar las columnas se suele emplear el carácter tabulador o el «;». Este tipo de ficheros son útiles para guardar información de forma ordenada, además puede ser leído por otras muchas aplicaciones, por ejemplo MS Excel.

La escritura tiene el terminal *Append To File?* para indicar si los nuevos datos tienen que sobrescribir a los que había o deben agregarse al final, y el terminal *Format* para indicar el formato numérico, este formato puede consultarse en la ayuda.

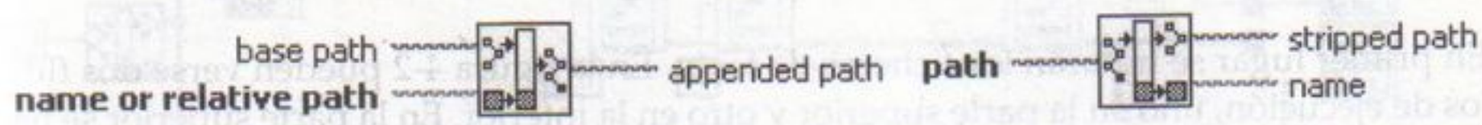
La lectura es desde la versión 8.20 un VI polimórfico, la instancia seleccionada condiciona el tipo de salida (ver terminal *Format*): doble precisión, entero o *string*; puede leerse todo el fichero o de sólo una parte. Estos VIs pueden considerarse como una forma compacta de los de leer y escribir junto con *Array To Spreadsheet String* y *Spreadsheet String To Array*.

Format y Scan



Estos VIs son equivalentes a usar los típicos de leer y escribir con *Scan From String* y *Format To String* respectivamente. Al escribir, con un solo VI se pueden convertir varios tipos de datos a *string*, concatenarlos y guardar en un fichero. El proceso contrario ocurre en la lectura.

Path



Build Path construye una ruta a partir de dos entradas.

Strip Path separa de una ruta el fichero o último directorio del resto, por ejemplo, si la ruta es «c:\windows\win.ini» separará por una parte «c:\windows» y por otra «win.ini», si se aplica sobre «c:\windows\system32» separa «c:\windows» de «system32».

También se puede construir una ruta con las funciones del menú *string*, pero no es recomendable hacerlo así, por ejemplo, presentaría problemas si el programa debe ejecutarse sobre Windows o Linux debido a que las barras que sirven para separar los directorios en el *path* son diferentes.

En el submenú *File Constants* hay varios nodos que devuelven la ruta de un determinado elemento, entre ellos el más usado es *Current VI s Path* que devuelve la ruta del fichero en el que se coloca.

- **Measurement File (VIs Express)**

  Estos VIs Express están especialmente preparados para almacenar y recuperar datos provenientes de formas de onda. Entre las opciones más destacadas se encuentra la posibilidad de guardar los datos en formato LVM, TDM o TDMS. El formato LVM es un fichero de texto que básicamente consta de una cabecera con información sobre el fichero (autor, fecha, etc.) y canales; cada canal a su vez también tiene una cabecera y los datos propiamente dichos. Los formatos TDM se estudiarán en el apartado dedicado a los ficheros *Storage*.

- **Ficheros ZIP**

  Estas funciones se han añadido a LabVIEW en la versión 8.0. Sirven para crear ficheros comprimidos en formato ZIP. La primera de ellas es *New Zip File*, su función es crear un fichero, la segunda es *Add File to Zip* que añade un fichero cualquiera al ZIP creado con la anterior, a este VI hay que indicarle la ruta absoluta del fichero a comprimir y la ruta relativa que tendrá dentro del ZIP. Finalmente con *Close Zip* File se cierra el fichero.

- **VIs avanzados**

  El submenú *Advanced File Functions* es un menú misceláneo de funciones para trabajar con ficheros y directorios. Por ejemplo, ya se ha hablado de las funciones para modificar el puntero de posición en un fichero; otras funciones de interés son las que trabajan con directorios, mueven y copian ficheros, etc.

4.1.1.1 Ejemplo I: Ficheros de texto y binarios

Este ejemplo se ha hecho con la intención de conocer las diferencias entre usar por un lado *path* o referencias para indicar el fichero y por otro lado entre ficheros de texto y binarios.

En primer lugar se tratarán los ficheros de texto. En la figura 4-2 pueden verse dos flujos de ejecución, uno en la parte superior y otro en la inferior. En la parte superior se ha usado el *path* del fichero como entrada para el resto de funciones. Primero se escribe un *string* y luego otro, pero en la segunda escritura también se usa *path*, por lo que el puntero que indica la posición del fichero empieza de nuevo en cero y sobrescribirá lo anterior, por lo tanto la salida del nodo de lectura (y el contenido del fichero) será únicamente la segunda constante escrita.

En la parte inferior se usa una referencia al fichero una vez abierto, al escribir la primera constante el puntero aumentará en cuatro posiciones, por lo que al volver a escribir otra constante se adjuntará después de la primera, y no se sobrescribirá. Después se usa una función *Set File Position* para manipular el puntero de posición y leer cuatro caracteres a partir de la segunda posición. El resultado serán cuatro caracteres: los dos últimos de la primera constante y los dos primeros de la segunda, «lamu».

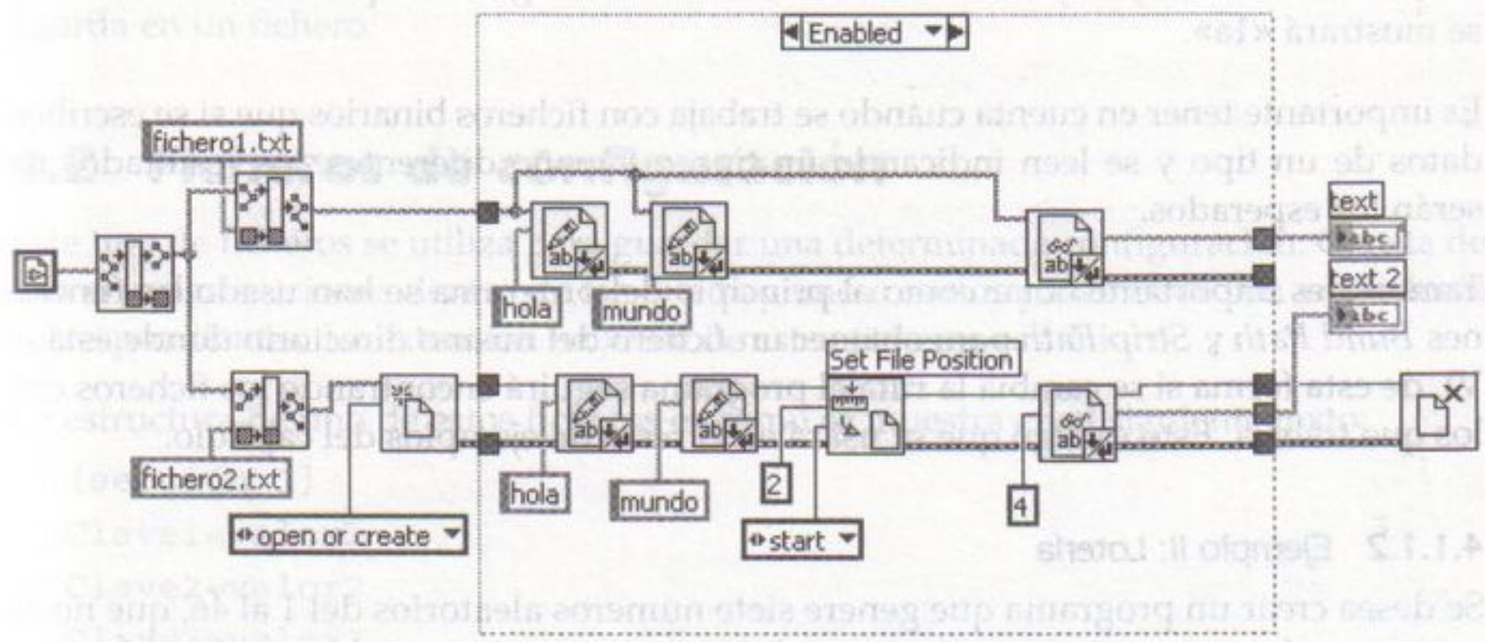


Figura 4-2. Ejemplo de manipulación de ficheros de texto

Ahora se hará lo mismo en la figura 4-3 con las funciones que trabajan en binario. Estas funciones admiten cualquier tipo de datos, en este ejemplo se usarán numéricos sin signo de 8 bits. En la parte superior se escriben dos números, primero el 126 y luego el 97 (son el valor en ASCII de los símbolos «~» y «a»), al forzar a que estos nodos abran el fichero nuevamente el segundo sobrescribirá al primero. Si se abre el fichero con un editor de texto sólo se verá una letra «a». En la lectura hay que indicar el tipo de datos que se va a leer, además se ha añadido un *Type Cast* para ver la representación ASCII del valor leído en un *string*.

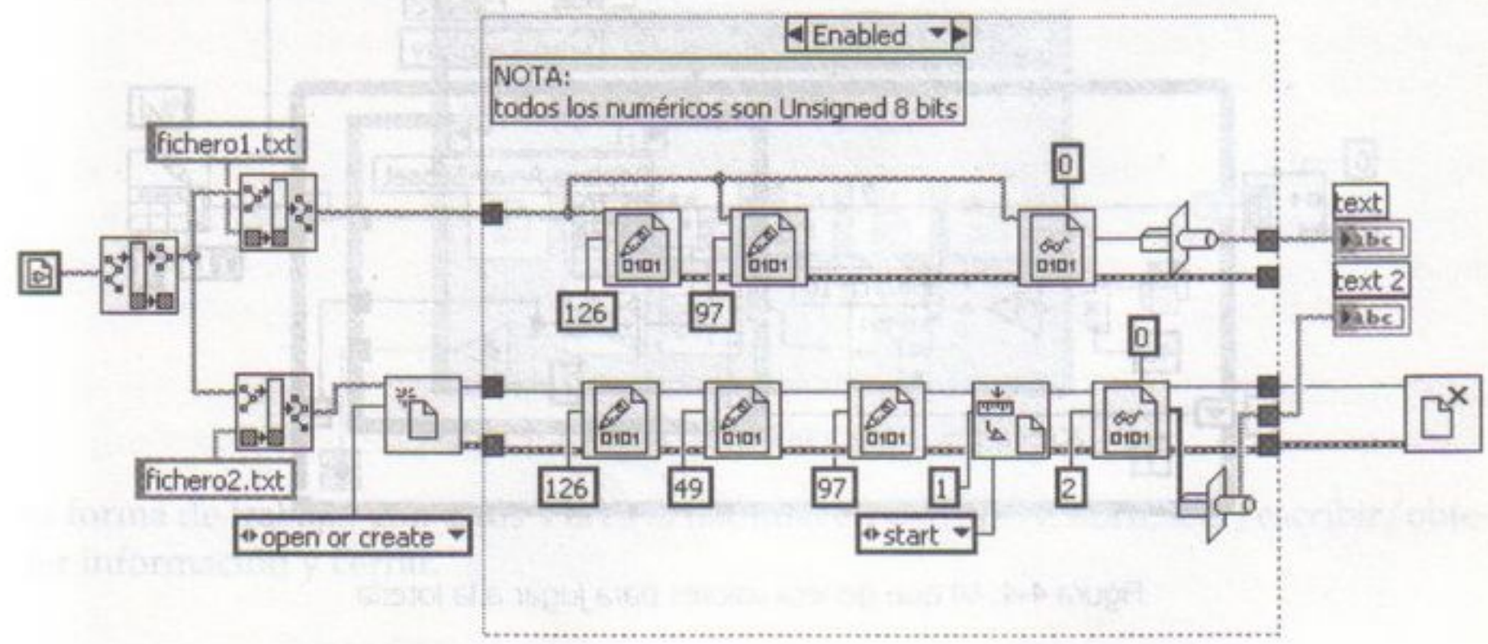


Figura 4-3. Ejemplo de manipulación de ficheros binarios

La parte inferior usa referencias a un fichero previamente abierto por lo que el puntero de posición no se reiniciará en cada nodo como en el caso anterior. En primer lugar se escriben tres números: 126, 49 y 97; estos números se escribirán uno junto al otro, por lo que el contenido del fichero será «~1a». En ese momento el puntero apuntará a la posición 3 pero se usa una función para variarlo y que apunte a la 1, después se leen dos bytes (se está trabajando con enteros sin signo de 8 bits) que son los que corresponden a los valores 49 y 97 y finalmente se convierte a *string* con lo que en el Panel Frontal se mostrará «1a».

Es importante tener en cuenta cuando se trabaja con ficheros binarios que si se escriben datos de un tipo y se leen indicando un tipo o tamaño diferentes, los resultados no serán los esperados.

También es importante notar como al principio del programa se han usado las funciones *Build Path* y *Strip Path* para obtener un fichero del mismo directorio donde está el VI, de esta forma si se cambia la ruta el programa seguirá encontrando los ficheros con los que trabaja. Esto es algo que se usará en el resto de ejemplos del capítulo.

4.1.1.2 Ejemplo II: Lotería

Se desea crear un programa que genere siete números aleatorios del 1 al 46, que no se repitan y almacenar el resultado en un fichero que pueda ser leído por otras aplicaciones.

Para escribir el fichero se usará *Write To Spreadsheet File* indicando que cada vez que se genere un nuevo resultado se escriba en el fichero una línea que contenga los números. En este VI se indica que se guarden los datos como enteros (%d) y que el carácter separador sea un «;».

El código se muestra en la figura 4-4.

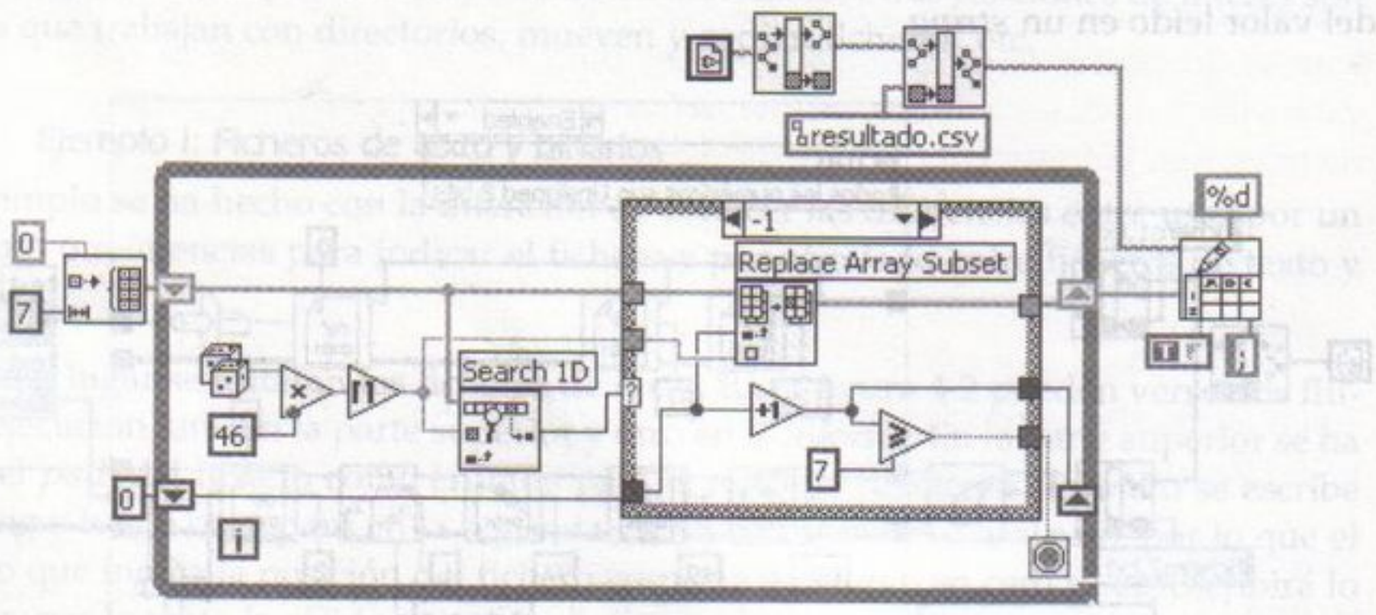


Figura 4-4. VI que genera valores para jugar a la lotería

Sobre el programa que genera los números se pueden comentar varias cosas. En primer lugar se crea un *array* del tamaño apropiado, cuando se genere un nuevo número válido se almacenará en una posición de ese *array*. Para generar números dentro de un rango se multiplican los números aleatorios que van entre cero y uno por el límite superior y después se redondea al entero inmediatamente superior. El siguiente paso es comprobar si el número generado ya había sido sacado, si no es así se almacena en una posición del *array*. Cuando ya se tienen los siete números acaba el bucle WHILE y se guarda en un fichero.

4.2 Ficheros de configuración

Este tipo de ficheros se utiliza para guardar una determinada configuración. Consta de una o varias secciones dentro de las que hay uno o varios parámetros con sus valores correspondientes. La extensión habitual de estos ficheros es *.ini.

La estructura de uno de estos ficheros es como se muestra en el siguiente texto:

```
[sección 1]
Clave1=valor1
Clave2=valor2
Clave3=valor3
[sección 2]
Clave4=valor4
Clave5=valor5
```

Si se va a la carpeta de instalación de LabVIEW se podrá ver un fichero llamado LabVIEW.ini que guarda la configuración sobre varios parámetros del funcionamiento de este programa.

En LabVIEW existe una paleta en *Programming > File I/O > Configuration File VIs* con funciones para manejar estos ficheros.

Figura 4-5. Paleta de configuración de ficheros

La forma de trabajar con estos VIs es la habitual en LabVIEW: abrir, leer/escribir/obtener información y cerrar.

4.2.1 Ejemplo

En el siguiente ejemplo se muestra la forma de trabajar con estos VIs.

En primer lugar se abre una referencia al fichero «config.ini» situado en la misma carpeta que el VI. A continuación se escribe la clave «`param`» con el valor «`valor`» dentro de la sección «`sub config`», ver figura 4-6.

La siguiente parte sería la de lectura, en ella se leerá todo el fichero y se mostrará en una tabla. Lo primero será obtener los nombres de las secciones para que sirvan como cabeceras de las columnas de la tabla. Después se entra en un bucle FOR que se repite tantas veces como secciones haya; en este bucle se obtienen los nombres de todas las claves y se entra en otro bucle FOR que se repite el mismo número de veces que claves hay en la sección actual. Dentro del segundo bucle simplemente se lee el valor de la clave y se genera un *string* con el formato «clave=valor» que será el mostrado en la tabla.

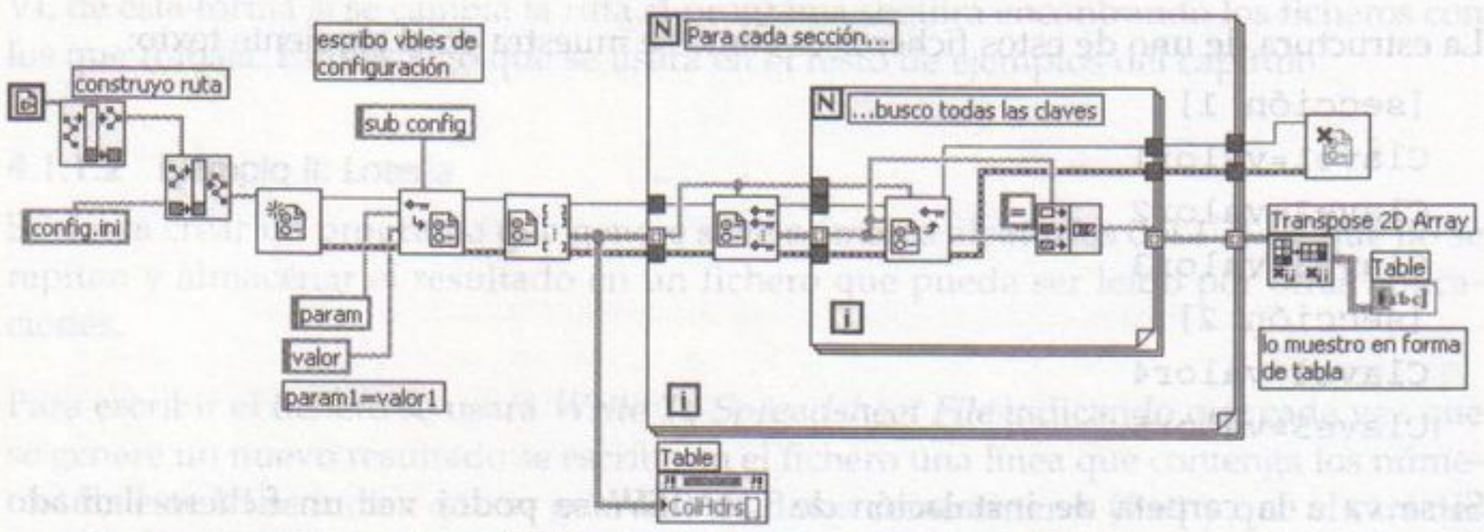


Figura 4-6. Ejemplo de configuración de ficheros

Si el fichero «config.ini», antes de ejecutar este programa, contenía:

```
[config principal]
param1=valor1
param2=valor2
[sub config]
param=valor
```

Después de la ejecución contendrá una sección más y el resultado será:

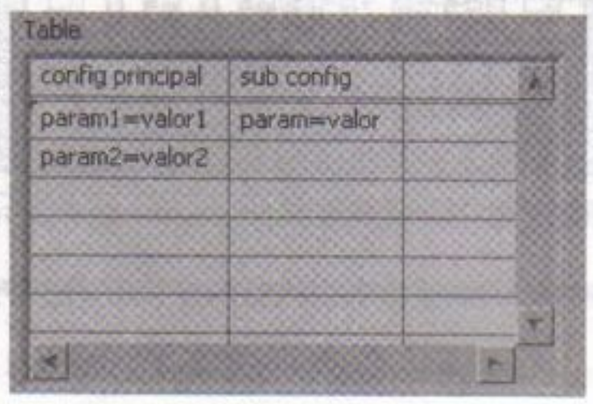


Table		
config principal	sub config	
param1=valor1	param=valor	
param2=valor2		

Figura 4-7. Resultado de la ejecución del VI dado por la figura 4-6

4.3 Ficheros XML

XML significa *eXtensible Markup Language*. El lenguaje XML es una recomendación del W3C de finales de los años 90, aunque se basa en otros estándares mucho más antiguos.

Su aplicación es el intercambio de información estructurada entre diferentes programas e incluso plataformas.

Básicamente es un método de aplicar etiquetas para describir partes de un documento. También se dice que es un metalenguaje, esto es un lenguaje para definir otros lenguajes.

Un ejemplo de fichero XML es el siguiente:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<clase>
    <alumno>
            <nombre>José</nombre>
            <apellido>García</apellido>
            <nota>4.99</nota>
    </alumno>
</clase>
```

Como se puede ver es un lenguaje de marcas muy parecido a HTML y aún más a XHTML (de hecho éste es un lenguaje basado en XML).

Las etiquetas van entre los símbolos <...>, la forma de colocar las etiquetas es muy estricta, es decir, se necesita un perfecto anidamiento y que todas las etiquetas abiertas se cierren. Para indicar que una etiqueta se cierra se usa `</...>`. Estas etiquetas de inicio y fin encierran en su interior datos u otras etiquetas de la forma `<marca>...</marca>`. Algunas de estas etiquetas no encierran nada, por lo que no es necesario poner otra marca para cerrar, en este caso se usa `<marca/>`, que puede entenderse como una forma resumida de abrir y cerrar una etiqueta. También se pueden incluir parámetros en las marcas como `<marca parámetro="valor">...`.

En la primera línea se indican varios atributos del fichero, el *standalone* sirve para decir si el documento puede ir sólo o si va acompañado de un DTD. El DTD es una definición de los elementos que se pueden incluir en las marcas y la forma de incluirlos, en caso de indicarse puede ir en el mismo fichero o indicar la ruta donde encontrarlo.

Para ver un fichero XML basta con abrir el fichero de un proyecto de LabVIEW (*.lvproj) con un editor de textos.

El menú para trabajar con XML en LabVIEW se encuentra en *Programming > String > XML*.

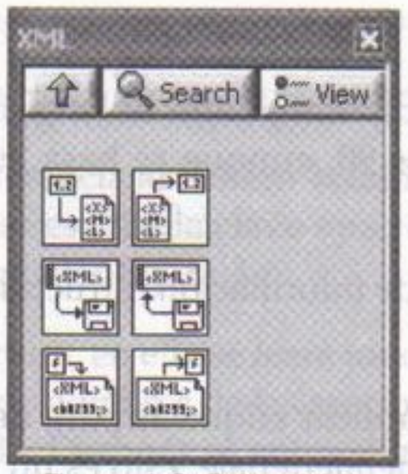


Figura 4-8. Paleta XML

Los dos primeros VIs (*Flatten to XML* y *Unflatten to XML*) se usan para transformar a formato XML y viceversa cualquier control de cualquier tipo de dato, de esta forma todos los tipos de datos se almacenan de una manera uniforme. El formato sería:

```
<Tipo de datos>
    <Name>nombre</Name>
    <Val>valor</Val>
</Tipo de datos>
```

Los dos siguientes VIs (*Write to XML* File y *Read from XML File*) son para escribir y leer el fichero de datos respectivamente. Mientras que los dos últimos (*Escape XML* y *Unescape XML*) se usan cuando se almacenan caracteres especiales, como letras con acentos y otros símbolos.

4.3.1 Ejemplo: Agenda

En este ejemplo se quiere programar un sistema para almacenar los datos del personal de la plantilla de una pequeña empresa. El formato del fichero debe ser XML para hacerlo compatible con otras herramientas.

La estructura del fichero consistirá en un registro por cada empleado, ese registro contendrá los campos de nombre, apellidos, dirección, teléfono, departamento dentro de la empresa, años de antigüedad y sueldo.

Se emplearán *arrays* en la lectura y escritura del fichero, cada elemento del *array* será un *cluster* que contenga los campos de datos.

En la figura 4-9 se muestra la estructura del programa. Se ha empleado un EVENT para cada acción: guardar el dato mostrado, borrarlo y desplazarse por las fichas de los empleados. El evento que se ve en la imagen es el correspondiente a guardar los datos, en él, primero se leen todas las fichas que había guardadas, se modifica la correspondiente a la que se muestra en el Panel Frontal y después se guardan en el fichero. Para que lo almacenado en el fichero no dependa del tipo de datos, éstos tienen que convertirse a XML con el VI *Flatten To XML*, también se ha usado *Escape XML* para cambiar los caracteres especiales a formato XML.

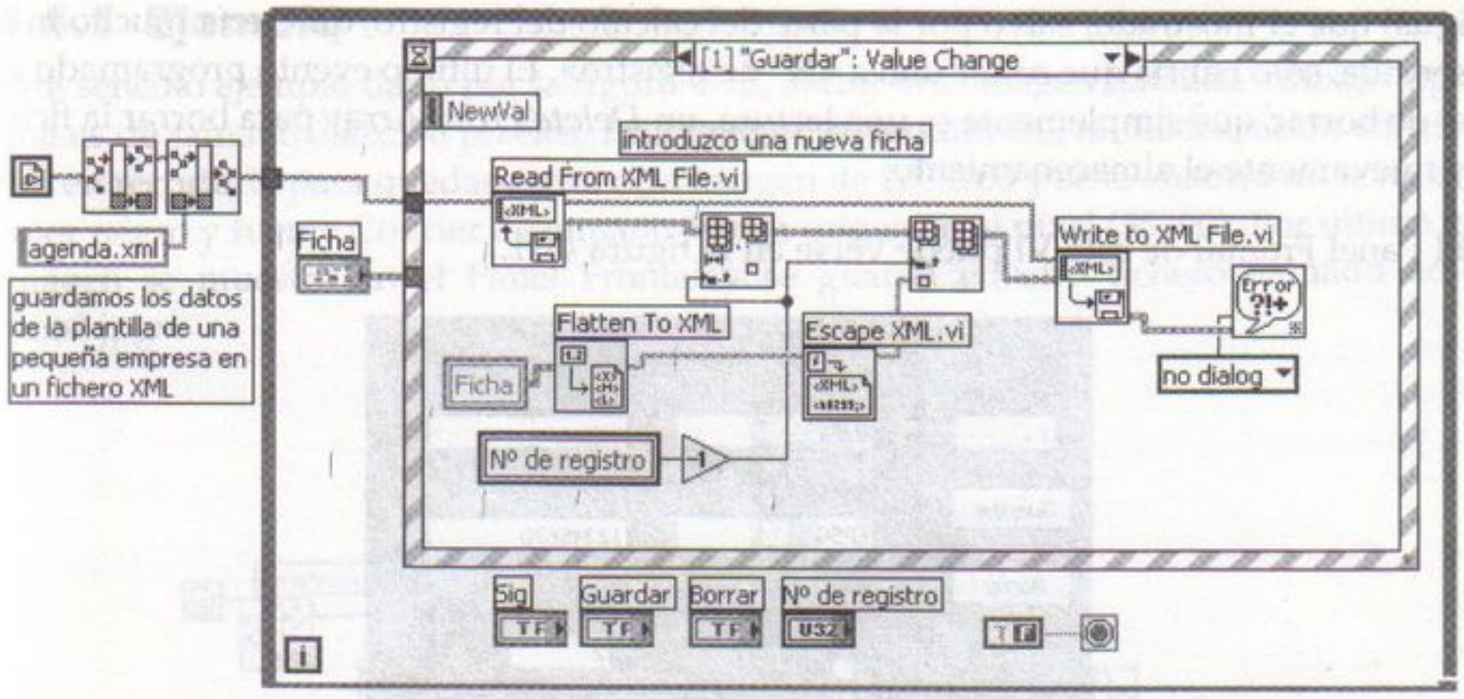


Figura 4-9. Uno de los eventos del programa que implementa una agenda

Antes de guardar los datos tienen que leerse, esto se hace así para poder modificar fichas; si solamente se guardaran sin leer todo el contenido antes sólo se podrían introducir nuevos registros, pero no cambiar uno existente.

En la figura 4-10 se puede ver el evento correspondiente al cambio de la ficha mostrada por medio de los botones «anterior» y «siguiente». En primer lugar se calcula qué elemento del *array* se debe mostrar, luego se lee el fichero y de él se selecciona sólo el elemento calculado antes. Con *Unflatten From XML* se convierten los datos de formato XML al tipo de datos y valor que le corresponderían en LabVIEW.

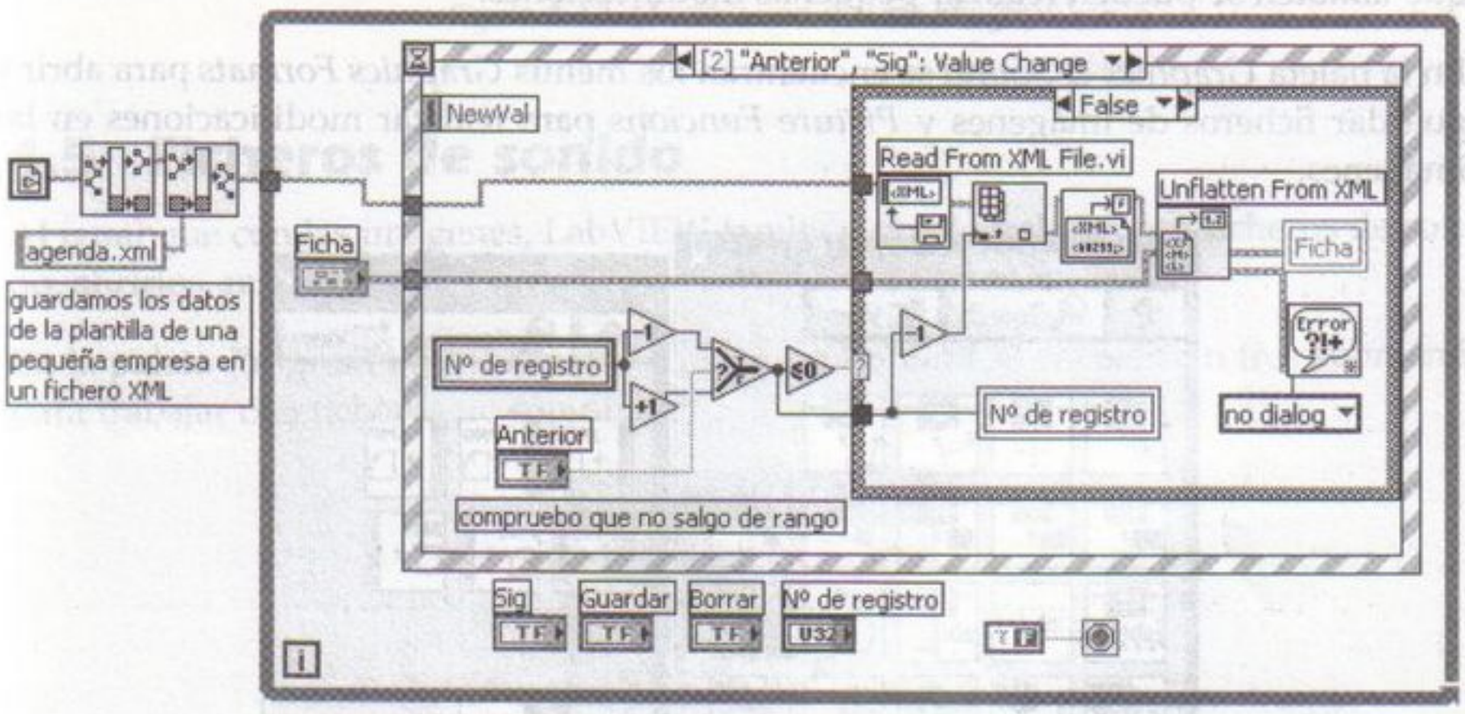


Figura 4-10. Evento de cambio de ficha de la agenda

En el ejemplo también se ha programado un evento cuando se cambia de forma manual el número de ficha a mostrar, es decir, cuando no se usan los botones «anterior» y «siguiente» y en su lugar se usa un indicador numérico. Este evento es prácticamente

igual que el mostrado, salvo por la parte del calculo del registro, que sería mucho más sencilla, sólo habría que restar uno a «Nº de registro». El último evento programado es el de borrar, que simplemente es una lectura, un *Delete From Array* para borrar la ficha y nuevamente el almacenamiento.

El Panel Frontal de este VI puede verse en la figura 4-11.

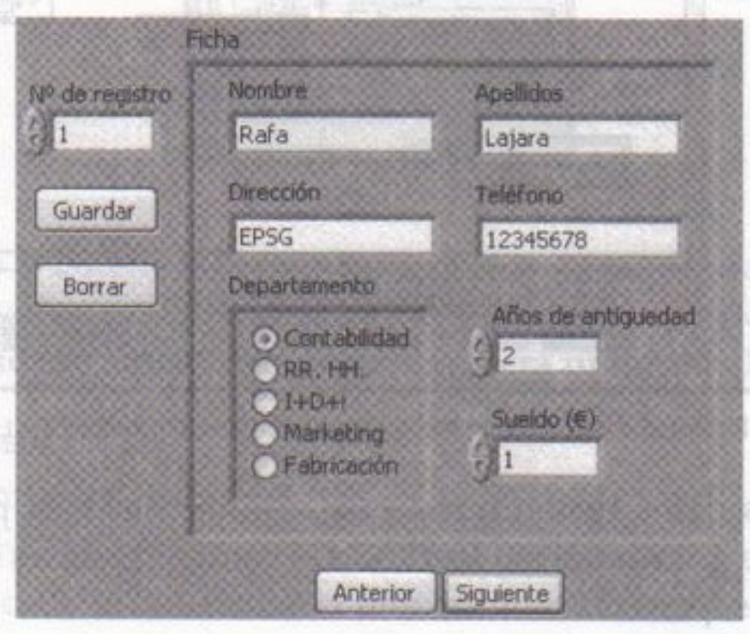


Figura 4-11. Panel Frontal de la Agenda

4.4 Ficheros de imagen

Con LabVIEW no sólo se pueden abrir ficheros de imágenes (*.jpg, *.bmp, *.png) sino que también se pueden realizar pequeñas modificaciones.

En la paleta *Graphics & Sound* se encuentran los menús *Graphics Formats* para abrir y guardar ficheros de imágenes y *Picture Funcions* para realizar modificaciones en las imágenes.

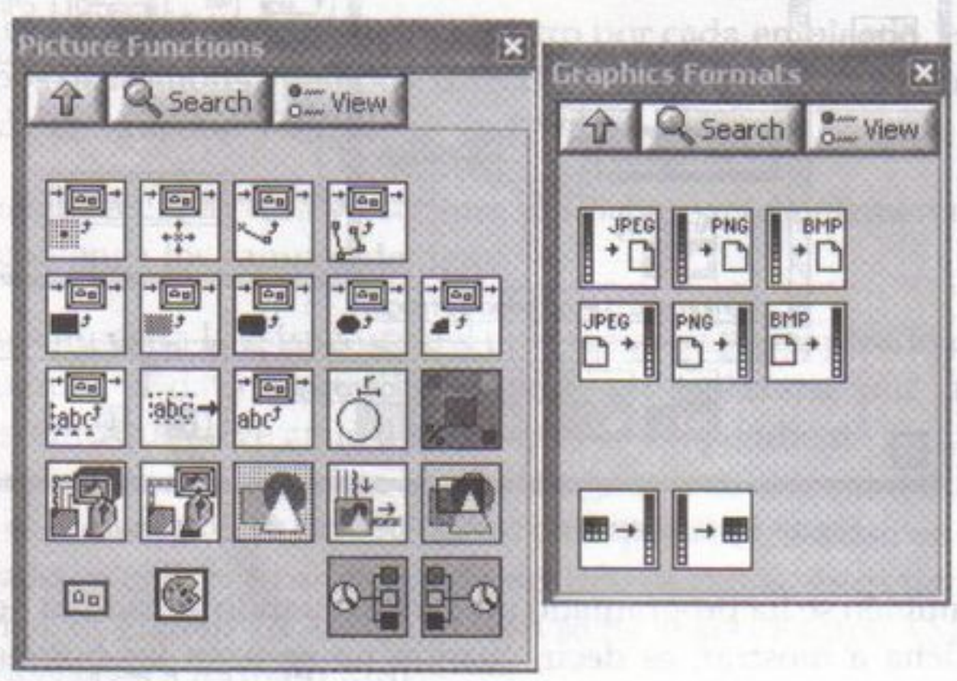


Figura 4-12. Paletas de funciones de imágenes y formatos gráficos

4.4.1 Ejemplo

Este sencillo ejemplo dado por la figura 4-13, abrirá una imagen llamada «dibujo.jpg» que es de tamaño 300x300 píxeles. Esta imagen se modificará, la modificación consistirá en recortarla para quedarse con otra imagen de 200x200 y se le añadirá un texto en color negro y fuente Courier de tamaño 30 con origen en el píxel (55, 90). Por último la imagen se muestra en el Panel Frontal y se guarda en otro fichero llamado «dibujo2.jpg».

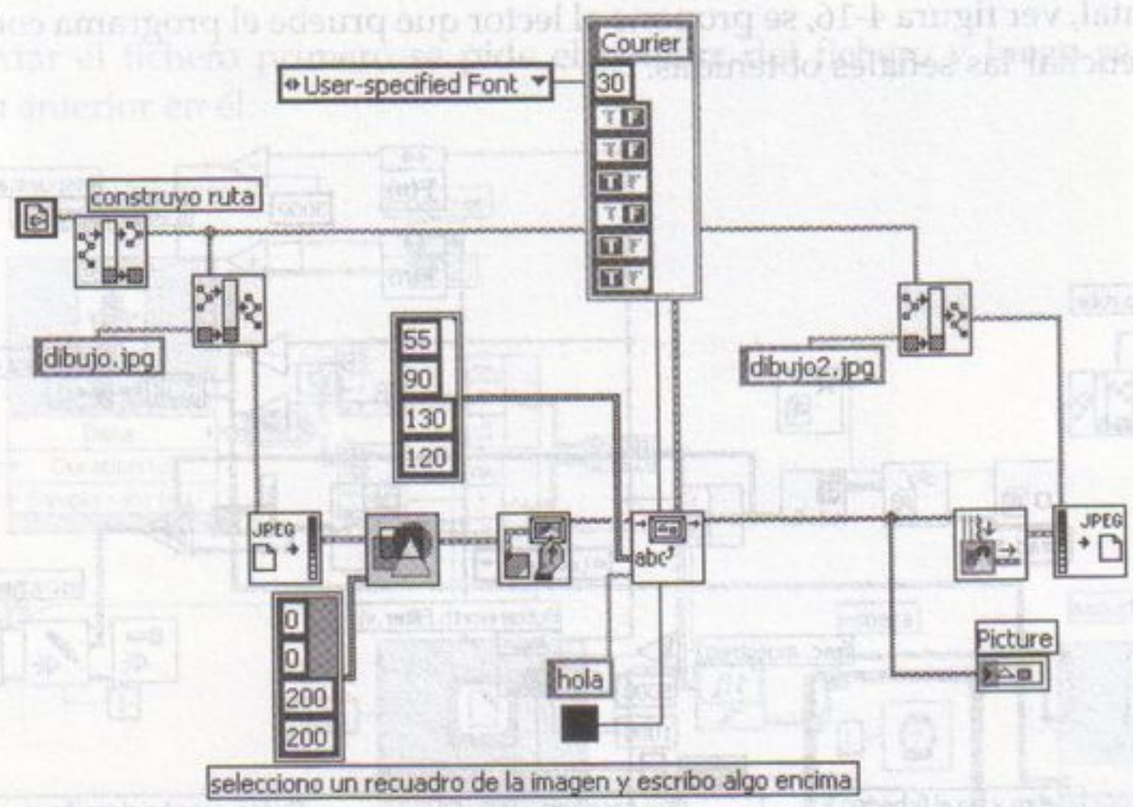


Figura 4-13. Modificación de una imagen

4.5 Ficheros de sonido

Al igual que con las imágenes, LabVIEW también puede trabajar con ficheros de sonido, abrirlos, modificarlos y guardarlos

En la paleta *Programming > Graphics & Sound> Sound* se encuentran tres submenús para trabajar con ficheros de sonido.

Figura 4-14. Paleta para trabajar con ficheros de sonido

4.5.1 Ejemplo I: Lectura de fichero y procesado

El programa, dado por la figura 4-15, en primer lugar abrirá un fichero *.wav y lo leerá, este fichero se procesará mediante un filtro pasa bajo de segundo orden con el que se eliminarán los tonos agudos y sólo quedarán los graves; a continuación se muestran en gráficas las respuesta en tiempo y frecuencia del sonido antes y después de filtrarlo, además también se puede elegir si se escuchará el sonido antes o después de filtrar.

Una vez ejecutado el programa se puede ver el resultado del filtro en las gráficas del Panel Frontal, ver figura 4-16, se propone al lector que pruebe el programa con un fichero para escuchar las señales obtenidas.

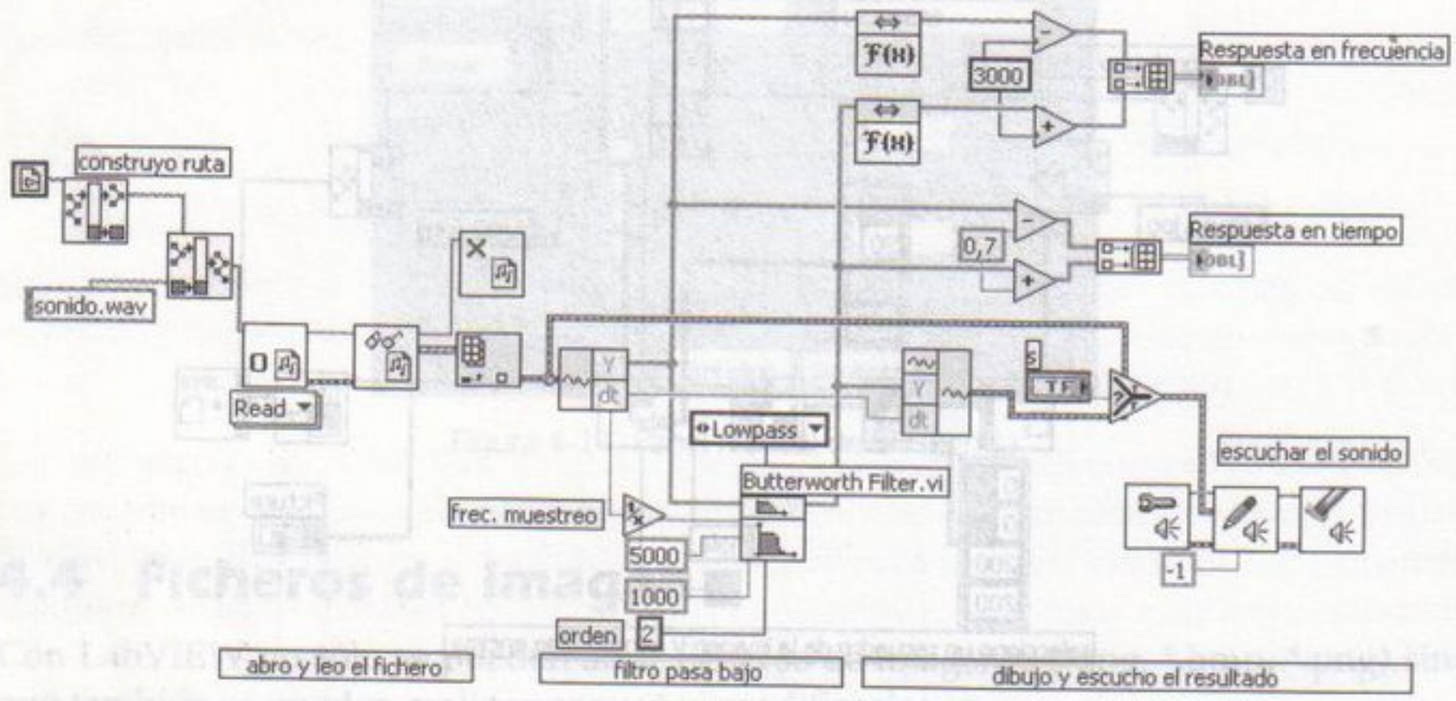


Figura 4-15. Programa que procesa ficheros de audio

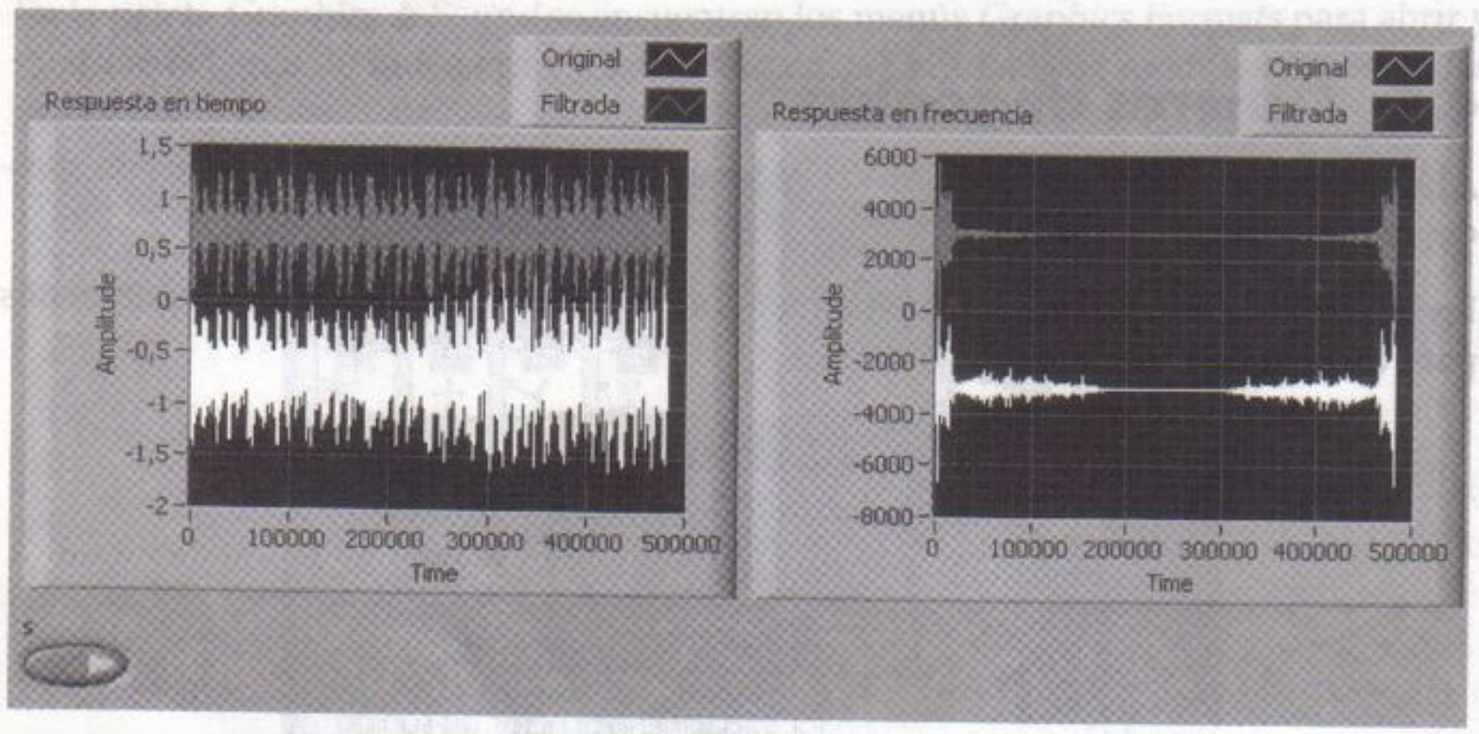


Figura 4-16. Resultado del filtrado del fichero de audio

4.5.2 Ejemplo II: Adquisición y VIs Express

Para variar respecto al ejemplo anterior, esta vez se usarán VIs Express. Este ejemplo consiste en adquirir una señal de audio estéreo y guardarla en un fichero. El programa se muestra en la figura 4-17.

La adquisición se realiza mediante el VI *Adquire Sound*, configurado adecuadamente. A continuación se han separado los dos canales en sendos *arrays* con los que construir un *waveform*.

Para guardar el fichero primero se pide el nombre del fichero y luego se escribe el *waveform* anterior en él.

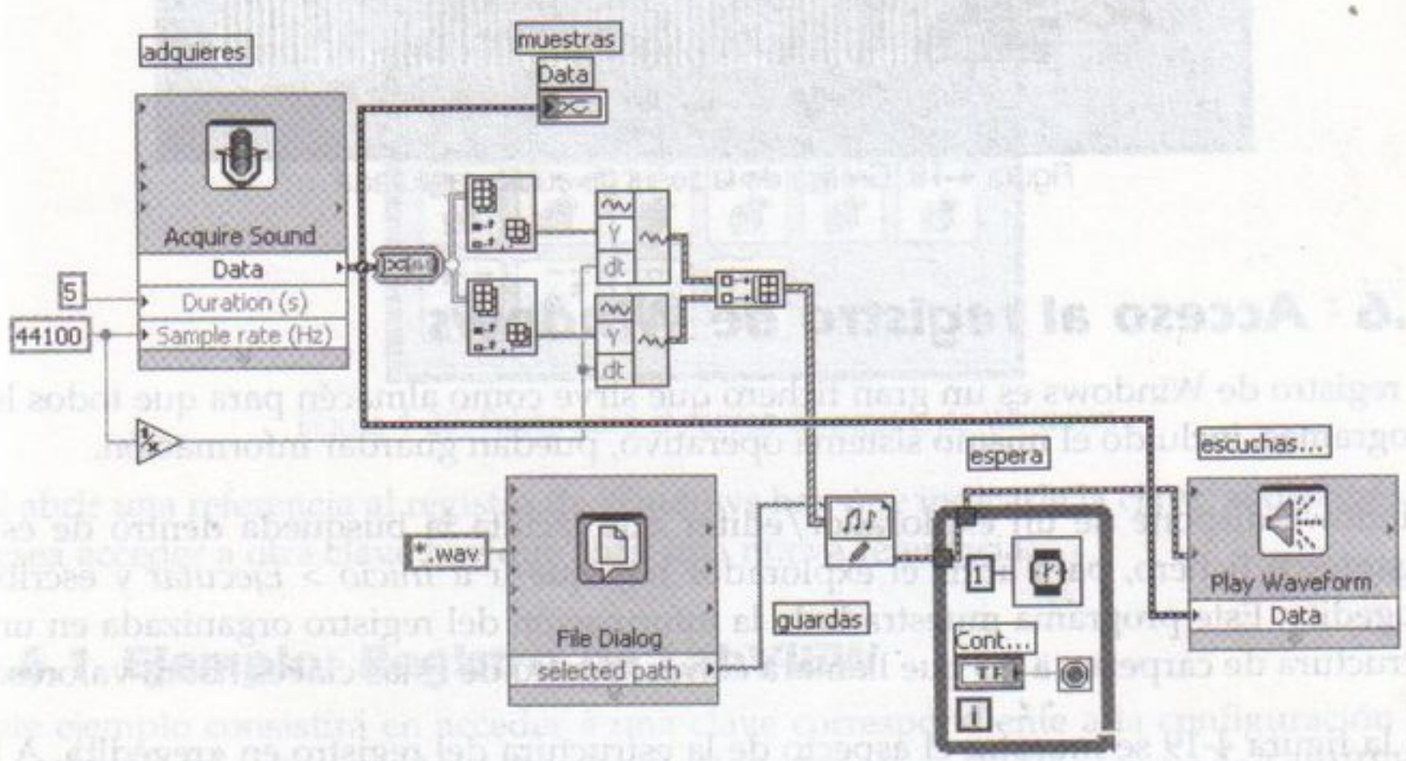


Figura 4-17. Adquisición y almacenamiento de una señal de audio

Para probar este programa se han conectado en el ordenador la salida de los altavoces con la entrada del micrófono de la tarjeta de sonido. Después se ha abierto un fichero de audio con un reproductor multimedia, el cual ha generado una señal por la salida de la tarjeta de sonido que es capturada por su entrada mediante la conexión descrita antes. Este programa ha capturado cinco segundos de esta señal y los ha guardado en un fichero.

Debido a este montaje más o menos rudimentario, se ha añadido un bucle WHILE que introduce una espera hasta que se presiona un botón; de esta forma se podrán cambiar las conexiones en la tarjeta de sonido antes de escuchar el resultado.

En la figura 4-18 puede verse la gráfica de uno de los sonidos capturados.

Nótese que en este caso la captura realizada con la tarjeta de sonido ha sido de una señal de audio, no obstante también se podría capturar cualquier señal eléctrica siempre que se respeten los límites de la tarjeta; de esta manera se tendría una pequeña y muy barata tarjeta de adquisición de datos.

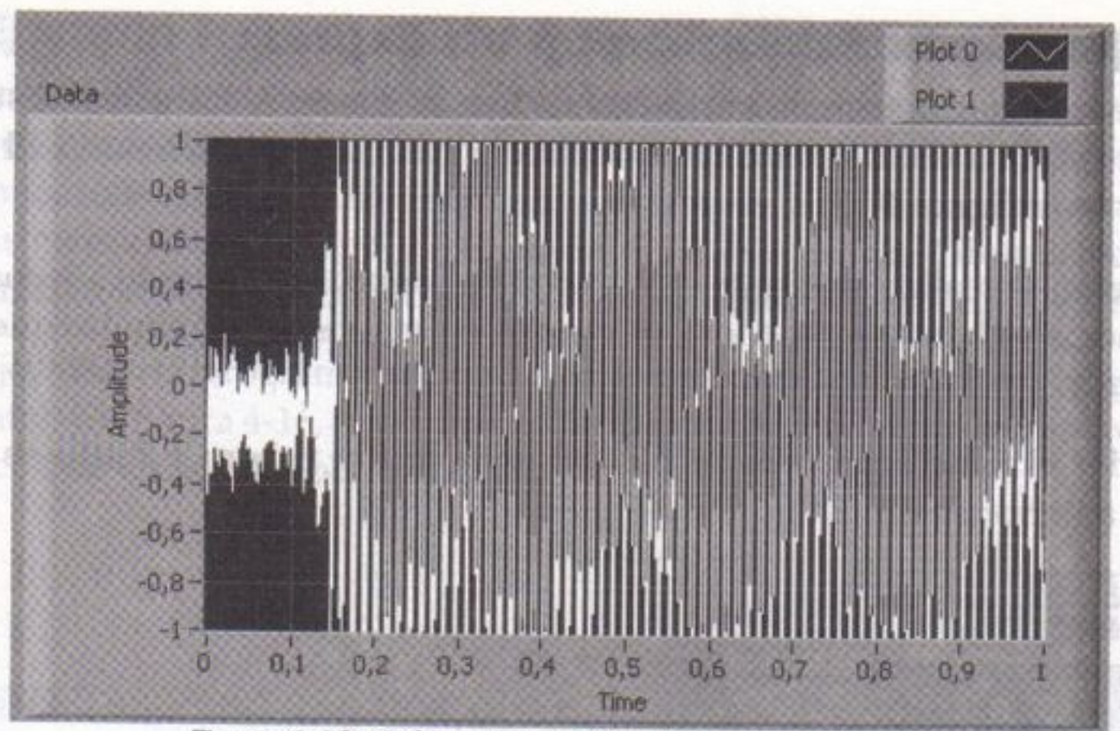


Figura 4-18. Gráfica de la señal de audio capturada

4.6 Acceso al registro de Windows

El registro de Windows es un gran fichero que sirve como almacén para que todos los programas, incluido el mismo sistema operativo, puedan guardar información.

Windows dispone de un explorador/editor que facilita la búsqueda dentro de este gigantesco fichero, para abrir el explorador hay que ir a *Inicio > Ejecutar* y escribir «regedit». Este programa muestra toda la información del registro organizada en una estructura de carpetas, a las que llamará claves, dentro de estas claves habrá valores.

En la figura 4-19 se muestra el aspecto de la estructura del registro en «regedit». A la izquierda pueden verse las claves principales (las de primer nivel) y de ellas se despliegan varias subclaves; a la derecha se ven los nombres, tipos y datos de los valores que contiene la clave seleccionada.

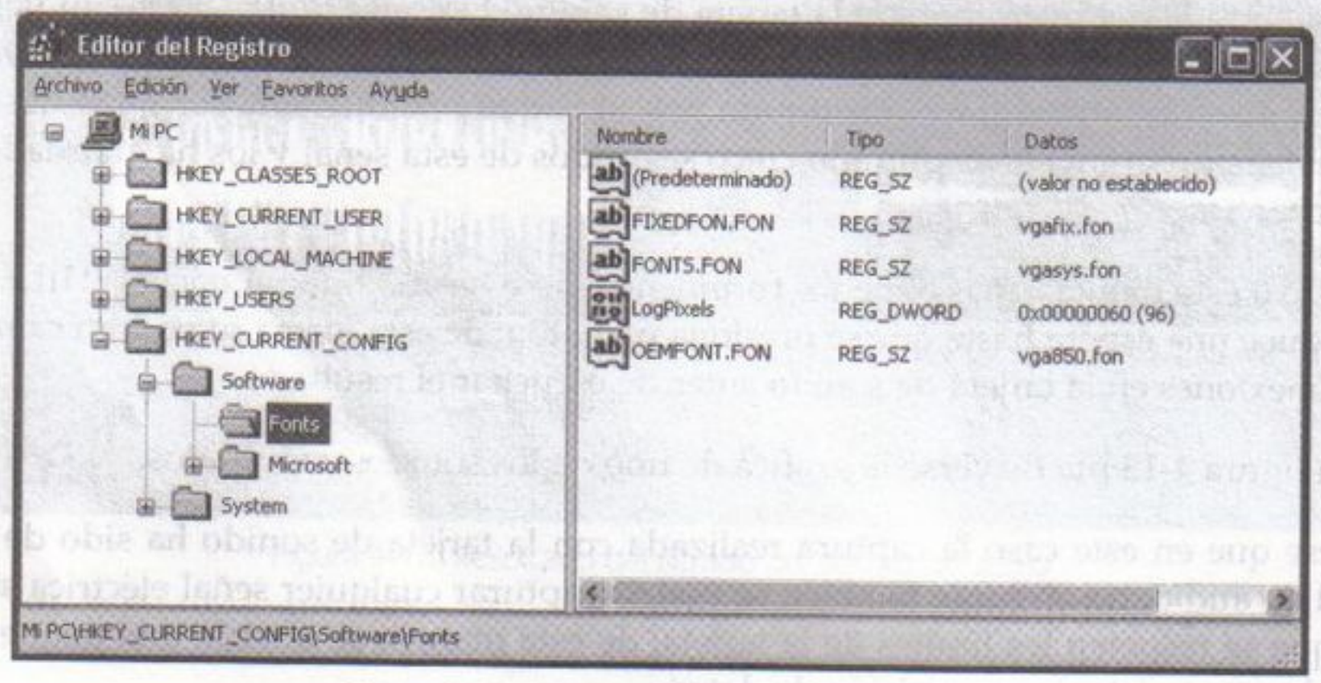


Figura 4-19. Editor del Registro de Windows

Dado que el funcionamiento del sistema operativo está determinado por los valores que están almacenados en el registro, conviene no modificarlos a menos que se sepa con seguridad lo que se está haciendo.

Desde LabVIEW se pueden crear, leer, escribir y borrar datos del registro de Windows, para ello existe una paleta dentro de *Connectivity > Windows Registry Access VIs* con las funciones mostradas en la figura 4-20.

El funcionamiento de estos VIs es muy parecido a los del apartado de ficheros de configuración. La forma de usarlos también: abrir, leer/escribir/borrar/obtener información y cerrar.

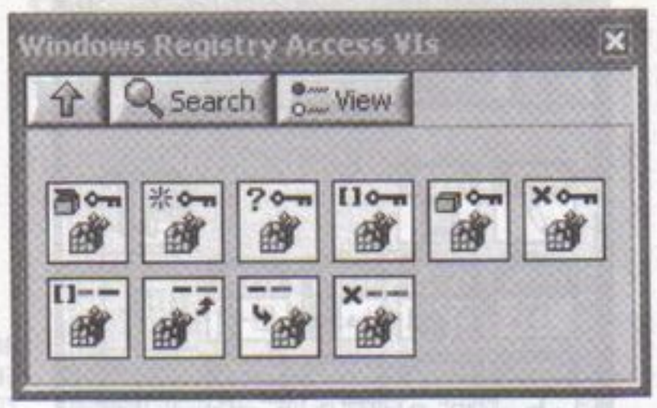


Figura 4-20. Paleta para el acceso al registro de Windows

Al abrir una referencia al registro de Windows hay que indicarle la clave; si después se desea acceder a otra clave hay que abrir una nueva referencia.

4.6.1 Ejemplo: Registro de LabVIEW

Este ejemplo consistirá en acceder a una clave correspondiente a la configuración de LabVIEW 8.20 y obtener el nombre y organización a la cual está registrado LabVIEW.

Para esto hay que ir a la clave:

```
HKEY_LOCAL_MACHINE\SOFTWARE\National Instruments\LabVIEW\8.2
```

Y leer los valores «RegisteredOwner» y «RegisteredOrganization».

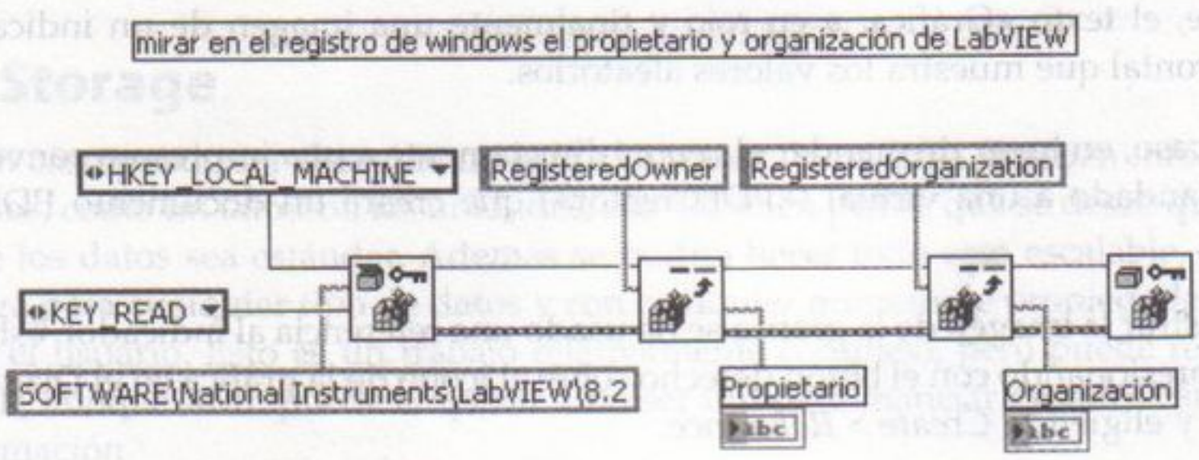


Figura 4-21. VI para manipular el registro de Windows con LabVIEW 8.20

4.7 Reports

Los *reports* son informes que pueden generarse con LabVIEW, pueden ser guardados como ficheros HTML o enviados directamente a una impresora.

Su utilización consiste en abrir una referencia al informe e ir construyéndolo poco a poco. Para construirlo se van añadiendo nuevos componentes, estos componentes pueden ser textos, imágenes externas, imágenes de los controles e indicadores del Panel Frontal, el propio Panel Frontal, la documentación y jerarquía de un VI, etc.

En la figura 4-22 puede verse la paleta con los VIs para crear informes.

Figura 4-22. Paleta para crear informes

Además de esta paleta, también existe un *toolkit* con el que se pueden crear *reports* para MS Word y MS Excel de una forma idéntica a ésta.

4.7.1 Ejemplo: Generación de un fichero PDF

En el siguiente ejemplo, figura 4-23, se crea un informe cuyo título es «Informe de prueba» y está compuesto por el icono del VI, una lista de diez números generados aleatoriamente, el texto «Gráfica: » en rojo y finalmente una imagen de un indicador del Panel Frontal que muestra los valores aleatorios.

En este caso, en lugar de mandar el *report* directamente a una impresora convencional se ha mandado a una virtual («PDFCreator») que creará un documento PDF con el informe.

Para mostrar la imagen de la gráfica se ha usado una referencia al indicador, ésta puede crearse presionando con el botón derecho sobre el icono de la gráfica en el Diagrama de Bloques y eligiendo *Create > Reference*.

Este informe quedaría como se muestra en la figura 4-24.

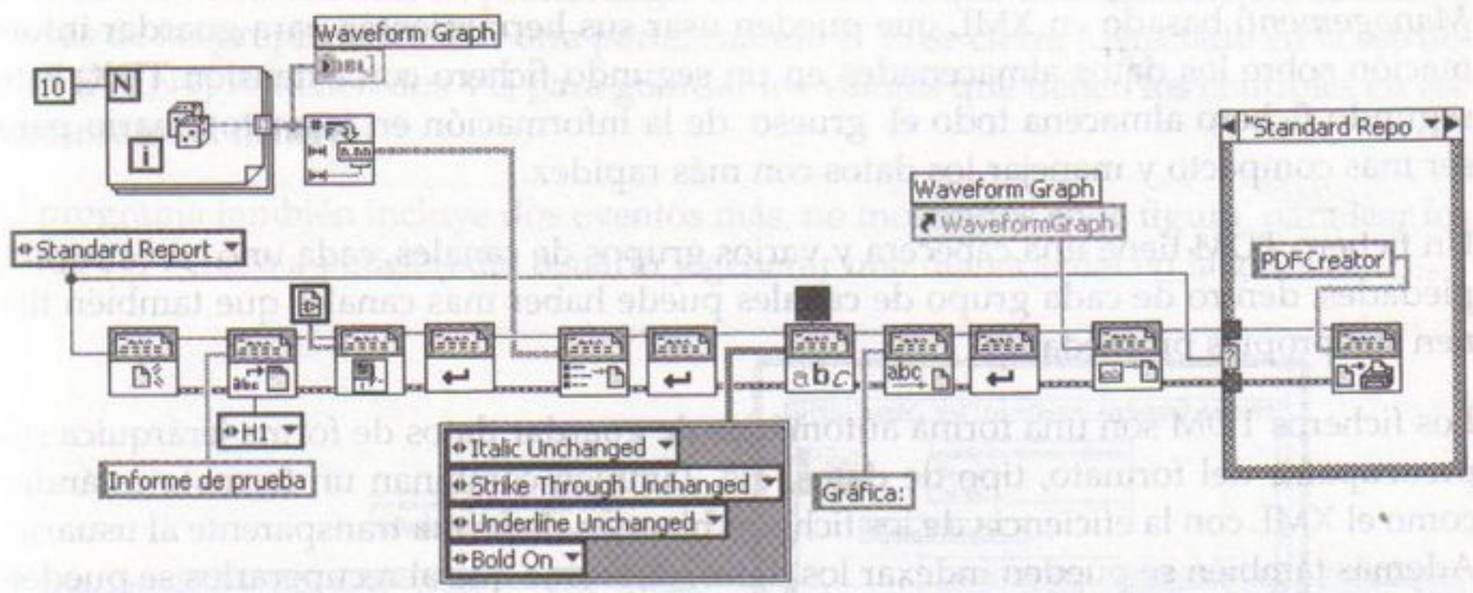


Figura 4-23. Generación de un fichero PDF

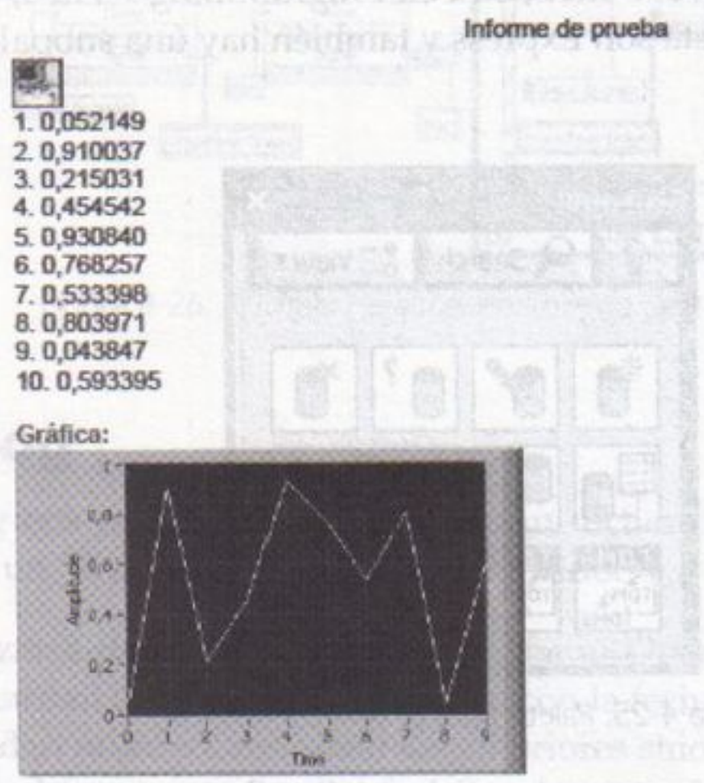


Figura 4-24. Resultado del Report generado

4.8 Storage

En algún caso puede ser deseable no sólo almacenar datos, sino también información sobre ellos, como un nombre, las unidades, etc. También puede que se desee que el formato de los datos sea estándar. Además se podría hacer todo esto escalable, es decir, que valga para cualquier tipo de datos y con cualquier número de propiedades definidas por el usuario. Esto es un trabajo relativamente complejo, pero puede resolverse con XML, aunque XML puede empezar a no ser útil para manejar grandes cantidades de información.

National Instruments ha creado un formato de fichero llamado TDM (*Technical Data Management*) basado en XML que pueden usar sus herramientas para guardar información sobre los datos almacenados en un segundo fichero con extensión TDX. Este segundo fichero almacena todo el grueso de la información en formato binario para ser más compacto y manejar los datos con más rapidez.

Un fichero TDM tiene una cabecera y varios grupos de canales, cada uno con sus propiedades, dentro de cada grupo de canales puede haber más canales que también tienen sus propias propiedades.

Los ficheros TDM son una forma automática de guardar datos de forma jerárquica sin preocuparse del formato, tipo de datos, etc. También combinan un formato estándar como el XML con la eficiencia de los ficheros binarios de forma transparente al usuario. Además también se pueden indexar los datos, de forma que al recuperarlos se pueden filtrar los resultados dependiendo de una condición.

La paleta que contiene estos VI se encuentra en *Programming > File I/O > Storage*. Los seis primeros VIs de esta paleta son Express y también hay una subpaleta con más funciones disponibles.

Figura 4-25. Paleta para el almacenamiento

Como cualquier otra función que accede a ficheros, los de *storage* tienen VIs para abrir, leer-escribir y cerrar. Entre las funciones de leer y escribir están las que trabajan con datos, propiedades y consultas para filtrar los datos.

TDMS es una optimización de TDM para trabajar con datos mediante *streaming*. El VI *Programming > File I/O > TDM Streaming > TDMS File Viewer* es una herramienta que permite ver de una forma sencilla los datos guardados con TDMS.

4.8.1 Ejemplo

En esta ocasión el objetivo será que el VI guarde el estado de dos controles cuando se cierre y lo recupere cuando vuelva a ejecutarse.

El programa de la figura 4-26, comienza abriendo una referencia al fichero TDM, se puede configurar para que si éste no existe lo genere. Una vez abierto ya se pueden leer

los valores almacenados; en primer lugar se selecciona qué dato en concreto se leerá a través de las propiedades. Por otra parte, cuando el VI se cierra (detectado en la estructura EVENT) se tienen dos VIs para guardar los valores que tienen los controles en ese instante en el fichero.

El programa también incluye dos eventos más, no mostrados en la figura, para leer los datos del fichero a petición del usuario y generar una nueva señal en la gráfica.

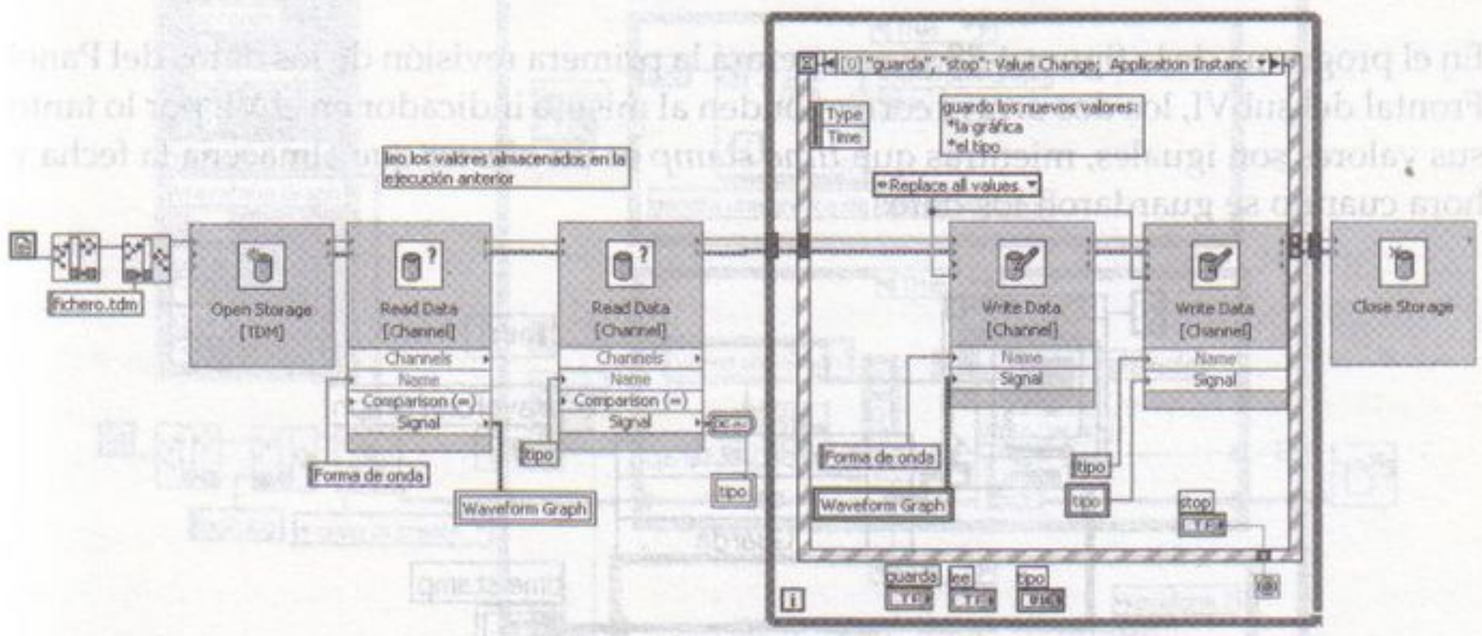


Figura 4-26. Ejemplo de almacenamiento de información

4.9 Datalog

Por *data logging* nos referimos a guardar en un fichero todos los datos relativos al Panel Frontal de un VI en un momento determinado.

Esto puede realizarse de forma manual en el menú *Operate > Data Logging*. Con la opción *Log* se guardan los datos actuales junto con la fecha en el archivo; cada vez que los datos se guardan no se sobrescriben los anteriores sino que se crea una nueva revisión. *Receive* hace lo contrario, lee del fichero y muestra los datos en el Panel Frontal, en ese momento aparecerá una barra en la que se indica el número de revisión y la fecha correspondiente.

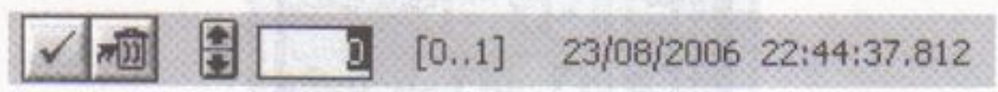


Figura 4-27. Barra de información utilizando *Receive*

Las otras opciones de este menú son borrar una revisión, cambiar el fichero donde se almacenan y borrar todos los datos almacenados.

También se puede recuperar una revisión de un VI desde otro distinto de forma programada. Para esto se añade el primero como subVI del segundo, después se selecciona la opción *Enable Database Access* del menú contextual del icono del subVI, con esto

aparecerá un recuadro alrededor del VI, este recuadro tiene un control en la esquina superior izquierda para indicar el número de revisión a recuperar (0 la primera, 1 la segunda, -1 la última, -2 la penúltima, etc.) y tres indicadores en las otras esquinas que sirven para mostrar si el proceso se ha realizado correctamente, un *cluster* con los datos recuperados y la fecha y hora de cuando se almacenó. También se puede leer (pero no escribir) el valor de los controles e indicadores a través del conector del VI. Un aspecto importante a tener en cuenta es que el subVI no se ejecutará, sólo se accede al fichero de revisiones.

En el programa de la figura 4-28 se recuperará la primera revisión de los datos del Panel Frontal del subVI, los dos *arrays* corresponden al mismo indicador en el VI, por lo tanto sus valores son iguales, mientras que *time stamp* es un *cluster* que almacena la fecha y hora cuando se guardaron los datos.

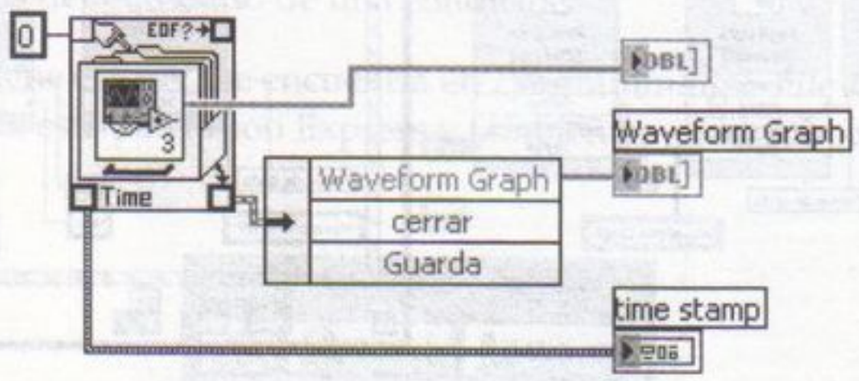


Figura 4-28. Recuperación de la primera revisión

Otra opción de los ficheros *datalog* es operar sobre ellos de forma programada, para ello se dispone de una paleta en *Programming > File I/O > Advanced File Functions > Datalog* con las funciones clásicas: abrir, leer, escribir y cerrar además de otras para indicar el número de revisión.

Figura 4-29. Paleta Datalog

4.9.1 Ejemplo

En la figura 4-30 se ve el uso de estos VIs. En primer lugar se abre una referencia al fichero indicando un *cluster* con los datos que serán guardados, este formato no tiene

por qué ser igual al que se crearía de forma manual (*Operate > Data Logging > Log*), pero tiene que coincidir con el que se usa en ese fichero concreto, es decir, que el *cluster* no debe cambiar su estructura de una escritura a la siguiente. Después ya se pueden usar las funciones para leer y escribir; por defecto la de escribir lo hace al final del fichero (una nueva revisión), en la parte de lectura se ha indicado que ésta sea de la última revisión con *Set Datalog Position*.

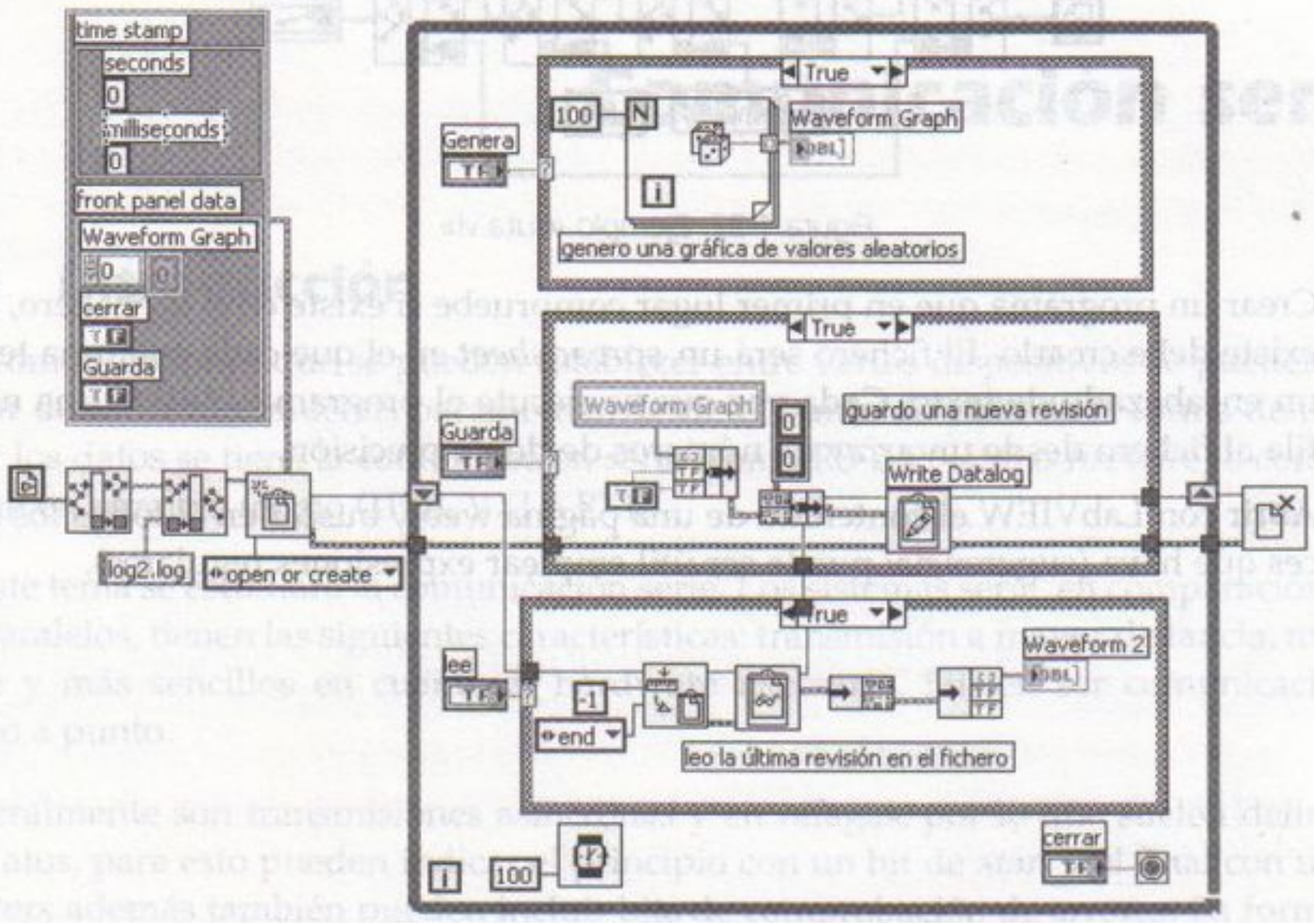


Figura 4-30. Uso de las funciones de la paleta Datalog

4.10 Waveform

Los VIs del menú *Programming > Waveform > Waveform File I/O* están especialmente preparados para trabajar con el tipo de datos *Waveform*.

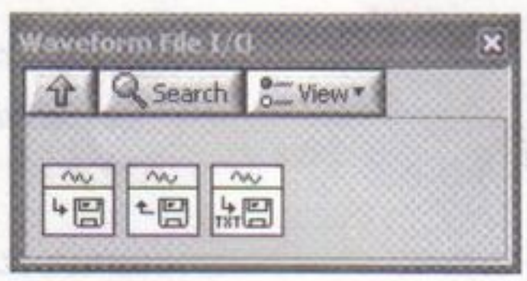


Figura 4-31. Paleta de ficheros Waveform

Estos VIs pueden abrirse y navegar por su jerarquía, si se hace puede comprobarse que están creados a partir de los genéricos estudiados al principio del capítulo. Después de ver los ejemplos de este capítulo, el uso de estos VIs no debe suponer ningún problema.

por qué ser igual al que se crearía de forma manual (*Operate > Data Logging > Log*), pero tiene que coincidir con el que se usa en ese fichero concreto, es decir, que el *cluster* no debe cambiar su estructura de una escritura a la siguiente. Después ya se pueden usar las funciones para leer y escribir; por defecto la de escribir lo hace al final del fichero (una nueva revisión), en la parte de lectura se ha indicado que ésta sea de la última revisión con *Set Datalog Position*.

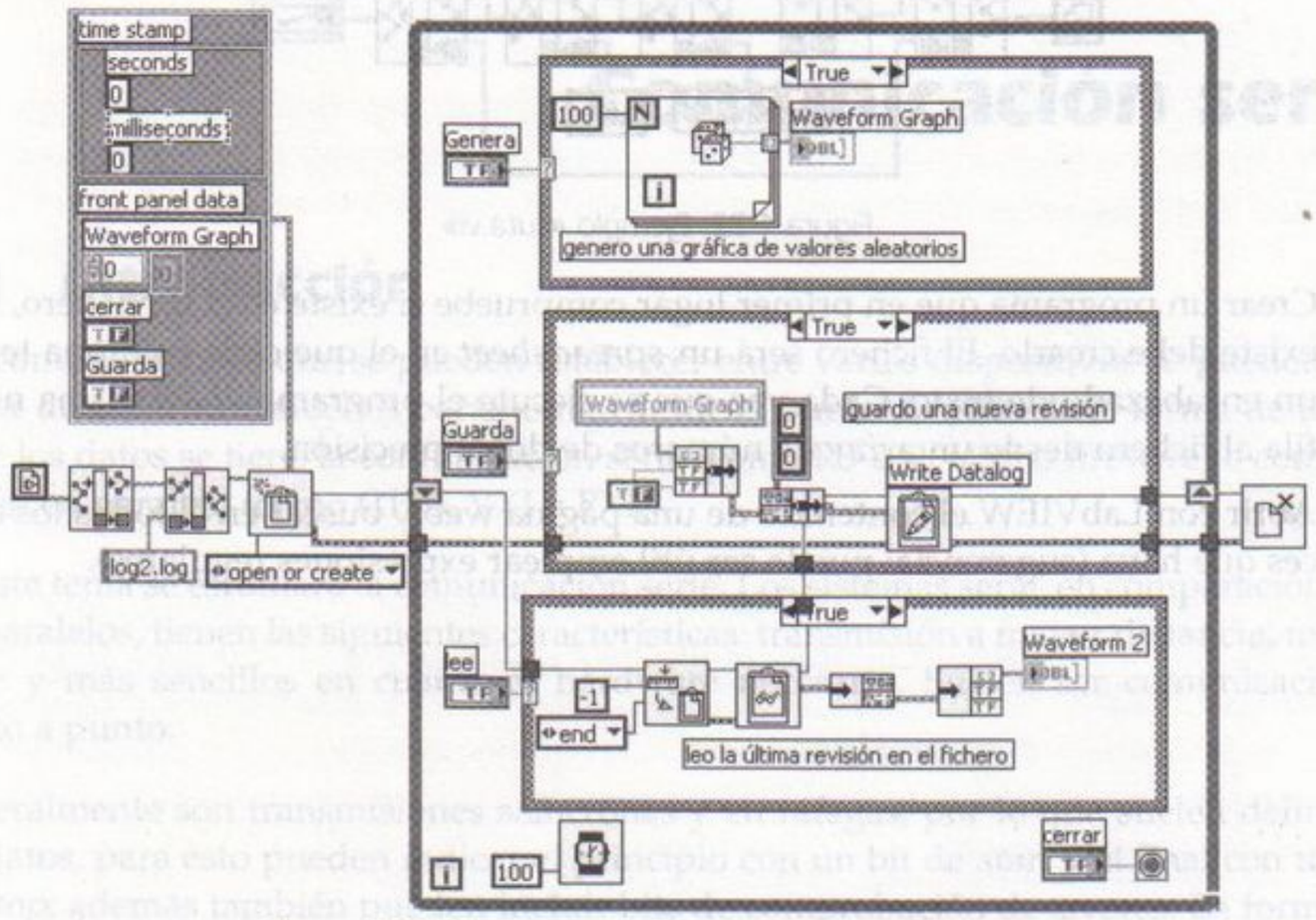


Figura 4-30. Uso de las funciones de la paleta Datalog

4.10 Waveform

Los VIs del menú *Programming > Waveform > Waveform File I/O* están especialmente preparados para trabajar con el tipo de datos *Waveform*.

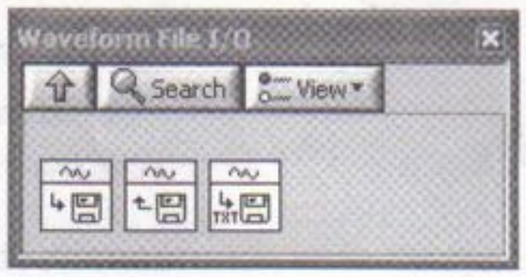


Figura 4-31. Paleta de ficheros Waveform

Estos VIs pueden abrirse y navegar por su jerarquía, si se hace puede comprobarse que están creados a partir de los genéricos estudiados al principio del capítulo. Después de ver los ejemplos de este capítulo, el uso de estos VIs no debe suponer ningún problema.

4.11 Ejercicios

1. Si el VI de la figura 4-32 se llama «ruta.vi» está situado en C:\Archivos de programa\National Instruments\LabVIEW 8.2\user.lib, ¿qué valor tendrá el indicador de la figura?

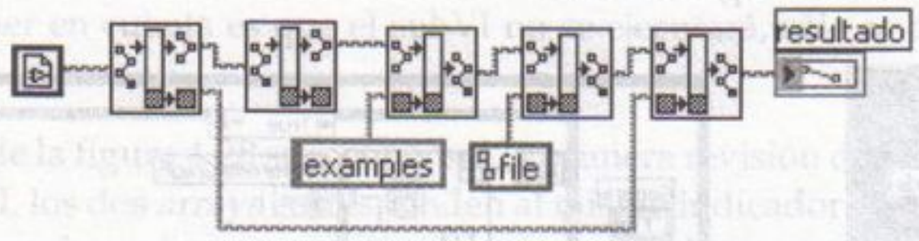


Figura 4-32. Ejemplo «ruta.vi»

2. Crear un programa que en primer lugar compruebe si existe o no un fichero, si no existe debe crearlo. El fichero será un *spreadsheet* en el que cada columna tendrá un encabezado de texto. Cada vez que se ejecute el programa añadirá una nueva fila al fichero desde un *array* de números de doble precisión.
3. Abrir con LabVIEW el contenido de una página web y buscar en él todos los enlaces que haya (sugerencia: puede ser útil emplear expresiones regulares).

Capítulo 5

Comunicación serie

5.1 Introducción

Las comunicaciones que se pueden establecer entre varios dispositivos se pueden clasificar de acuerdo a muchos parámetros, desde el punto de vista de la forma de transmitir los datos se tiene la comunicación serie (como RS-232, USB o *firewire*) o comunicación en paralelo (como GPIB, VXI o PXI).

En este tema se estudiará la comunicación serie. Los sistemas serie, en comparación con los paralelos, tienen las siguientes características: transmisión a mayor distancia, menor coste y más sencillos en cuanto al hardware necesario. Suelen ser comunicaciones punto a punto.

Generalmente son transmisiones asíncronas y en ráfagas, por lo que suelen delimitar los datos, para esto pueden indicar el principio con un bit de *start* y el final con un bit de *stop*; además también pueden incluir bits de comprobación de errores. La forma de comprobar errores más sencilla es añadir un bit de paridad, esta paridad puede ser par o impar, por ejemplo, si el dato es 0100111 y se trabaja con paridad impar, el bit de paridad valdrá 1 (para tener un número de bits impar en la transmisión), así el dato transmitido será 01001111, donde el último bit es el de paridad.

Una trama son todos los bits transmitidos, desde el de *start* al de *stop*. Las características principales de una trama son:

- La velocidad, medida en baudios.
- El número de bits de datos.
- El tipo de paridad.

Algunos de los estándares de comunicación serie más utilizados son RS-232, RS-422 y RS-485.

5.2 Historia

El RS-232 se diseñó en 1962 y desde entonces ha tenido varias revisiones, la última es la «F» de 1997. Comenzó a ser usado para la comunicación entre un módem y una

impresora de teletipo. La *Electronic Industries Association* (EIA) hizo una importante revisión del estándar en 1969 y creó el RS-232-C.

Desde entonces el RS-232 se usó en ordenadores, impresoras, instrumentos de test y en casi cualquier dispositivo electrónico que necesitara comunicarse.

El estándar ha sido renombrado varias veces de acuerdo con la organización que hacía una nueva revisión, así se le conoce como EIA RS 232, EIA 232, TIA 232. También es muy parecido al V.24.

Algunos fabricantes decidieron no seguir la norma y crearon terminales con otra asignación de pines o distintos usos de las líneas, lo que es causa de problemas de compatibilidad entre dispositivos.

5.3 RS232

RS-232 significa *Recomend Standard* 232, está definido por el ANSI (*American National Standard Institution*) como «la interfase entre un equipo terminal de datos y un equipo de comunicación de datos utilizando un intercambio binario en modo serie».

Los dispositivos son llamados DTE *Data Terminal Equipment* (por ejemplo un ordenador) y DCE *Data Circuit-terminating Equipment* (por ejemplo un módem). El DTE usa un terminal macho mientras que al DCE se conecta un terminal hembra y es el encargado de establecer y mantener la conexión. Un equipo puede necesitar ser DTE y DCE, por ejemplo un ordenador será el DTE al comunicarse con un módem y un DCE al hacerlo con una impresora.

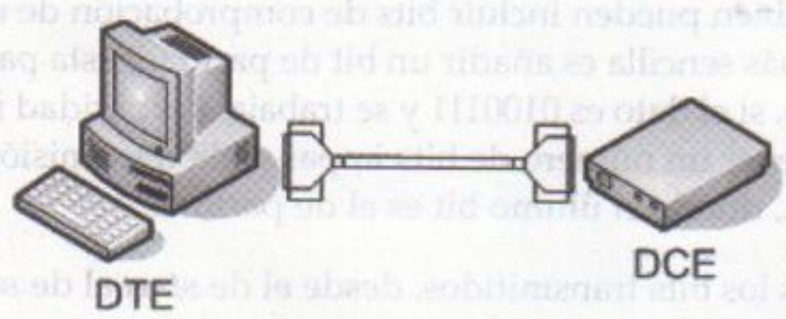


Figura 5-1. Dispositivos en una comunicación serie: DTE y DCE

Las características más importantes del estándar son:

- Velocidad máxima original era 20 kbps, hay aplicaciones que llegan a 116 kbps.
- Longitud máxima del cable de 15 m (revisión «C»).
- Tensión en modo común máxima de ±25 V.
- Impedancias de 3 a 7 k?.
- Modo de operación simple.
- Un emisor y un receptor.
- Transmisión asíncrona o síncrona (usando líneas extra para el reloj).

En una transmisión asíncrona las tramas siguen el esquema básico de la 5-2.

Start	Datos (de 5 a 8 bits)	Paridad	Stop (1 o 2 bits)

Figura 5-2. Trama típica

5.3.1 Conectores

El RS-232 puede utilizar varios tipos de conectores siendo los más usados los de 25 pines, de 68 pines, *Modular Jack* de 10 pines, RJ45 de 8 pines y los de 9 pines. Los conectores que tienen muchos pines implementan varios canales secundarios y también tienen otras señales de temporización para transmisiones síncronas. El conector DB-9 es el más habitual, consta de 5 líneas de recepción y 3 de transmisión. Las líneas se listan en la Tabla 1.

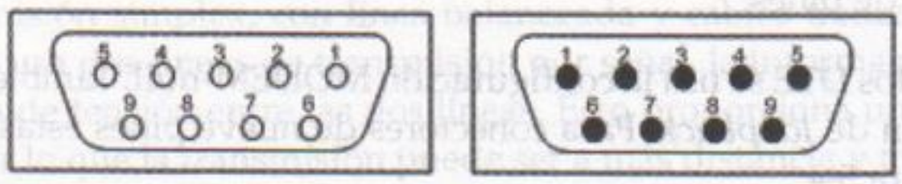


Figura 5-3. Conectores DB-9

Tabla 1. Señales en RS-232

Pin DB25	Pin DB9	Nombre	Descripción
8	1	DCD	Data Carrier Detect
6	6	DSR	Data Set Ready
3	2	RD	Receive Data Line
4	7	RTS	Request To Send
2	3	TD	Transmit Data Line
5	8	CTS	Clear To Send
20	4	DTR	Data Terminal Ready
22	9	RI	Ring Indicador
7	5	GND	Common Ground

DCD: el DCE la pone a 1 para indicar que está recibiendo una señal portadora.

DSR: el DCE la pone a 1 para indicar al DTE que está conectado a la línea.

RD: la entrada de datos. Si DCD=0 debe estar en un estado llamado Mark.

RTS: el DTE pone a 1 para indicar que puede transmitir datos. El DCE pondrá a 1 la línea CTS para recibir los datos. Al acabar la transmisión RTS pasa a 0 y el DCE pone CTS también a 0.

TD: salida de datos (del DTE al DCE). Si no se transmite estará en Mark. DSR, DTR, RTS y CTS deben estar a 1 para transmitir.

CTS: el DCE la pone a 1 para indicar que está preparada para recibir datos. Si RTS, DSR y DTR están a 1, CTS también se pone a 1.

DTR: el DTE la pone a 1 para indicar que puede recibir o quiere transmitir datos. El DTE pone un 0 para finalizar la transmisión.

RI: el DCE la pone a 1 cuando está recibiendo una llamada.

GND: tensión de referencia. Debe estar aislada de la toma de tierra del equipo.

Las líneas RTS, CTS, DSR, DCD y DTR se usan para implementar un protocolo de *handshake*. Puede ocurrir que en una transmisión no se usen todas las líneas, para establecer una comunicación solamente hacen falta GND, TD y RD, otros sistemas como por ejemplo un ratón de ordenador sólo usa GND, RI, TD y RD. También puede darse el caso que algunas aplicaciones usen las líneas para otros propósitos o que no respeten la asignación de pines.

Para comunicar dos DTE se usa la configuración MODEM-null. También se puede crear una configuración de *loopback*. Para conectores de nueve pines estas configuraciones son las de la figura 5-4.

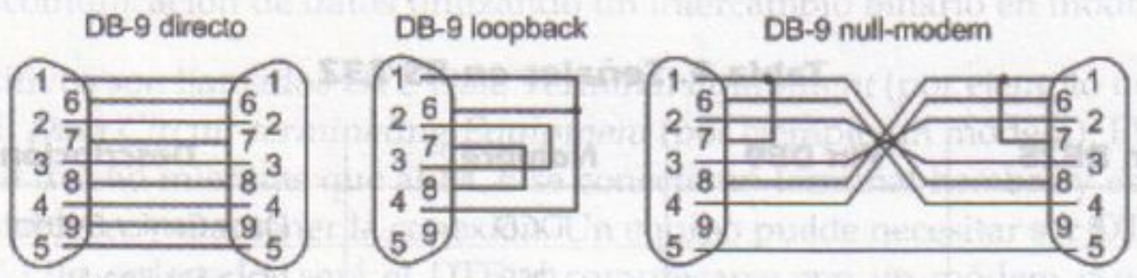


Figura 5-4. Configuraciones habituales de cables

5.3.2 Puertos

Los puertos constan de un registro donde se escriben los bits para ir mandándolos y otro registro donde se almacenan los que llegan.

Para enviar y recibir tramas empleando una transmisión asíncrona se usa un circuito llamado UART (*Universal Asyncronous Receiver/Transceiver*). Una vez que la UART está programada, la comunicación se reduce a leer y escribir bytes en ella. La principal función de la UART es hacer la conversión de serie a paralelo.

Los buffers son un conjunto de registros en los que se almacenan los bytes que deben ser transmitidos o que son recibidos hasta ser leídos. Cuando no se usan todas las líneas podría aparecer un error si se sobrepasan los límites (*overflow*) de los buffers, por lo que se perderían datos.

5.3.3 Handshake

Los protocolos de comunicación deben evitar que los buffers se saturen. Para esto se puede indicar al transmisor que debe parar de enviar datos cuando el buffer de entrada del receptor esté lleno. Este protocolo puede ser hardware o software (o no existir).

LabVIEW llama *Hardware Handshaking* a usar las líneas. RTS, CTS, DSR, DCD y DTR como se han indicado antes para evitar la saturación del buffer.

Un protocolo de software para evitar la saturación es el XON/XOFF. Cuando el buffer de recepción está cerca de saturarse, se envía al emisor el carácter XOFF (valor 19 en decimal o 13 en hexadecimal). Entonces el emisor debe detener la transmisión. Cuando vuelve a haber espacio en el buffer de entrada del receptor, éste envía XON (17 o 11h) para reanudar la transmisión. Si el emisor y el receptor están configurados para usar XON y XOFF, estos caracteres no pueden usarse en la transmisión porque serán interpretados como comandos del protocolo.

5.3.4 Otros estándares

Hay varios estándares derivados del RS-232. El estándar RS-422 se definió también para una transmisión simplex, con línea balanceada y cables trenzados. La transmisión balanceada usa dos líneas de transmisión por señal, la información va codificada por la diferencia de tensión entre las dos líneas. Esto proporciona una mayor inmunidad al ruido, por lo que la transmisión puede ser a más distancia y más velocidad que con RS-232.

El estándar RS-485 es una actualización del RS-422 que permite comunicación bidireccional, multipunto y con más dispositivos.

Las líneas en estos dos estándares se pueden ver en la Tabla 2, compárese con la Tabla 1 que mostraba las líneas del RS-232.

Tabla 2 - Señales de RS-422 y RS-485

Pin DB25	Pin DB9	Nombre	Descripción
8	1	GND	*Common Ground*
3	2	CTS+	*Clear to Send*
2	3	RTS+	*Ready to Send*
20	4	RX+	*Received Data*
7	5	RX-	*Received Data*
6	6	CTS-	*Clear to Send*
7	7	RTS-	*Ready to Send*
5	8	TX+	*Transmitted Data*
22	9	TX-	*Transmitted Data*

- Datos: TX+, TX-, RX+, RX-.
- Handshake: RTS+, RTS-, CTS+, CTS-.
- Referencia: GND.

La Tabla 3 muestra una comparativa de estas tecnologías.

Tabla 3 - Comparativa de los tres estándares serie

Parámetro	RS-232	RS-422	RS-485
Modo de operación	simple	diferencial	diferencial
Nº de dispositivos	1 emisor 1 receptor	1 emisor 10 receptores	32 emisores 32 receptores
Longitud del cable	15 m	1.200 m	1.200 m
Velocidad máxima	20 kbps	10 Mbps	10 Mbps
Carga driver	de 3 a 7 k?	100 ? mínimo	60 ? mínimo
Entrada receiver	de 3 a 7 k?	4 k?	12 k?
Tensión modo común	±25 V	±7 V	de -7 a 12 V
Salida	de 5 a 25 V	de 2 a 6 V	de 1,5 a 6 V

5.4 Comunicaciones serie en LabVIEW

Las versiones antiguas de LabVIEW incluían unos VIs específicos para manejar el puerto serie, pero en las nuevas versiones se han integrado con el resto de VIs de la librería VISA. Los antiguos VIs todavía se conservan por razones de compatibilidad, aunque ofrecen menos posibilidades que los nuevos y su uso está desaconsejado, estos VIs pueden verse en la figura 5-5.

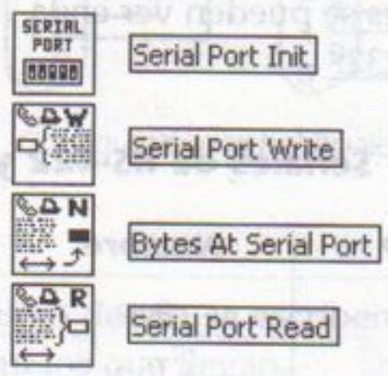
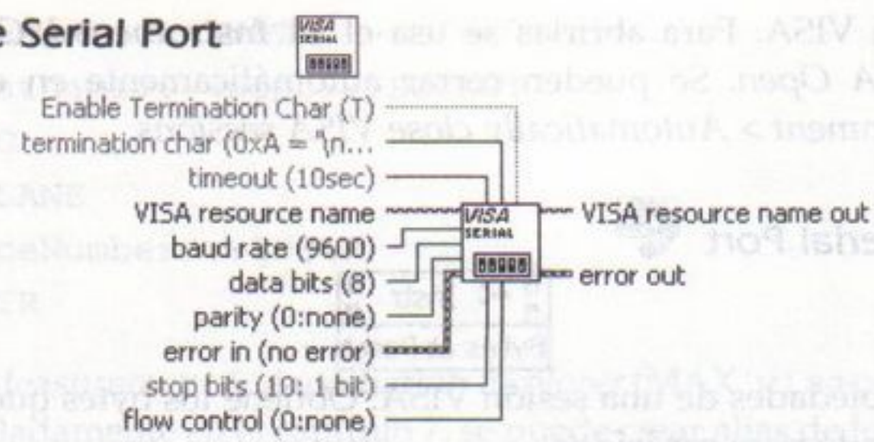


Figura 5-5. Antiguos VIs para comunicación serie

La nueva paleta para el puerto serie está en *Instrument I/O > Serial.*

Figura 5-6. Menú Serial

VISA Configure Serial Port

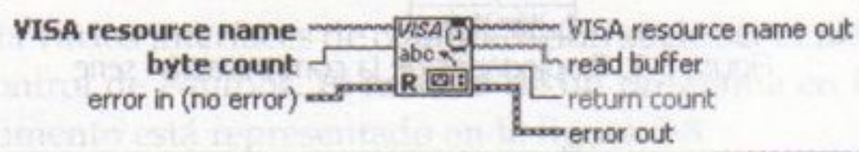


Configura el puerto serie con todos los parámetros que se han visto en la parte teórica de este capítulo: velocidad, protocolo, paridad, etc. El puerto se selecciona con *VISA resource name*, al crear un control o una constante en el Diagrama de Bloques o en el Panel Frontal se escanean los puertos disponibles en el ordenador y se muestran en una lista. *VISA resource name* es un identificador lógico único que sirve para comunicarse con un recurso manteniendo una sesión en la que se pueden realizar varias operaciones. Los nombres de los recursos, habitualmente, siguen el formato `Interface Type[board index]::Address::INSTR` (para comunicación serie asíncrona Interface vale ASRL y no se usa Address), aunque se puede usar un alias. Para crear alias se puede usar el programa MAX. Sería el equivalente a un *refnum* de los VIs que trabajan con ficheros, de aquí en adelante se llamaran *handler* o manejadores. Para más información sobre la configuración se puede consultar la ayuda.

VISA Write

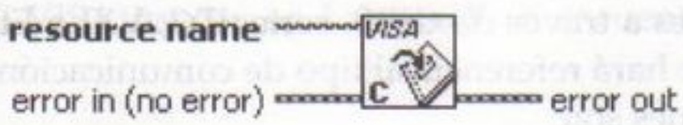
Escribe datos desde el buffer hasta el otro dispositivo. Se puede hacer que este VI y el siguiente utilicen una la transmisión síncrona o asíncrona, para cambiarlo hay que presionar con el botón derecho del ratón sobre el VI y seleccionar *Do I/O Synchronously/Asynchronously*, al hacerlo aparecerá o desaparecerá un pequeño reloj en la esquina superior derecha del icono.

VISA Read



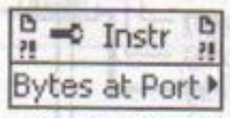
Es el contrario al anterior. Hay que indicarle el número de bytes que debe leer del buffer, para darle un valor a este parámetro suele usarse la salida de la propiedad *VISA Bytes at Serial Port*.

VISA Close



Cierra una sesión VISA. Para abrirlas se usa el VI *Instrument I/O > VISA > VISA Advanced > VISA Open*. Se pueden cerrar automáticamente en el menú *Tools > Options > Environment > Automatically close VISA sessions.*

VISA Bytes at Serial Port



Es una de las propiedades de una sesión VISA. Obtiene los bytes que hay en el buffer del puerto esperando para ser leídos.

A parte de *Bytes at Port* hay otras propiedades, mediante ellas se puede leer y cambiar la configuración del puerto. Hay un total de 16 propiedades para el puerto serie como puede verse en la figura 5-7, también pueden ser útiles las propiedades de la clase *Modem Line Settings.*

Para crear una propiedad se debe ir a un control, en este caso al *VISA resource name*, y en el menú desplegable ir a *Create > Property*. También pueden crearse en la paleta *Programming > Aplication Control > Property Node*, cableándolo de forma adecuada y seleccionando la propiedad que se desea. En la figura 5-7 se muestran dos nodos *Property* con varias propiedades relacionadas con el puerto serie.

Otras funciones que podrían ser útiles para el manejo del puerto serie, como definir el tamaño de los buffers, se encuentran en las paletas de VISA.

Figura 5-7. Propiedades de la comunicación serie

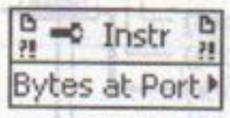
5.4.1 VISA

VISA (*Virtual Instrument Software Architecture*) es un API o librería desarrollada por varios fabricantes de equipos que proporciona un estándar software para las operaciones de lectura y escritura en instrumentación.

NI-VISA es la implementación de National Instruments de este estándar, puede establecer comunicaciones a través de GPIB, serie, PXI, VXI o Ethernet. En las direcciones de los dispositivos se hará referencia al tipo de comunicación y al dispositivo. Algunos ejemplos de direcciones son:

Cierra una sesión VISA. Para abrirlas se usa el VI *Instrument I/O > VISA > VISA Advanced > VISA Open*. Se pueden cerrar automáticamente en el menú *Tools > Options > Environment > Automatically close VISA sessions.*

VISA Bytes at Serial Port



Es una de las propiedades de una sesión VISA. Obtiene los bytes que hay en el buffer del puerto esperando para ser leídos.

A parte de *Bytes at Port* hay otras propiedades, mediante ellas se puede leer y cambiar la configuración del puerto. Hay un total de 16 propiedades para el puerto serie como puede verse en la figura 5-7, también pueden ser útiles las propiedades de la clase *Modem Line Settings.*

Para crear una propiedad se debe ir a un control, en este caso al *VISA resource name*, y en el menú desplegable ir a *Create > Property*. También pueden crearse en la paleta *Programming > Aplication Control > Property Node*, cableándolo de forma adecuada y seleccionando la propiedad que se desea. En la figura 5-7 se muestran dos nodos *Property* con varias propiedades relacionadas con el puerto serie.

Otras funciones que podrían ser útiles para el manejo del puerto serie, como definir el tamaño de los buffers, se encuentran en las paletas de VISA.

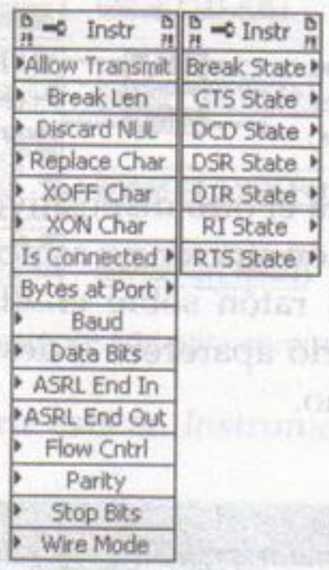


Figura 5-7. Propiedades de la comunicación serie

5.4.1 VISA

VISA (*Virtual Instrument Software Architecture*) es un API o librería desarrollada por varios fabricantes de equipos que proporciona un estándar software para las operaciones de lectura y escritura en instrumentación.

NI-VISA es la implementación de National Instruments de este estándar, puede establecer comunicaciones a través de GPIB, serie, PXI, VXI o Ethernet. En las direcciones de los dispositivos se hará referencia al tipo de comunicación y al dispositivo. Algunos ejemplos de direcciones son:

```
TCPIP::<hostname>::INSTR
TCPIP::<hostname>::<port>::SOCKET
GPIB::INTFC
VXI::BACKPLANE
PXI::<DeviceNumber>::INSTR
ASRL1::INSTR
```

En el programa *Measurement & Automation Explorer* (MAX, `visaconf` en UNIX) que se verá más detalladamente en el capítulo 7, se puede crear alias de los dispositivos instalados en *My System > Software > NI-VISA > VISA Options*. De esta forma en lugar de usar, por ejemplo, `ASRL1::INSTR` se utilizará `COM1`.

NI-VISA viene acompañado de una serie de utilidades como:

- *VISA Driver Development Wizard* (DDW): asistente que permite crear un driver para un dispositivo PXI, PCI, USB o firewire que, al instalarlo en Windows, habilita a NI-VISA para acceder a ese dispositivo.
- *VISA Interactive Control* (VISAIC): es un asistente para probar de una forma rápida y sencilla la conectividad y funcionalidad de los dispositivos mediante las funciones de NI-VISA.
- *VISA Server*: es un servidor que permite controlar de forma remota un equipo a través de VISA. El servidor debe ser ejecutado en la máquina donde esté conectado el equipo. En la máquina cliente se podrán encontrar los equipos instalados dando el valor `visa://[servidor]/[instrumento]` al terminal *VISA resource name*.
- *NI-Spy*: se trata de un programa para ayudar a la depuración en bajo nivel. Permite capturar las llamadas que se realizan a las funciones del API y las respuestas que producen.

Mediante NI-VISA también se puede acceder a dispositivos de otros fabricantes creando e instalando un driver a través de *VISA Driver Development Wizard*. Para testear el funcionamiento de la comunicación se puede emplear MAX o *VISA Interactive Control*.

Como VISA soporta varios interfaces de comunicación suele ser el método elegido para crear drivers de control de equipos. El modelo de un programa en LabVIEW para un driver de un instrumento está representado en la figura 5-8.

Hay un asistente en *File > New... > Project > Instrument Driver Project* o en *Tools > Instrumentation > Create Instrument Driver Project* que crea el esqueleto de un proyecto en el que se puede programar el driver de un instrumento usando NI-VISA y siguiendo el esquema anterior.

Las funciones del API NI-VISA usadas en LabVIEW se encuentran en el menú *Instrument I/O > VISA*.

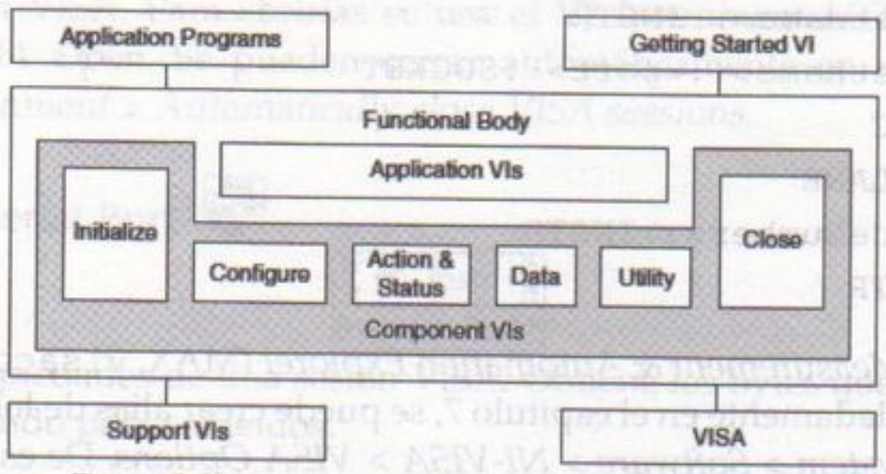


Figura 5-8. Esquema de los drivers de instrumentos

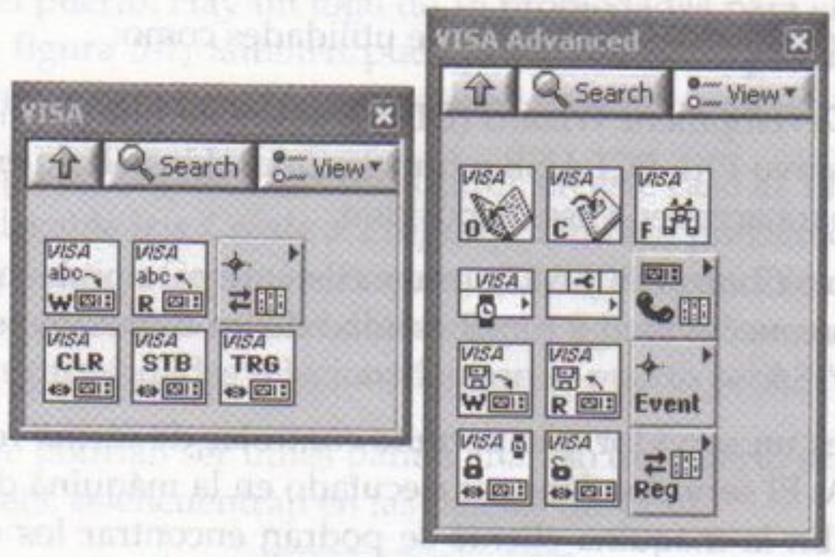


Figura 5-9. Menú VISA

Hay docenas de VIs dedicados a VISA, por eso no se estudiarán todos en detalle, simplemente decir que los típicos son *VISA Open, VISA Write, VISA Read* y *VISA Close*.

Las direcciones deben indicarse en un control o constante de tipo *VISA resource name*. Este control o constante puede cambiarse de clase presionando con el botón secundario del ratón sobre una constante o control y yendo al menú *Select VISA Class*. Cada clase tendrá unas propiedades y métodos diferentes.

5.5 Ejemplos

5.5.1 Ejemplo I: Comunicación con un microcontrolador

5.5.1.1 Explicación teórica

En el primer ejemplo con RS-232 se crearán dos programas que se comunicarán entre sí. Uno de los programas se ejecutará en un PC y estará desarrollado con LabVIEW. El segundo de los programas se implementará en un microcontrolador y por sencillez se programará usando un lenguaje de alto nivel.

El programa del PC enviará por el puerto serie dos números de un carácter (del 0 al 9) y un signo de operación («+» ó «-»). El programa en el microcontrolador lo leerá, rea-

lizará la operación indicada y el resultado lo enviará de vuelta por el puerto RS-232. Finalmente el programa del PC leerá el resultado y lo mostrará en su Panel Frontal.

5.5.1.2 Código del microcontrolador

El siguiente código es un pequeño programa en lenguaje C para un microcontrolador PIC18F8720 de Microchip compilado con *CCS Compiler*. El programa esperará a que llegue un byte por el puerto serie, cuando esto ocurre se activa el flag `kbhit`. El byte llegado se lee con `getchar`, éste puede ser uno de los dos números que se guardarán en las variables «a» y «b» o puede indicar la operación a realizar. Cuando se realiza la operación se enviará el resultado por el interfaz serie con la función printf.

```
/*************************************************************/
/*                Programa para el PIC 18F8720               */
/*                Lee un dato del puerto serie               */
/*                y lo muestra por un LCD.                   */
/*************************************************************/
#include <18f8720.h>
#FUSES HS,NOWDT,NOPROTECT,PUT,BROWNOUT,NOLVP   //fusibles
#use DELAY(CLOCK=20000000)                     //reloj del PIC
#use RS232 (BAUD=9600, XMIT=PIN_C6, RCV=PIN_C7)      //config RS232
/*************************************************************/
/*                          MAIN                             */
/*************************************************************/
void main(){
    char c;
    int a, b, operacion, resultado, estado=0;
/**********************PROGRAMA CALCULADORA******************/
    while(1){
        while( !kbhit() );                     //espera un caracter
        if ( estado==0 ){
            a=getchar()-0x30;                  //primer numero
            estado=1;
        }else if( estado==1 ){
            operacion=getchar();                     //operacion
            estado=2;
        }else if( estado==2 ){
            b=getchar()-0x30;                  //segundo numero
            estado=0;
            if ( operacion=='+' ) resultado=a+b;
            else if ( operacion=='-' ) resultado=a-b;
            else break;
            printf("%D %c %D = %D", a, operacion, b, resultado);
        }else estado=0;
    }
}
```

5.5.1.3 Código en LabVIEW

En el código primero se configura el puerto serie, la configuración debe ser igual que la del microcontrolador (directiva `#use RS232`). El siguiente paso escribirá un *string* que contendrá la operación a realizar, por ejemplo «1+1». Después se introduce un retardo de 50 ms para dar tiempo al microcontrolador a realizar la operación y enviar el resultado por el puerto RS-232; este método es una opción muy simple y habitual para comunicarse con dispositivos lentos. Después del retardo se utiliza la propiedad *Bytes at Port* para indicarle a *VISA Read* el número de bytes que tiene que leer, el *string* leído es el resultado que devuelve el microcontrolador.

El código puede verse en la figura 5-10.

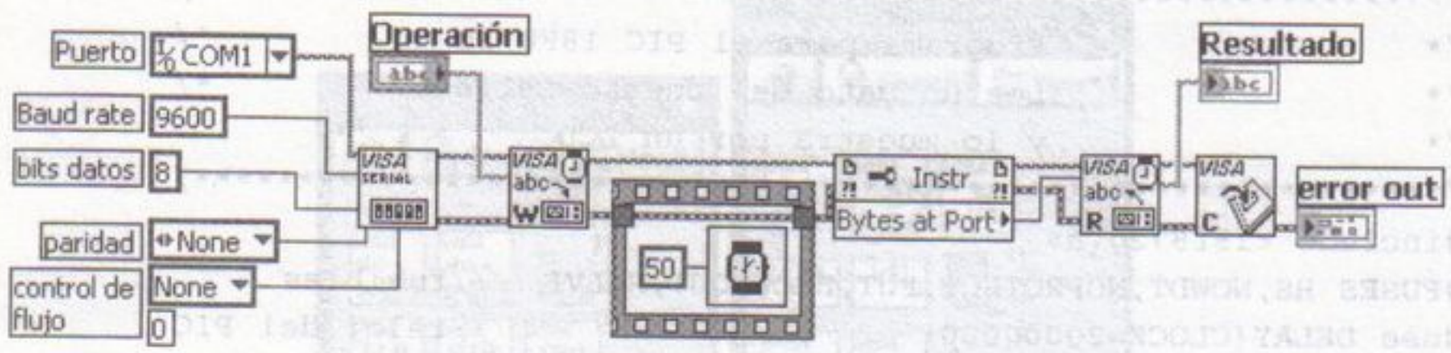


Figura 5-10. Configurar, leer y escribir por el puerto serie

5.5.1.4 Resultado

Una vez programado el microcontrolador y conectado al PC ya puede ejecutarse el programa de LabVIEW. El Panel Frontal del resultado de este VI se puede ver en la figura 5-11.

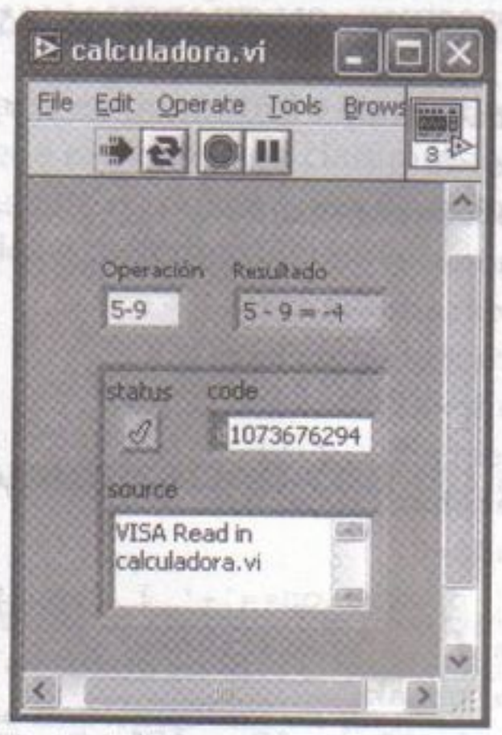


Figura 5-11. Resultado de la comunicación con un microcontrolador

5.5.2 Ejemplo II: Control de un instrumento de laboratorio

5.5.2.1 Explicación teórica

Los instrumentos de laboratorio normalmente tienen varios interfaces para que se puedan comunicar con otros dispositivos. Estos interfaces suelen ser el puerto serie y el bus GPIB, además últimamente también se añade un conector RJ45 para ethernet o un USB.

Enviando las órdenes adecuadas sobre este interfaz se puede acceder a cualquier parámetro del instrumento, leer datos o cambiar cualquier configuración, incluso las relativas al mismo interfaz.

En este ejemplo se desea crear un barrido de frecuencias con el generador de funciones. El generador debe proporcionar una salida senoidal de frecuencia entre 1 Hz y 10 MHz en potencias de 10, es decir 1 Hz, 10 Hz, 100 Hz, 1 kHz...

Para poner el generador de funciones (HP33120A) en modo RS-232 se debe acceder a su menú de configuración: *Shift > Menu > I/O Menu > Interface > RS-232 > Enter.*

Las órdenes se verán más detenidamente en el próximo capítulo. En este ejemplo se muestran a continuación los comandos enviados con su descripción:

`*IDN?`	Pide una identificación al equipo.
`APPL:SIN 1,1,0`	Aplica una señal senoidal de 1 V y 1 Hz.
`FREQ XX`	Cambia la frecuencia de la señal a XX Hz.

5.5.2.2 Código

El código consta de dos VIs, uno de ellos será el que envíe los comandos a través del puerto, éste será un subVI; el VI principal*f* calculará las frecuencias, compondrá los comandos adecuados y usará el subVI anterior para enviar los comandos al instrumento.

El código del VI para enviar comandos al generador a través del puerto serie se muestra en la figura 5-12, en ella se ha incluido un retardo entre el envío de comandos y las respuestas del equipo (en caso que las hubiera) para dar tiempo a que el generador pueda recibir, procesar y responder a la orden dada.

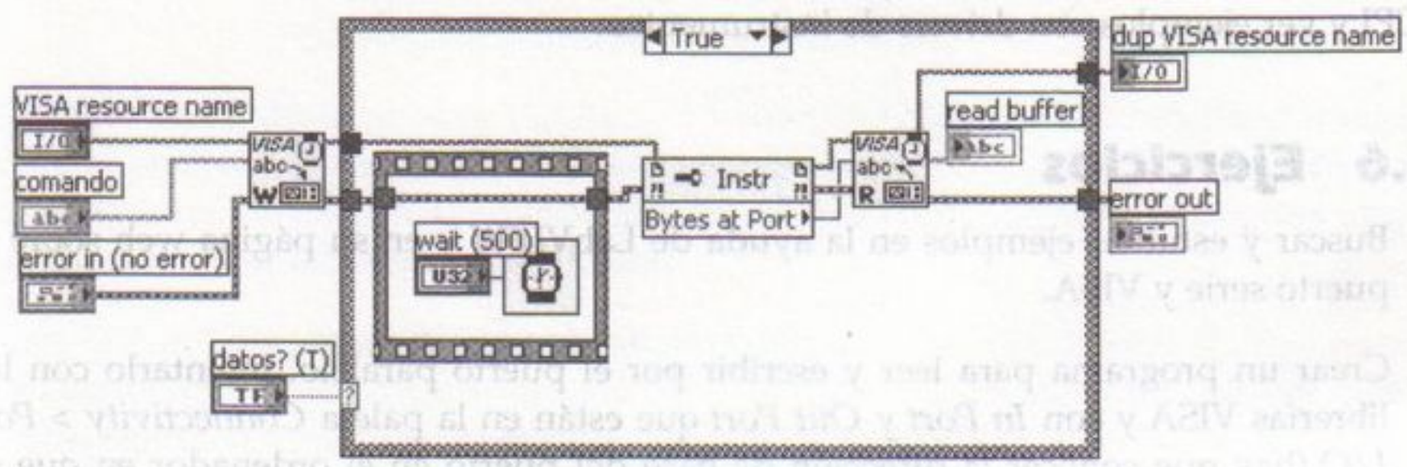


Figura 5-12. SubVI para leer y escribir por el puerto serie

En este VI se han asociado convenientemente los controles e indicadores con los terminales en el icono para que pueda ser usado como subVI.

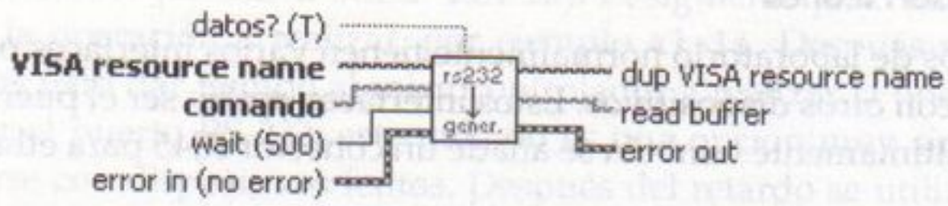


Figura 5-13. Terminal del subVI anterior

Dentro del VI principal*f* se usará el anterior, las funciones del nuevo VI serán la configuración inicial del interfaz de comunicaciones y la configuración del equipo. Después entrará en un bucle FOR en el cual calculará la nueva frecuencia de la señal y le enviará el comando correspondiente al generador. El código es el de la figura 5-14.

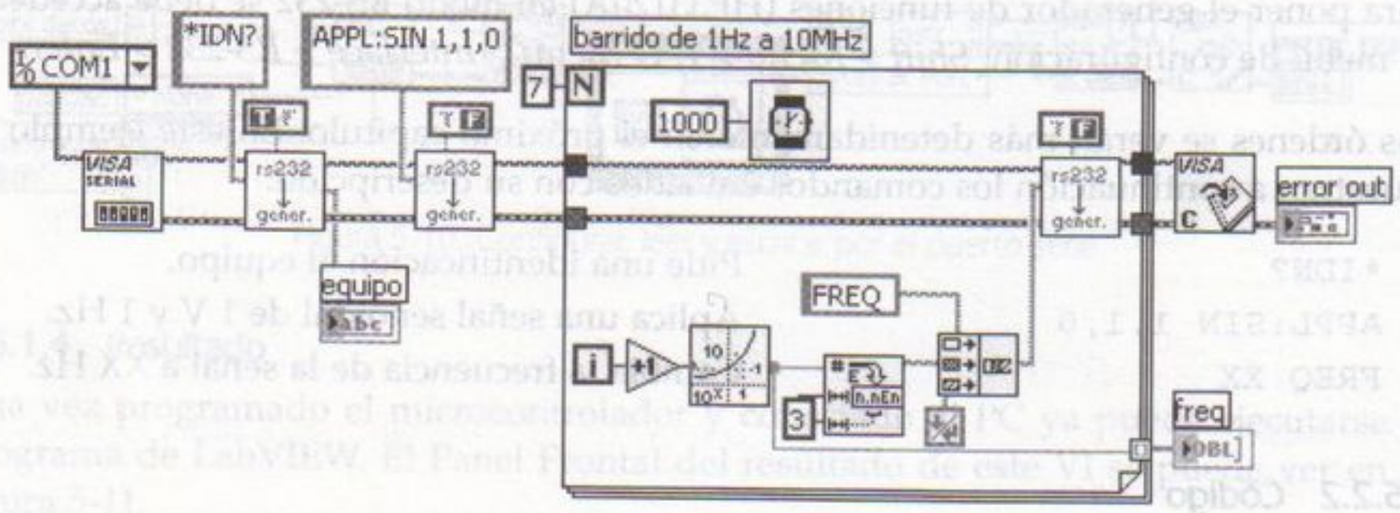


Figura 5-14. Comunicación con el Agilent 33120A a través de un interfaz serie

5.5.2.3 Resultado

Para comprobar el resultado hay que conectar un cable serie al COM1 entre el ordenador y el generador, además se debe configurar el generador para que use el interfaz RS-232. Una captura de pantalla del resultado se puede ver en la figura 5-15.

En este ejemplo se ha visto la forma de controlar un equipo a través del puerto serie. El control de instrumentos será ampliado en el tema sobre GPIB al explicar los comandos SCPI y ver ejemplos con drivers de instrumentos.

5.6 Ejercicios

1. Buscar y estudiar ejemplos en la ayuda de LabVIEW y en su página web sobre el puerto serie y VISA.

2. Crear un programa para leer y escribir por el puerto paralelo. Intentarlo con las librerías VISA y con *In Port* y *Out Port* que están en la paleta *Connectivity > Port I/O* (hay que conocer la dirección de base del puerto en el ordenador en que se

Figura 5-15. Resultado del programa anterior

usará el programa, se puede ver en el administrador de dispositivos, normalmente es 378h). Para Windows 2000 o XP hace falta AccessHW si se usa *In* y *Out Port*.

5.7 Bibliografía

Dan Mondrik, *Advanced Software Techniques for Instrument Control*, National Instruments, 2000.

National Instruments, *Serial Hardware and Software for Windows*, 2000.

National Instruments, *VISA Help*, 2004.

Capítulo 6

Bus de comunicaciones GPIB

6.1 Introducción

Una de las claves de la instrumentación virtual es la comunicación. Un bus de instrumentación es una forma de interconectar varios equipos electrónicos destinados a realizar medidas o tests de forma conjunta. Algunos ejemplos de estos equipos pueden ser osciloscopios, fuentes de alimentación, generadores de funciones, etc. Normalmente todos están controlados por un ordenador, el cual los programa, procesa los resultados y ofrece un interfaz al usuario final.

GPIB es el bus de instrumentación más popular, aceptado por muchos fabricantes como Agilent, Cec, Iotech, Keithley, Mcc, NI... En los últimos años los fabricantes de equipos tienden a incluir otros tipos de interfaz de comunicaciones como ethernet o USB aunque estos tipos, al ser más genéricos, carecen de algunas características deseables que sí tiene GPIB al ser específico para instrumentación. Otros tipos de instrumentos usan buses planos (de bastidor) en lugar de cableados como PXI, VME o VXI.

Figura 6-1. Cable GPIB y controladora GPIB con interfaz USB

El primer bus a destacar es el HP-IB de Hewlett-Packard (*Hewlett-Packard Interface Bus*) que se desarrolló en 1965. En aquella época HP-IB fue muy popular para conectar dispositivos como impresoras a ordenadores, constaba de dieciséis líneas y una velocidad máxima de transferencia de 1 MBps.

Ya en 1975 el IEEE, basándose en HP-IB, propone la norma IEEE 488/1975 y se renombra como GP-IB (*General Purpose Interface Bus*). En 1987, vistas las carencias de esta norma, aparecería una nueva norma que complementa a la anterior: la IEEE 488.2, pasando la anterior a llamarse 488.1. En 1992 se revisó de nuevo esta última versión y

un año después National Instruments propuso un nuevo GPIB de alta velocidad (HS488).

Otras organizaciones adoptaron soluciones muy parecidas, como es el caso del IEC 625-1 (difiere en el conector), B.S. 6146 (igual que IEC 625-1) y el ANSI MC1.1 (igual que GPIB).

El HP-IB original y, por tanto, los demás estándares y normas, describen las especificaciones mecánicas (conector, cableado, etc.), eléctricas (niveles lógicos, temporización, etc.) y funcionales (funciones del dispositivo, funciones de interfaz y codificación de mensajes). Para desarrollar una aplicación haría falta como mínimo un nivel extra: el operacional, en el que se definirían las funciones del dispositivo. En IEEE 488, ANSI MC1.1 y IEC 625-1 no se incluyó este último nivel para dar más flexibilidad a los diseñadores de instrumentos y ordenadores. El resultado de esto fue que los diseños realizados eran incompatibles entre sí; con las sucesivas revisiones se ha intentado solucionar este problema extendiendo el ámbito de definición original, como se muestra en la figura 6-2.

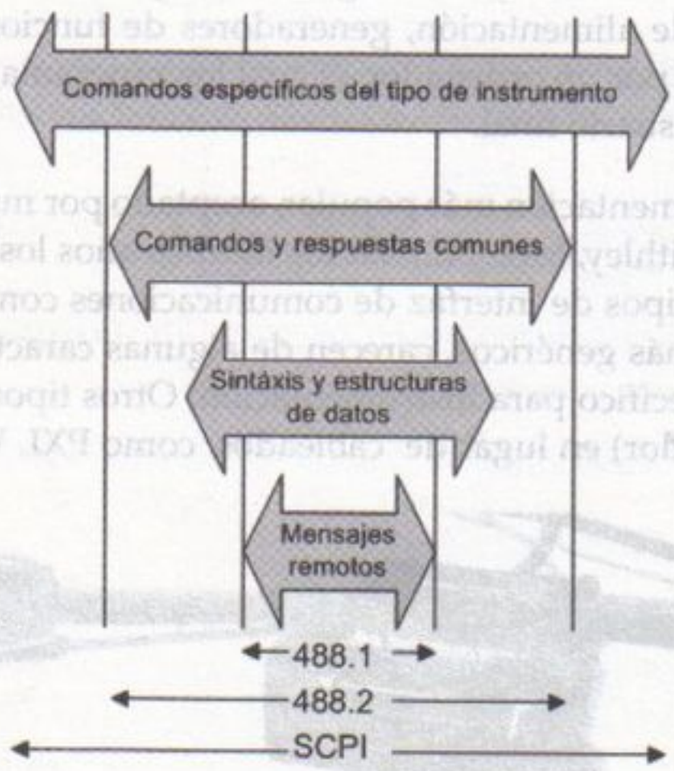


Figura 6-2. Evolución de los estándares relativos a GPIB

6.2 El estándar IEEE 488.1

Como se ha comentado antes, el IEEE 488.1 define los requisitos mecánicos, eléctricos y funcionales que deben cumplir los dispositivos para comunicarse.

Algunas características son:

- Un máximo de 363 dispositivos conectados, pero sólo 15 en funcionamiento simultáneo.

- Interconexión en línea, en estrella o mixta. Entre dos instrumentos no puede haber más de 2 metros de longitud del cable. La longitud total de todo el bus del sistema (entre todos los instrumentos) no puede exceder 20 metros a menos que se utilicen técnicas de extensión.
- Transferencia asíncrona para adaptar velocidades. Velocidad máxima 1 MBps (se ha aumentado en nuevas versiones llegando hasta los 4 MBps y 8 MBps en el HS 488).
- Compatibilidad con la tecnología TTL y lógica inversa.
- Capacidad de direccionamiento de hasta 31 direcciones primarias, desde la 0 a la 30 (la 31 se reserva para «desdireccionar»), cada una con varias direcciones secundarias. El formato de la dirección es:

Tabla 1 - Direcciones GPIB

Bit	7	6	5	4	3	2	1	0
Descripción	0	TA	LA	Dirección primaria (de la 0 a la 30)				

Para gestionar la información hay tres tipos de elementos: receptor (*listener*), emisor (*talker*) y controlador (*controller*). Nótese que se habla de elementos y no de equipos o instrumentos, un equipo puede tener implementados varios de estos elementos.

- El *Listener* sólo puede recibir datos cuando es direccionado. Sólo puede haber 14 *listeners* activos simultáneamente.
- El *talker* sólo puede transmitir datos cuando es direccionado. Sólo puede haber un *talker* activo en un momento determinado en el bus.
- El *controller* es el elemento que sirve para direccionar a los *talkers* y *listeners* y que lleven a cabo la comunicación. Sólo puede haber uno activo en un momento dado.

Si se observa la Tabla 1, donde se describe el formato de direcciones, se puede ver como para una misma dirección primaria puede haber un *talker* (bit 6 a 1) o un *listener* (bit 5 a 1).

Sólo puede haber un controlador en el bus en un momento dado, pero éste puede cambiar a lo largo del tiempo. Este dispositivo es el encargado de asignar las funcionalidades al resto de instrumento del bus.

6.2.1 Cableado

Los cables son apantallados de 24 hilos acabados en ambos extremos con un conector doble macho-hembra de tipo Microribbon 57. Como se ha dicho al principio de esta sección, la longitud de los cables está limitada a 2 metros entre equipos y a 20 metros entre todos los dispositivos. Un conector tiene el esquema de la figura 6-3.

Como puede observarse en la figura 6-3, el conector tiene varios cables de masa porque cada uno de ellos (excepto el 24) forma un par trenzado con el cable que tiene enfrente

(23 con 11, 22 con 10, etc.), de esta forma se consigue disminuir el ruido. Las líneas de datos (DI1 ... DI8) tienen un retorno común. Todas están apantalladas.

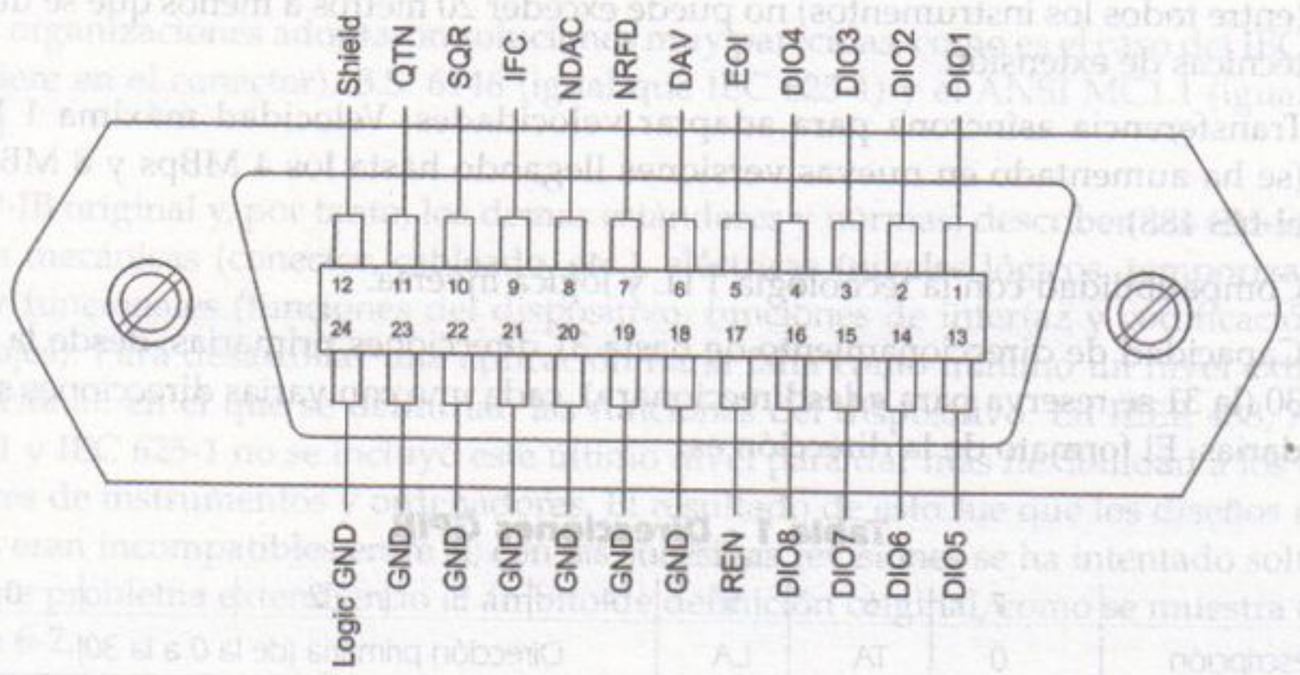


Figura 6-3. Líneas del conector GPIB

Las 16 líneas de señal se reúnen formando tres grupos:

- Control de transferencia (*handshake*)
  - DAV (*Data Valid*): indica que el dato presente en DIOx es válido.
  - NRFD (*Not Ready For Data*): indica la disposición de aceptar un dato.
  - NDAC (*Not Data Accepted*): indica la aceptación de un dato.
- Control del bus
  - ATN (*Attention*): el controlador indica si envía comandos o datos.
  - IFC (Interface Clear): permite al controlador inicializar el bus.
  - SRQ (*Service Request*): sirve para que algún dispositivo pida una solicitud de atención al controlador. Para que éste sepa el dispositivo que produjo la solicitud tiene que hacer un sondeo (*polling*) que puede ser serie o paralelo.
  - REN (*Remote Enable*): el controlador usa esta línea y otros mensajes para habilitar o deshabilitar controles locales que tienen su correspondencia remota.
  - EOI (*End or Identify*): sirve para que un *talker* marque el final de un mensaje de varios bytes o (junto con ATN) para indicar que el controlador está ejecutando un sondeo.
- Datos (bidireccional)
  - DIO1-DIO8: son líneas bidireccionales que transmiten los datos codificados normalmente en ASCII de 7 bits, siendo DIO1 el menos significativo, DIO7 el más significativo y DIO8 el bit de paridad.

Las líneas NRFD, NDAC y SRQ son de colector abierto. Las líneas DIO, DAV, IFC, ATN, REN y EOI pueden ser colector abierto o triestado. Un driver para una salida de colec-

tor abierto permite realizar una OR cableada con el resto de drivers conectados a la misma línea. En la figura 6-4 (a) puede verse la configuración típica de una salida; en (b) se muestra el driver de una línea de colector abierto.

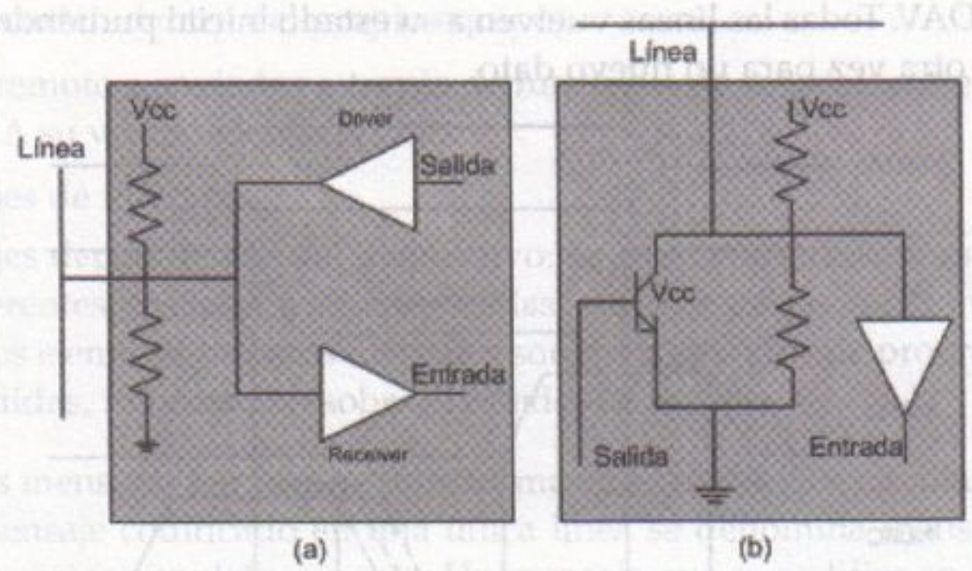


Figura 6-4. Bloque IO de las líneas del bus GPIB

6.2.2 Handshake

El protocolo de comunicación para coordinar la transferencia sobre el bus de datos ha de asegurarse que todos los receptores (*listeners*) están preparados antes que el emisor (*talker* o *controller*) empiece a transmitir y además tiene que asegurar la integridad de los datos.

La comunicación será asíncrona, en lugar de reloj se usan las tres líneas del bus de control de la transferencia: DAV, NRFD, NDAC. Mediante este mecanismo se consigue ajustar la velocidad de la transmisión al dispositivo más lento.

Un ejemplo del uso del bus de control de la transferencia se muestra en la figura 6-5, donde se marcan los instantes clave:

a) Cuando un dispositivo esté direccionado como listener tratará de ponerse en estado de escucha. En el caso que un dispositivo conectado al bus no pueda recibir el dato, la señal NRFD en el dispositivo estará a TRUE, forzando a que la tensión en la línea sea de 0 V (independientemente del estado del resto de dispositivos), como puede comprobarse en la figura 6-4 (b). Cuando todos los dispositivos estén listos para recibir un dato, la señal NRFD en todos ellos será FALSE (*Not Ready for Data*=FALSE à *Ready for Data*), por lo que la línea podrá tener un nivel de tensión alto.

b) Al subir el nivel de la señal NRFD, el *talker* sabrá que todos los dispositivos están preparados, por lo que validará mediante la señal DAV los datos que previamente había puesto en el bus (*Data Valid*=TRUE, línea a bajo nivel).

c) El primer *listener* ha leído el dato e impone un cambio en el estado de la línea NRFD. También intenta elevar la tensión de la línea NDAC, pero al ser también colector abierto deberá esperar a que hagan lo propio el resto de *listeners*.

d) Todos los *listeners* terminan de leer el dato, por lo que la línea NDAC sube su nivel para confirmar la lectura.

e) El *talker* retira el dato del bus indicando que su contenido ya no es válido mediante la línea DAV. Todas las líneas vuelven a su estado inicial pudiéndose repetir todo el proceso otra vez para un nuevo dato.

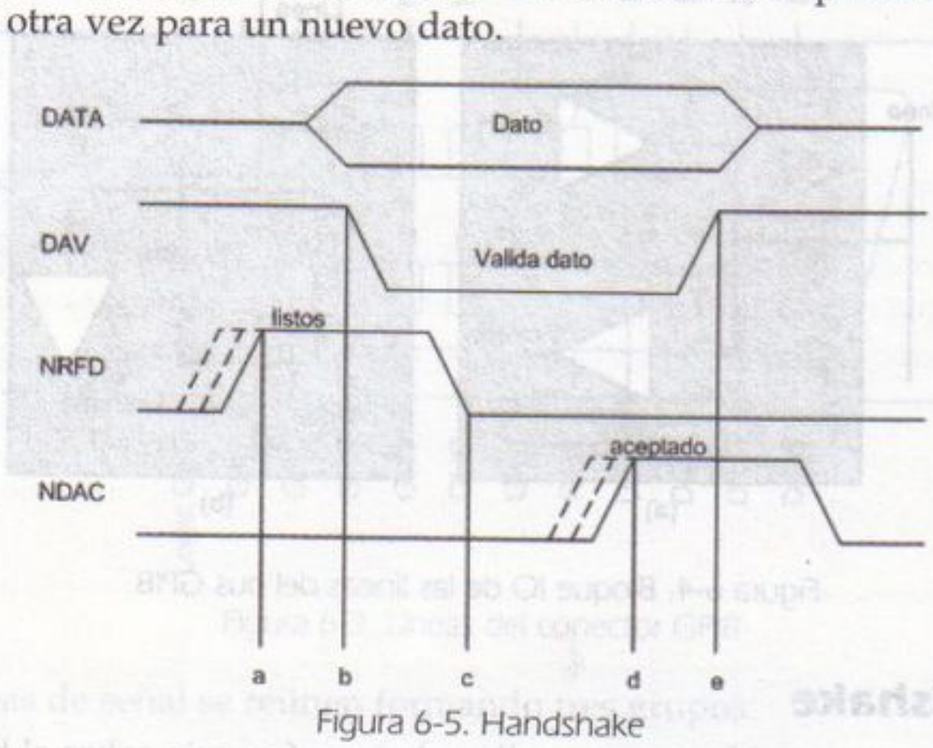


Figura 6-5. Handshake

6.2.3 Funcionamiento

Las funciones de interfaz son las capacidades predefinidas que los diseñadores de equipos pueden implementar a través de las cuales se puede enviar, recibir y procesar mensajes; se describen mediante diagramas de estados. Un determinado equipo no necesita implementarlas todas, se suele marcar con una o dos letras las capacidades del dispositivo cerca del conector. Algunas de ellas son:

- SH (*Source Handshake*) y AH (*Acceptor Handshake*): permiten al dispositivo enviar y recibir respectivamente mensajes multilínea.
- T (*Talker*), L (*Listener*) y C (*Controller*): permiten al dispositivo ser un *talker*, un *listener* o un *controller*.
- SR (*Service Request*): puede pedir atención al controlador de forma asíncrona.
- RL (*Remote Local*): permite decidir la entrada de información al dispositivo: Panel Frontal o remotamente a través del bus.
- PP (*Parallel Poll*): proporciona al dispositivo la capacidad de identificarse cuando ha pedido atención al controlador y éste está en el proceso de descubrir al equipo que la pidió.
- DC (*Device Clear*): permite que el equipo pase a un estado inicial.
- DT (*Device Trigger*): el dispositivo inicializará su operación dependiendo del bus.

También existe otro tipo de funciones llamadas funciones dependientes del dispositivo o simplemente funciones de dispositivo. Estas funciones no están definidas por la norma 488.1 y, en principio, serían definidas por el diseñador del equipo.

Las comunicaciones entre funciones de interfaz y su entorno se realizan a través de mensajes. Los mensajes enviados pueden ser:

- Mensajes locales: se producen entre una función de dispositivo y una función de interfaz (ocurren dentro del propio equipo).
- Mensajes remotos: enviados a través de funciones de interfaz entre diferentes dispositivos. A su vez pueden ser:
  - Mensajes de interfaz.
  - Mensajes dependientes del dispositivo: ocurren entre funciones de dispositivo en diferentes equipos, van codificadas a través de las funciones de interfaz. Algunos ejemplos de estos mensajes son los comandos de programación, datos de medidas, información sobre el estado del equipo, etc.

Al codificar los mensajes remotos se transforman en señales que circulan por las líneas del bus. Un mensaje codificado en una única línea se denomina mensaje unilínea; se pueden enviar varios simultáneamente. Un mensaje que se codifica en varias líneas se llama multilínea.

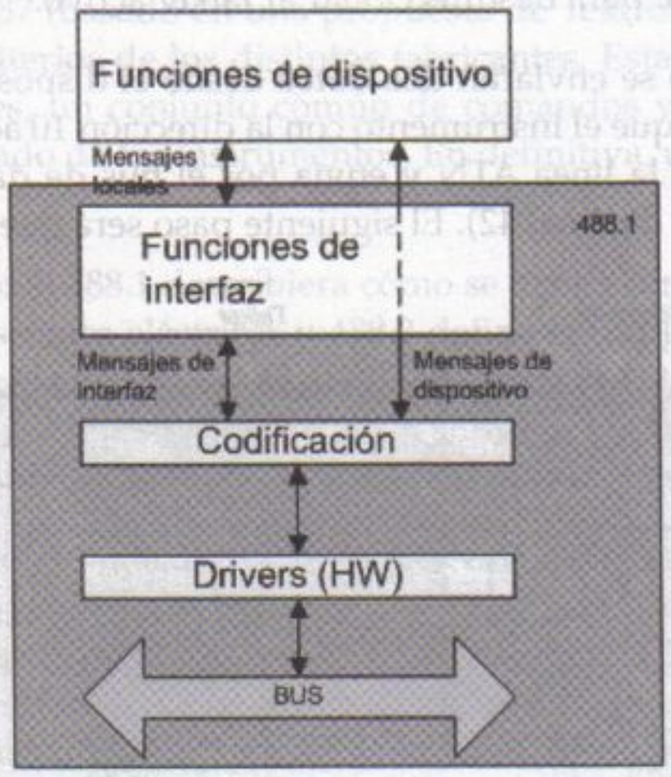


Figura 6-6. Descripción funcional de un dispositivo 488.1

Los mensajes unilínea son: IFC, REN, ATN e IDY. Algunos ejemplos de mensajes remotos multilínea son:

- DCL (*Device Clear*): provoca la reinicialización de todos los dispositivos del bus.
- GET (*Group Execute Trigger*): permite activar o disparar varios instrumentos simultáneamente.
- GTL (*Go To Local*): desactiva el modo remoto y activa el modo local en el dispositivo seleccionado.
- LLO (*Local Lockout*): deshabilita el modo local en el panel del instrumento.

- MLA[X] (*My Listen Address*): hace que el dispositivo con la dirección X sea un *listener*.
- MTA[X] (*My Talker Address*): igual que el anterior pero con el *talker*.
- PPC (*Parallel Poll Configure*): se utiliza en un sondeo paralelo.
- PPU (*Parallel Poll Unconfigure*): reinicia los dispositivos de muestreo paralelo.
- SDC (*Selected Device Clear*): resetea el dispositivo direccionado.
- SPD (*Serial Poll Disable*): indica que el proceso de sondeo serie ha finalizado.
- SPE (*Serial Poll Enable*): después de una interrupción (SRQ) se usa este comando para llevar a cabo un sondeo serie y averiguar el dispositivo que produjo la interrupción, para ello se envía este comando al dispositivo direccionado y éste responde depositando su byte de estado en las líneas de datos del bus.
- TCT (*Take Control Talker*): transfiere el control del bus al *talker*, que se convierte en el nuevo *controller*.
- UNL (*Unlisten*): desdirecciona (desactiva) los *listeners*.
- UNT (*Untalk*): sirve para desdireccionar al *talker* activo.

En el siguiente ejemplo se enviarán dos datos desde el dispositivo 4 al 10: en primer lugar debe conseguirse que el instrumento con la dirección 10 actúe como *listener*, para ello el *controller* activa la línea ATN y envía por el bus de datos el mensaje MLA10 (codificado con el valor decimal 42). El siguiente paso será que el dispositivo 4 se con-

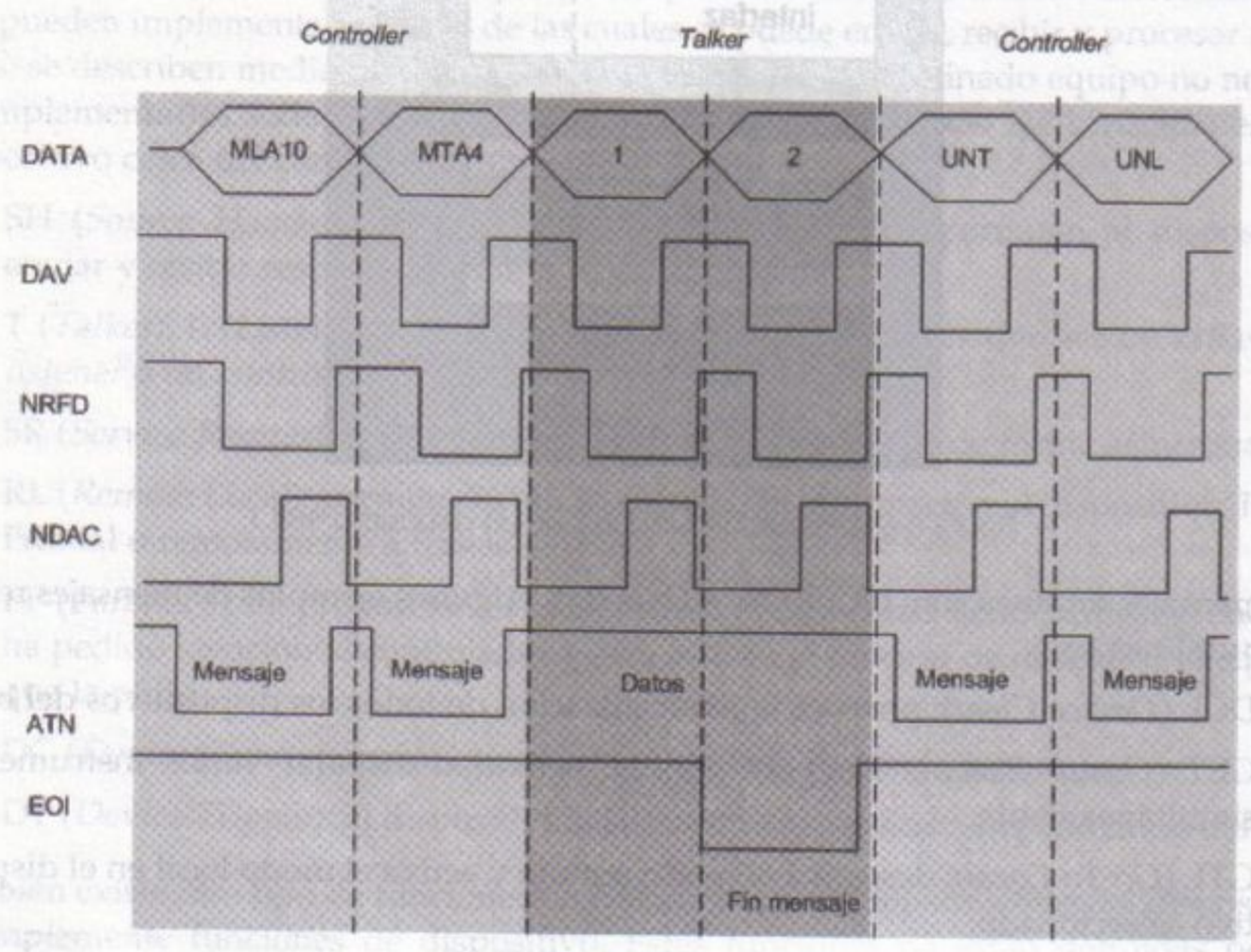


Figura 6-7. Transmisión de datos completa con 488.1

vierta en el *talker*, por lo que el *controller* enviará el mensaje MTA4. A continuación el *talker* podrá depositar los dos datos en el bus. Finalmente el *controller* puede enviar un mensaje UNT para desactivar el *talker* y UNL para hacer lo mismo con los *listeners*.

6.3 El estándar IEEE 488.2

Como se ha visto, el estándar IEEE 488 (luego renombrado a IEEE 488.1) define la forma de interconectar físicamente los dispositivos y el control del bus pero no dice nada sobre los datos que se transmiten (los valores «1» y «2» de la figura 6-7). Esto se hizo así para dar más libertad y flexibilidad a los diseñadores de instrumentos, sin embargo lo que ocurrió es que cada fabricante creó su propia forma de comunicación, siendo cada una incompatible con la de otros fabricantes; de esta forma la libertad del diseñador de equipos se transformó en quebraderos de cabeza del programador que necesitaba crear un sistema de instrumentación con varios equipos diferentes y cada uno funcionando de una manera distinta.

El IEEE 488.2 *Codes, Formats Protocols and Common Commands For use with Std 488.1-1987* surgió en 1987 (basado en una propuesta de Tektronix en 1985) de la necesidad de unificar los criterios de los distintos fabricantes. Establece un estándar en el formato de los mensajes, un conjunto común de comandos y una forma única para poder comprobar el estado de los instrumentos. En definitiva, unifica el control de instrumentos.

Una analogía sería como si 488.1 describiera cómo se deben conectar los cables de telégrafos y transmitir las señales eléctricas y 488.2 define el código Morse. El IEEE 488.2 se establece como una capa superior a la 488.1, en la figura 6-2 la flecha inferior representa a 488.1 (mensajes remotos) y las dos centrales se englobarían en 488.2 (sintaxis, estructuras de datos, comandos y respuestas comunes).

Sobre el formato de datos y sintaxis, IEEE 488.2 define el formato de números decimales, en coma flotante y *strings*; la codificación puede ser ASCII de 7 bits (como en 488.1) o binaria de 8 bits (más eficiente que ASCII). Por ejemplo, el formato de los números decimales se muestra de forma resumida en la figura 6-8, de esta forma son válidos: -12.3, .123, .12 E - 3, +.1e+2... El estándar define de una forma parecida el resto de tipos de datos.

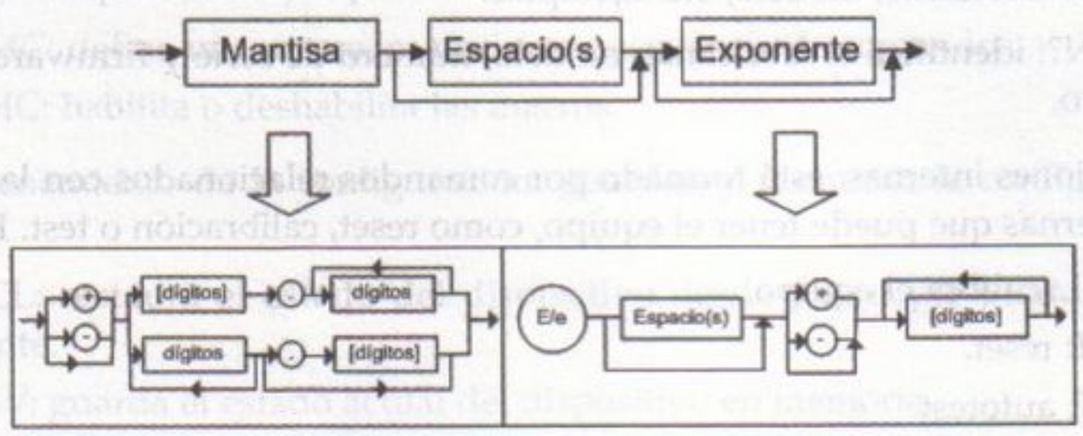


Figura 6-8. Formato de números decimales

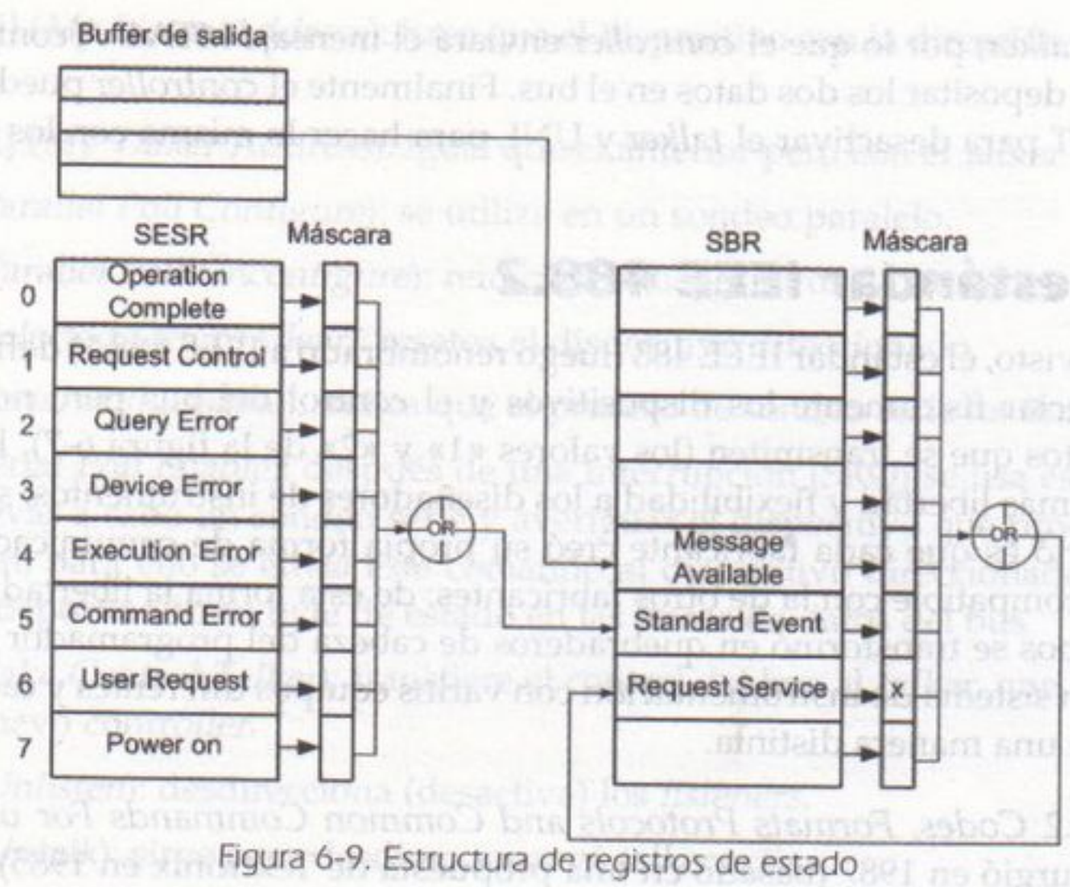


Figura 6-9. Estructura de registros de estado

El 488 original definía un byte de estado que era enviado en un sondeo serie. En 488.2 hay tres registros de estado (ver figura 6-9), cada uno puede ser enmascarado con su propio registro de habilitación.

Los registros son:

- *Standard Event Status Register* (SESR): informa de si se ha completado una operación, hay un error, etc.
- *Questionable Data*: indica sucesos como sobretensiones, sobrecorrientes, etc.
- *Status Byte Register* (SBR): es el mismo registro que especificaba 488.1, sirve para indicar si el equipo quiere enviar algún mensaje, ha pedido atención al controlador, etc.

En la norma 488.2 también se definen comandos, algunos optativos y otros obligatorios que son comunes a todos los dispositivos, éstos se agrupan según su funcionalidad:

- Datos del sistema: son comandos relacionados con información sobre el sistema, como su fabricante, modelo, etc. Ejemplos:
  - *IDN?: identifica el fabricante, modelo, número de serie y firmware del dispositivo.
- Operaciones internas: está formado por comandos relacionados con las capacidades internas que puede tener el equipo, como reset, calibración o test. Ejemplos:
  - *CAL: calibra el equipo.
  - *RST: reset.
  - *TST: autotest.

- Estado y eventos: controlan las estructuras de estado del equipo. Ejemplos:
  - *CLS: borra el registro de estado del dispositivo.
  - *ESE: fija un determinado registro interno (*Standard Event Status Enable*).
  - *ESE?: lee el valor del registro anterior.
  - *ESR?: lee el valor del registro de incidencias (*Standard Event Status*).
  - *SRE?: lee el valor del registro SRER (*Service Request Enable Register*).
  - *STB?: lee el valor de un registro (*Status Byte*).
  - *SER: fija un valor en un registro interno (*Service Request Enable*).
- Sincronización. Ejemplos:
  - *OPC: activa el bit de operación completa (bit 0 del registro *Status Byte*).
  - *OPC?: comprueba el estado del bit anterior.
  - *WAI: fuerza a completar todos los comandos anteriores antes de continuar procesando los comandos siguientes al *WAI.
- Sondeo paralelo. Ejemplos:
  - *IST?: petición del *individual status bit*.
- Trigger. Definen como un dispositivo tiene que responder ante un mensaje GET. Ejemplos:
  - *DDT: graba una secuencia de comandos que el equipo ejecutará cuando reciba el mensaje GET o el comando *TRG.
  - *TRG: realiza la misma función que el mensaje GET de 488.1.
- Controller. Ejemplos:
  - *PCB: con este comando se le dice al controlador la dirección a la que debe enviar un mensaje TCT que, como se ha visto, pasa la función de controlador de un dispositivo a otro.
- Autoconfiguración. Ejemplos:
  - *DLF: deshabilita la función de *listener*.
- Macros: estos comandos permiten al usuario definir nuevos comandos basándose en los ya disponibles. Ejemplos:
  - *DMC: define una macro, es decir, una secuencia de comandos.
  - *EMC: habilita o deshabilita las macros.
- Almacenamiento de la configuración: guardan y restauran la configuración del equipo.
  - *RCL: restaura el estado del dispositivo desde una copia guardada previamente.
  - *SAV: guarda el estado actual del dispositivo en memoria.

6.4 SCPI

En 488.2 se definen comandos comunes a todos los equipos, mediante estos comandos se pueden efectuar tareas que afectan a las comunicaciones o a algunas funciones básicas de los equipos como autotest. Pero para trabajar realmente con los equipos falta aún otro paso: configurar todas sus funciones, por ejemplo, la captura o generación de señales. Estas funciones lógicamente dependerán del tipo de instrumento, en un osciloscopio se configurarán parámetros como la escala de tiempo mientras que en una fuente de alimentación puede configurarse el límite de corriente. Con este objetivo Hewlett-Packard formalizó una propuesta en 1990 que se convirtió en SCPI (*Standard Commands for Programmable Instruments*). Hoy el consorcio SCPI lo forman fabricantes como Agilent (HP), Tektronix, Keithley, Fluke (Wavetek), National Instruments, etc.

En la figura 6-2 SCPI se sitúa por encima de GPIB aunque en realidad es independiente de la conexión física, puede utilizarse sobre GPIB, RS-232, VXI, USB, ethernet, etc.

Se trata de un lenguaje basado en comandos, consistente, English-Like e independiente del fabricante del equipo. En SCPI los distintos comandos se agrupan en varios subsistemas o familias, cada una de ellas identificada con un bloque funcional del instrumento, lógicamente no todos los instrumentos tienen los mismos bloques. En la figura 6-10 pueden verse los bloques principales de un instrumento. La agrupación se realiza de forma jerárquica empezando por la familia más genérica y particularizándose cada vez más. Cada familia o subfamilia está representada por una *keyword*, de esta forma no hace falta memorizar cientos de mnemotécnicos. Las familias principales son: `CALCulate`, `CALibration`, `CONTrol`, `DIAGnostic`, `DISPlay`, `FORMat`, `HCOPy`, `INPut`, `INSTrument`, `MEMory`, `MMEMory`, `OUTPut`, `PROGram`, `ROUTe`, `SENSe`, `SOURce`, `STATus`, `SYSTem`, `TEST`, `TRACe`, `TRIGger`, `UNIT` y `VXI`.

Un ejemplo de comando es:

```
SYSTem:COMMunicate:SERial:BAUD 9600
```

Se puede ver como la familia raíz es SYSTem, luego van las subfamilias `COMMunicate`, `SERial` y `BAUD`. Entre las familias se pone el carácter dos puntos «:» para separarlas. Las letras en minúscula se pueden omitir para conseguir mayor rapidez. Los parámetros que puedan tener los comandos van al final separados de la última *keyword* por un espacio. El comando acaba con un fin de línea.

En el caso anterior el comando sirve para poner la velocidad del puerto serie del equipo a 9.600 baudios.

Otros comandos devuelven datos, por ejemplo, para preguntar al equipo la velocidad de comunicación del puerto serie se puede usar:

```
SYST:COMM:SER:BAUD?
```

Para mejorar más la eficiencia de la comunicación (aunque no la legibilidad) se pueden agrupar varios comandos de la familia en una misma línea. De esta forma sólo se cam-

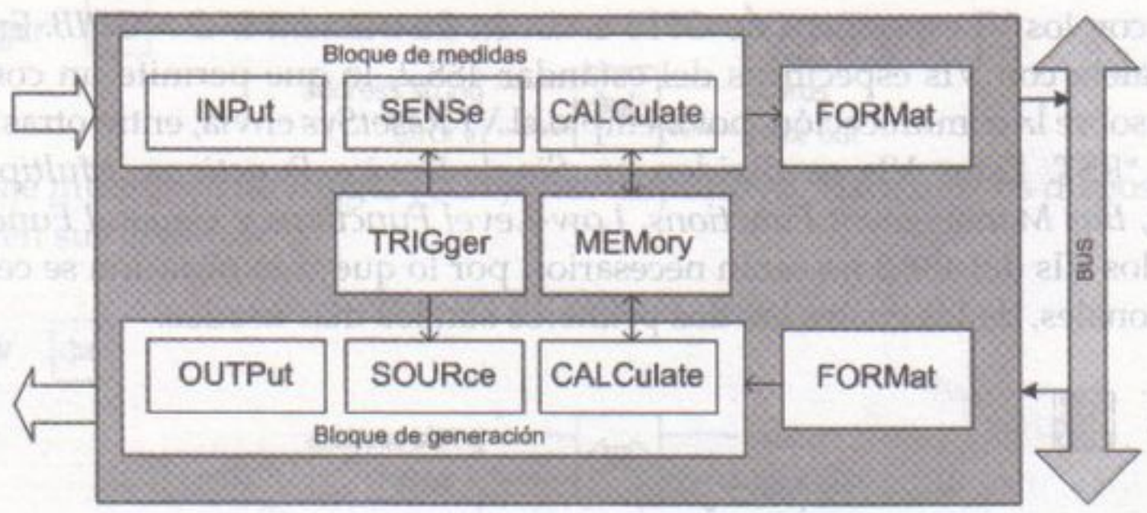


Figura 6-10. Modelo de instrumento en SCPI

bia la subfamilia más específica. Para separar los comandos se usa el carácter punto y coma «;». Un ejemplo que equivale a los dos anteriores es:

```
SYST:COMM:SER:BAUD 9600; BAUD?
```

Si al «;» le sigue el carácter «:» se indica que la escritura del siguiente comando empieza por una familia raíz (hay que escribirlo completamente). Por ejemplo:

```
SYST:COMM:SER:BAUD 9600; :SYST:COMM:SER:BITS 8
```

6.5 GPIB en LabVIEW

Todo lo visto anteriormente sobre 488.1 y 488.2 puede ser, si no se entra en detalles, totalmente transparente al programador que use las funciones GPIB en LabVIEW. Para esto LabVIEW proporciona una serie de librerías que ayudadas por el sistema operativo implementan todas las rutinas necesarias.

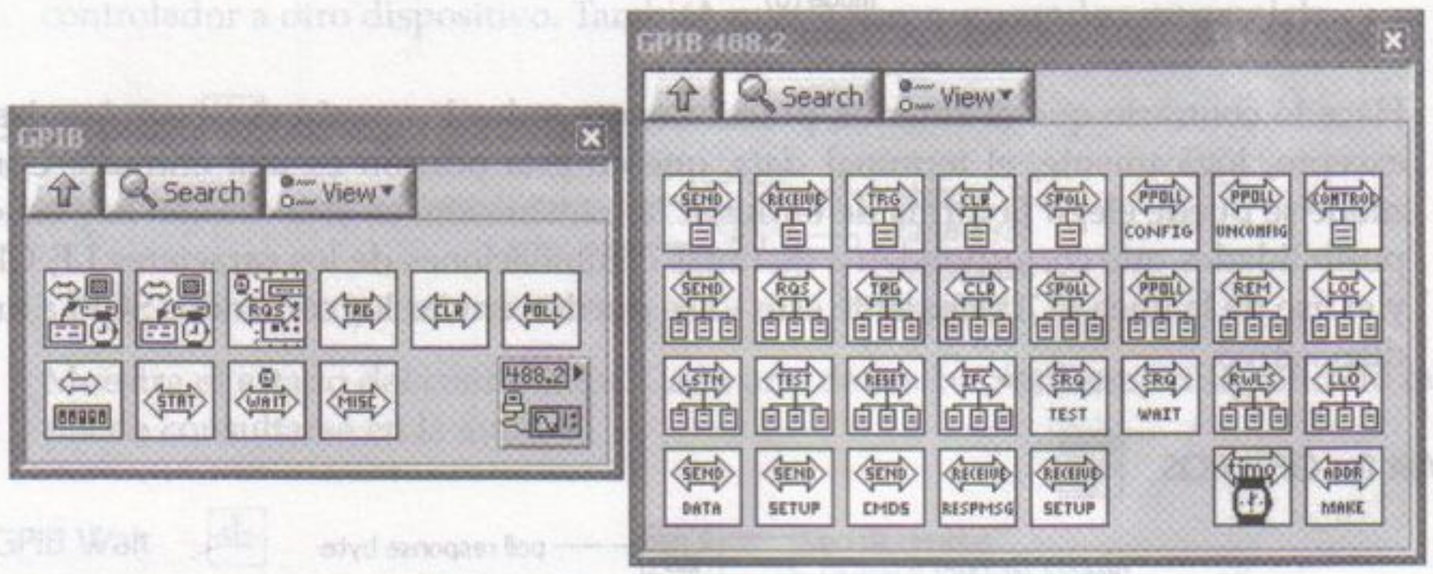
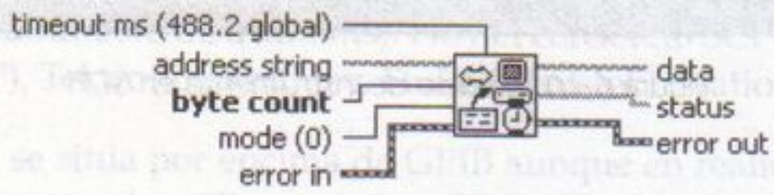


Figura 6-11. Funciones específicas de GPIB

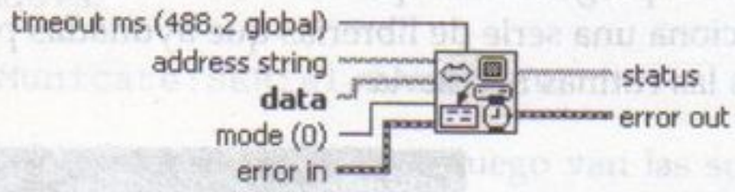
La paleta con los VI específicos de GPIB están en *Instrument I/O > GPIB*. En ella hay otra subpaleta con VIs específicos del estándar 488.2, lo que permite un control más profundo sobre la comunicación, por ejemplo el VI *ResetSys* envía, entre otras cosas, un comando *RST. Estos VIs se dividen en *Single-Device Functions, Multiple-Device Functions, Bus Management Functions, Low-Level Functions* y *General Functions*. En principio los VIs del 488.2 no serán necesarios, por lo que la explicación se centrará en los tradicionales, de los cuales los dos primeros son los más usados.

GPIB Read



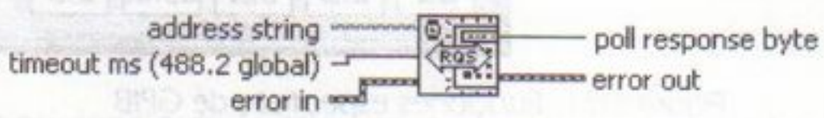
Lee la cantidad de bytes indicada en *byte count* que el dispositivo señalado con *address string* ha debido dejar previamente en el bus GPIB. Se pueden indicar dirección primaria y secundaria escribiendo «X+Y» donde X es la dirección primaria e Y es la secundaria. También se puede escribir la dirección de forma «0:2», aunque lo normal es escribir sólo la dirección primaria (la que muestran los equipos en sus displays). El parámetro *mode* sirve para indicar una de las tres formas en que el comando puede finalizar: 0 sin ningún carácter extra, 1 añadiendo CR (0Dh \r) o 2 añadiendo LF (0Ah \n). En los dos últimos casos el VI leerá datos hasta que se encuentre el carácter indicado, hasta *byte count* o hasta que se active la línea EOI. Esta función devuelve el dato leído y el estado del controlador.

GPIB Write



Hace lo contrario que el anterior, prácticamente todos los parámetros son iguales, excepto, lógicamente, el terminal *data*, que en esta ocasión es una entrada. Con *mode* se puede elegir la forma de terminar la transmisión, en esta ocasión hay ocho posibilidades que corresponden a distintas combinaciones de los caracteres LF, CR y el uso de la línea EOI. Mediante este VI se pueden enviar los comandos SCPI a un dispositivo.

Wait for GPIB RQS

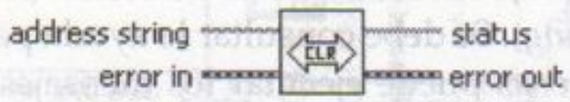


Espera hasta que el dispositivo seleccionado envíe un mensaje de petición de servicio (RQS).

GPIB Trigger

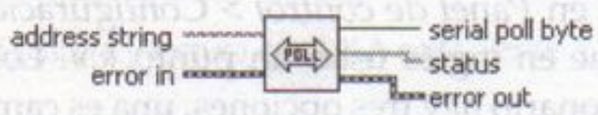
Mensaje multilínea de *trigger* (GET) a un dispositivo. Hace que los dispositivos sincronicen sus acciones.

GPIB Clear



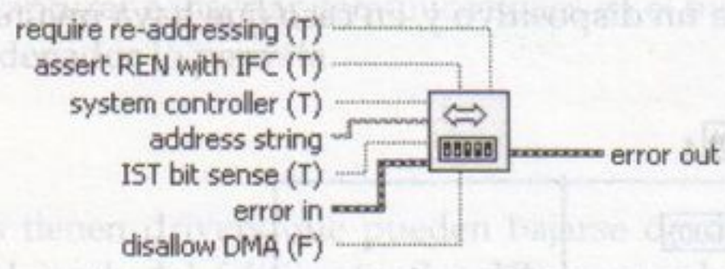
Mensaje de *clear* (SCL) a un dispositivo. El dispositivo se resetea a un estado predefinido por el fabricante.

GPIB Serial Poll



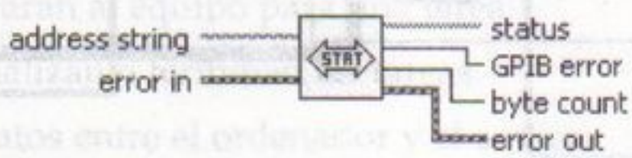
El controlador realiza un sondeo (*polling*) serie a un dispositivo para saber si ha hecho una petición de servicio (ver línea SRQ en la sección 488.1).

GPIB Initialization



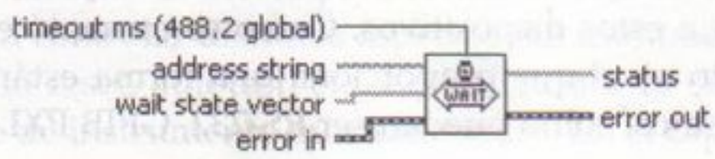
Configura el interfaz de un equipo, sirve entre otras cosas para pasar el estatus de controlador a otro dispositivo. También puede hacer un sondeo en paralelo.

GPIB Status



Muestra el estado del controlador. Cada uno de los 16 bits tiene un significado que puede consultarse en la ayuda.

GPIB Wait



Espera a que el dispositivo indicado pase a un cierto estado. El estado se indica con la misma codificación que en el VI anterior.

GPIB Misc

Ejecuta un comando de bajo nivel sobre una dirección (la dirección y el comando se indican en el mismo *string*). Se debe consultar la ayuda para saber el mnemónico del comando. Entre otras cosas puede ejecutar los mensajes que envían funciones del interfaz de IEEE 488.1 (por ejemplo GET, PPC, etc.).

Con LabVIEW suele haber un fallo recurrente cuando se programa por primera vez, el problema es con la interpretación de la coma de los decimales. La causa es un pequeño detalle: el idioma español usa como carácter de separación de decimales la coma «,» (en Windows XP puede verse en *Panel de control > Configuración regional y de idioma > Personalizar*), mientras que en inglés usan un punto «.». Los equipos se guían por la norma inglesa. Para solucionarlo hay tres opciones, una es cambiarlo en la configuración del idioma, otra es sustituir todas las comas por puntos en los *strings* antes de enviarlos al instrumento o al leerlos, y otra es configurar LabVIEW para que no use el separador de decimales regional (*Tool > Options > Front Panel > Use localized decimal point*).

Un ejemplo sencillo del uso de estos VIs es el que se muestra en la figura 6-12, donde se envía un comando a un dispositivo y, en caso que haya una respuesta, se lee.

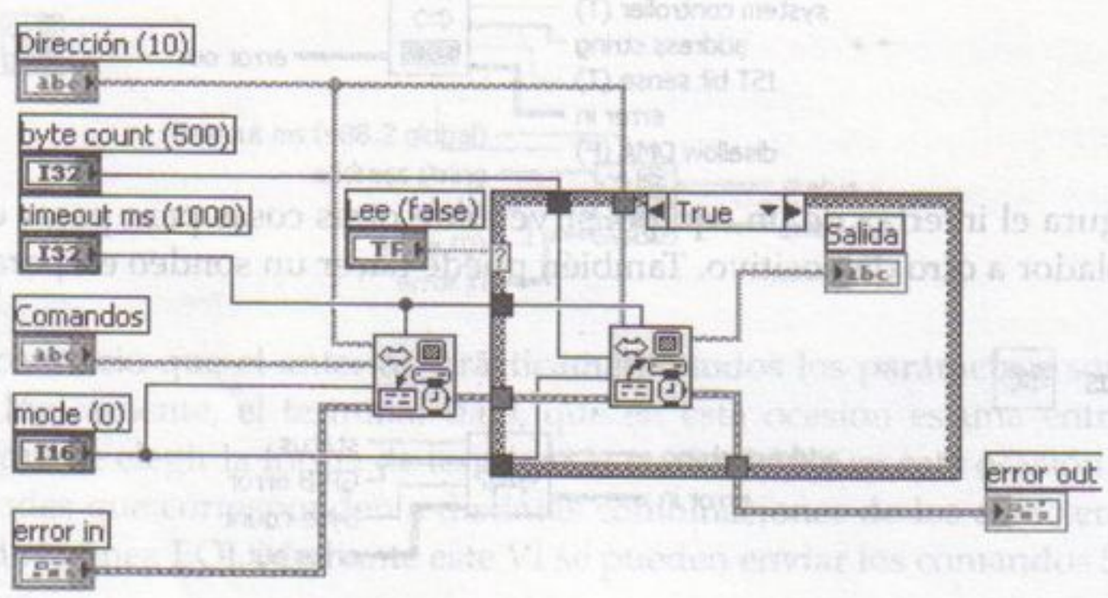


Figura 6-12. Escritura y lectura con GPIB

6.5.1 VISA

Además de las funciones específicas de GPIB, también se pueden emplear las funciones VISA para acceder a estos dispositivos. Como se comentó en el capítulo anterior, VISA es un API de alto nivel que proporciona una forma estándar de comunicación sobre diversos mecanismos, como puedan ser RS-232, GPIB PXI o VXI.

Al usar las funciones VISA a través de un interfaz GPIB se indicará la dirección del equipo de la forma GPIB[tarjeta]::[dirección primaria][::dirección secundaria][::INSTR]. En la figura 6-13 se muestra un programa equivalente al de la figura 6-12 pero en esta ocasión utilizando VISA, como dirección se ha usado simplemente «GPIB::10», donde 10 es la dirección del equipo.

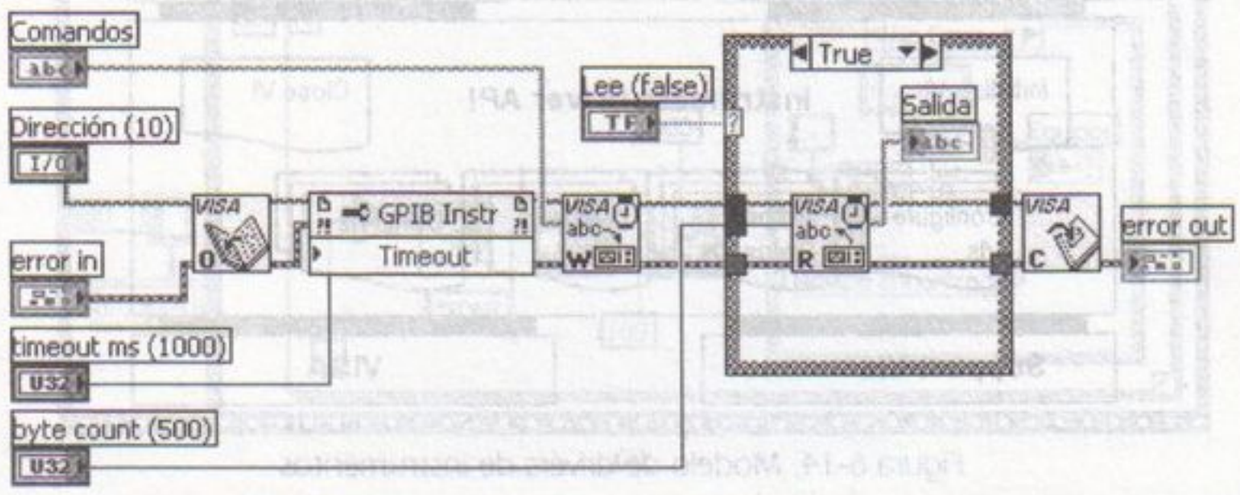


Figura 6-13. Escritura y lectura con VISA equivalente a la figura 6-12

Las herramientas de depuración como *NI-Spy*, estudiada en el capítulo anterior, pueden utilizarse tanto con VISA como con las funciones específicas GPIB. Además también puede utilizarse *Measurement & Automation Explorer* (MAX) para configurar la tarjeta controladora GPIB del ordenador o descubrir los dispositivos conectados a ella. Una nueva herramienta disponible para las funciones de GPIB es *GPIB Analyzer*, que permite monitorizar, capturar e insertar comunicaciones en el bus siempre que la tarjeta controladora del ordenador lo permita.

6.5.2 Drivers

Muchos instrumentos tienen drivers que pueden bajarse desde la web de National Instruments o desde la web del fabricante. Son VIs que se han creado usando los comandos SCPI. El modelo de programación con drivers añade una capa extra de abstracción, los drivers normalmente tienen la misma estructura:

- Inicialización: su función principal es configurar la comunicación.
- Configuración: preparan al equipo para una tarea.
- Acción/estado: inicializan o terminan las tareas.
- Datos: transfieren datos entre el ordenador y el equipo.
- Cierre: cierra la conexión.

Los drivers suelen usar las funciones VISA en lugar de las GPIB para ser independientes del medio de comunicación.

LabVIEW incluye por defecto un driver para el multímetro HP34401A. En él hay subpaletas que sirven para controlar distintos aspectos del multímetro, para leer medidas y otras utilidades como reset, autotest, etc. En la figura 6-15 se puede ver como esta paleta sigue el modelo de instrumento mostrado antes. En el capítulo anterior también

se comentó que existe un asistente en *File > New... > Project > Instrument Driver Project*, o también en *Tools > Instrumentation > Create Instrument Driver Project* para ayudar a crear drivers.

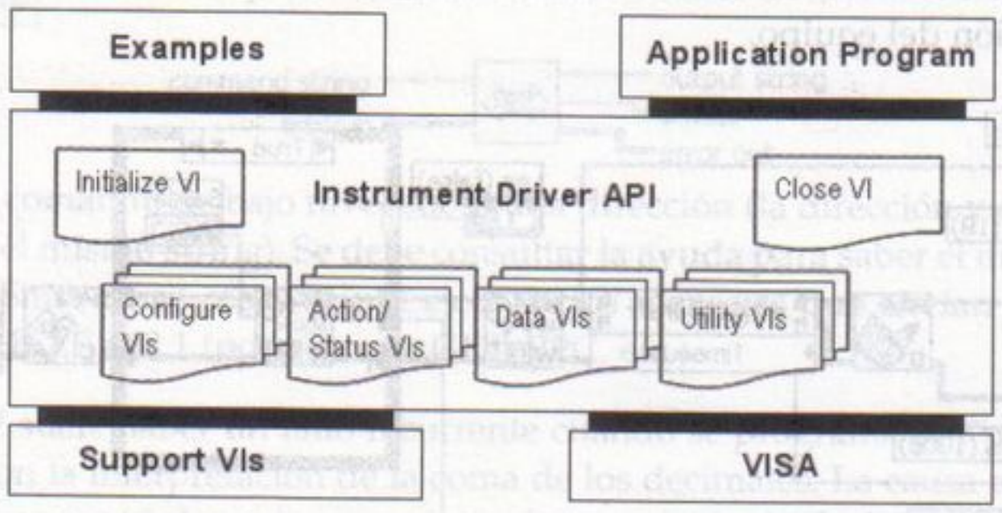


Figura 6-14. Modelo de drivers de instrumentos

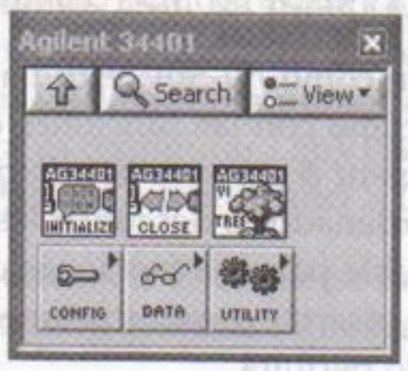


Figura 6-15. Menú de driver de un instrumento

Como novedad, LabVIEW 8.x dispone de una herramienta en *Tools > Instrumentation > Find Instrument Drivers* para facilitar la localización e instalación de los drivers de los instrumentos conectados al ordenador. Para ello el asistente realiza un escáner para detectar los equipos conectados, cuando el usuario seleccione uno el programa se conectará a la web de National Instruments (requiere registro), mostrará las distintas versiones de los drivers que existen para ese equipo y los instalará automáticamente. En versiones anteriores el proceso era bajar un fichero comprimido de la web del fabricante y descomprimirlo en el directorio *instr.lib*.

6.6 Ejemplos

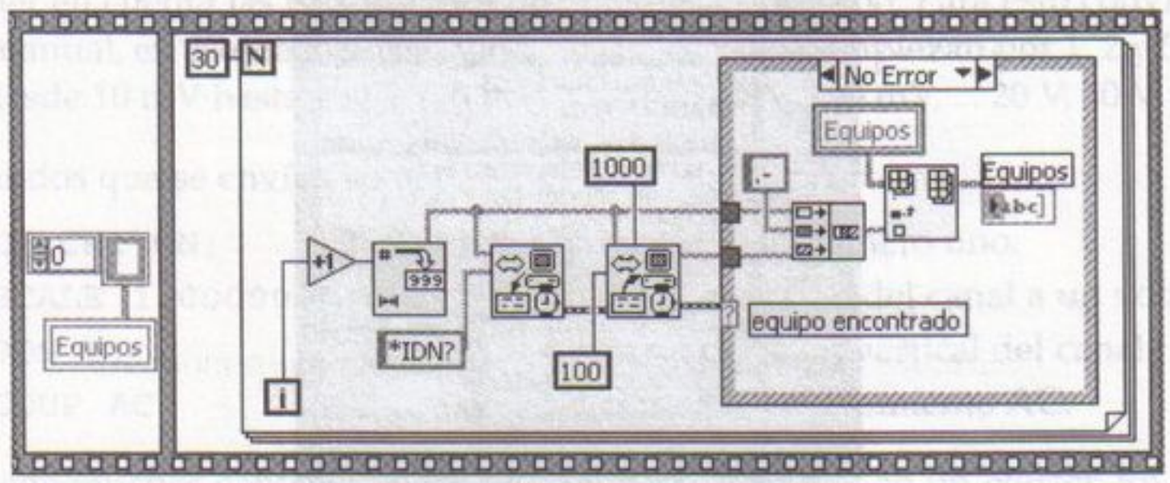
6.6.1 Ejemplo I: Escáner del bus

6.6.1.1 Explicación teórica

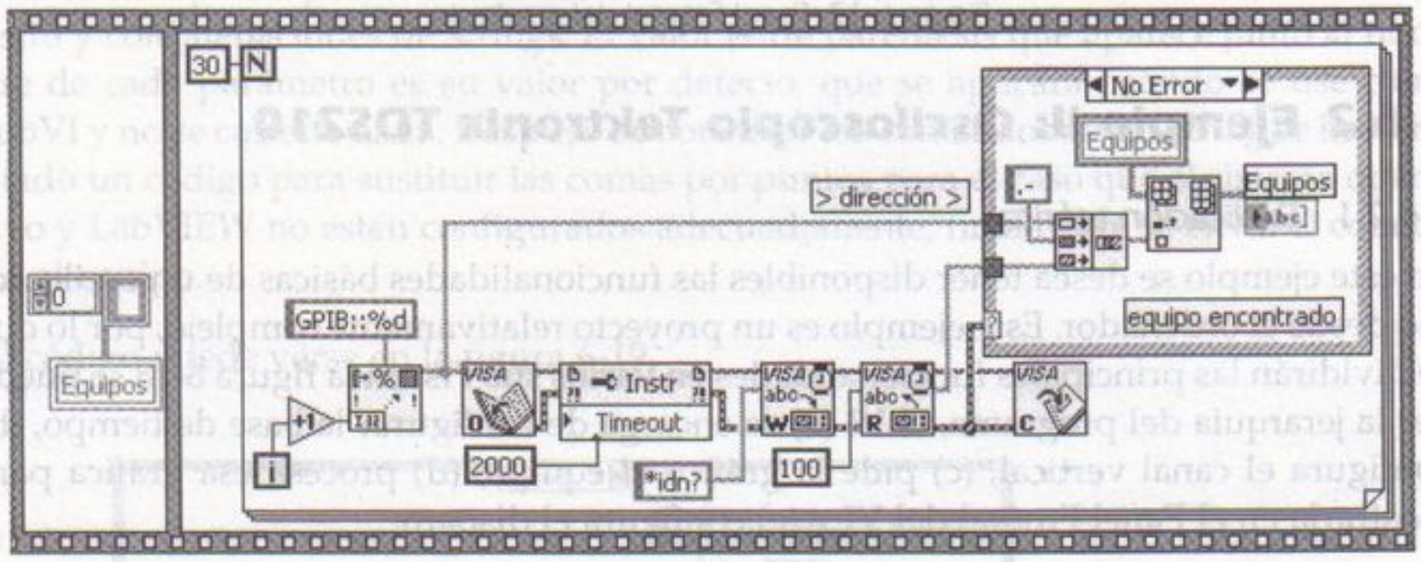
Este primer ejemplo será un sencillo programa que escanee los dispositivos del bus para pedirles que se identifiquen. Este programa usa el comando *IDN? que se vió en la sección sobre 488.2. Se creará usando primero las funciones específicas de GPIB y después las funciones VISA.

6.6.1.2 Código

El programa simplemente es un bucle que recorre todas las direcciones posibles en el bus. Para cada una envía un comando *IDN? y espera una respuesta. Las respuestas las almacena en un *array* que se muestra al finalizar el programa.



(a)



(b)

Figura 6-16. Escáner del bus GPIB

Al no indicar otro valor en las funciones de leer y escribir de GPIB, se usa por defecto el modo 0. Para mostrar en el *array* que contiene el resultado únicamente los equipos conectados, se usa la función *Insert Into Array* dentro del subdiagrama del CASE que se ejecuta cuando se recibe una respuesta.

6.6.1.3 Resultado

Este VI no requiere configuración, sólo hay que ejecutarlo. A la hora de probarlo se ha conectado un osciloscopio Tektronix con la dirección 2, una fuente de alimentación Kepco con la dirección 6, un generador de funciones Agilent en la dirección 10, un gauss-meter en la dirección 15 y un sourcemeter en la 24. El resultado se muestra en la figura 6-17.

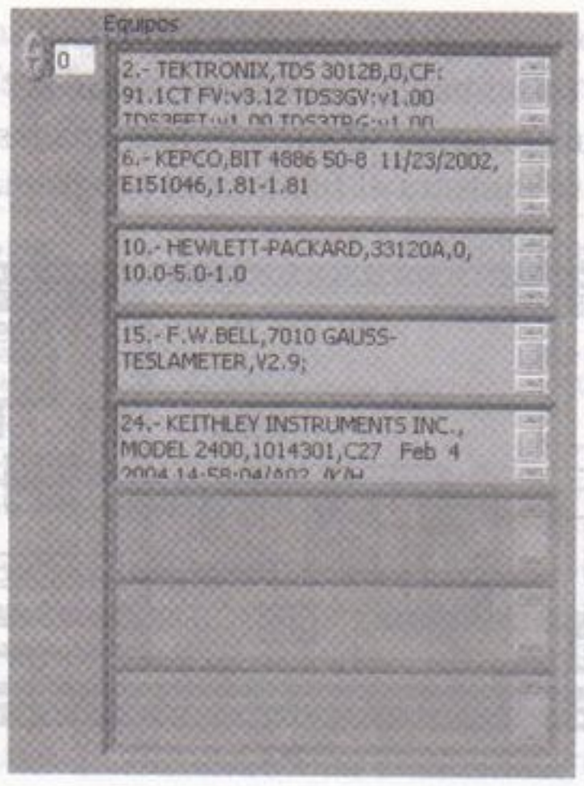


Figura 6-17. Panel Frontal del resultado

6.6.2 Ejemplo II: Osciloscopio Tektronix TDS210

6.6.2.1 Explicación teórica

En este ejemplo se desea tener disponibles las funcionalidades básicas de un osciloscopio desde el ordenador. Este ejemplo es un proyecto relativamente complejo, por lo que se dividirán las principales funcionalidades en varios subVIs; en la figura 6-18 se puede ver la jerarquía del programa, el VI (a) se encarga de configurar la base de tiempo, (b) configura el canal vertical, (c) pide la gráfica al equipo, (d) procesa esa gráfica para mostrarla en el Panel Frontal del VI y (e) configura el disparo.

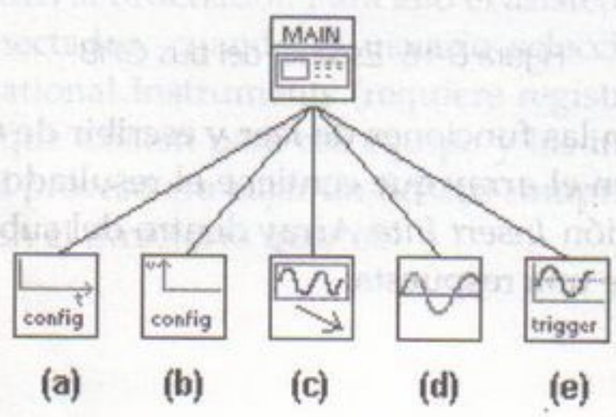


Figura 6-18. Jerarquía del programa

En el programa principal hay que decidir cuando se deben enviar los comandos de configuración y cuando se debe pedir la gráfica. La configuración debe hacerse cuando cambie algún parámetro. Sin embargo la lectura de la forma de onda debe realizarse de forma periódica, por lo que será necesario crear dos hilos (*threads*) independientes y paralelos, uno para la configuración y otro para la lectura.

6.6.2.2 Código

6.6.2.2.1 Configuración vertical

Este VI se encarga de configurar el canal del osciloscopio con los parámetros escala (Voltios/división), *position* (offset) y acoplamiento. Al introducir valores en la escala se deben tener en cuenta las restricciones que presenta el equipo. Para esto conviene consultar el manual, en este caso serían todos los valores que empiezan por 1, 2 y 5 en cada década, desde 10 mV hasta 100 V (10 mV, 20 mV, 50 mV, 100 mV, ... 20 V, 50 V y 100 V).

Los comandos que se envían son:

`SELECT:CH1 ON;`	Activa el canal número uno.
`CH1:SCALE 1.000000E+0;`	Ajusta la ganancia del canal a un voltio.
`CH1:POS 0;`	Ajusta la posición vertical del canal.
`CH1:COUP AC`	Selecciona el acoplamiento AC.

Este VI tiene muchos controles, por lo que se han agrupado en un *cluster*. En esta ocasión se ha optado por construir los comandos a base de transformaciones número a texto y concatenaciones de *strings*. El valor entre paréntesis que aparece junto al nombre de cada parámetro es su valor por defecto, que se aplicará cuando se use como subVI y no se conecte nada. Después de construir los comandos en un *string* se ha habilitado un código para sustituir las comas por puntos para el caso que el sistema operativo y LabVIEW no estén configurados adecuadamente, finalmente se envía el comando al equipo.

El código puede verse en la figura 6-19.

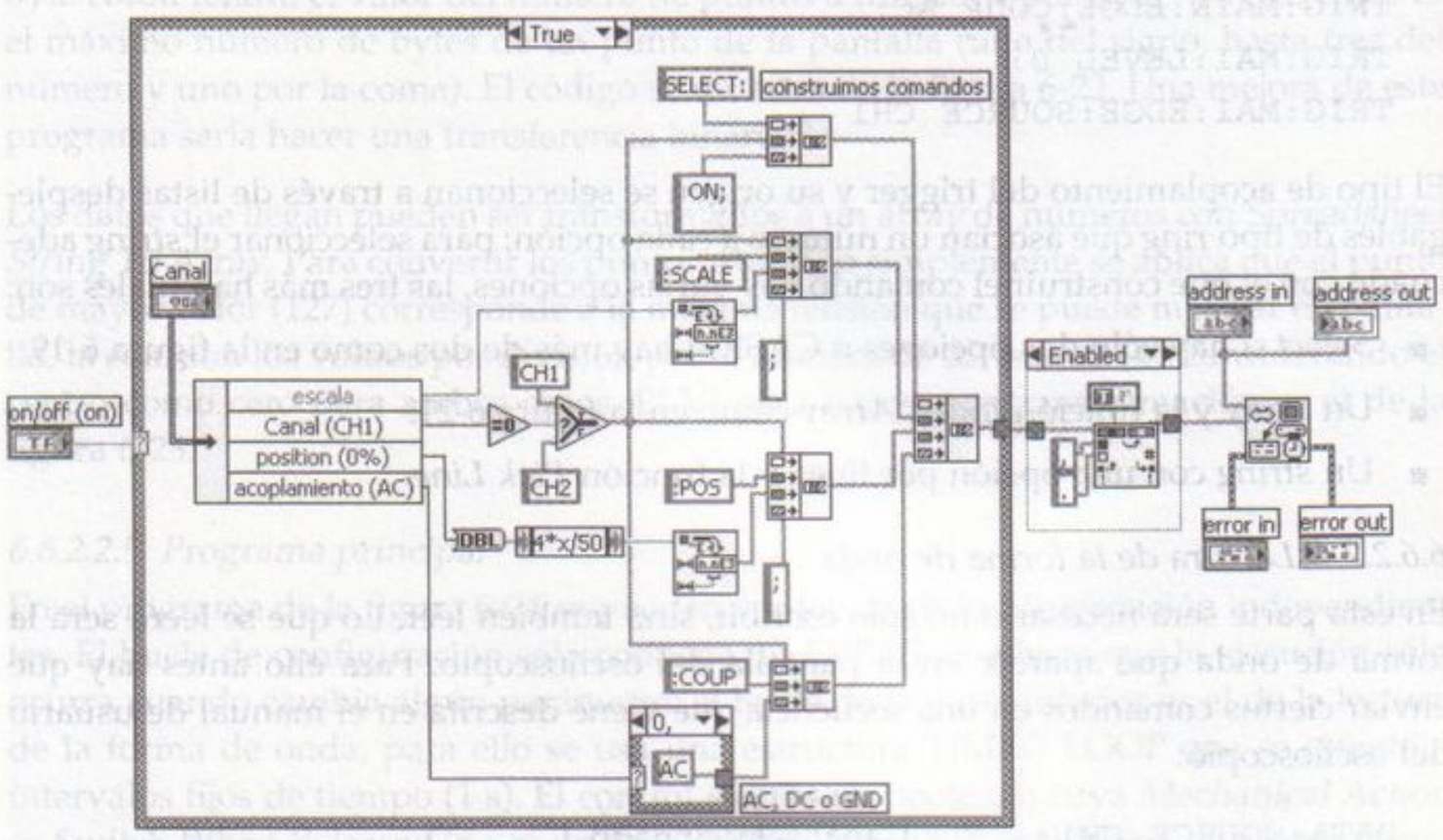


Figura 6-19. Configuración del canal de un osciloscopio

6.6.2.2.2 Configuración horizontal

Este VI (figura 6-20) se encarga de configurar la base de tiempos (HOR) en cuanto a su escala (SCA) y offset (POS). Los comandos SCPI son:

`HOR:SCA 1E+0;` Ajusta los segundos por división.

`HOR: TRIG:POS 50` Ajusta el origen en %.

Como variación respecto al anterior, esta vez los comandos se construirán usando *Format Into String* en lugar de conversiones y concatenaciones. National Instruments recomienda este método.

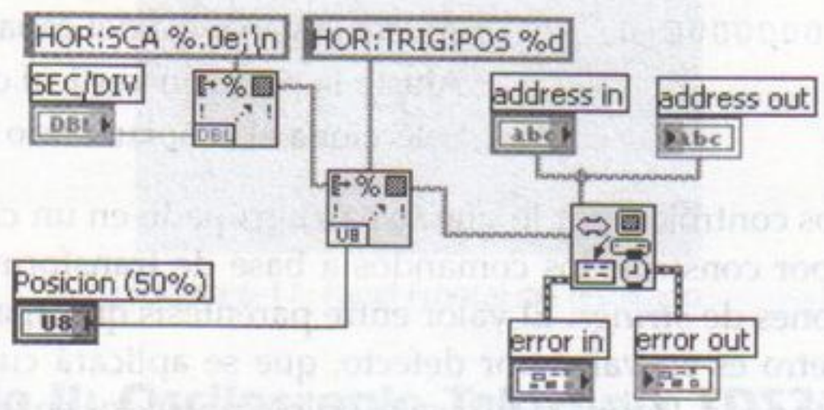


Figura 6-20. Configuración de la escala de tiempos de un osciloscopio

6.6.2.2.3 Configuración del trigger

El tercero de los VIs de configuración es el que controla el trigger; en él se seleccionará el tipo de acoplamiento, nivel y origen.

```
TRIG:MAIN:EDGE:COUP AC;
TRIG:MAI:LEVEL 0;
TRIG:MAI:EDGE:SOURCE CH1
```

El tipo de acoplamiento del trigger y su origen se seleccionan a través de listas desplegables de tipo *ring* que asocian un número a cada opción; para seleccionar el *string* adecuado con el que construir el comando hay varias opciones, las tres más habituales son:

- *Select* si hay sólo dos opciones o CASE si hay más de dos como en la figura 6-19.
- Un *array* y la función *Index Array* como en la figura 6-21.
- Un *string* con una opción por línea y la función *Pick Line*.

6.6.2.2.4 Lectura de la forma de onda

En esta parte será necesario no sólo escribir, sino también leer. Lo que se leerá será la forma de onda que aparece en la pantalla del osciloscopio. Para ello antes hay que enviar ciertos comandos en una secuencia que viene descrita en el manual de usuario del osciloscopio:

`DATA:SOURCE CH1;` Canal seleccionado

`DATA:ENC ASCII;` Formato de codificación (en este caso ASCII)

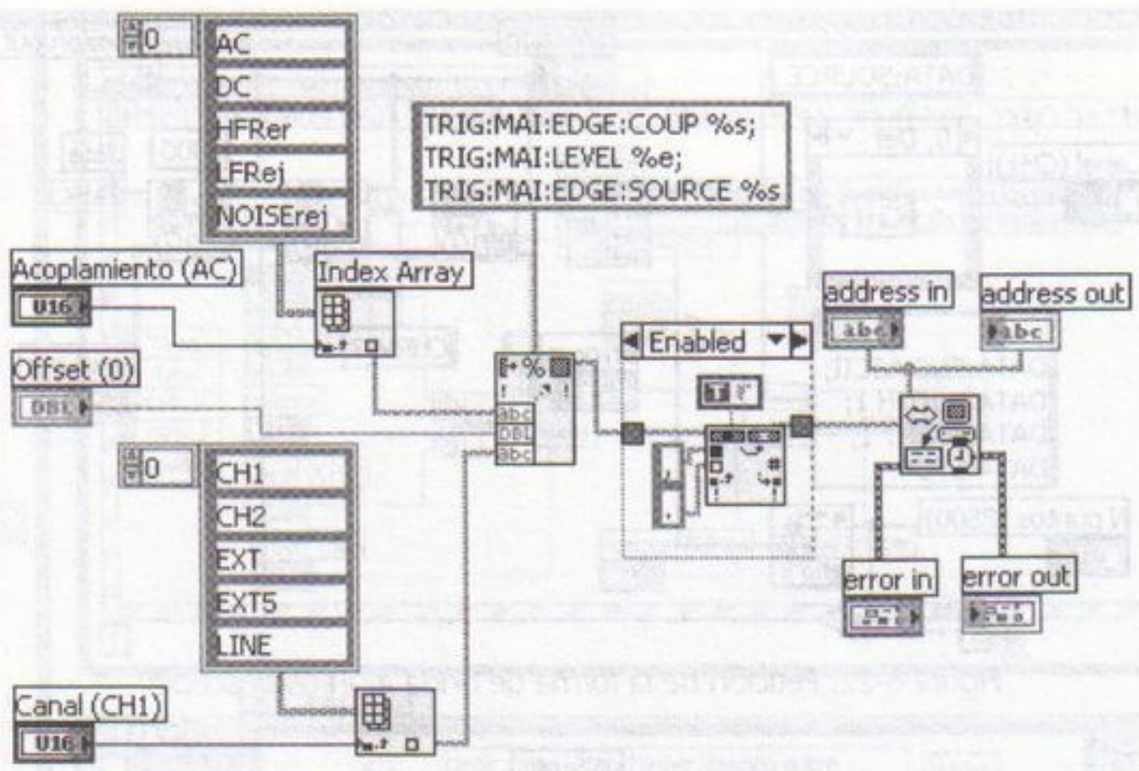


Figura 6-21. Configuración del trigger de un osciloscopio

`DATA:WIDTH 1;`	Número de bytes por punto en la transferencia
`DATA:START 1;`	Punto de inicio de la transferencia
`DATA:STOP 2500`	Último punto para transferir (máx. 10.000)

Después del comando CURVE? se recibirá la forma de onda. Ésta viene en forma de un *string* de números dentro del rango -128 a 127 (para tamaño 1 byte) separados por comas, un ejemplo es «`-100,50,120,-2,0,1`». En la función *GPIB Read* el terminal *byte count* tendrá el valor del número de puntos a transferir por cinco, ya que cinco es el máximo número de bytes de un punto de la pantalla (uno del signo, hasta tres del número y uno por la coma). El código se muestra en la figura 6-22. Una mejora de este programa sería hacer una transferencia binaria.

Los datos que llegan pueden ser transformados a un *array* de números con *Spreadsheet String To Array*. Para convertir los puntos a tensión simplemente se aplica que el punto de mayor valor (127) corresponde a la máxima tensión que se puede mostrar en pantalla, la cual son los voltios por división por el número de divisiones (4), considerando el centro como cero para ambos casos. El VI para hacer esta transformación es el de la figura 6-23.

6.6.2.2.5 Programa principal

En el programa de la figura 6-24 se pueden ver los dos hilos de ejecución independientes. El bucle de configuración sólo contiene un EVENT que hace que la ejecución sólo ocurra cuando cambia algún parámetro; el bucle de la parte inferior es el de la lectura de la forma de onda, para ello se usa una estructura TIMED LOOP que se ejecute a intervalos fijos de tiempo (1 s). El control *stop* es un booleano cuya *Mechanical Action* es *Switch When Released* (no es de tipo *Latched* porque de esta forma no acepta variables locales) por lo que necesita inicializarse.

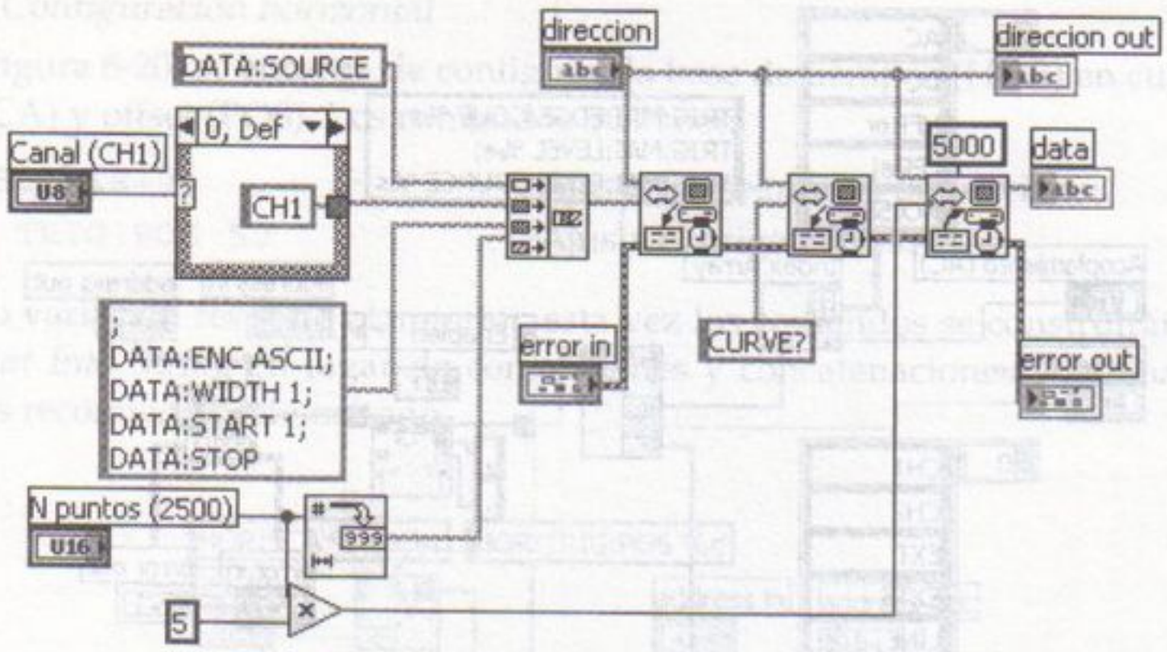


Figura 6-22. Petición de la forma de onda a un osciloscopio

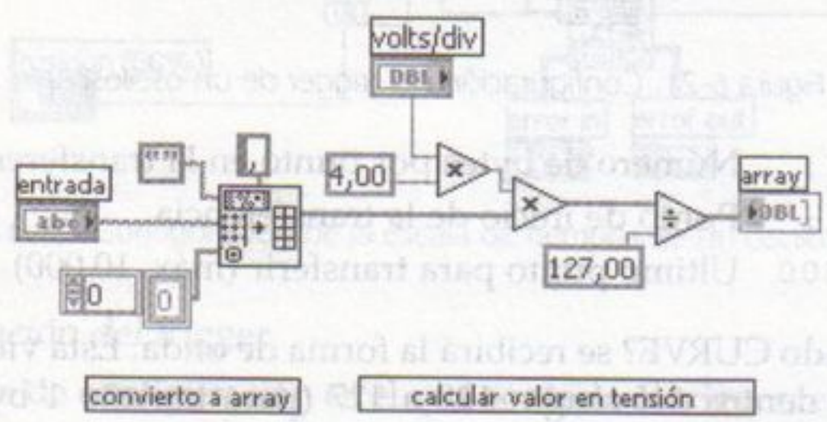


Figura 6-23. Transformar la respuesta del osciloscopio a un array de números

Para que la gráfica sea igual que la pantalla del osciloscopio se escala su eje vertical con ±4*Voltios/div en el bucle superior; en el bucle inferior se construye un dato de tipo *Waveform* con un periodo entre muestras (*dt*) igual a los segundos por división de la escala de tiempo por las 10 divisiones horizontales entre los 10.000 puntos que se reciben.

El último detalle del programa tendrá una doble utilidad: estética y funcional. Los controles para ajustar la base de tiempos y el canal (*V/div* y *sec/div*) serán de tipo *Knob*, parecidos a los controles que tiene el osciloscopio. Además, accediendo a sus propiedades, como muestra la figura 6-25, se pueden re*string*ir sus valores a una lista conocida de manera que el usuario que maneje el programa sólo pueda insertar ciertos valores.

6.6.2.3 Resultado

El Panel Frontal resultante se muestra en la figura 6-26.

Una vez creado el programa se pueden sacar algunas conclusiones: en primer lugar se ha mostrado un ejemplo relativamente complejo que incluye la lectura y escritura para manejar un instrumento a través de GPIB. Además se muestran varios detalles que deben tenerse en cuenta cuando se usa este bus, por ejemplo, la construcción de comandos, los rangos de datos válidos, las transformaciones de *string* a números y viceversa,

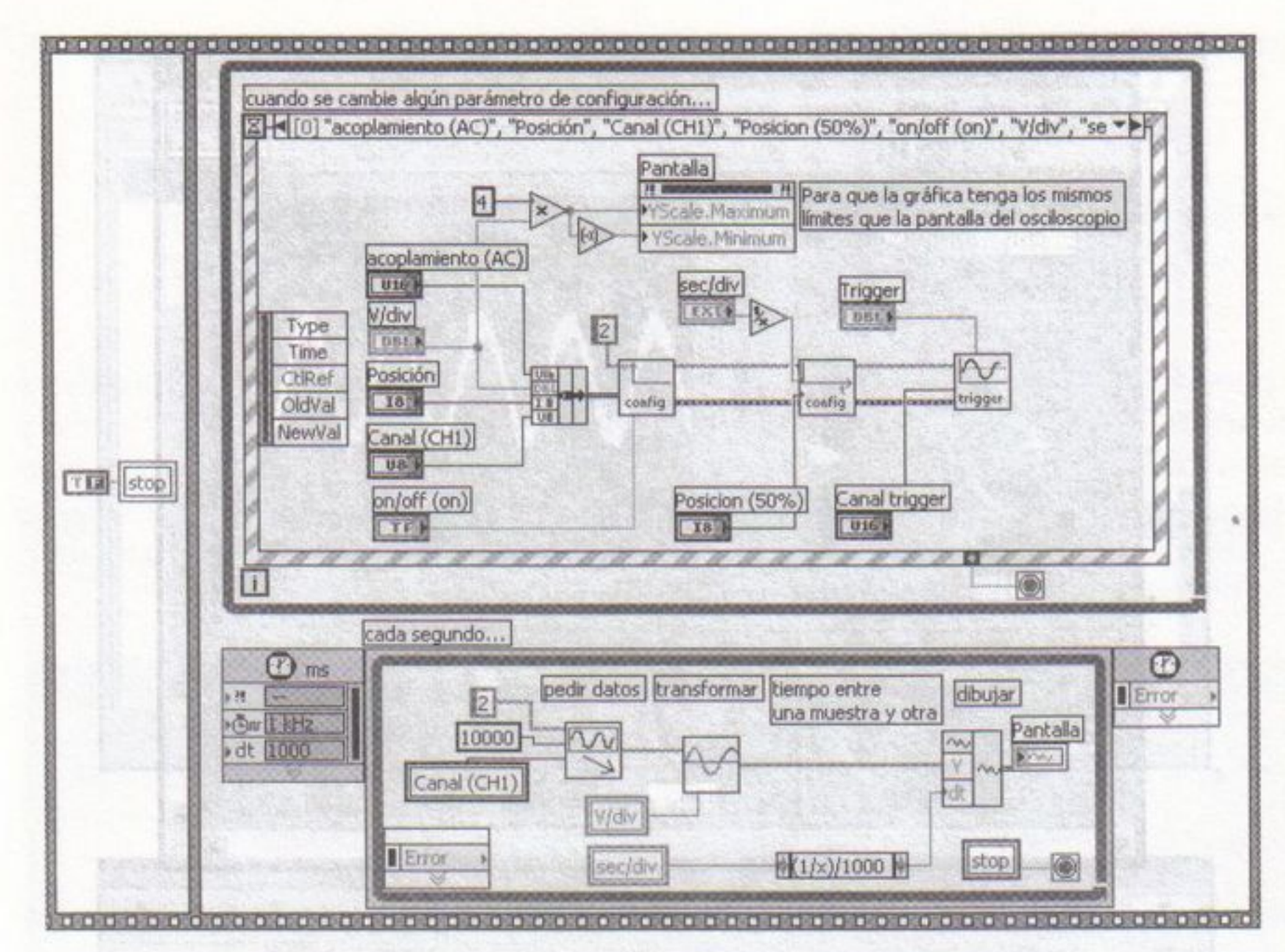


Figura 6-24. Programa final para controlar un osciloscopio

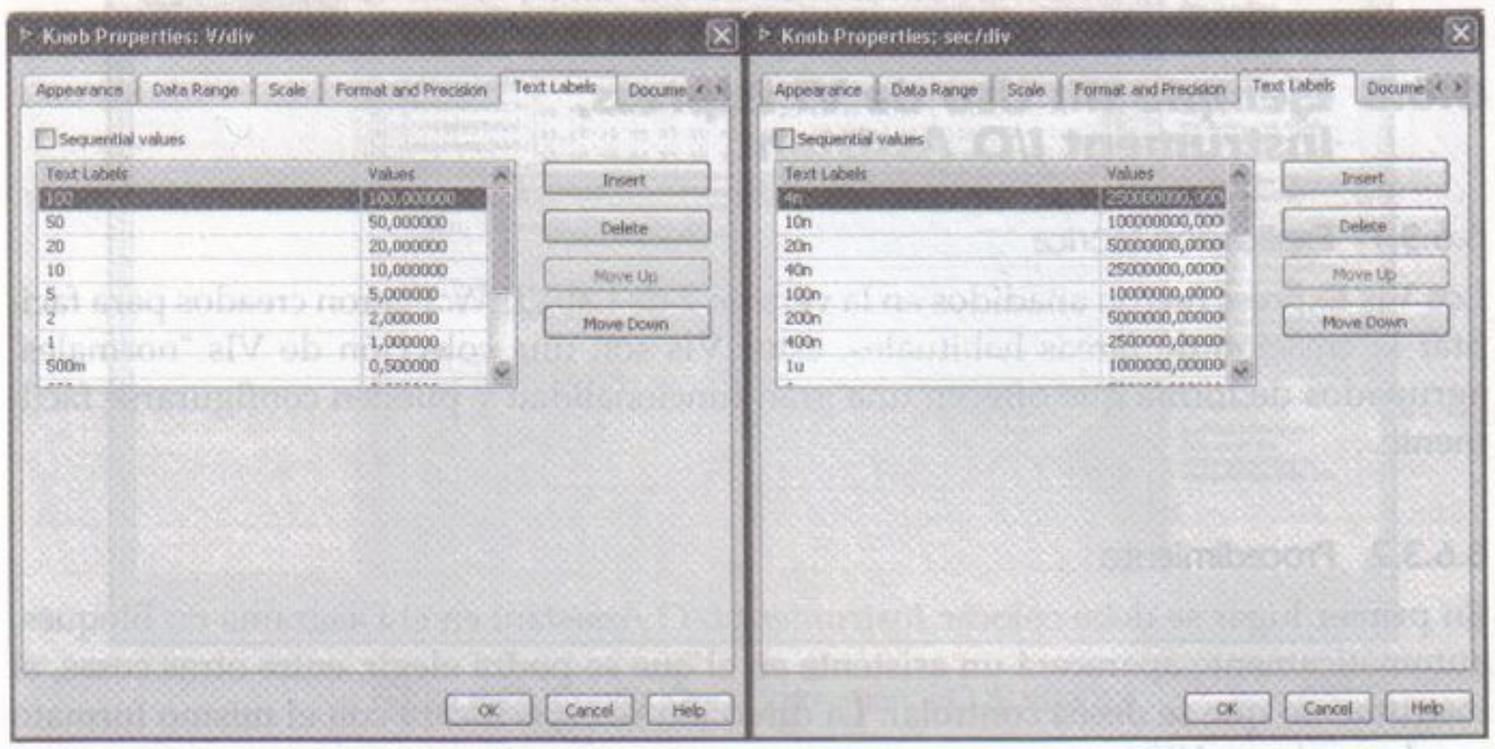


Figura 6-25. Propiedades de los controles *Knob* (voltios por división y segundos por división)

las secuencias de instrucciones en un orden preestablecido, etc. También cabe mencionar el estilo de los VIs: documentados y pudiéndose encadenar a través de *error in* y *error out*.

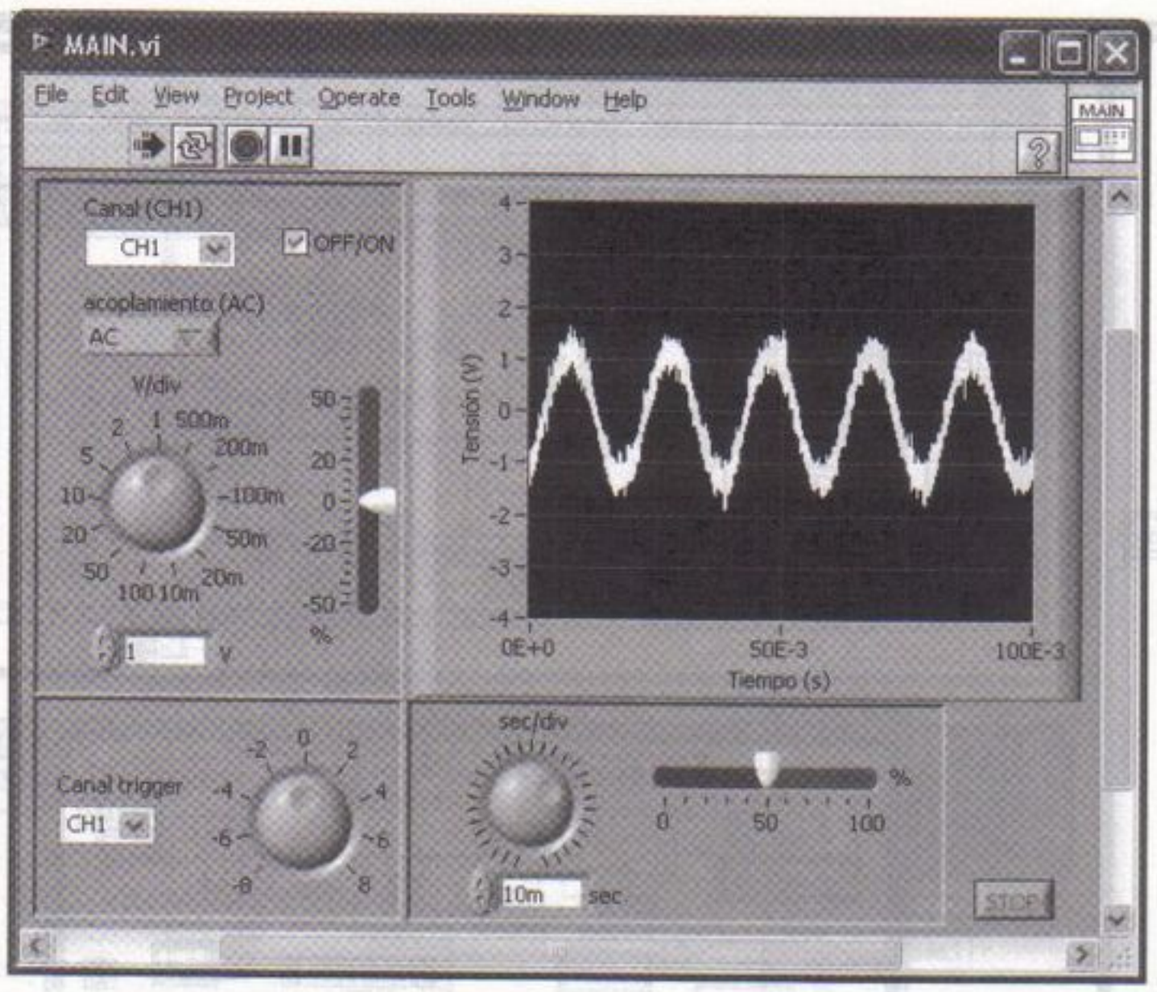


Figura 6-26. Panel Frontal del resultado

Por otra parte se han visto otros conceptos en una aplicación práctica, como la programación multihilo, la estructura TIMED LOOP y EVENT, variables locales, etc.

6.6.3 Ejemplo III: Uso de VI Express, Instrument I/O Assistant

6.6.3.1 Explicación teórica

Los VIs Express fueron añadidos en la versión 7 de LabVIEW. Fueron creados para facilitar la creación de tareas habituales, estos VIs son una colección de VIs "normales" agrupados de forma que ofrecen una gran funcionalidad y pueden configurarse fácilmente.

6.6.3.2 Procedimiento

En primer lugar se debe colocar *Instrument I/O Assistant* en el Diagrama de Bloques, automáticamente aparecerá un asistente en el que se podrá elegir, entre otras cosas, el instrumento que se desea controlar. La dirección se especificará con el mismo formato que las sesiones VISA.

Con el botón *Add Step* se podrán añadir nuevas funciones al instrumento, estas funciones se ejecutarán en el orden en que se muestran en la parte izquierda del VI (de arriba hacia abajo). En cada paso se pueden enviar varios comandos al instrumento, por ejemplo en la figura 6-28 se le pide una identificación (`*idn?`) y se espera la respuesta.

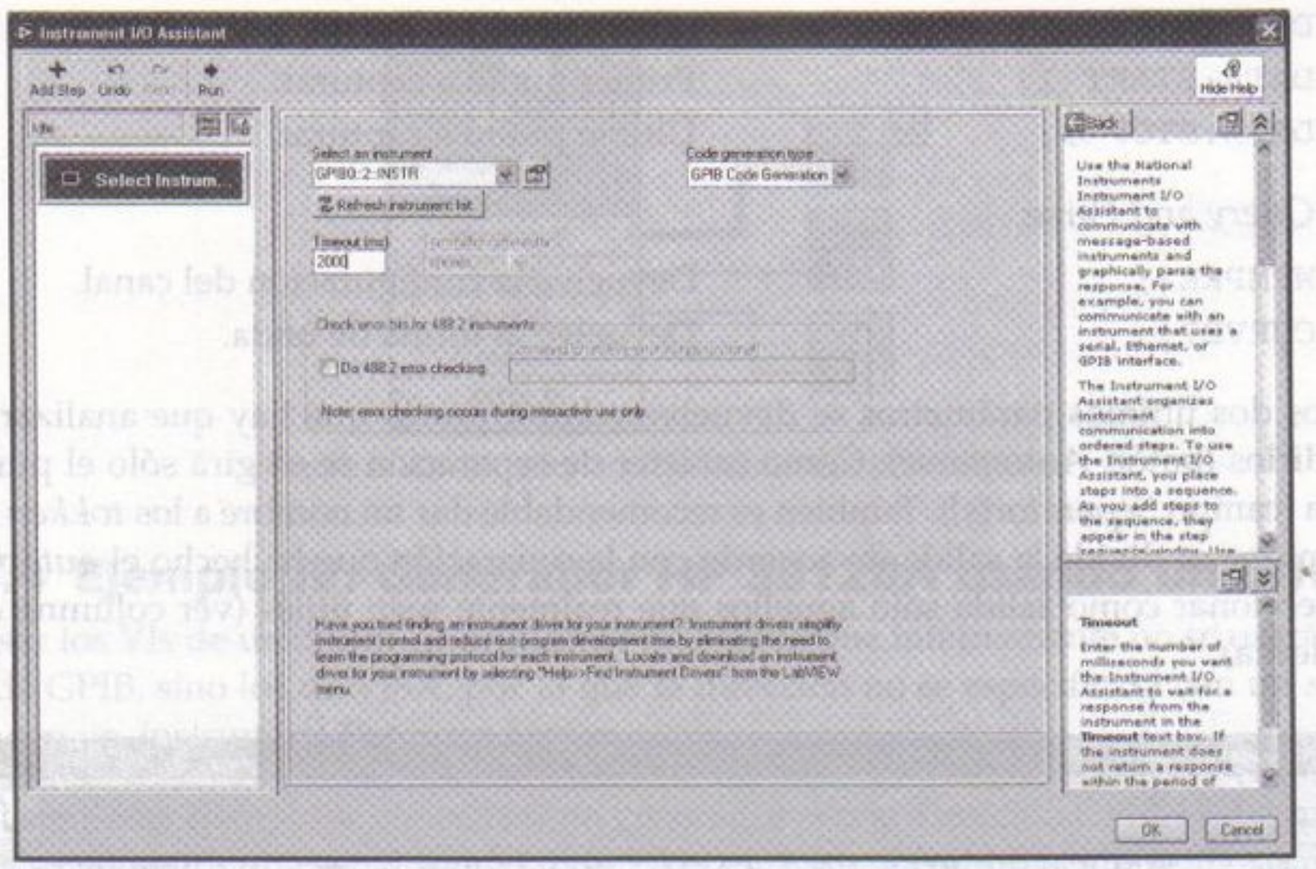


Figura 6-27. Configurando Instrument I/O Assistant I

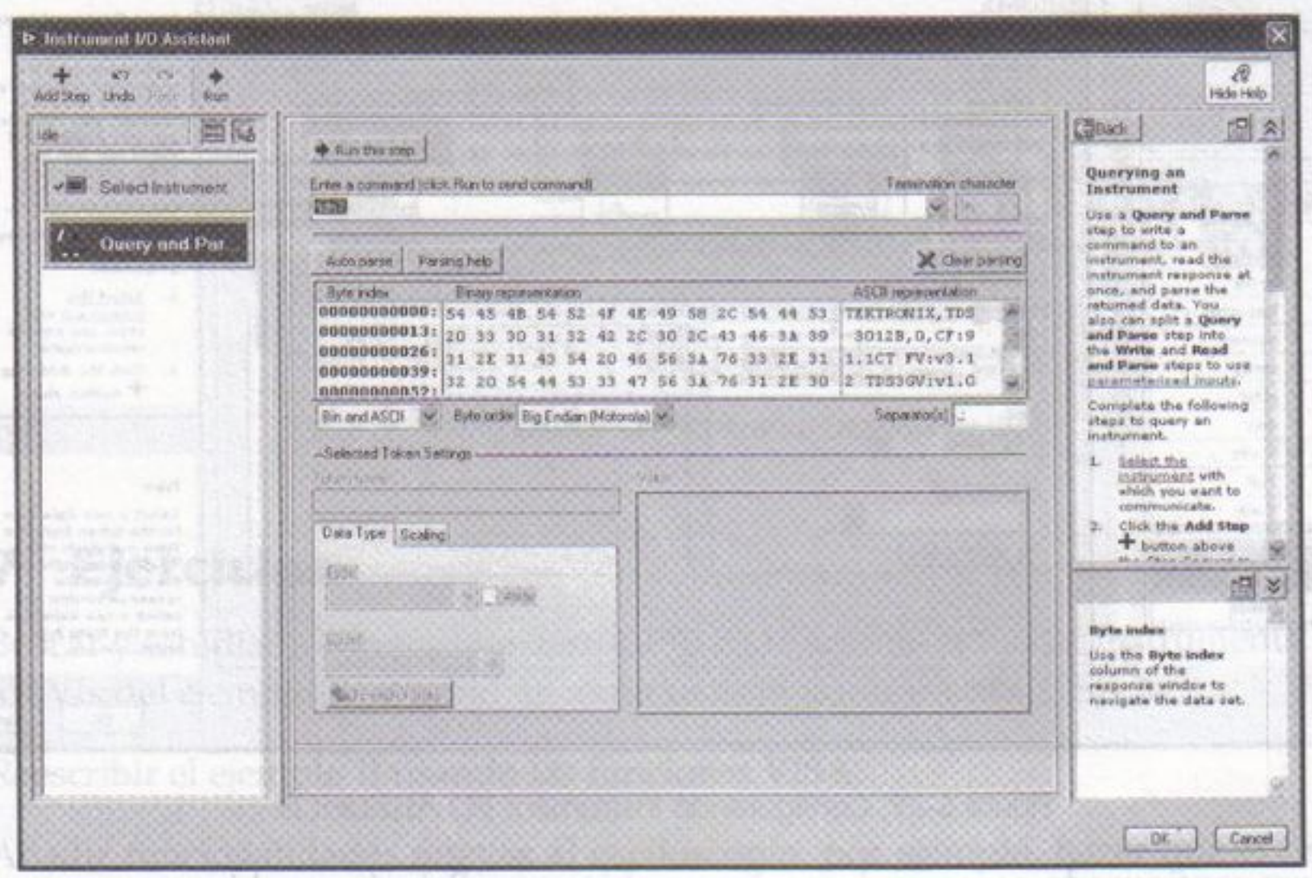


Figura 6-28. Configurando Instrument I/O Assistant II

En el ejemplo de la figura 6-29 se enviarán diversos comandos:

- *Step Write*:

`DATA:SOURCE CH1;`	Para seleccionar la el canal 1.
`DATA:ENCDG ASCII;`	Codificación ASCII.

`DATA:WIDTH 1;`	Ancho de datos 1 byte.
`DATA:START 1;`	Primer punto a capturar.
`DATA:STOP 9;`	Último punto a capturar.

- *Query and Parse*:

`WFMPRE?;`	Devuelve la configuración del canal.
`CURVE?;`	Devuelve la forma de onda.

En los dos últimos parámetros se devuelven datos, por lo que hay que analizarlos y dividirlos (botón *Auto parse*). Como carácter de separación se elegirá sólo el punto y coma (campo *Separator(s)*). También es recomendable dar un nombre a los *tokken* (partes en que se divide la salida de acuerdo con la separación que ha hecho el *autoparse*) y seleccionar como salida sólo aquellos que realmente sean útiles (ver columna de la izquierda).

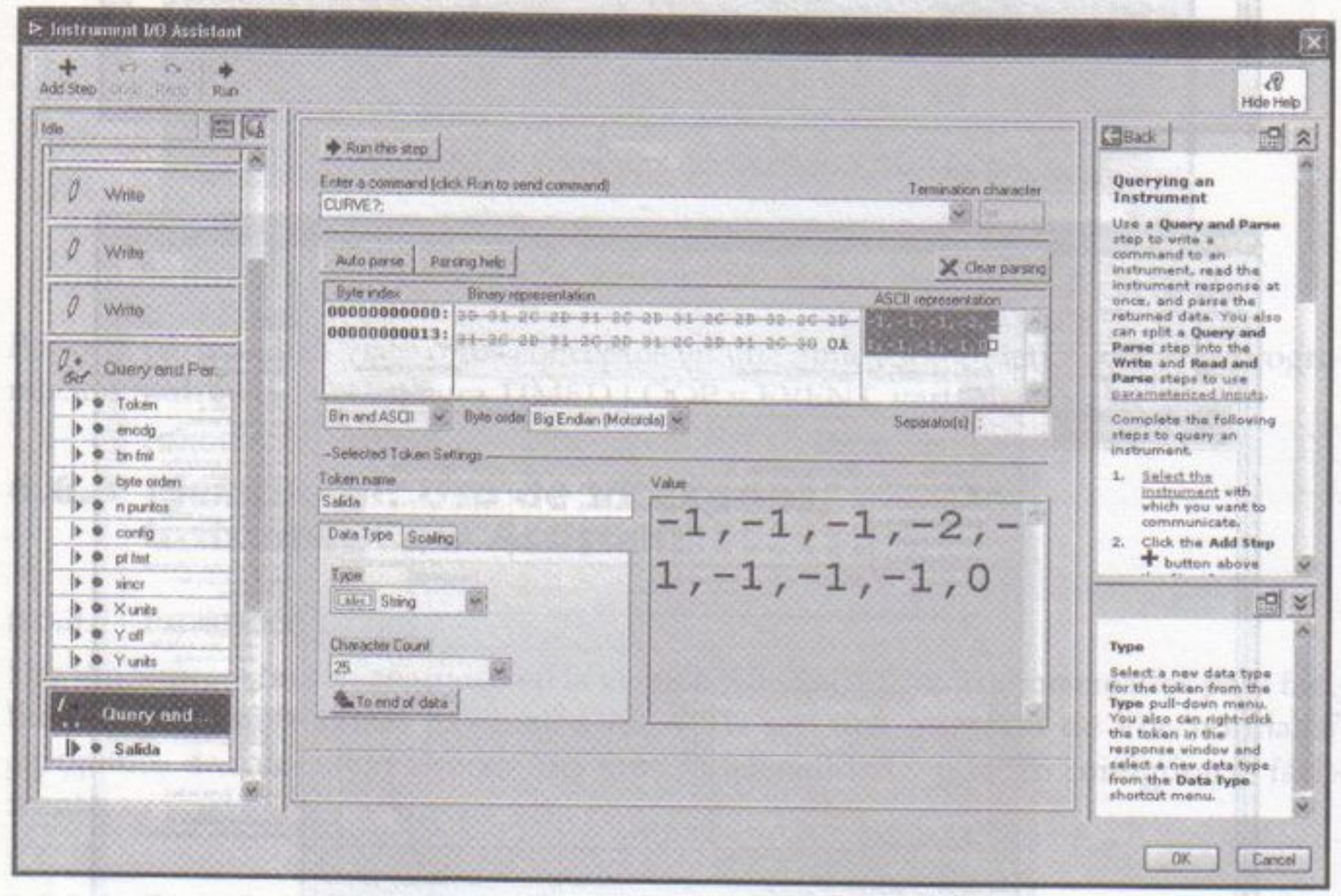


Figura 6-29. Configurando Instrument I/O Assistant III

Una vez configurado como se desea, los cambios se reflejarán en el icono puesto en el Diagrama de Bloques. Sólo resta cablearlo correctamente y ejecutarlo.

Además, presionando con el botón derecho sobre el *Instrument I/O Assistant* y seleccionando *Open Front Panel* se convierte a código clásico de LabVIEW.

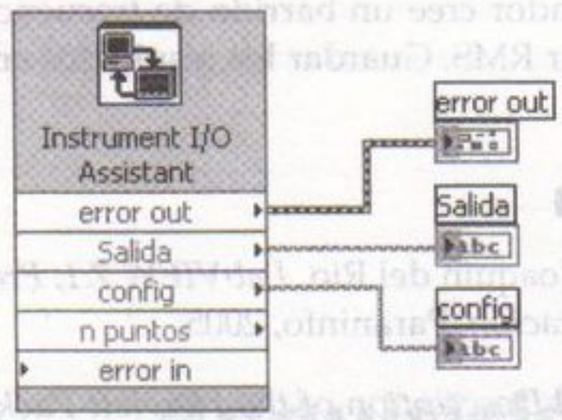


Figura 6-30. Conexionado de Instrument I/O Assistant

6.6.4 Ejemplo IV: Generador HP 33120A usando un driver

Al usar los VIs de un driver se debe tener en cuenta que internamente no se usarán los VIs de GPIB, sino los de VISA, por lo que la dirección no se especificará con un *string* sino con un *Instrument Resource Name*.

En el siguiente ejemplo se controlará un generador con la dirección GPIB 10. En primer lugar se generará una señal senoidal de 2 kHz y 2 Vp, a continuación se mostrará un mensaje en la pantalla del generador y finalmente se generará un sonido.

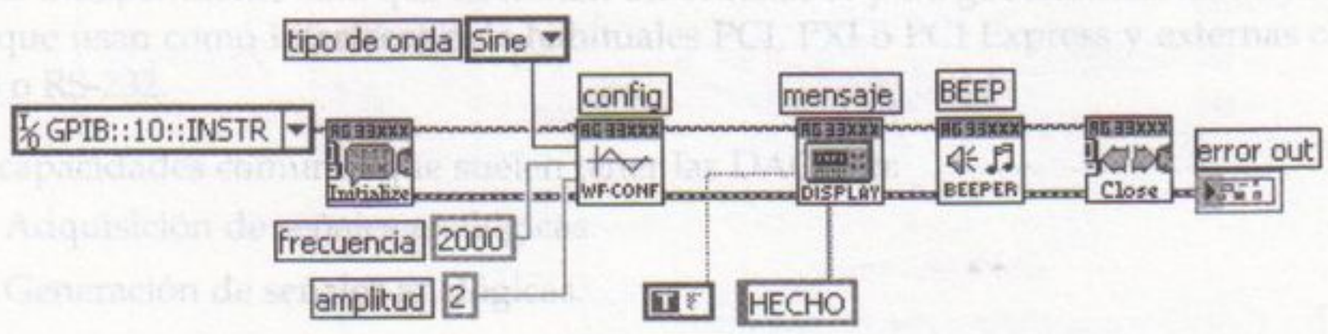


Figura 6-31. Control de un generador usando su driver

6.7 Ejercicios

1. Buscar y estudiar ejemplos que usen GPIB en la web de National Instruments. Abrir los VIs del ejemplo IV y estudiar cómo se usan las funciones VISA.

2. Reescribir el ejemplo II usando las funciones VISA.

3. Añadir funcionalidades nuevas al osciloscopio, por ejemplo, hacer que el usuario pueda elegir entre mostrar una gráfica lineal o logarítmica, hacer la transformada de Fourier a la entrada y aplicarle un filtro digital.

4. Crear un sistema de instrumentación que incluya un generador de señales y un osciloscopio. Aplicar a un circuito RC montado en una placa board una señal sinusoidal con el generador y adquirir la tensión entre los extremos de uno de los componentes con el osciloscopio (adquirir sólo un valor, no hace falta toda la gráfica), por ejemplo seleccionar como medida 1 el valor RMS del canal donde se hace la medi-

da. Hacer que el generador cree un barrido de frecuencias y para cada nueva frecuencia adquirir el valor RMS. Guardar los resultados en un fichero.

6.8 Bibliografía

Antoni Manuel Lázaro y Joaquín del Río, *LabVIEW 7.1, Programación Gráfica para el Control de Instrumentación*, Paraninfo, 2005.

Hewlett Packard, *Tutorial Description of the Hewlett-Packard Interface Bus*, 1999.

José Pelegrí y José Francisco Toledo, *Instrumentación de medida en sistemas avanzados*, Servicio de publicaciones Universidad Politécnica de Valencia, 2001.

Kamran Shah, *AN 140: Using IVI Drivers in LabVIEW*, National Instruments, 1999.

SCPI Consortium, *Standard Commands for Programmable Instruments (SCPI), 99*, 1999.

The Institute of Electrical and Electronics Engineers, Inc, *ANSI/IEEE Std 488.1-1987: IEEE Standard Digital Interface for Programmable Instrumentation*, ANSI/IEEE, 1988.

Capítulo 7

Adquisición de datos

7.1 Introducción

En este capítulo se tratará uno de los principales usos que se le da a LabVIEW: la adquisición y generación de señales eléctricas a través de tarjetas de adquisición de datos.

Hay varios tipos de sistemas de adquisición de datos. Los *Data Loggers* son sistemas que operan de forma independiente (*stand-alone*), la única función del ordenador es el volcado de los datos adquiridos. Las tarjetas DAQ (*Data Acquisition*) no operan de forma independiente sino que necesitan un ordenador para gobernarlas, las hay internas que usan como interfaces más habituales PCI, PXI o PCI Express y externas como USB o RS-232.

Las capacidades comunes que suelen tener las DAQ son:

- Adquisición de señales analógicas.
- Generación de señales analógicas.
- Generación y adquisición de señales digitales.
- Contadores y timers.
- Triggers (pre-trigger y post-trigger).
- Autocalibración, sensores, etc.

El Diagrama de Bloques típico de una DAQ se muestra en la figura 7-1.

7.2 Adquisición de señales analógicas

No hay que olvidar que la entrada de señales analógicas es sólo una parte de todas las que tiene una DAQ; eso sí, es el bloque más importante y por tanto se estudiará más detenidamente. Su estructura habitual se muestra en la figura 7-2, en ella se puede ver que todos los canales de entrada se multiplexan, ésta es la configuración más habitual, no obstante también pueden encontrarse tarjetas con los canales sin multiplexar, lo cual eleva notablemente sus prestaciones y precio; el siguiente bloque es un amplificador de ganancia programable que se usa para adecuar el nivel de la señal al convertidor con el fin de aprovechar todo su rango; los siguientes bloques son un filtro antialiasing, un cir-

cuito de muestreo y retención (SH) y un convertidor analógico a digital (A/D); finalmente las muestras adquiridas se almacenan en una memoria FIFO dentro de la propia tarjeta.

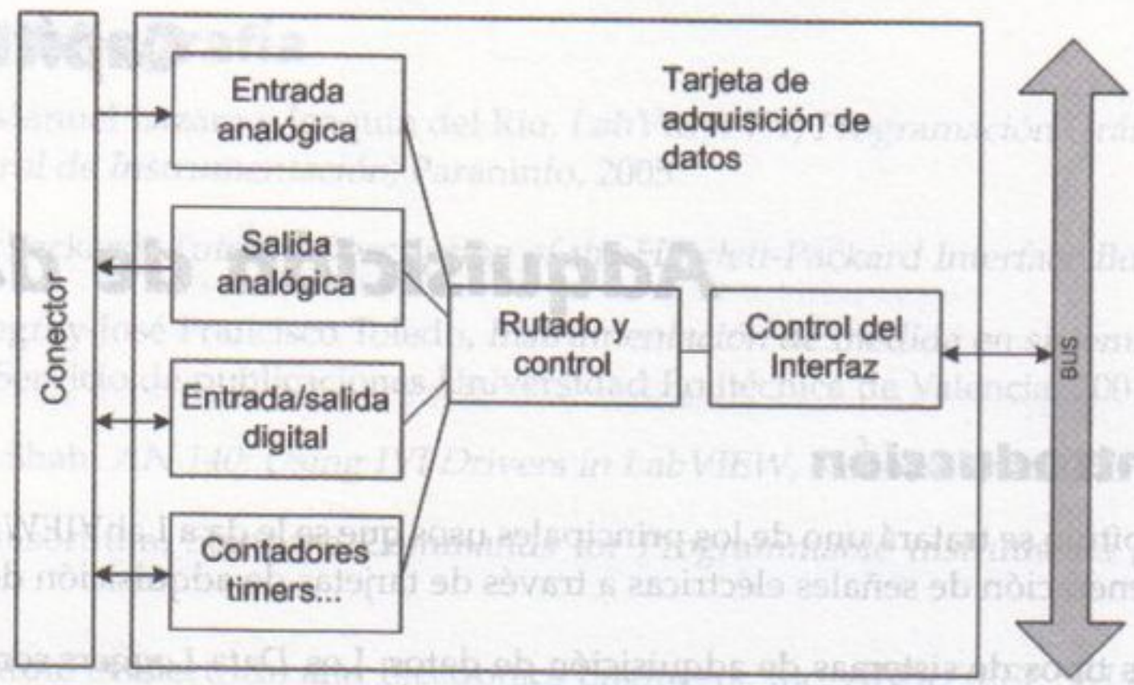


Figura 7-1. Esquema de bloques de una tarjeta de adquisición de datos

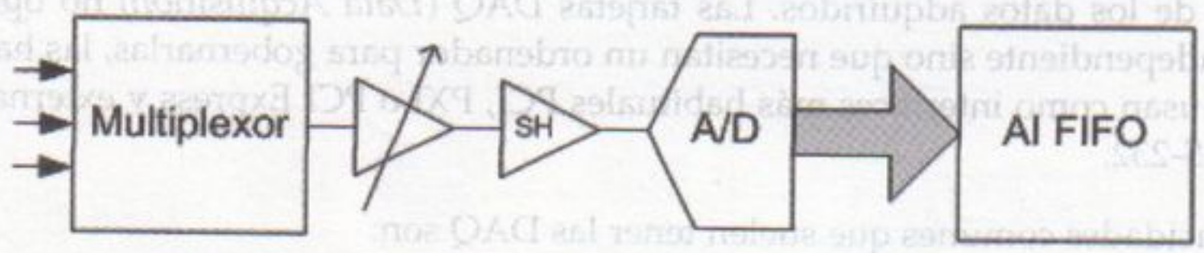


Figura 7-2. Esquema típico de un canal de entrada analógico

7.2.1 Configuraciones de terminales

- La adquisición de la señal puede ser diferencial, referenciada a un punto o no referenciada.
- En la medida diferencial (DIFF) se tendrá en cuenta la diferencia de tensión entre una línea y otra. En este caso el circuito externo y la tarjeta no tienen una referencia común. En la figura 7-3 se puede ver cómo el valor final es la diferencia[1] entre las dos líneas de uno de los ocho canales diferenciales, por ejemplo tomando el primer canal.
- Como regla general es preferible una medida diferencial a las otras porque presenta menos ruido, mejor rechazo al modo común, etc.
- Las medidas con referencia (*referenced single-ended*, RSE) sí tienen un punto común entre el exterior y la tarjeta DAQ. En el esquema de la figura 7-4 el circuito exterior y la tarjeta DAQ compartirían la misma referencia: AIGND (*analog input ground*).

1 La salida de un amplificador de instrumentación es un cierto valor multiplicado por la resta de las tensiones de sus dos entradas.

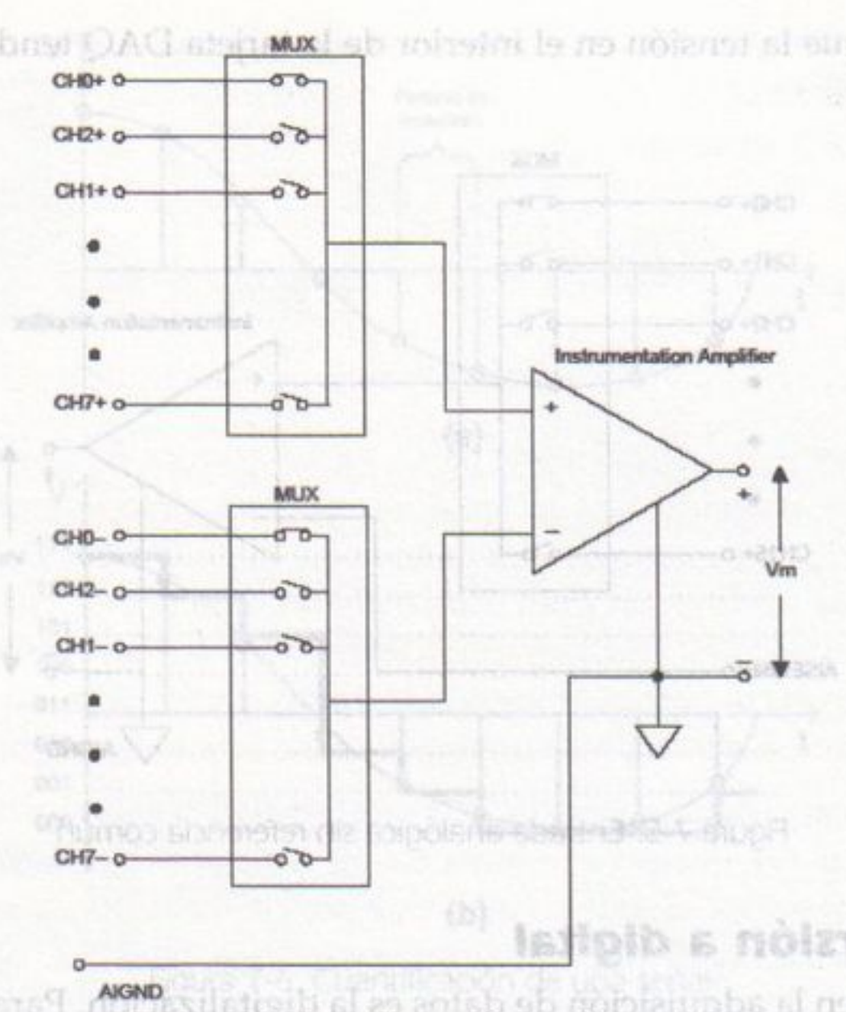


Figura 7-3. Entrada analógica diferencial

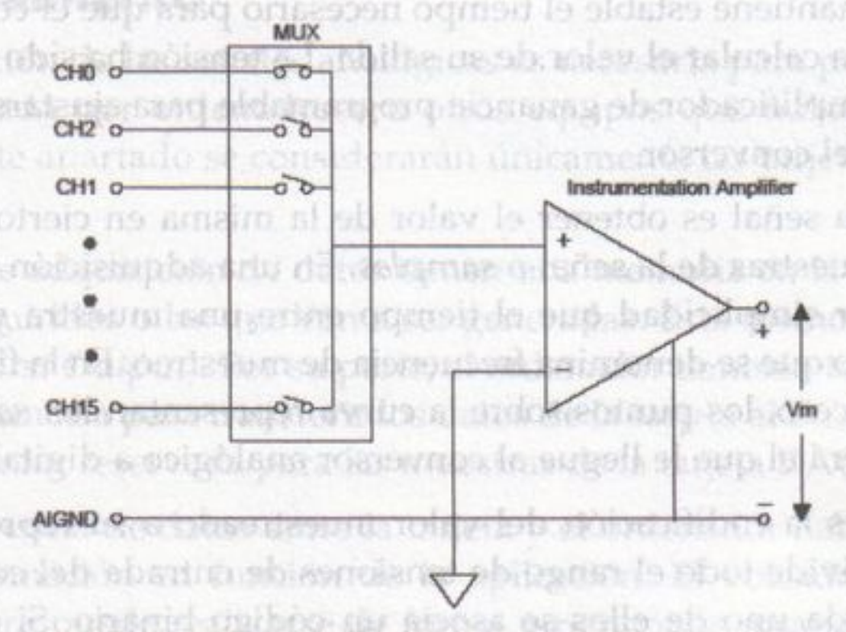


Figura 7-4. Entrada analógica con referencia común

- La ventaja de esta medida respecto a la anterior es que habrá más canales disponibles, pero todas esas entradas deben tener la misma referencia. No se recomienda si las tensiones son pequeñas, si hay una gran distancia entre el acondicionador y la tarjeta, los cables no están apantallados o se está inmerso en un ambiente ruidoso.
- En las medidas sin referencia (*nonreferenced single-ended,* NRSE) la señal exterior y la circuitería de la tarjeta DAQ no tienen una referencia común. En el circuito de la figura 7-5 se muestra cómo la señal exterior está referenciada al terminal AISEN-

SE mientras que la tensión en el interior de la tarjeta DAQ tendrá como referencia AIGND:

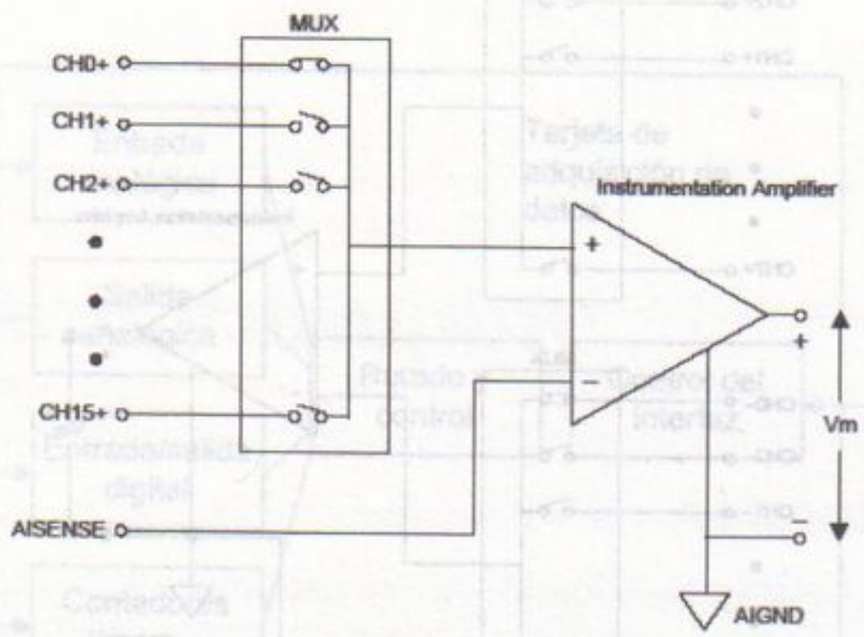


Figura 7-5. Entrada analógica sin referencia común

7.2.2 Conversión a digital

- El paso clave en la adquisición de datos es la digitalización. Para ello se emplea un circuito de muestreo y retención (*Sample and Hold*) que captura una tensión de la entrada y la mantiene estable el tiempo necesario para que el conversor analógico a digital pueda calcular el valor de su salida. La tensión ha sido previamente escalada por un amplificador de ganancia programable para ajustarse lo máximo posible al rango del conversor.
- Muestrear una señal es obtener el valor de la misma en ciertos momentos, esos valores son muestras de la señal o *samples*. En una adquisición continua se puede considerar por simplicidad que el tiempo entre una muestra y la siguiente será constante, es lo que se denomina frecuencia de muestreo. En la figura 7-6 (a) puede verse este proceso, los puntos sobre la curva representan los *samples*, el valor en esos puntos será el que le llegue al conversor analógico a digital.
- El otro paso es la codificación del valor muestreado a su representación digital. Para esto se divide todo el rango de tensiones de entrada del conversor en varios niveles y a cada uno de ellos se asocia un código binario. Si el conversor tiene muchos niveles para codificar la señal, necesitará más bits para codificar cada nivel, pero tendrá una mayor fidelidad al reproducir la señal. Al haber un número de niveles finito se producirá un error que corresponde a la diferencia entre el valor real de la señal analógica y el valor discreto que le es asignado. Este error se denomina error de cuantificación y es uno de los muchos errores que puede haber en un sistema de adquisición. En la figura 7-6 (b) se ha mostrado este proceso usando ocho niveles.

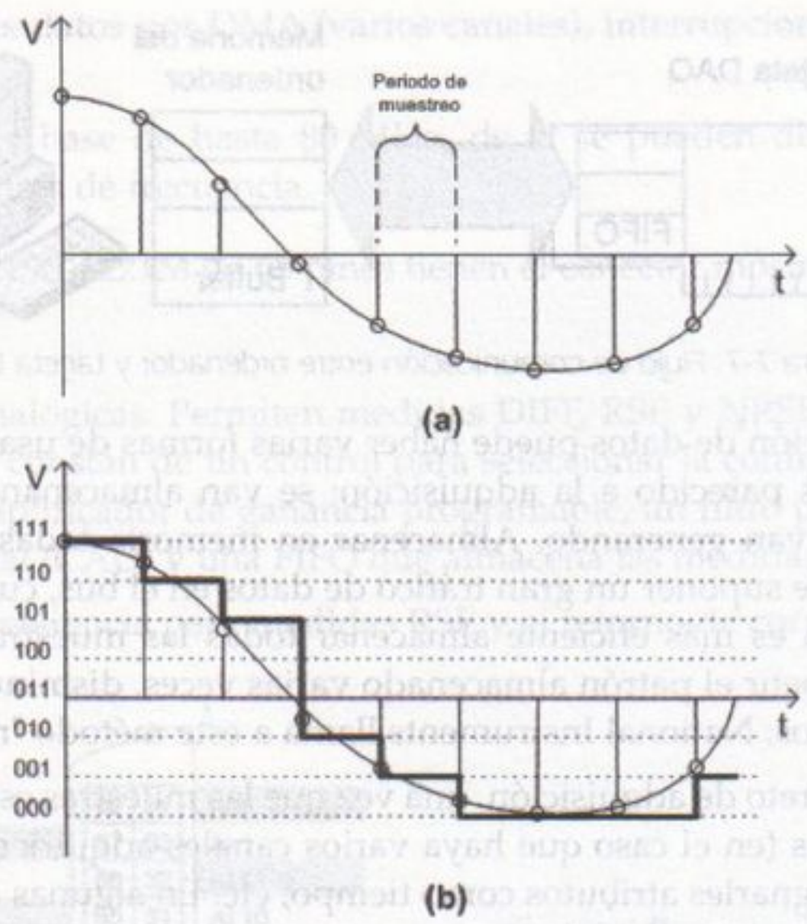


Figura 7-6. Cuantificación de una señal

7.2.3 Procesamiento

- La digitalización de las señales analógicas es necesaria para poder ser procesada y/o almacenada por ordenadores u otros equipos que trabajan en el dominio digital. En este apartado se considerarán únicamente las tarjetas internas a ordenadores.
- Las tarjetas de adquisición de datos tienen una memoria en la que almacenan las muestras adquiridas o las que van a ser generadas. Esta memoria es de tipo FIFO (*First Input First Output*). Por su parte, el ordenador también tendrá reservado un espacio de memoria para transferir los datos de la tarjeta al PC; la parte de memoria del ordenador reservada para las muestras de la tarjeta DAQ se llamará buffer.
- En la transferencia de datos entre la tarjeta y el ordenador habrá mayor o menor flujo de información en función de la aplicación. El volcado de datos de una memoria a otra puede realizarse de forma programada, usando interrupciones o mediante una transferencia DMA (*Direct Memory Access*); este último método suele ser el más eficaz al realizarse todo el proceso sin intervención de la CPU, la transferencia es gobernada por un controlador DMA que toma posesión del bus, mientras tanto la CPU puede realizar otras tareas (que no necesiten acceder al bus).
- Las tarjetas DAQ pueden tener varias FIFO, por ejemplo uno para la adquisición analógica, otro para la generación y otro para los canales digitales. Tanto las memorias FIFO como los *buffers* tienen un tamaño limitado, por lo que se deben evitar los errores de *overflow* y *underflow* que se producen si se saturan las memorias.

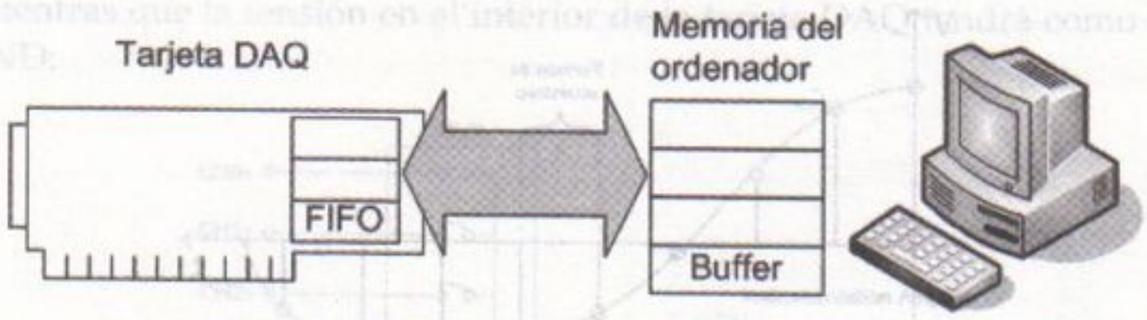


Figura 7-7. Flujo de comunicación entre ordenador y tarjeta DAQ

- Para la generación de datos puede haber varias formas de usar los buffers. El primer método es parecido a la adquisición: se van almacenando las muestras en memoria y se van generando. Almacenar en memoria todas y cada una de las muestras puede suponer un gran tráfico de datos en el bus, cuando se genera una señal periódica es más eficiente almacenar todas las muestras en memoria una única vez y repetir el patrón almacenado varias veces, disminuyendo así la transferencia de datos; National Instruments llama a este método "regeneración".
- En el caso concreto de adquisición, una vez que las muestras están en el buffer sólo resta ordenarlas (en el caso que haya varios canales adquiriendo datos a la vez), escalarlas y asignarles atributos como tiempo, etc. En algunas aplicaciones de alto rendimiento estos pasos podrían posponerse para hacerlos offline en lugar de hacerlos en tiempo real, y así evitar una sobrecarga de tareas a la CPU que podría provocar los errores antes comentados.
- Actualmente existe una tendencia a integrar en la propia tarjeta parte o todo el procesado que puedan requerir las medidas, es lo que National Instruments llama "DAQ inteligente". Se trata de dispositivos que usan elementos reprogramables como FPGAs que permiten realizar en la misma tarjeta y mediante hardware complejos cálculos a alta velocidad, para más información puede consultarse el último capítulo en la seccion dedicada a los sistemas CompactRIO.

7.3 Tarjeta de adquisición de datos NI PCI-6221M

La serie M de las DAQ de National Instruments está formada por:

- Entradas analógicas: una precisión de hasta 18 bits, una velocidad de 250 kS/s y hasta 80 entradas analógicas (la 6221 tiene 16, 8 en modo diferencial). Rangos de entrada programables de ±10, ±5, ±1 o ±0,2 V. FIFO de 4.095 samples.
- Salidas analógicas: hasta 4 salidas de 16 bits, rango de ±10 V y velocidad de 833 kS/s (la 6221 tiene 2 salidas). FIFO de 8.191 samples para todos los canales.
- Líneas digitales: hasta 48 líneas de entrada-salida TTL/CMOS (la 6221 tiene un puerto de 8 líneas, pero puede tener otros dos canales digitales usando las líneas PFI).
- Contadores: dos contadores/timers de 32 bits.
- Triggering analógico o digital, bus RTSI para sincronizarse con otras tarjetas.

- Transferencia de datos por DMA (varios canales), interrupciones o de forma programada.
- Reloj interno de base de hasta 80 MHz, de él se pueden derivar otras señales mediante divisores de frecuencia.

Las tarjetas NI PCI/PXI-6221M de 68 pines tienen el conector mostrado en la figura7-8. Las líneas son:

- AI: entradas analógicas. Permiten medidas DIFF, RSE y NRSE. Los canales están multiplexados, constan de un control para seleccionar la configuración de los terminales, un amplificador de ganancia programable, un filtro pasa bajo, conversor analógico-digital (CAD) y una FIFO que almacena las medidas.
- AIGND: es la referencia para medidas RSE y el retorno de corrientes para DIFF.

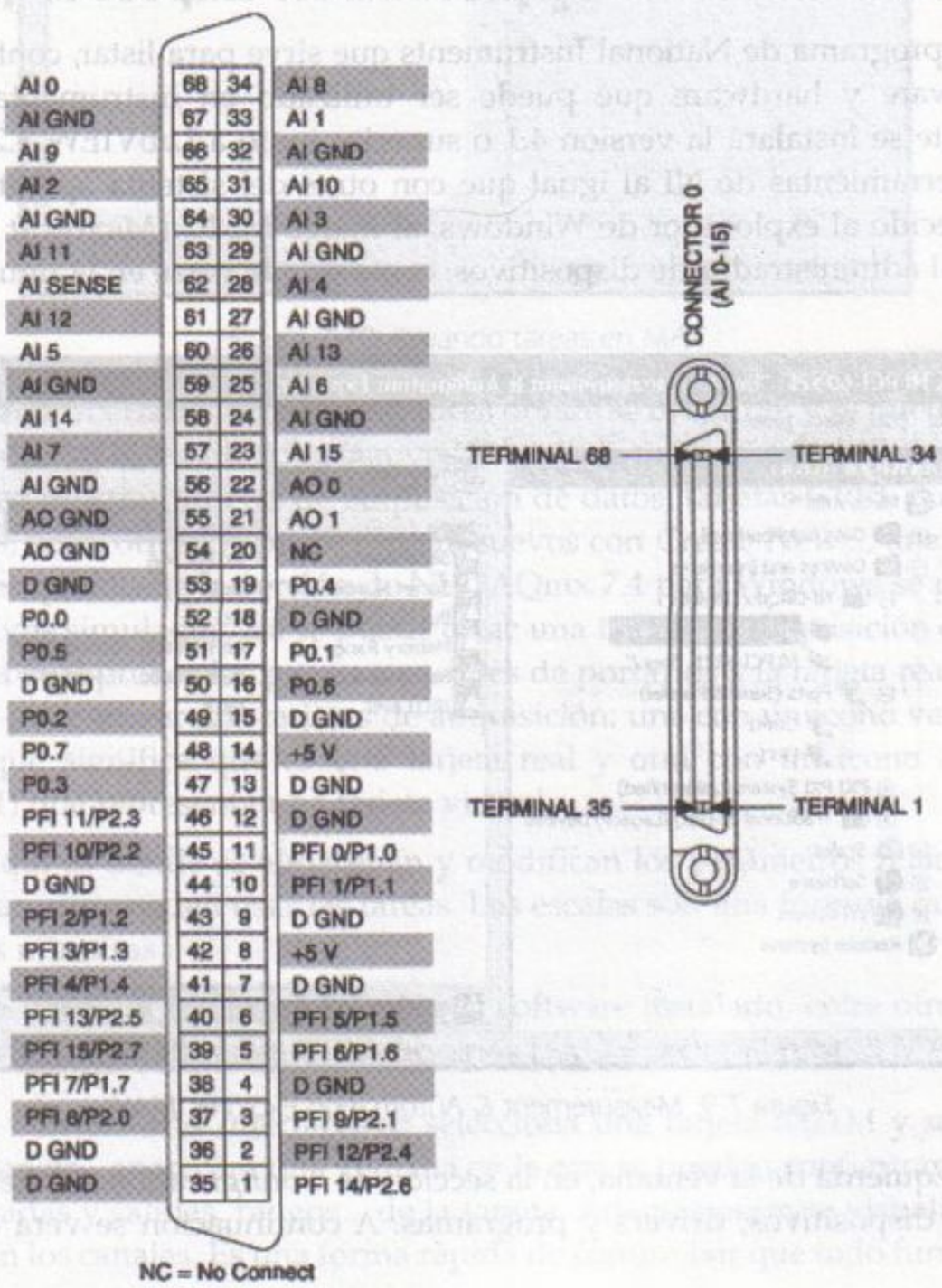


Figura 7-8. Conector de una tarjeta de adquisición de datos

- AISENSE: es la referencia para medidas NRSE.
- AO: salidas analógicas. Constan de una FIFO donde se almacenan las muestras que deben escribirse y conversores digital-analógico.
- AOGND: referencia para las salidas analógicas.
- PO: canales digitales, sirven tanto para entrada como para salida. En la serie M todas las líneas pertenecen al puerto 0. Las líneas PFI (*Programmable Function Interface*) también pueden configurarse como el canal digital P1 y P2.
- DGND: referencia para las líneas digitales.

Las tres referencias AIGND, AOGND y DGND están conectadas internamente en la tarjeta.

7.4 Measurement & Automation Explorer (MAX)

MAX es un programa de National Instruments que sirve para listar, configurar y probar el software y hardware que puede ser utilizado en instrumentación virtual. Normalmente se instalará la versión 4.1 o superior junto a LabVIEW 8.20. Interactúa con otras herramientas de NI al igual que con otras del sistema operativo; tiene un aspecto parecido al explorador de Windows, al marco MMC (*Microsoft Management Console*) y al administrador de dispositivos, como puede verse en la figura 7-9.

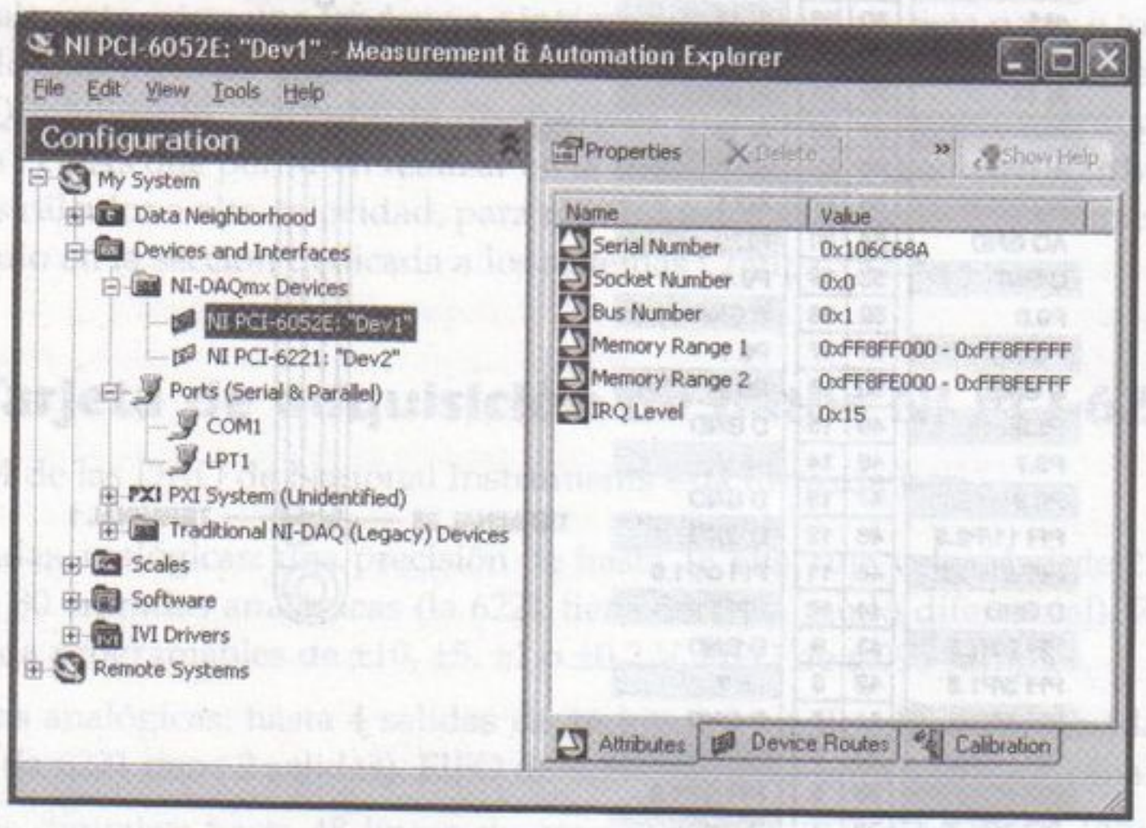


Figura 7-9. Measurement & Automation Explorer (MAX)

En la parte izquierda de la ventana, en la sección de *Configuration* se pueden explorar los distintos dispositivos, drivers y programas. A continuación se verá cada uno de ellos.

- **Data Neighborhood**: en esta carpeta es donde se crean los canales virtuales, tareas, etc., de los diferentes dispositivos que van a ser utilizados. Para ello se debe seleccionar esta opción, se presiona el botón *Create New...* de la barra de herramientas o del menú contextual, y seguir el asistente; en el caso de elegir *NI-DAQmx Task* aparecerá la pantalla de la figura 7-10.

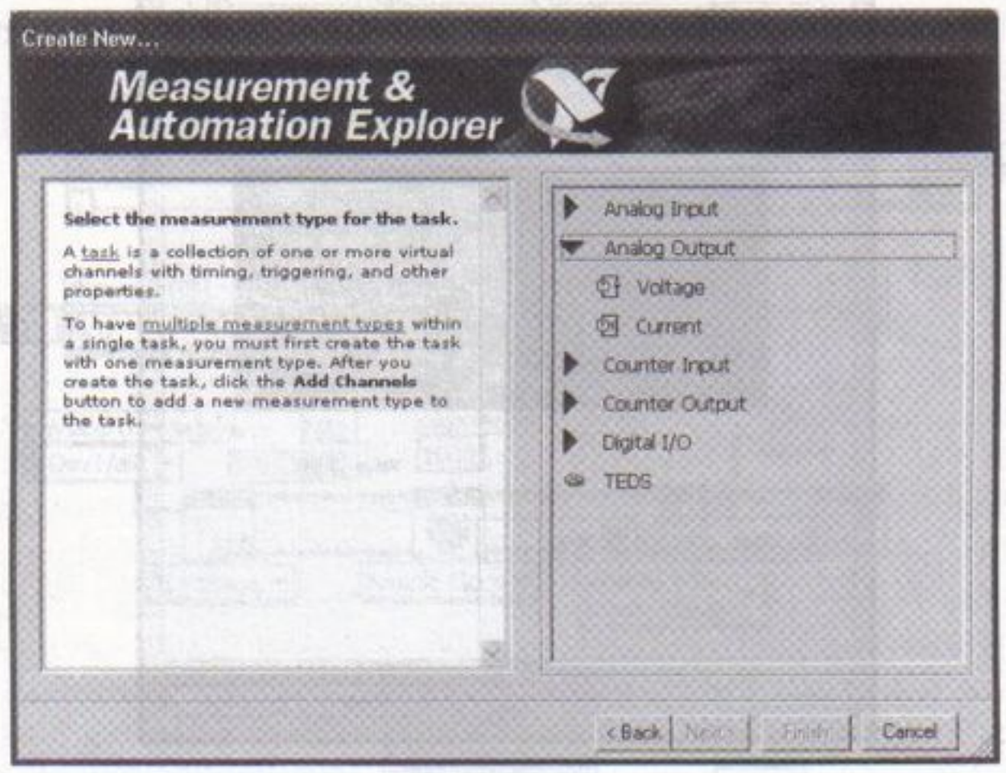


Figura 7-10. Creando tareas en MAX

- **Devices and Interfaces**: en esta carpeta es donde se muestran y se pueden editar los atributos de los diferentes dispositivos físicos que están instalados en el ordenador. Se pueden mostrar tarjetas de adquisición de datos, tarjetas GPIB, puertos serie o paralelos, entre otros. Se pueden crear nuevos con *Create New...*, una opción interesante es que desde la versión de NI-DAQmx 7.4 para Windows se pueden crear dispositivos simulados, así se puede crear una tarjeta de adquisición de datos virtual en la cual probar los programas antes de portarlos a la tarjeta real. En la figura 7-9 se pueden ver dos tarjetas de adquisición: una con un icono verde (NI PCI-6052E) que significa que es una tarjeta real y otra con un icono amarillo (NI PCI-6221) que representa una tarjeta virtual.
- **Scales**: aquí es donde se almacenan y modifican los parámetros relacionados con las escalas que pueden usar las tareas. Las escalas son una fórmula que se aplica a todas las muestras.
- **Software**: muestra información sobre el software instalado, entre otros LabVIEW, las librerías VISA, librerías GPIB, librerías DAQ y sobre el propio MAX.

Si dentro de *Devices and Interfaces* se selecciona una tarjeta 6221M y se presiona el botón *Test Panels...* aparecerá una ventana en la que se pueden configurar los distintos tipos de entradas y salidas, rangos... de la tarjeta. Además permite visualizar las señales que hay en los canales. Es una forma rápida de comprobar que todo funciona correctamente.

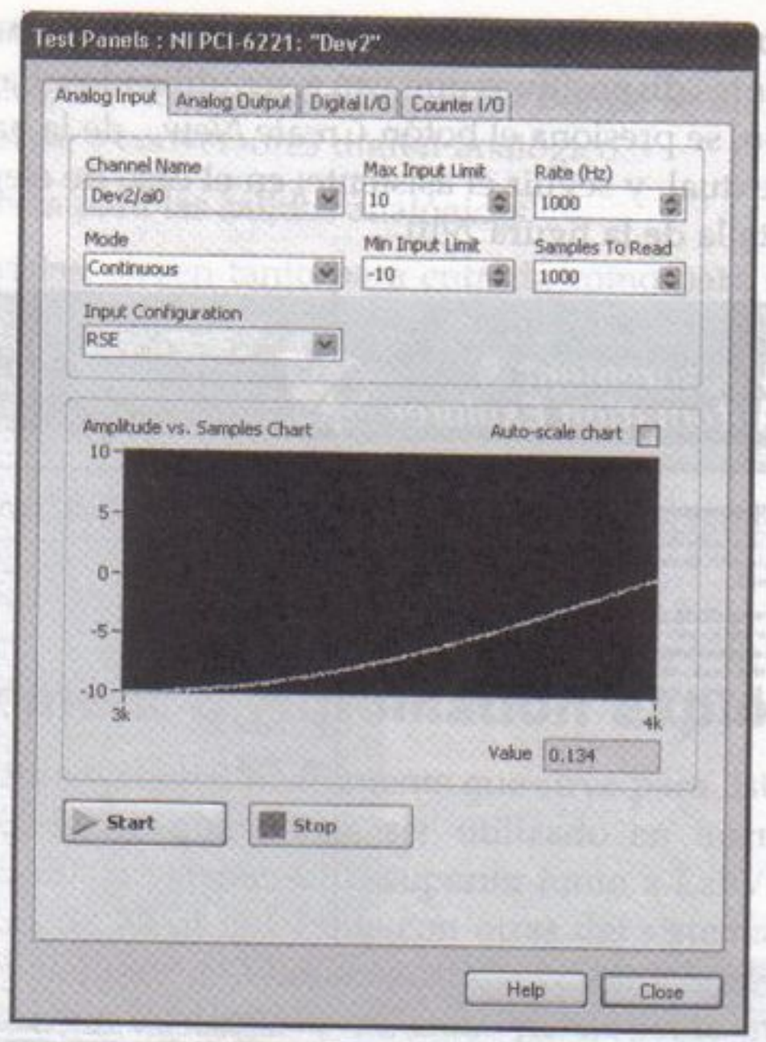


Figura 7-11. Panel de test en MAX

- Una configuración hecha con MAX puede guardarse y recuperarse más tarde en *File > Import/Export*. El fichero resultado puede importarse en un proyecto de LabVIEW y distribuirse junto con los programas realizados. En LabVIEW 8.20 también puede importarse y exportarse la configuración de MAX en *Measurement I/O > MAX Configuration*.

7.5 DAQ en LabVIEW

LabVIEW proporciona tres tipos de drivers para adquisición de datos: Traditional NI-DAQ, VIs Express y NI-DAQmx.

- Los *Traditional NI-DAQ* son los de más bajo nivel, usados en versiones antiguas de LabVIEW, actualmente se conservan por razones de compatibilidad. National Instruments sólo aconseja usarlos con algunos dispositivos antiguos y en situaciones muy concretas. No presentan la capacidad de multiproceso que tienen las nuevas versiones.
- Los VIs Express se introdujeron en LabVIEW 7 y hacen la programación mucho más sencilla y compacta que con los VIs normales al agrupar a varios VIs en un único nodo muy configurable.
- Los *NI-DAQmx* también se han introducido con la versión 7 de LabVIEW para Windows. Presentan muchas ventajas respecto a los tradicionales, como por ejemplo mayor integración con MAX, un nivel de abstracción mayor, multiproceso

(*multithread*), más robustez, mayor estabilidad, etc. La sencillez la consiguen básicamente a través de VIs polimórficos (agrupación de varios VIs distintos bajo un único fichero e icono, sería el equivalente a la sobrecarga de funciones), con lo que logran agrupar las tareas que hacen varios VIs tradicionales en uno sólo.

La figura 7-12 es una comparación sobre un programa para leer una entrada analógica de las tres formas mencionadas anteriormente.

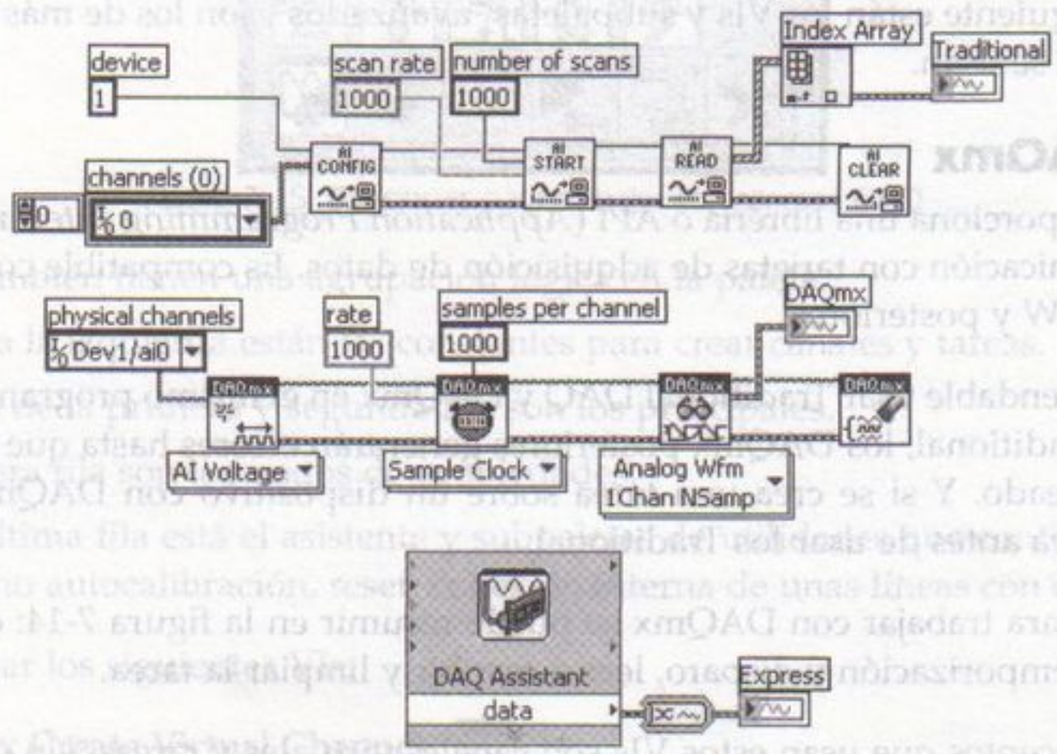


Figura 7-12. Comparativa de código (Tradicional DAQ, DAQmx y Express)

7.5.1 Traditional DAQ

Ya se han comentado las características de estos VIs en el apartado anterior. Su menú está en NI *Measurements > Data Acquisition*. Esta paleta presenta a su vez una serie de subpaletas, cada una de ellas para un tipo de configuración: entradas y salidas analógicas y digitales, contadores y calibración.

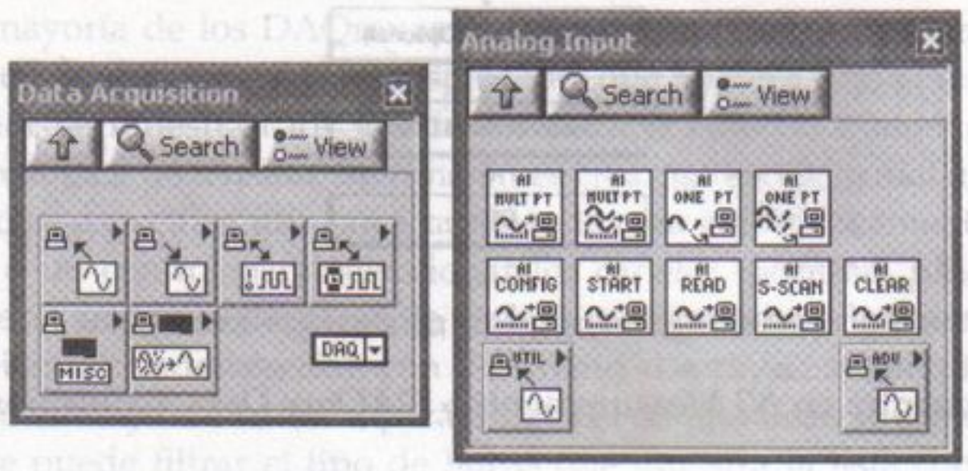


Figura 7-13. Antiguos VIs para trabajar con tarjetas DAQ

La organización de las subpaletas es la siguiente:

- En la fila superior están los VIs "sencillos". Hacen funciones típicas como leer un dato. Entre otras cosas tienen un manejo automático de los errores. Están hechos para que funcionen "tal cual", si se abren se puede comprobar como están hechas a partir de varios VIs "intermedios" y de otras utilidades.
- En la segunda fila están los VIs "intermedios" y abajo a la izquierda las utilidades. Tienen más funcionalidad y eficiencia que los "sencillos". Uno sólo de estos VIs no puede realizar una tarea completa, por lo que necesita de otros.
- En la siguiente están los VIs y subpaletas "avanzados", son los de más bajo nivel y rara vez se usan.

7.5.2 DAQmx

DAQmx proporciona una librería o API (*Application Programming Interface*) que facilita la comunicación con tarjetas de adquisición de datos. Es compatible con la versión 7 de LabVIEW y posteriores.

No es recomendable usar Traditional DAQ y DAQmx en el mismo programa, ya que si se usa un Traditional, los DAQmx posteriores generarán errores hasta que el dispositivo sea reseteado. Y si se crea una tarea sobre un dispositivo con DAQmx, hay que borrar la tarea antes de usar los Traditional.

El método para trabajar con DAQmx se puede resumir en la figura 7-14: crear tareas, configurar temporización y disparo, leer o escribir y limpiar la tarea.

Algunos conceptos que usan estos VIs son canales virtuales y tareas. Un canal virtual es una colección de líneas o puertos físicos que se agrupan para realizar la misma tarea e información asociada, como la configuración o las escalas. Una tarea es algo más general, representa la medida o generación de señal que quiere realizarse, agrupa los canales, *timing*, *triggering* y otras propiedades.

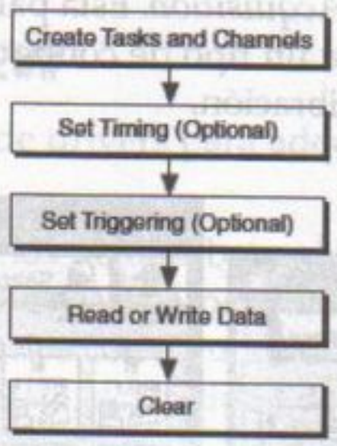


Figura 7-14. Flujo de programa para trabajar con DAQmx

La paleta se encuentra en *NI Measurements > DAQmx Data Acquisition.*

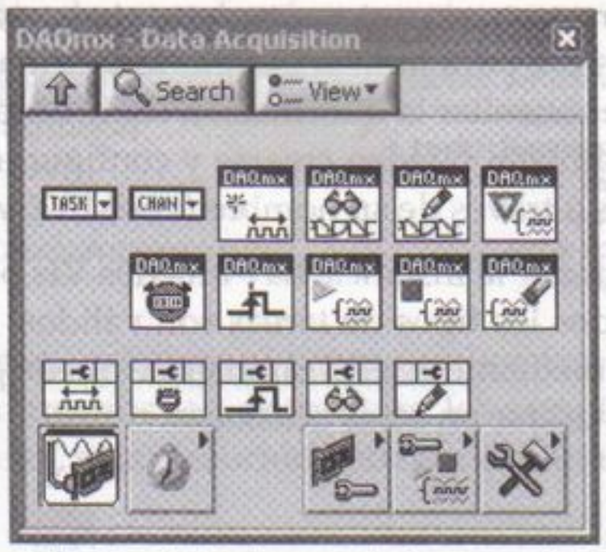


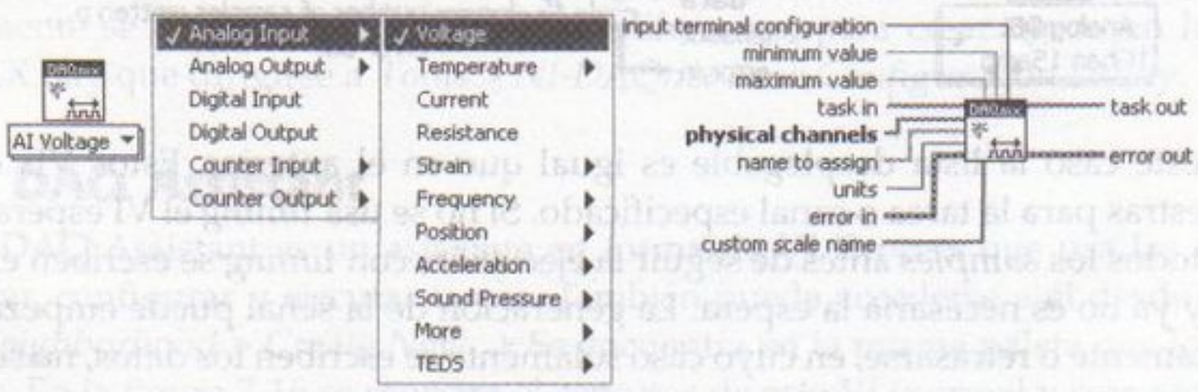
Figura 7-15. Funciones para trabajar con tarjetas DAQ

Estos VIs también tienen una agrupación lógica en la paleta:

- Arriba a la izquierda están las constantes para crear canales y tareas.
- Los VIs de la primera y segunda fila son los principales.
- La tercera fila son los nodos de propiedades.
- En la última fila está el asistente y subpaletas de utilidades que contienen funciones como autocalibración, reset, conexión interna de unas líneas con otras, etc.

Cabe destacar los siguientes VIs:

- **DAQmx Create Virtual Channel**

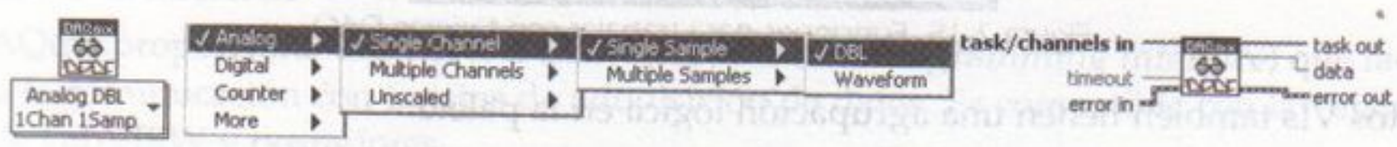


Como la mayoría de los DAQmx, este VI es polimórfico y la instancia a usar es seleccionada mediante una lista desplegable que aparece bajo el icono. En la imagen anterior se muestran los terminales para el caso de querer configurar una entrada analógica de tensión. Su función es crear un canal virtual y añadirlo a una tarea, si no se especifica una tarea también la crea automáticamente. Al crear una constante o un control para seleccionar los canales (terminal *physical channels*) debe aparecer una lista desplegable en la que se muestran los que están disponibles, también se pueden escribir con el formato «`Dev1/ai0`», donde `1` es el número de dispositivo y `ai0` la línea, para usar más de una línea se separan por comas. Además se puede filtrar el tipo de líneas que muestra la lista con la opción *I/O Name Filtering* del menú desplegable. También se puede aceptar un *string* con el mismo contenido.

Sobre el VI particular mostrado antes simplemente hay que decir que el *input terminal configuration* sirve para indicar el tipo de medida: diferencial RSE o NRSE. En la ayuda se puede consultar la información sobre cada una de las instancias que componen este VI. También pueden crearse y configurar tareas mediante un asistente y acceder a ellas a través de su nombre (ver MAX y *DAQ Assistant Express*).

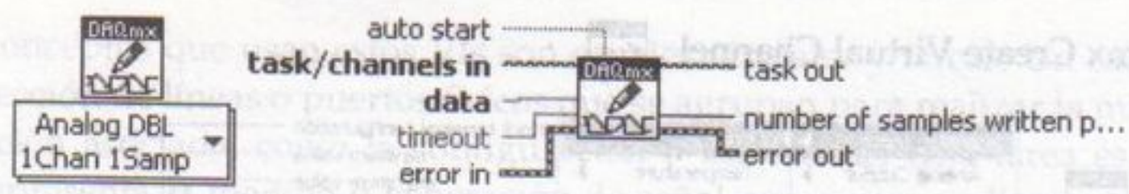
En la lista para seleccionar la instancia hay muchos VIs para elegir. Para una determinada tarjeta no todos los tipos de tareas son válidos. Por último hay que decir que una tarea creada mediante este VI también se puede guardar para poder modificarse desde MAX o usarse en otras ocasiones.

- **DAQmx Read**



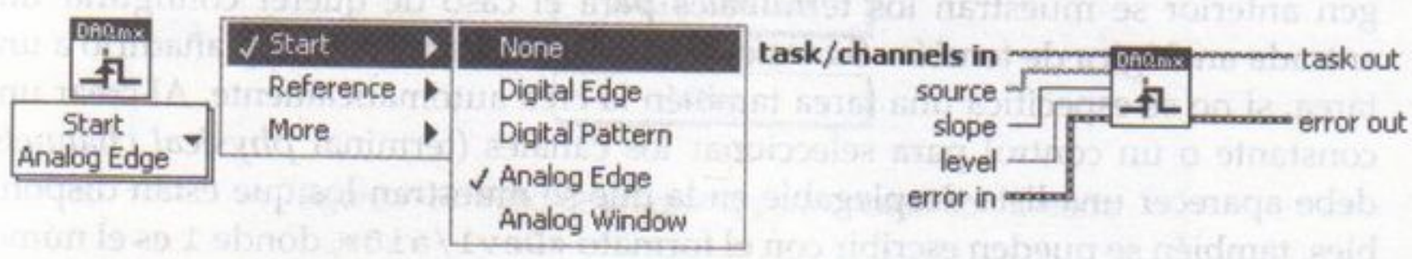
Lee muestras para el canal o tarea especificada, ésta puede haber sido generada con el VI anterior o con MAX. Al seleccionar una instancia en concreto se determina el formato de la lectura: un simple dato, *arrays*, etc. Todas las instancias incluyen un parámetro de *timeout* y, en las que se puede aplicar, otro parámetro para indicar el número de *samples* por canal.

- **DAQmx Write**



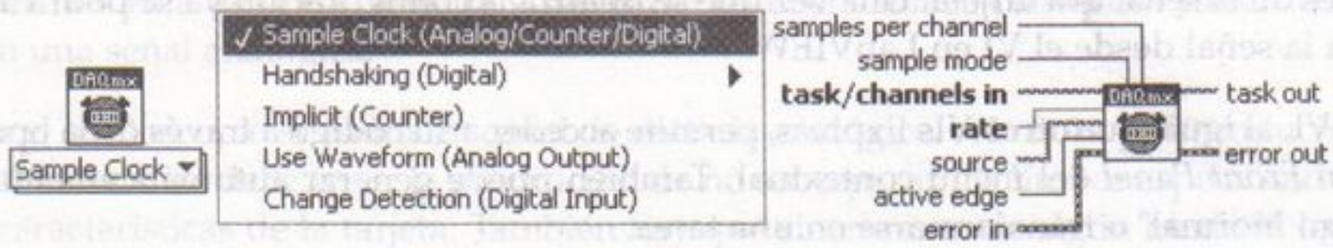
En este caso la lista desplegable es igual que en el anterior. Estos VIs escriben muestras para la tarea o canal especificado. Si no se usa *timing* el VI espera a generar todos los *samples* antes de seguir la ejecución, con *timing* se escriben en el buffer y ya no es necesaria la espera. La generación de la señal puede empezar inmediatamente o retrasarse, en cuyo caso solamente se escriben los datos, no se "sacan" al exterior.

- **DAQmx Trigger**



Configura la forma en que comienza una tarea, es decir, su condición de disparo. Un *trigger* puede tener diversas fuentes, tanto analógicas como digitales, internas a la tarjeta o externas. Un *trigger* de tipo *start* indica la condición de comienzo de la tarea, el tipo *reference* indica la condición de parada.

- **DAQmx Timing**



Configura los parámetros relativos al tiempo en una tarea y crea un buffer cuando es necesario. Estos parámetros son, entre otros, los *samples* por segundo, un terminal para usar como reloj, si la ejecución debe ser continua o finita, etc.

- **DAQmx Clear Task**

Detiene y borra la tarea. Es útil cuando se crea una tarea dentro de un bucle.

Además de estos VIs y otros, DAQmx también tiene un gran número de propiedades con las que se puede tener un gran nivel de control sobre la tarjeta.

NI-DAQmx Base es otro API para adquisición de datos que no debe confundirse con NI-DAQmx, aunque externamente son VIs muy parecidos a los de NI-DAQmx, el método de programación es el mismo y el nombre puede llevar a confusión. Un pequeño detalle es que los iconos de DAQmx Base tienen en su parte superior las letras «mxBase», mientras que los "normales" tienen «DAQmx». Internamente se basan en APIs totalmente diferentes; para crear tareas en lugar del MAX hay que dirigirse a *Tools > NI-DAQmx Base Configuration Utility*.

7.5.3 DAQ Assistant

DAQ Assistant es un asistente en forma de VI Express que usa los DAQmx para crear, configurar y ejecutar tareas. También puede accederse a él desde el MAX (*Data Neighborhood > Create New...*). Se encuentra en la misma paleta que los VIs de DAQmx. En la figura 7-16 se muestra el aspectos de este VI (normal y como icono).

Figura 7-16. DAQ Assistant

Al situarlo en el Diagrama de Bloques aparece automáticamente un asistente como el de la figura 7-10. En él hay que elegir el tipo de medida (por ejemplo una entrada analógica de tensión). Después se deben seleccionar los canales, el asistente muestra una lista de los disponibles. A continuación aparecerá la ventana de la figura 7-17 en la que configurar la adquisición, modificando parámetros como el rango de entrada, tipo de conexión, etc. Al igual que en MAX, también aparece la pestaña *Connection Diagram*

en la que se muestra de una manera esquemática la forma correcta de realizar las conexiones de la señal a la tarjeta. Una vez que se finaliza la configuración ya se podrá acceder a la señal desde el VI en LabVIEW.

Este VI, al igual que otros VIs Express, permite acceder a su código a través de la opción *Open Front Panel* del menú contextual. También puede generar automáticamente un código "normal" o transformarse en una tarea.

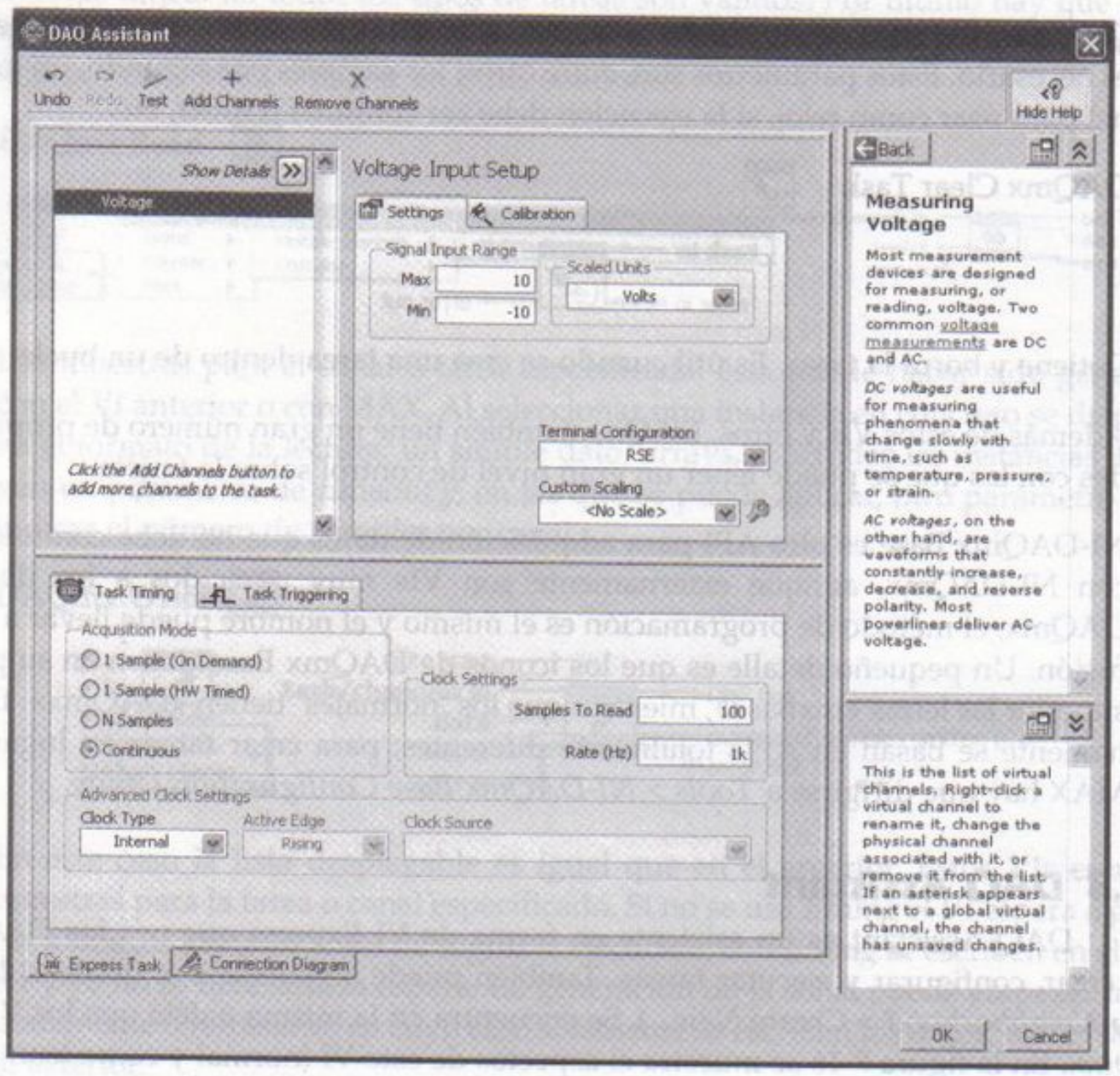


Figura 7-17. Configuración de DAQ Assistant

7.6 Ejemplos

7.6.1 Ejemplo I: Crear un osciloscopio y un generador de funciones con la tarjeta PCI-6221M

7.6.1.1 Explicación teórica

En este primer ejemplo se usarán las entradas y salidas analógicas de la tarjeta para crear un osciloscopio y un generador de funciones.

Para el generador de funciones se usará un canal de salida analógico. Se escribirá en el buffer la forma de onda deseada y mediante regeneración se convertirá continuamente en una señal analógica.

Para el osciloscopio se usará un canal de entrada analógico. A la hora de introducir una señal externa se debe tener la precaución de no sobrepasar los límites especificados en las características de la tarjeta. También hay que sincronizar de algún modo la lectura de datos en el buffer con la toma de muestras para que no haya errores de ningún tipo.

7.6.1.2 Código

Para el generador se creará una tarea y se configurarán los parámetros relativos a la temporización, después se creará una señal con el VI *Basic Function Generator* que es pasada a *DAQmx Write*, el cual está configurado con *auto start* para que empiece inmediatamente la generación. A continuación se ha colocado un bucle WHILE en el que el programa se mantendrá un tiempo indefinido antes de finalizar la tarea. Para no ocupar todo el tiempo del procesador en un bucle que no hace nada, se ha colocado en su interior un VI *Wait (ms)*. Mientras el programa esté en este bucle la tarjeta generará una y otra vez los datos escritos antes.

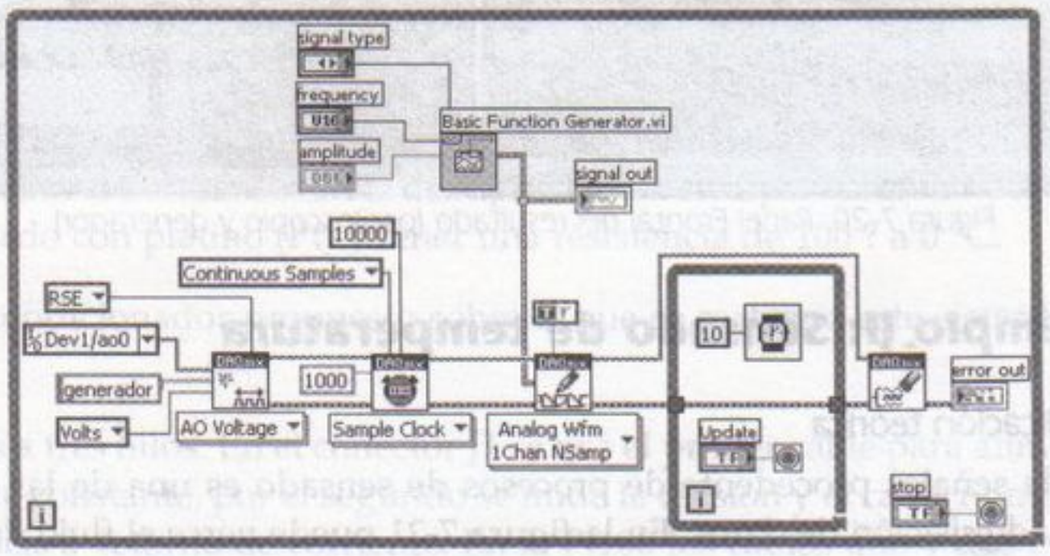


Figura 7-18. Generador con una tarjeta DAQ

El código del osciloscopio es parecido al anterior, sólo que esta vez en vez de configurar la tarea como salida analógica (*AO Voltage*), se hace como entrada (*AI Voltage*).

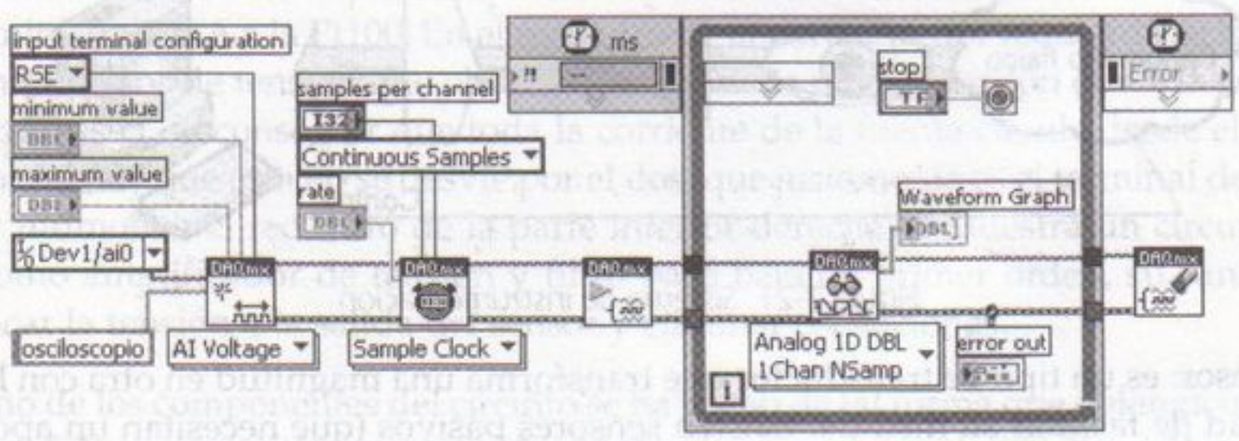


Figura 7-19. Osciloscopio con una tarjeta DAQ

También se ha usado un TIMED LOOP en lugar de un WHILE clásico para conseguir un intervalo de tiempo entre muestras más preciso. El bucle se repite cada segundo, dentro de él se leen todas las muestras almacenadas en el buffer.

7.6.1.3 Resultado

Para probar el funcionamiento de ambos programas se han programado canales diferentes y éstos se han cortocircuitado. El resultado puede verse en la figura 7-20.

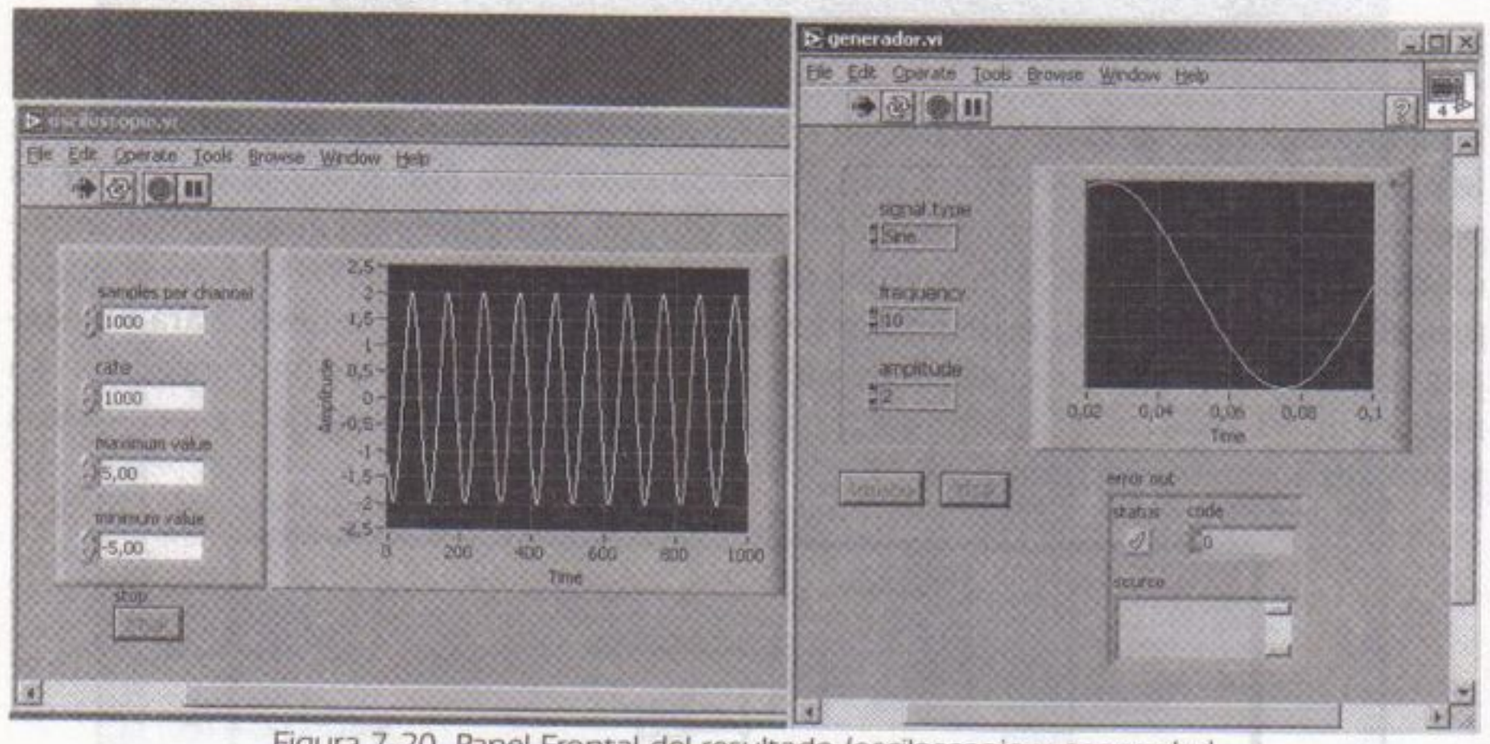


Figura 7-20. Panel Frontal del resultado (osciloscopio y generador)

7.6.2 Ejemplo II: Sensado de temperatura

7.6.2.1 Explicación teórica

La captura de señales procedente de procesos de sensado es una de las aplicaciones típicas de la adquisición de datos. En la figura 7-21 puede verse el flujo de un sistema de instrumentación para medir y controlar una variable física.

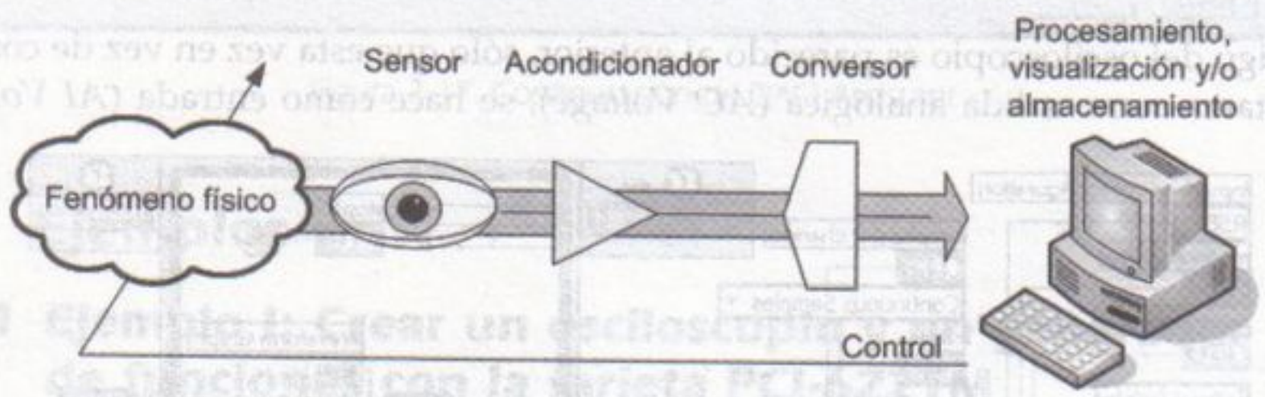


Figura 7-21. Sistema de instrumentación

- **Sensor**: es un tipo de transductor que transforma una magnitud en otra con la finalidad de facilitar su medida. Existen sensores pasivos (que necesitan un aporte de energía externo) y otros activos (que son capaces de generar su propia energía).

Los sensores eléctricos también se pueden clasificar en resistivos, capacitivos e inductivos, dependiendo de cual sea la característica eléctrica que varía.

- **Acondicionador**: la salida de los sensores no suele ser apta para su procesamiento, a veces no es lineal, otras veces tiene niveles demasiado pequeños, etc. La función del acondicionador es adaptar esa salida para facilitar que las etapas posteriores puedan procesarla. Dependen de las características del sensor utilizado.
- **Conversión a digital**: hasta ahora siempre se ha supuesto que se trabajaba con señales analógicas, si se quiere procesar la señal en el ámbito digital es necesario convertirla. Para ello se usan conversores analógicos digitales.
- **Procesamiento, visualización y/o almacenamiento**: esta parte puede estar en el ámbito digital o en el analógico. En la figura 7-21 se ha representado con un ordenador.
- **Control**: esta etapa no siempre está presente y, en caso de que exista, es la que cierra el bucle. Esto quiere decir que a partir de la señal procesada puede haber una actuación para modificar la variable medida con el fin de que ésta se mantenga dentro de unos límites o reaccione ante alguna variación.

Para el caso concreto de temperatura, los sensores más habituales son: termopares, termistores, detectores de temperatura resistivos (RTD) y sensores de temperatura de estado sólido (circuitos integrados).

Este ejemplo trata sobre como adquirir datos provenientes de un sensor de temperatura, concretamente de un sensor RTD del tipo Pt100. Este tipo de sensor debe su nombre a estar fabricado con platino (Pt) y tener una resistencia de 100 ? a 0 ºC.

El circuito acondicionador propuesto sobre el que se realizará este ejemplo es el de la figura 7-22.

La medida es a tres hilos. En el conector J1 se usa el primer cable para alimentar el sensor a corriente constante, por el segundo se mide la tensión y el cable número tres sirve como referencia y retorno de corriente. En la Pt100 los cables uno y dos están unidos. La ventaja de esta configuración es que la resistencia de los cables uno y dos no afecta a la medida.

En este circuito se han señalado las partes más importantes, en la figura 7-22 se ha marcado con el recuadro de la parte superior izquierda una fuente de corriente constante que suministra 1 mA a la Pt100. En el recuadro de la parte inferior izquierda se ha señalado un seguidor de tensión que obtiene a su salida la misma tensión que a la entrada, su función es la de conseguir que toda la corriente de la fuente circule desde el terminal uno al tres y que ésta no se desvíe por el dos, que justamente es el terminal de medida. Por último, en el recuadro de la parte inferior derecha se muestra un circuito que sirve como amplificador de tensión y filtro paso bajo de primer orden, su función es amplificar la tensión a la salida del sensor y eliminar oscilaciones.

El diseño de los componentes del circuito se ha hecho de tal forma que obtenga una tensión de 0,5 V para una temperatura de -50 ºC aproximadamente (80 ?) y 4 V para una

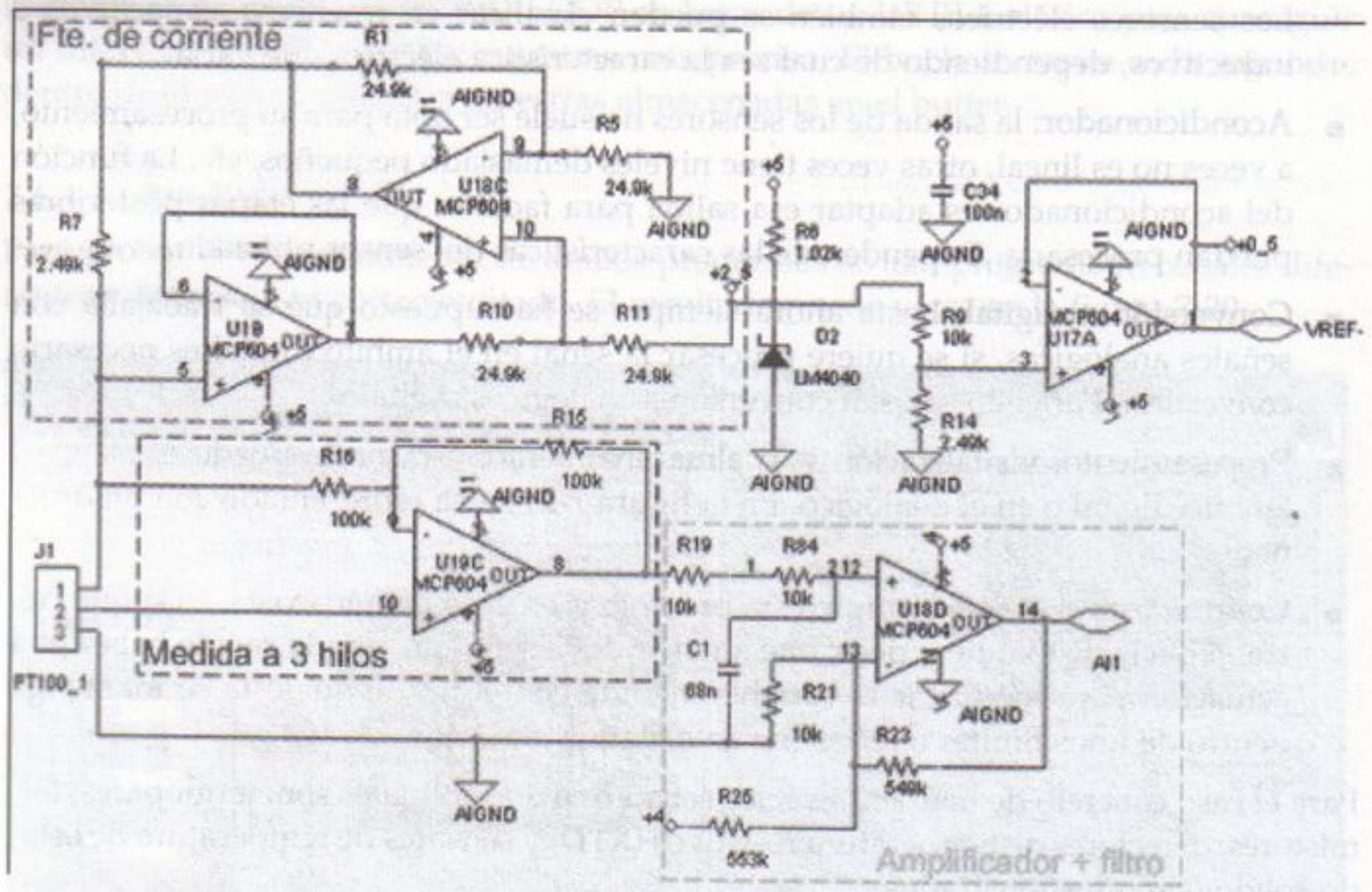


Figura 7-22. Circuito acondicionador

temperatura de unos 100 ºC (145 ?). El análisis del circuito, realizando algunas aproximaciones, se puede expresar como $V_{salida} = 55 \cdot 0{,}001 \cdot R_{Pt100} - 4$.

Para la conversión a digital se empleará la tarjeta una adquisición de datos.

7.6.2.2 Código

En principio se podría optar por realizar la conversión de la tensión leída a temperatura mediante una fórmula o método similar. Sin embargo en esta ocasión se utilizará el escalado, para ello sobre la entrada *custom scale name* del VI *DAQmx Create Virtual Channel* se creará una constante y en ella se elegirá la opción *Browse*, en la ventana que aparece se creará una nueva escala de tipo lineal. Después de elegir el nombre se

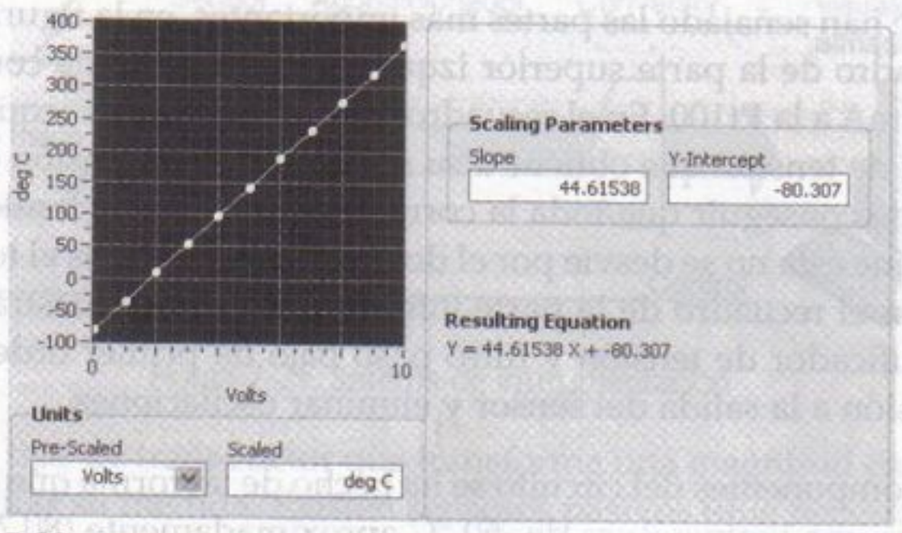


Figura 7-23. Escalado de las medidas. Conversión de tensión a temperatura

podrán introducir los coeficientes que configurarán el escalado de las lecturas de esa tarea. También se puede emplear MAX para crear la escala.

Mediante una tabla de calibración de una Pt100 y una caja de décadas se pueden obtener experimentalmente los valores de los coeficientes mostrados en la figura 7-23.

El resto del programa de la figura 7-24 ya ha sido explicado anteriormente. Simplemente adquiere una muestra cada cierto tiempo. El valor obtenido de esa muestra ya ha sido escalado y sus unidades serán grados centígrados.

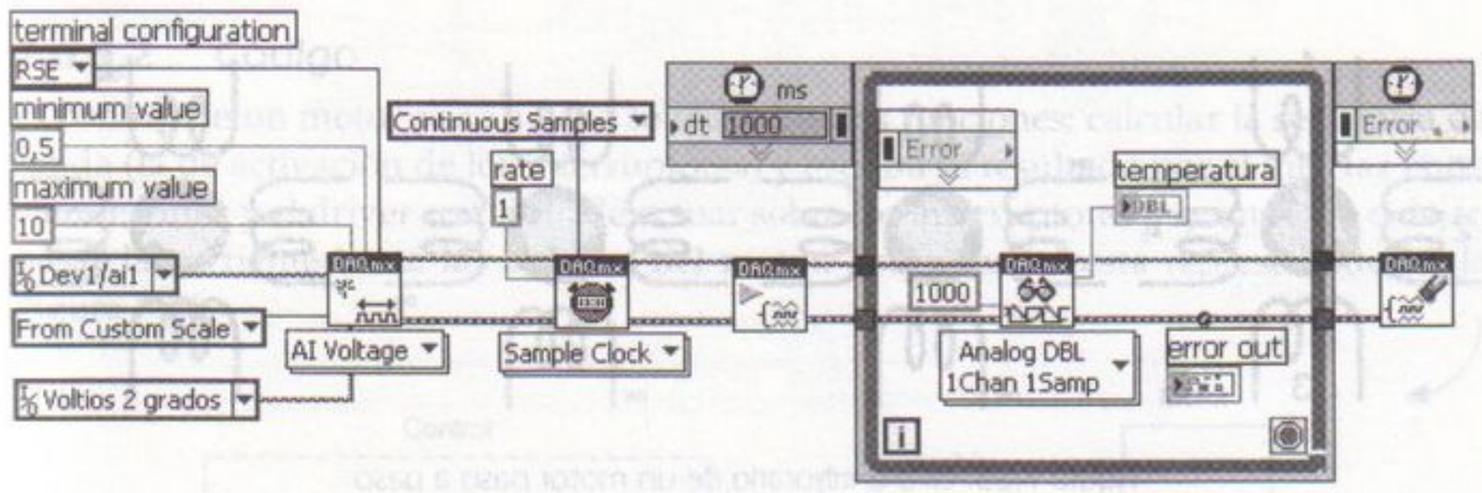


Figura 7-24. Programa para leer temperatura

7.6.3 Ejemplo III: Control de un motor paso a paso

7.6.3.1 Explicación teórica

Un motor paso a paso es un conversor electromecánico que transforma la energía eléctrica en mecánica. Convierten un tren de impulsos eléctricos en un movimiento angular. En ellos su eje gira un determinado ángulo, también llamado paso, dependiendo de los impulsos eléctricos que le sean aplicados. El paso determinará la precisión del motor, así para un paso pequeño el motor podrá girar con mayor precisión, pero necesitará más pasos para dar una vuelta completa. El paso de los motores suele variar desde 90° a ángulos menores de 1°.

Los motores paso a paso de imán permanente son los más comunes. El rotor es un imán permanente y el estator contiene unos polos salientes en forma de dientes sobre los que van las bobinas. El modo en que están conectadas las bobinas determina el tipo de motor paso a paso:

- Bipolar: sobre las bobinas del estator se puede hacer pasar una corriente eléctrica que creará un campo magnético. Este campo magnético hará que el rotor gire hasta encontrar una posición de equilibrio. Si en ese momento las corrientes cambian, el campo magnético también cambia y el rotor deberá girar otra vez.
- Unipolar: las bobinas del estator están divididas en dos debido a que su punto central está puesto a referencia. La corriente puede circular por una de las dos partes de la bobina dependiendo de un conmutador. De esta forma la posición del con-

mutador determina el sentido de la corriente en la bobina y ésta a su vez determina la polaridad del campo magnético generado y el sentido del giro.

El control de los motores se reduce a activar y desactivar las bobinas para hacer que el rotor gire. La activación y desactivación se realiza a través de un driver que consta de interruptores; cuando los interruptores se cierran circula una corriente por la bobina que provoca un campo magnético el cual atrae o repele el imán del rotor. En la figura 7-25 puede verse una secuencia de activación de bobinas para hacer girar el rotor en sentido antihorario.

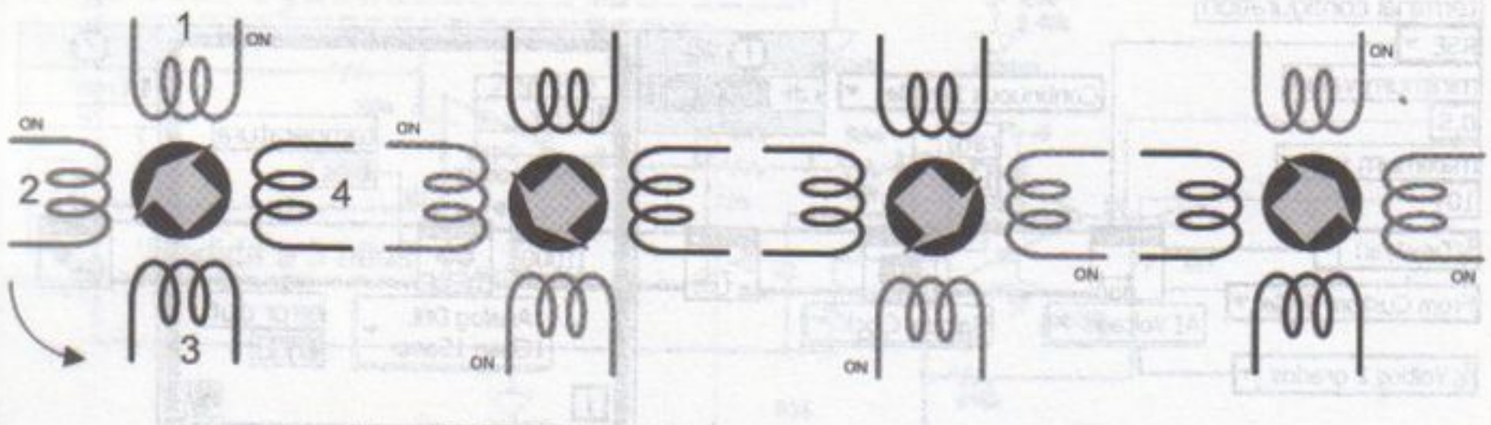


Figura 7-25. Giro antihorario de un motor paso a paso

En la secuencia de paso completo (*full step*) el rotor gira un paso por cada pulso, es la secuencia representada en la figura 7-25 y en la Tabla 1 (b). Para girar en sentido horario se sigue la secuencia[2] a y para sentido antihorario b . En la secuencia de medio paso (*half step*), como su nombre indica, el motor avanza medio paso por cada pulso; se basa en la activación de una y dos bobinas, como puede verse en la Tabla 2.

Tabla 1 - Secuencia de paso entero para giro horario (a) y antihorario (b)

Paso	S1 (a)	S2 (a)	S3 (a)	S4 (a)	S1 (b)	S2 (b)	S3 (b)	S4 (b)
1	ON			ON	ON	ON		
2			ON	ON		ON	ON	
3		ON	ON				ON	ON
4	ON	ON			ON			ON
1	ON			ON	ON	ON		

Tabla 2 - Secuencia de medio paso para giro horario (a) y antihorario (b)

Paso	S1 (a)	S2 (a)	S3 (a)	S4 (a)	S1 (b)	S2 (b)	S3 (b)	S4 (b)
1	ON			ON	ON	ON		
2				ON		ON		
3			ON	ON		ON	ON	

2 Las casillas que no contienen nada están en OFF.

Paso	S1 (a)	S2 (a)	S3 (a)	S4 (a)	S1 (b)	S2 (b)	S3 (b)	S4 (b)
4			ON				ON	
5		ON	ON				ON	ON
6		ON						ON
7	ON	ON			ON			ON
8	ON				ON			
1	ON			ON	ON	ON		

7.6.3.2 Código

El control de un motor paso a paso se reduce a dos funciones: calcular la secuencia de salida (la de activación de los interruptores) y escribir el resultado por el interfaz entre el ordenador y el driver con el fin de actuar sobre los interruptores que impiden o dejan pasar la corriente hasta las bobinas del motor. Este proceso está representado en la Figura 7-26.

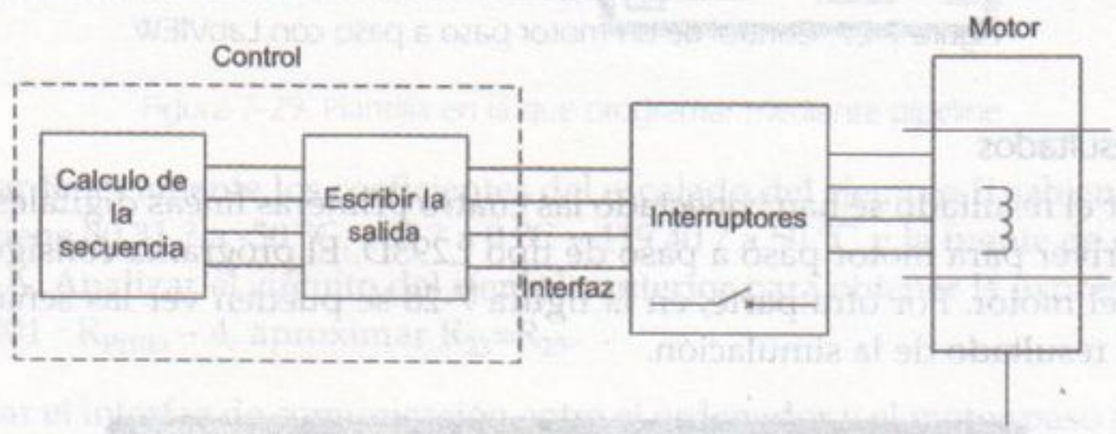


Figura 7-26. Control por software del driver de un motor paso a paso

El interfaz entre los interruptores y el ordenador serán las salidas digitales de una tarjeta de adquisición de datos. El programa primero crea una tarea *Digital Output*; dentro de un bucle se escribirá sobre esas líneas cada cierto tiempo, el dato que se escribe es una de las secuencias (S1, S2, S3, S4) de las tablas anteriores, las cuales están representadas por dos *arrays*, además en cada iteración se rotará el *array* para obtener el dato en la siguiente iteración. El código de la mitad superior del bucle sirve para dibujar sobre el Panel Frontal una gráfica polar (*Controls > Modern > Graph > Controls > Polar Plot Indicator*) donde se simulará el giro.

Dentro del bucle se seleccionará mediante un CASE uno de los dos *arrays*: medio paso o paso completo. Por una parte se rota el *array* para la próxima iteración y por otra parte, en caso de ser medio paso, se obtiene el primer elemento del *array*, si es paso completo se convierte el *array* de booleanos a número; finalmente se escribe el resultado en la tarjeta DAQ.

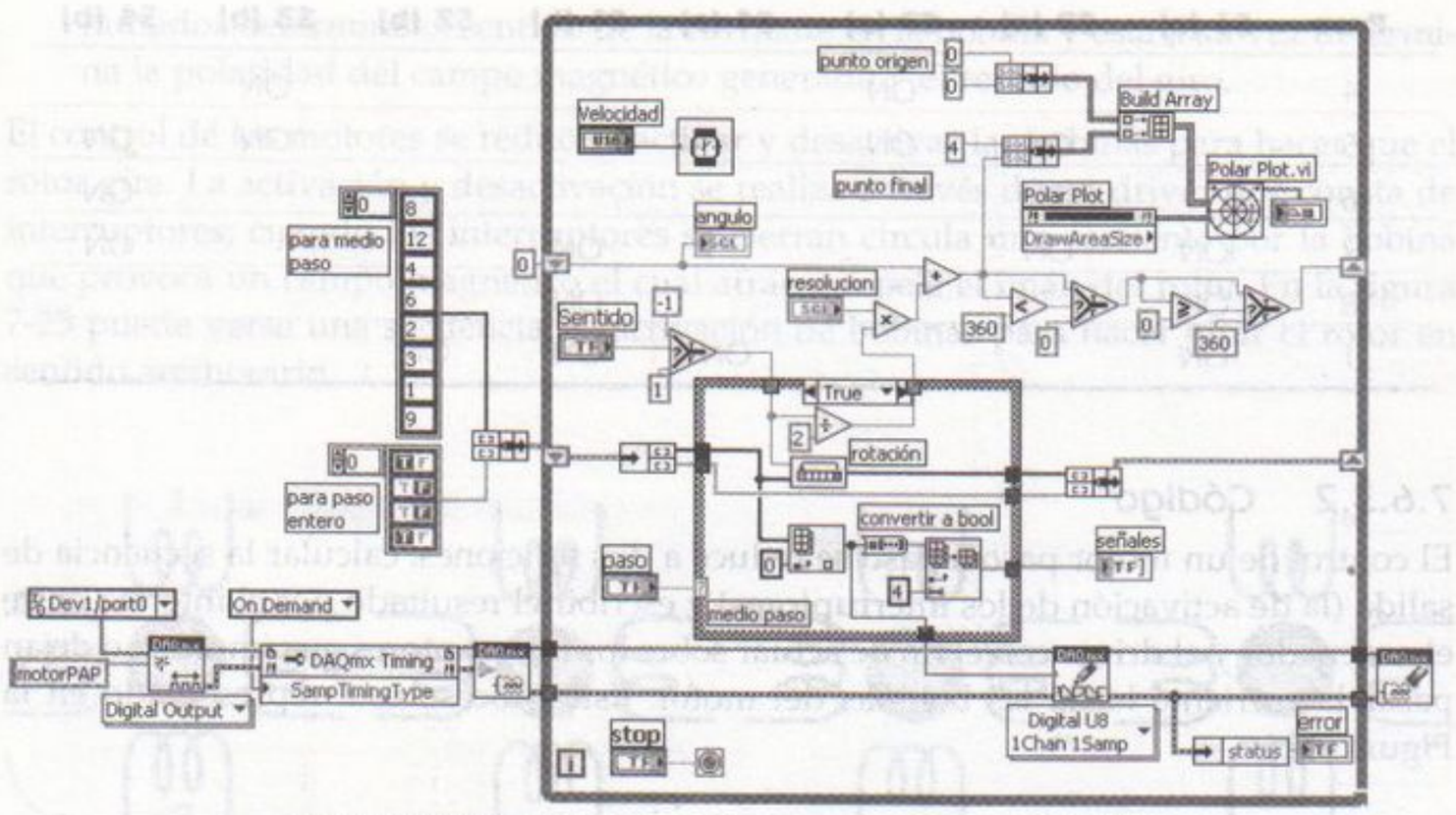


Figura 7-27. Control de un motor paso a paso con LabVIEW

7.6.3.3 Resultados

Para probar el resultado se han conectado las cuatro primeras líneas digitales del puerto 0 a un driver para motor paso a paso de tipo L293D. El programa consigue un giro continuo del motor. Por otra parte, en la figura 7-28 se pueden ver las señales que se envían y el resultado de la simulación.

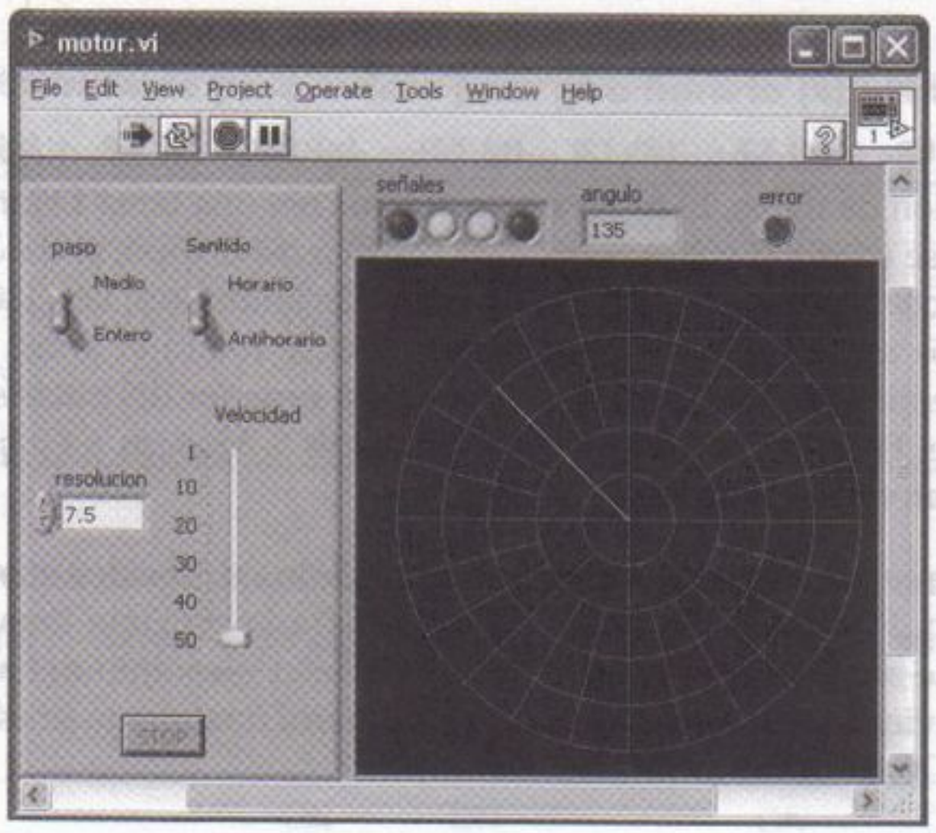


Figura 7-28. Panel Frontal del resultado

7.7 Ejercicios

1. Buscar y estudiar ejemplos en la ayuda de LabVIEW y en la web de National Instruments sobre adquisición de datos.

2. Comprobar los límites de frecuencia de los distintos tipos de generación/adquisición en la tarjeta de adquisición.

3. Usando la plantilla mostrada a continuación encontrar un método para adquirir datos y procesarlos en paralelo usando *pipeline* (segmentación).

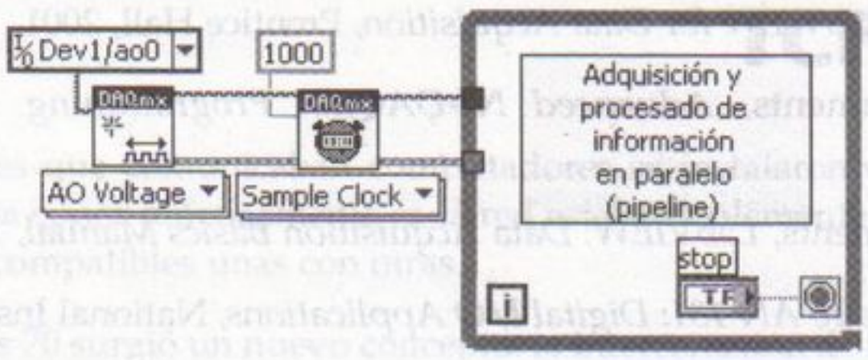


Figura 7-29. Plantilla en la que programar mediante pipeline

4. Calcular teóricamente los coeficientes del escalado del ejemplo II sabiendo que una Pt100 tiene 80.31 ? a -50 ºC, 100 ? a 0 ºC y 119.40 ? a 50 ºC y la fuente de corriente es de 1 mA. Analizar el circuito del ejemplo anterior para obtener la expresión $V_{salida} = 55 \cdot 0{,}001 \cdot R_{Pt100} - 4$, aproximar $R_{23}=R_{25}$.

5. Cambiar el interfaz de comunicación entre el ordenador y el motor paso a paso, usar en esta ocasión el puerto paralelo.

6. Realizar un sistema de alarma que muestre un aviso cuando en una entrada analógica se superen los 5 V. Para ello emplear un trigger. Con el fin de evitar que haya activaciones falsas cuando la tensión esté próxima al límite, se añadirá un ciclo de histéresis de ±0,1 voltios. Cuando se active la alarma mostrar un mensaje por pantalla.

7. Realizar un medidor del caudal de agua que pasa por una tubería empleando uno de los contadores de la tarjeta. La tubería tiene un aspa que gira de acuerdo con el flujo de agua. Cada varilla del aspa está imantada y al girar inducirán una tensión

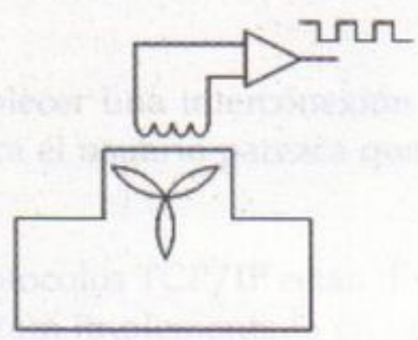


Figura 7-30. Esquema de un medidor de flujo

en una bobina colocada cerca de ellas. Esta tensión, una vez acondicionada, se introducirá directamente en la tarjeta de adquisición de datos. La figura 7-30 muestra el esquema propuesto. En este ejemplo, por simplificación, se supondrá que el número de pulsos de la señal tendrá una dependencia directa con el flujo de agua: .

7.8 Bibliografía

Bonnie C. Baker, *AN 687: Precision Temperature-Sensing With RTD Circuits*, Microchip, 2003.

Bruce Mihura, *LabVIEW for Data Acquisition*, Prentice Hall, 2001.

National Instruments, *Advanced NI-DAQmx Programming Techniques with LabVIEW*.

National Instruments, *LabVIEW: Data Acquisition Basics Manual*, 1998.

Niranjan Maharajh, *AN 131: Digital I/O Applications*, National Instruments, 1998.

Rajesh S. Vaidya, *AN 129: Tips and Techniques in DAQ Triggering*, National Instruments, 1998.

Richard House, *AN 092: Data Acquisition Specifications ¡ a Glossary*, National Instruments, 1997.

STMicroelectronics, *L293D Push-Pull Four Channel Driver With Diodes*, 2003.

Capítulo 8

Protocolos de comunicación: TCP y UDP

Las primeras redes que comunicaban computadores se instalaron en EE.UU. en los años 60. En aquella época prácticamente cada red estaba implementada con una tecnología distinta e incompatibles unas con otras.

A principios de los 70 surgió un nuevo concepto: la interconexión de redes. Ya no basta con comunicar computadores sino que había que comunicar redes de computadoras distintas.

TCP/IP fue desarrollado en 1972 por un grupo de investigadores subvencionados por el departamento de defensa de los EE.UU. ejecutándose en ARPANET. TCP/IP es una familia de protocolos de comunicación agrupados por niveles. Permiten la comunicación entre dispositivos y entre redes.

En los siguientes años TCP/IP fue mejorándose y extendiéndose, se incluyó en los antiguos sistemas operativos UNIX de Berkeley. A la vez ARPANET se extendió por todo Estados Unidos uniendo nuevas redes, el resultado fue lo que hoy conocemos como Internet.

Las RFC (*Request For Comments*) son documentos que, en su mayoría, describen protocolos o actualizaciones de éstos. Las RFC pueden encontrarse en el IETF (*Internet Engineering Task Force*). Las RFC se clasifican en: estándar, estándar provisional, propuesto como estándar, experimental, informativo e histórico. Varios RFC pueden formar parte de un mismo estándar.

8.1 TCP/IP

El objetivo de TCP/IP es establecer una interconexión entre redes para proporcionar servicios de tal manera que para el usuario parezca que sólo hay una única red homogénea.

Como se ha dicho antes, los protocolos TCP/IP están divididos en capas formando una pila de protocolos. Esta pila estará implementada en cada uno de los nodos de la red. Dentro de un nodo los mensajes se pasarán de un nivel al siguiente, desde arriba hacia

abajo en la transmisión y desde abajo hacia arriba en la recepción, añadiendo y quitando campos respectivamente, de forma que el mensaje que llega a un nivel sea el mismo en el transmisor y en el receptor. Así se puede decir que hay una comunicación directa entre los niveles equivalentes del emisor y del receptor, pues el resto de niveles serían "transparentes".

Los campos que se añaden y se quitan son básicamente cabeceras para que la red proporcione servicios. Por ejemplo, se puede añadir un código para comprobar o corregir errores, un número que indique la secuencia de un mensaje que forma parte de una transmisión mayor o la dirección del destino. En la figura 8-1 se pueden ver las pilas de protocolos de un emisor y un receptor y el paso de mensajes entre ellas dentro del propio host y a través de la red. Además, junto a cada protocolo pueden verse los datos y cómo se van añadiendo y eliminando cabeceras.

Tabla 1 - Protocolos de diferentes capas

Nivel de la pila	Ejemplo de protocolos
Aplicación	HTTP, FTP, SMTP...
Transporte	TCP, UDP...
Interred	IP, ARP, X.25...
Enlace	Ethernet, ATM, Frame Rely, Wi-Fi
Físico	Medio físico y técnicas de codificación

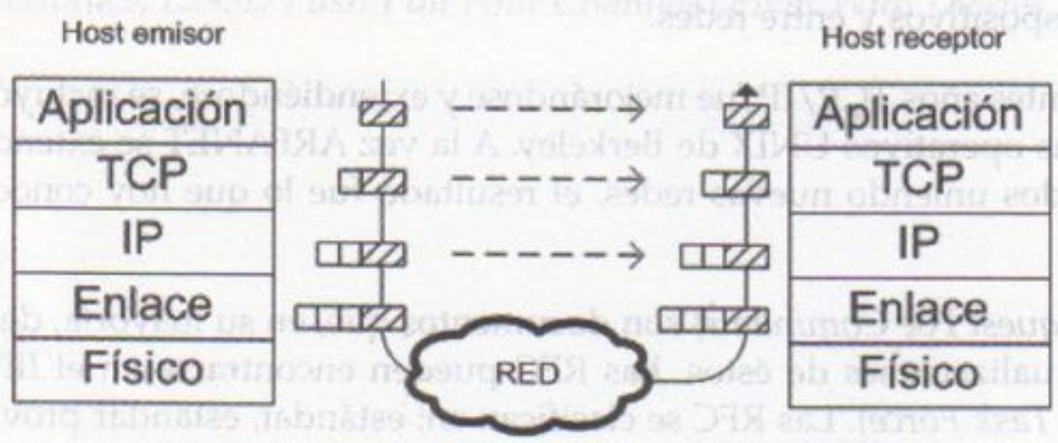


Figura 8-1. Encapsulado de datos

Para interconectar redes distintas haría falta un tercer elemento además del emisor y el receptor que haga de traductor entre las dos tecnologías distintas de un mismo nivel.

8.1.1 Nivel de red

El protocolo IP (*Internet Protocol*) es el protocolo de interconexión de redes más usado. Pertenece a la capa de red y está definido en la RFC 791. La unidad básica de datos en IP se llama datagrama. La misión de IP en un nodo es conformar los paquetes que serán pasados al protocolo inferior y desencapsular los paquetes entrantes para pasárselos al protocolo superior. Si los datos que recibe son de mayor tamaño que el aceptado por la red, este protocolo debe fragmentar la información en el emisor y reconstruirla en el

receptor. Otra característica fundamental es el encaminamiento, dos de los campos de la cabecera IP son la dirección del host origen y destino. Esta información es usada por la red para hacer llegar los paquetes de un nodo a otro aunque no tengan conexión directa.

El protocolo IP proporciona un sistema de distribución poco fiable incluso en una base sólida. Los datagramas se pueden retrasar, perderse, crear duplicados, ser enviados en una secuencia incorrecta o fragmentados intencionadamente para permitir que un nodo con un buffer más pequeño que el tamaño del datagrama pueda coger todo el datagrama. En algunas situaciones de error, los datagramas son descartados sin mostrar ningún mensaje, mientras que en otras situaciones los mensajes de error son recibidos por la máquina origen (mediante el protocolo ICMP).

Entre los campos de la cabecera IP destacan las direcciones, tanto origen como destino. Cada uno de estos campos está compuesto por cuatro bytes, aunque se suelen representar como cuatro números decimales separados por puntos. Las direcciones IP se utilizarán para identificar el origen y destino de la información en la red. Las direcciones *multicast* o multidifusión son un tipo especial, ya que no hacen referencia a una máquina en concreto sino a un conjunto de ellas. En IPv4 se reservan las direcciones que empiezan por 1110 (de la 224.0.0.0 a la 239.255.255.255) para crear grupos *multicast*. Otra dirección especial es la de *loopback* o *localhost*, cuyo valor es 127.0.0.1.

8.1.2 Nivel de transporte

El protocolo TCP (*Transmission Control Protocol*) se describe en la RFC 793 y otras posteriores. Es un protocolo de la capa de transporte orientado a conexión. Se diseñó para proporcionar una corriente de bytes confiable a través de una interred no confiable, es decir, con TCP el flujo de datos entre el origen y el destino parecen continuos: se proporciona un circuito virtual para los datos que es llamado conexión. Los conjuntos de datos en TCP se llaman segmentos.

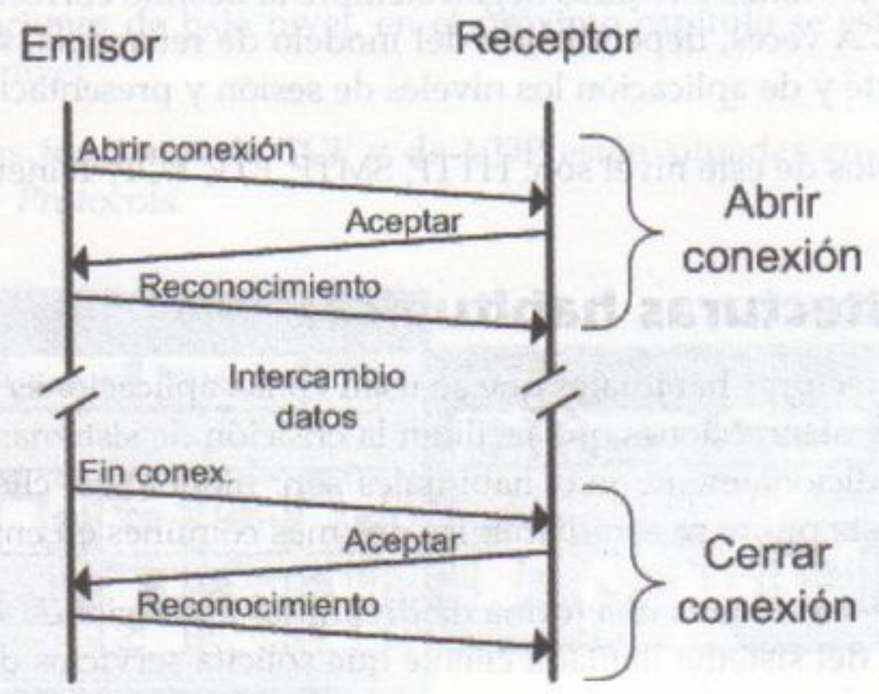


Figura 8-2. Pasos en una conexión

Como se aprecia en la figura 8-2, una conexión básica implica tres pasos: abrir una conexión, intercambio de datos y un cierre de la conexión.

Al igual que se hacía en IP con las direcciones, TCP tiene unos campos equivalentes en la cabecera dedicados a los puertos. Cada host puede tener varias conexiones TCP abiertas simultáneamente, cada una en un puerto distinto. Otro de los campos de la cabecera es una serie de flags que sirven para indicar petición de conexión, de finalización, reconocimientos, etc.

Otro protocolo es UDP (*User Datagram Protocol*), definido en la RFC 768, que está en el mismo nivel que TCP. No añade fiabilidad, control de flujo o recuperación de errores a IP cuando funciona sobre él, simplemente trabaja como un multiplexor/demultiplexor para enviar y recibir datagramas, usando los puertos para dirigir los datagramas.

A nivel práctico podemos considerar a UDP como una simplificación hasta el extremo de TCP ya que no proporciona ninguno de sus servicios, a cambio es un protocolo más "ligero" y rápido. UDP se suele usar en aplicaciones que requieran poco intercambio de información y en redes que no tengan mucho tráfico, mientras que TCP se usa cuando hay que transmitir mucha información o cuando el volumen de tráfico en la red es medio o alto.

En IP hay direcciones que sirven para encaminar la información de un host a otro a través de la red; y en TCP y UDP hay puertos que sirven para identificar qué aplicación está asociada a esa información. Por tanto para establecer una conexión entre dos equipos se necesitan estos dos datos básicos: puertos y dirección IP. A esta pareja de datos se le llama *socket*.

8.1.3 Nivel de aplicación

Por encima del nivel de transporte va el de aplicación. En este nivel, si se usa sobre TCP, se considera que los datos enviados llegan siempre al destino correcto, sin fallos y en el orden adecuado. A veces, dependiendo del modelo de referencia, se insertan entre la capa de transporte y de aplicación los niveles de sesión y presentación.

Algunos protocolos de este nivel son: HTTP, SMTP, FTP, POP, Telnet...

8.2 Arquitecturas habituales

Hay varias arquitecturas habituales que se usan en las aplicaciones de red, se trata de reglas, patrones y abstracciones que facilitan la creación de sistemas informáticos. Las arquitecturas tradicionalmente más habituales son: monolítica, cliente-servidor y de tres niveles. En este punto se estudiarán las dos más comunes en entornos de red.

El modelo cliente-servidor es una forma de dividir las aplicaciones. En esta arquitectura hay una parte del sistema llamada cliente que solicita servicios de la otra parte llamada servidor.

El servidor por lo general está escuchando (*listener*) y cuando un cliente se conecta, el servidor atiende su petición. En entornos de red el cliente estará escuchando en un puerto de un host. Tanto la dirección del host como el puerto deben ser conocidos por el cliente; éste, en principio, usará cualquier puerto que esté disponible en su host para establecer la conexión. Normalmente el sistema operativo es el encargado de gestionar los puertos.

Los clientes pueden ser livianos (con poca carga computacional) o pesados (el cliente realiza la mayoría de las tareas). Por su parte algunos servidores podrán atender a varios clientes simultáneamente. Tanto cliente como servidor pueden residir en la misma máquina o en máquinas diferentes.

Un ejemplo sería una aplicación web en la que el usuario utilizaría un navegador web (cliente) para abrir una conexión TCP/IP con un servidor. El usuario pude escribir en la barra de direcciones algo como `http://158.42.148.222:80`, como se puede ver se indica la dirección IP destino 158.42.148.222 y un puerto, el 80. En el host destino y en ese puerto debe haber un programa escuchando: el servidor web. Este servidor que estaba escuchando en el puerto 80, acepta la conexión y, una vez que el cliente confirme que realmente hay un servidor al otro lado, enviará una petición HTTP en la que pide un fichero. El servidor leería esa petición y buscaría el fichero para enviarlo al cliente. Una vez que el cliente recibe el fichero que había pedido se puede cerrar la conexión.

La arquitectura entre pares o iguales (*Peer to Peer* o P2P) consta de varios nodos que en un momento determinado pueden tener funciones de cliente o de servidor. Se trata de una red descentralizada en la que los nodos comparten sus recursos (capacidad de cómputo, almacenamiento, ancho de banda) para llevar a cabo una tarea.

8.3 TCP/IP en LabVIEW

LabVIEW tiene implementadas funciones para crear aplicaciones que usen TCP o UDP. Se consideran funciones de bajo nivel, en el próximo capítulo se estudiarán algunas funciones de más nivel.

Las paletas con las funciones de TCP y de UDP están situadas en *Funcions > Data Communication > Protocols*.

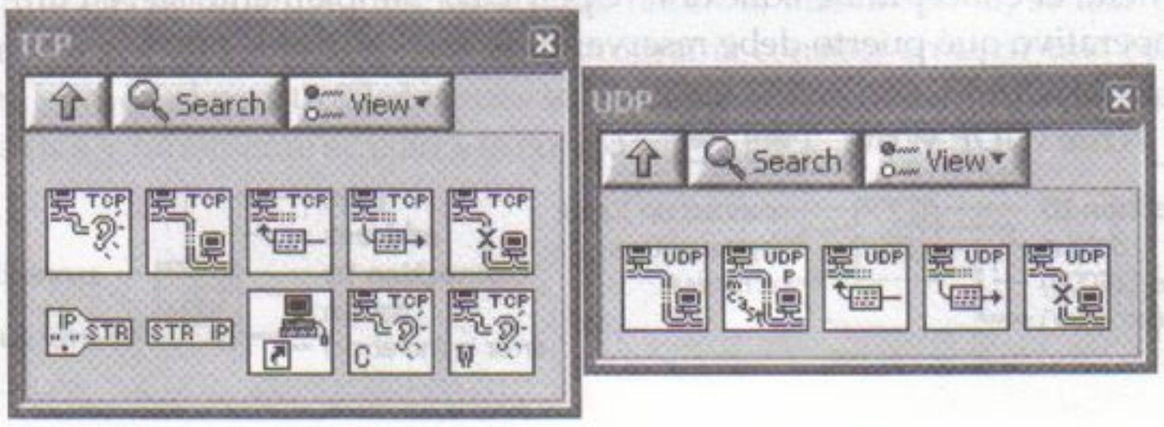
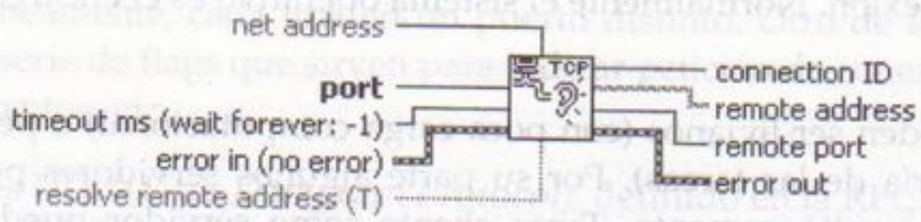


Figura 8-3. Menús de funciones TCP y UDP

En TCP y UDP se sigue el mismo modelo de programación que con ficheros: el concepto de abrir, leer/escribir y cerrar.

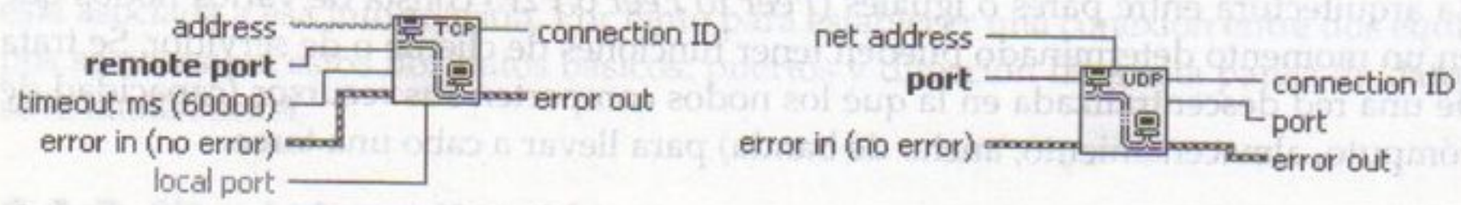
■ TCP Listen



Crea un *listener* y espera en el puerto seleccionado a que llegue una conexión. Nótese que si no se especifica un tiempo máximo para esperar, el flujo de ejecución del hilo del programa se detiene hasta que llegue una petición de conexión. Por lo general sólo hace falta especificar el puerto en el que tiene que escuchar, en caso de tener más de un adaptador de red también se puede seleccionar uno de ellos con *net address*. Entre los parámetros que devuelve está la dirección IP, el puerto del host que ha abierto la conexión y un manejador o *handler* (*connection ID*) que sirve para manejar esa conexión desde otros VIs.

Este VI suele usarse para crear servidores que acepten conexiones. Internamente usa *TCP Create Listener* y *TCP Wait On Listener*.

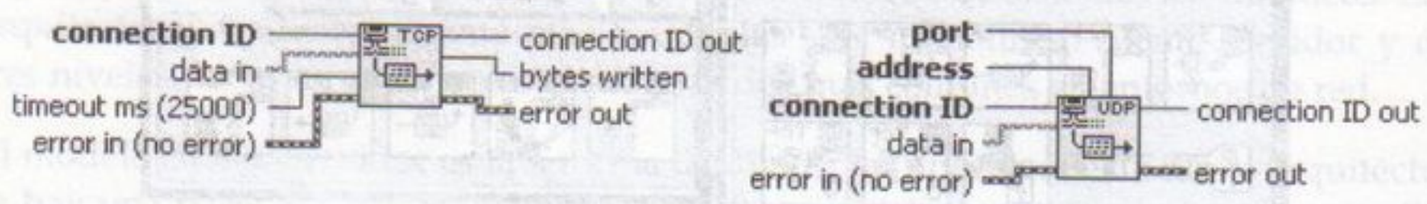
■ TCP Open Connection\UCP Open



Abre una conexión TCP con otro host. Como parámetros de entrada debe tener el puerto destino. Los parámetros de entrada opcionales son la dirección del destino (si no se especifica usa la del ordenador local) que puede ser la dirección IP o el nombre del host (en este caso el sistema operativo se encargará de traducir el nombre a dirección IP), el puerto local (si no se indica ninguno lo asigna el sistema operativo) y el tiempo de espera para la confirmación del cliente. Devuelve un manejador (*connection ID*) para ser usado en otros VIs.

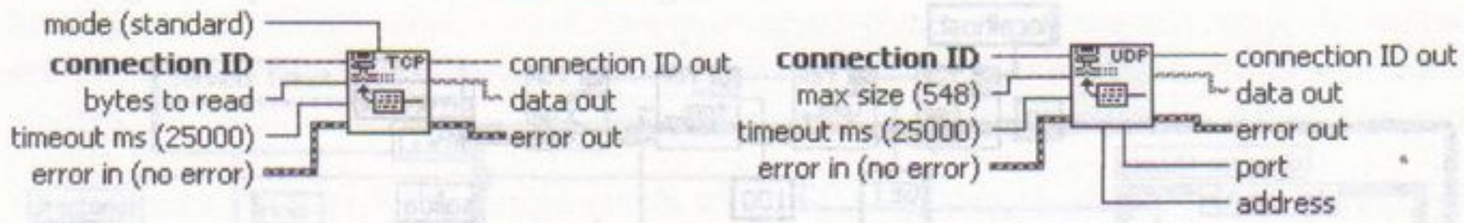
UDP no tiene el concepto de conexión, *Open UDP* simplemente sirve para indicar al sistema operativo qué puerto debe reservar para las siguientes operaciones.

■ TCP Write\UDP Write



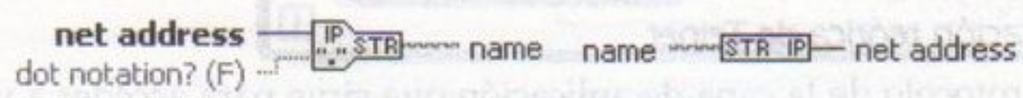
Estos VIs usan un manejador para enviar datos. Como se puede ver, tanto en la lectura como en la escritura, los datos son de tipo *string*. Si se quiere enviar otro tipo de datos habría que convertirlos a *string* como se indicaba en el capítulo 3. En TCP el destino está especificado en el manejador mientras que en UDP hay que indicar tanto el puerto como la dirección sobre la que escribir, para esto último se puede usar *String To IP*.

- **TCP Read\UDP Read**



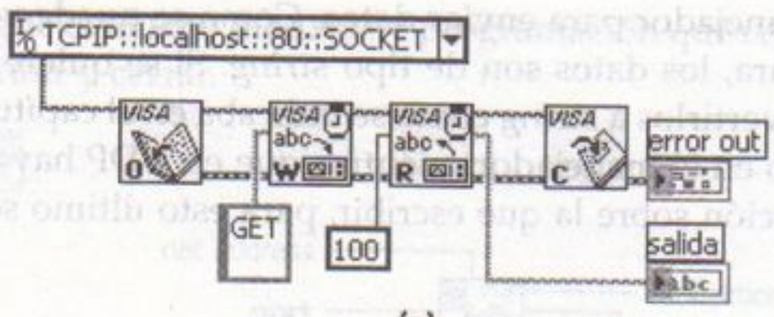
En TCP este VI trabaja sobre una conexión ya establecida que se indica mediante el manejador (*connection ID*). Además, para indicar cuantos datos se deben leer se usan los parámetros *bytes to read* y *mode*. Cuando el tamaño de los datos que se deben leer es variable hay dos opciones: usar un campo extra en los datos a modo de cabecera de tamaño fijo que indique la longitud de los datos o usar un carácter especial para indicar el final, para esto último se puede usar el valor *CRLF* en el terminal *mode*.

- **IP To *String* y *String* To IP**

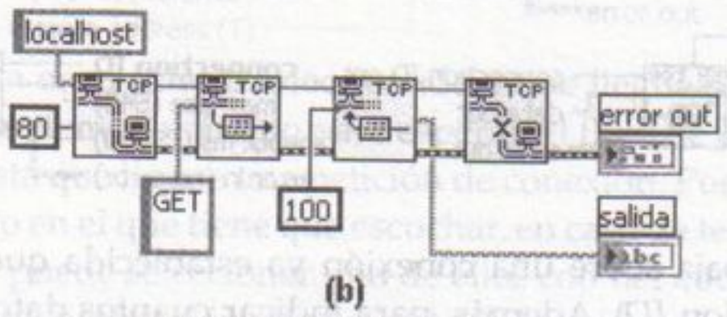


IP To String convierte una dirección IP representada mediante un número entero a un *string* que puede ser el nombre del ordenador o la dirección IP. *String To IP* hace lo contrario, a partir del nombre de un ordenador o un *string* que indica la dirección IP (por ejemplo un *string* cuyo valor sea 192.168.0.1) obtiene un número que representa su dirección IP. Si la dirección IP es A.B.C.D, el número valdrá . $A \cdot 2^{24} + B \cdot 2^{16} + C \cdot 2^{8} + D$. Si se selecciona la opción de *Multiple Output* en el menú contextual puede obtener más de una salida en forma de *array*, en este caso cada posición del *array* sería la dirección IP para cada interfaz de red.

Para finalizar también hay que comentar que, como se dijo en los temas anteriores, las librerías VISA también pueden comunicarse a través de TCP/IP. En la figura 8-4 se puede ver un ejemplo de dos programas que hacen lo mismo: en (a) se usa VISA y en (b) las funciones TCP estudiadas antes. El programa consiste en abrir una conexión al ordenador local en el puerto 80, después se escribe GET\n , a continuación se leen 100 bytes de datos y finalmente se cierra la conexión.



(a)



(b)

Figura 84. TCP/IP con VISA

8.4 Ejemplos

8.4.1 Ejemplo I: Servidor de Telnet

8.4.1.1 Explicación teórica de Telnet

Telnet es un protocolo de la capa de aplicación que sirve para acceder a una máquina de forma remota y poder ejecutar comandos en una consola. Por extensión también se llama telnet al programa cliente que permite acceder a un servidor.

Para funcionar es necesario tener un servidor de telnet funcionando en la máquina que se quiere controlar, normalmente en el puerto 23 y un cliente en la máquina desde la que se controlará.

Todos los sistemas operativos suelen tener un cliente de telnet. Por ejemplo, en Windows se puede acceder a él desde una ventana de MS-DOS escribiendo «telnet». Para abrir una conexión sólo hay que escribir «open 127.0.0.1 80» donde 127.0.0.1 es la dirección IP y 80 es el puerto remoto al que conectarse. Si no se especifica ningún puerto usa por defecto el de telnet: el 23.

En este ejemplo se creará un servidor de telnet al que se podrán conectar clientes desde otros ordenadores para ejecutar comandos de MS-DOS remotamente.

8.4.1.2 Código

Este ejemplo consta de dos partes principales: comunicación TCP/IP y ejecución de comandos. Una descripción más detallada de las tareas del servidor sería:

- Esperar a recibir una petición de conexión, cuando esto ocurra se aceptará la conexión y se mostrará un mensaje de presentación.

- Leer el comando que envía el cliente. El cliente irá enviando carácter a carácter según el usuario los teclea. Se empleará el carácter fin de línea para indicar el final del comando, lo cual dará paso a la ejecución del mismo. También se debe comprobar si el comando es el de salida (en este caso se ha elegido «exit»). Si no es «exit» se pasará lo recibido al sistema para que lo ejecute por medio del VI *System Exec* del menú *Functions > Connectivity > Libraries & Executables*.
- El resultado de este comando se envía al cliente.

El código completo se muestra en la figura 8-5.

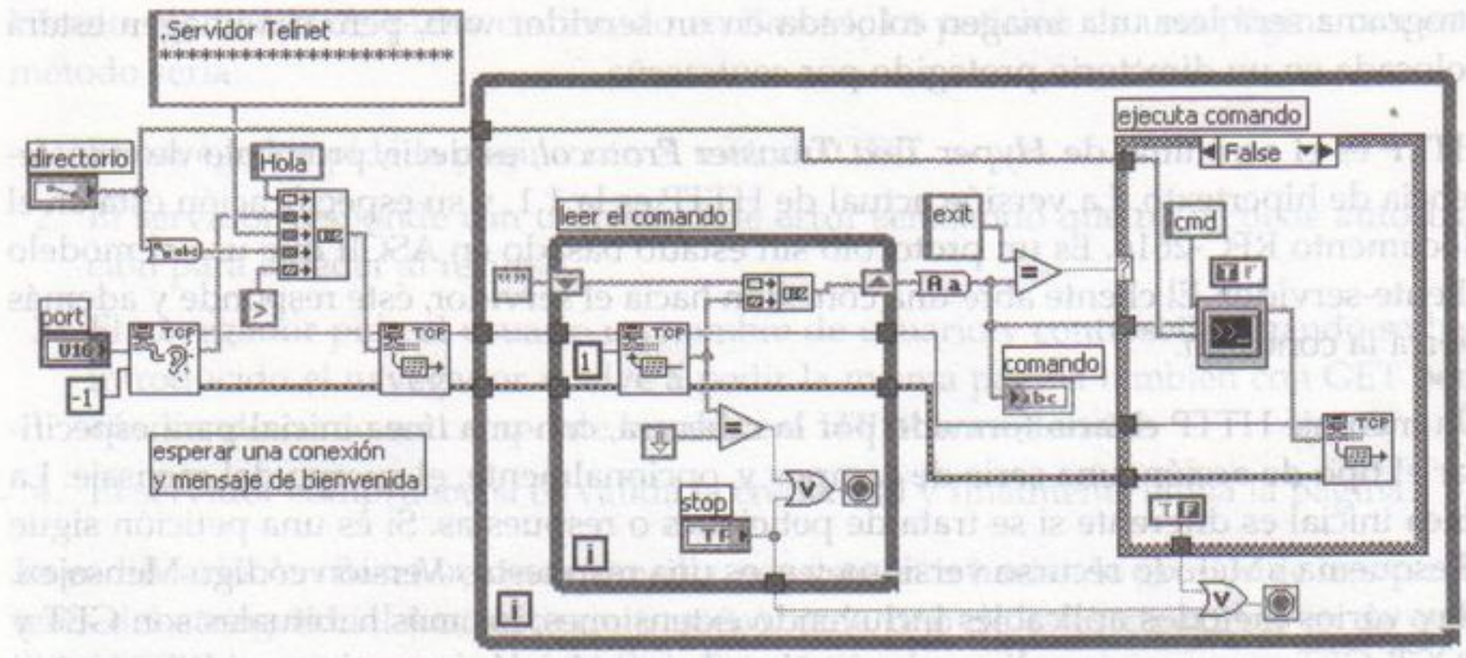


Figura 8-5. Servidor Telnet

8.4.1.3 Resultados

Para poner el servidor en funcionamiento primero hay que configurar su puerto y directorio de trabajo. Después hay que ejecutar el programa y abrir una conexión telnet con el ordenador donde se ejecuta el servidor, para esto basta escribir desde la consola «`telnet 127.0.0.1 23`».

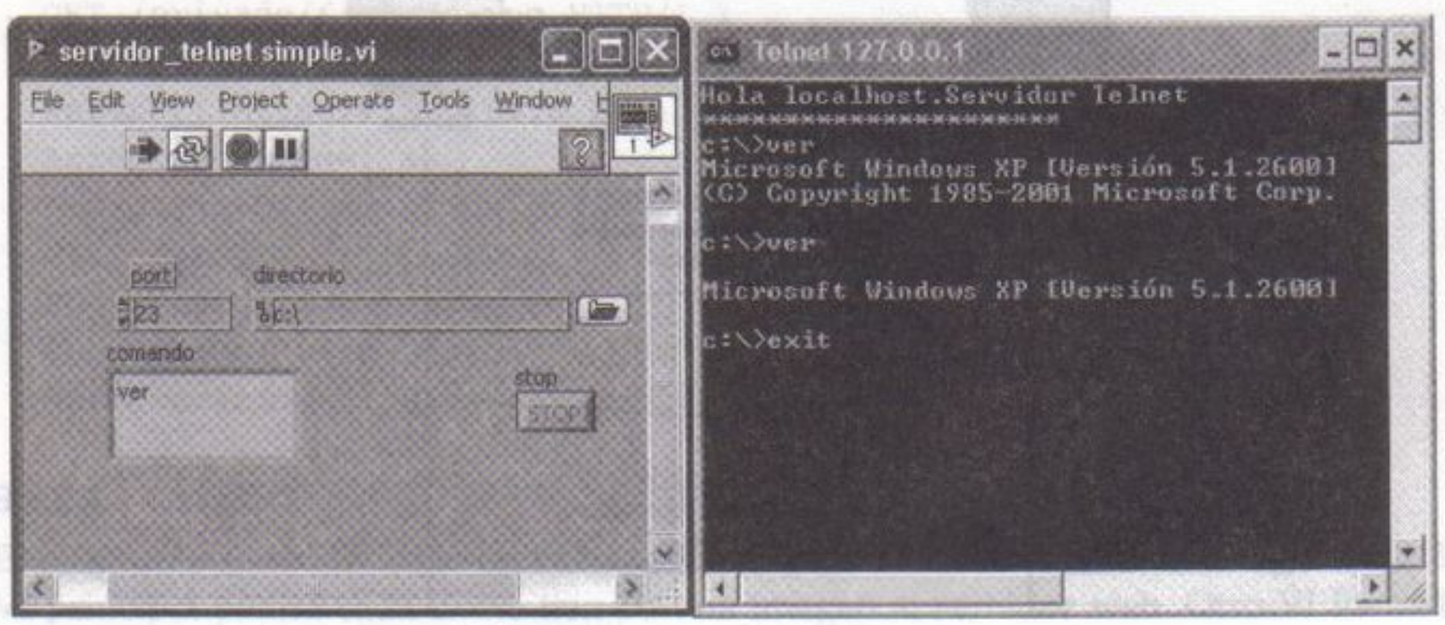


Figura 8-6. Servidor y cliente telnet

Hay que tener en cuenta que este programa presenta algunas limitaciones. Una de ellas es que sólo acepta una conexión cada vez. Una mejora sería hacer un servidor que pudiera aceptar varias conexiones a la vez. Otra limitación es que *System Exec* siempre ejecuta los comandos en el mismo directorio.

8.4.2 Ejemplo II: Protocolo estándar

8.4.2.1 Explicación teórica

En este ejemplo se implementará una pequeña parte de un cliente HTTP. La función del programa será leer una imagen colocada en un servidor web, pero esa imagen estará colocada en un directorio protegido por contraseña.

HTTP es el acrónimo de *Hyper Text Transfer Protocol*, es decir, protocolo de transferencia de hipertexto. La versión actual de HTTP es la 1.1, y su especificación está en el documento RFC-2616. Es un protocolo sin estado basado en ASCII que usa el modelo cliente-servidor. El cliente abre una conexión hacia el servidor, éste responde y además cierra la conexión.

Un mensaje HTTP estaría formado por la cabecera, con una línea inicial para especificar el tipo de acción, una serie de campos y, opcionalmente, el cuerpo del mensaje. La línea inicial es diferente si se trata de peticiones o respuestas. Si es una petición sigue el esquema «Método recurso versión» y si es una respuesta «Versión código Mensaje». Hay varios métodos aplicables incluyendo extensiones, los más habituales son GET y POST. GET se usa para pedir cualquier tipo de información, cuando se pide una página web o cualquier otro fichero se usa este método. En el ejemplo de la figura 8-7 se puede ver como el cliente realiza una petición al servidor compuesta por la línea «GET /ruta/index.htm HTTP/1.0» (método, recurso y versión). La línea inicial de una respuesta tiene también tres campos separados por un espacio: «versión código mensaje».

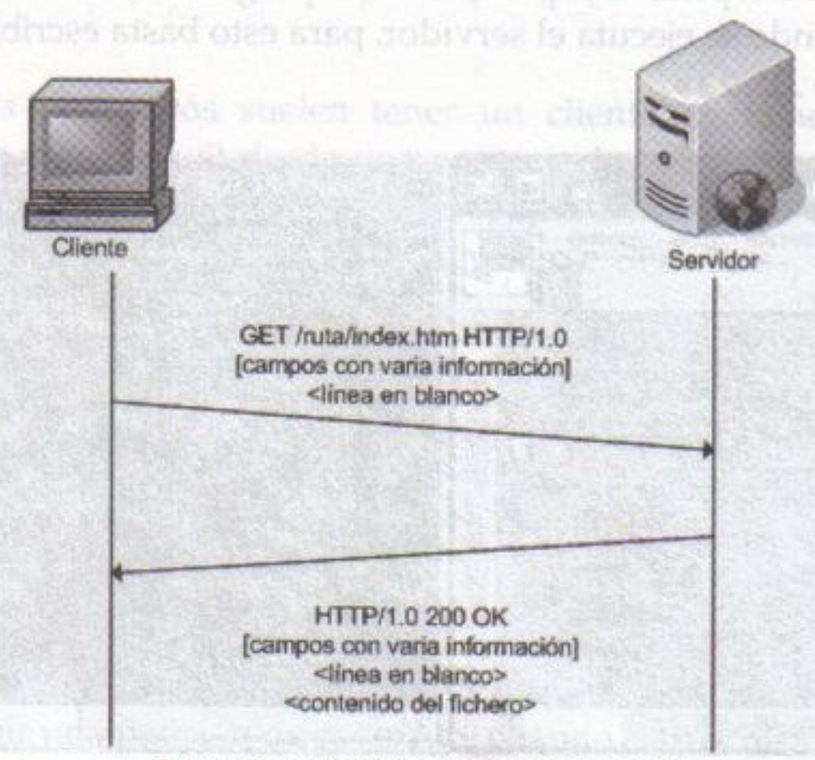


Figura 8-7. Petición y respuesta HTTP

Entre los campos de la cabecera HTTP, los típicos suelen ser:

- Por parte del cliente la identificación del navegador, lenguaje preferido, tipos de ficheros aceptados, etc.
- Por parte del servidor se tiene la fecha, versión del servidor, etc.

HTTP admite además otras muchas opciones, entre otras la denominada *basic authentication scheme*, que es un método por el que el usuario proporciona al servidor unas credenciales (nombre de usuario y contraseña) para acceder a un recurso. Este método presupone que la conexión es segura porque el envío de las credenciales no se hace cifrado, sino en texto plano codificado en Base64. La petición de una página con este método sería:

1. El navegador pide la página con el método GET.
2. El servidor responde con un código de error señalando que no se tiene autorización para acceder al recurso.
3. El navegador pide al usuario un nombre de usuario y contraseña, cuando se han introducido el navegador vuelve a pedir la misma página también con GET pero añadiendo como un campo de la cabecera HTTP la credencial.
4. El servidor comprueba si es válida la credencial y finalmente envía la página.

La codificación en Base64 es muy sencilla, consiste en ir haciendo grupos de tres caracteres del *string* inicial (usuario+contraseña), cada uno de estos caracteres es de ocho bits (ASCII), por lo tanto habrá 24 bits. Estos bits se agrupan ahora en grupos de seis bits que se corresponden con un subconjunto de 65 caracteres de la tabla ASCII. Estos cuatro caracteres son codificados nuevamente con ocho bits por lo que se pasa de tres caracteres a cuatro y el tamaño total se incrementa en un 33%. La implementación puede ser muy sencilla usando tablas.

Resumiendo, una petición de un fichero que requiera contraseña por parte de un cliente web a un servidor podría consistir en:

```
GET /privado/fichero.jpg HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Referer: http://192.168.0.3
Accept-Language: es
Accept-Encoding: gzip, deflate
User-Agent: MiprogramaenLV/1.0
Host: 127.0.0.1
Connection: Keep-Alive
Authorization: Basic cmFmYTptb25rZXk=
```

8.4.2.2 Código

El programa se ha dividido en varios subVIs de la forma indicada en la figura 8-8. El VI principal pedirá cada cierto tiempo una imagen al servidor web. Otro VI llamado «base64.vi» será el encargado de construir el campo `Authorization` de la petición

del fichero a partir de un nombre de usuario y contraseña, este VI usará otros dos subVIs llamados «codifica.vi» y «tablaB64.vi». El VI «lee de web.vi» realizará la petición al servidor web y recibirá su respuesta.

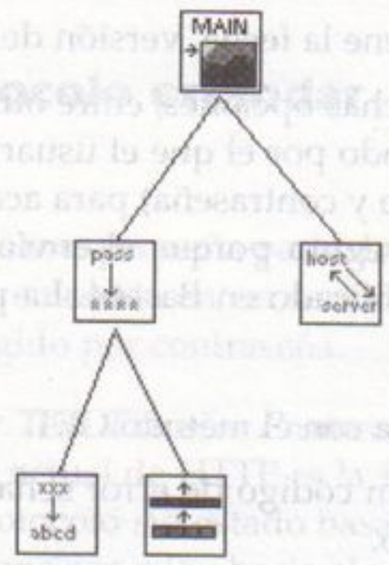


Figura 8-8. Estructura del programa

El primero de los subVIs de la figura 8-8 (fila central a la izquierda) es «base64.vi» (figura 8-9). No tiene mucha relación con los contenidos de este capítulo, su tarea es dividir en *string* «usuario:contraseña» en grupos de tres caracteres, estos tres caracteres de 8 bits cada uno (24 bits) son divididos por «codifica.vi» (fila inferior, izquierda de la figura 8-8) en cuatro grupos de 6 bits cada uno y convertidos a números. Los cuatro valores obtenidos se pasan al VI «tablaB64.vi» (fila inferior, derecha de la figura 8-8) donde se codifican en ASCII direccionando alguna de las posiciones de un gran *array* que contiene todos los valores posibles. Los cuatro caracteres resultantes se concatenan y se repite el proceso para el siguiente grupo de tres caracteres.

Una vez codificado el nombre de usuario y la contraseña se realizará la petición HTTP al servidor web con «lee de web.vi».

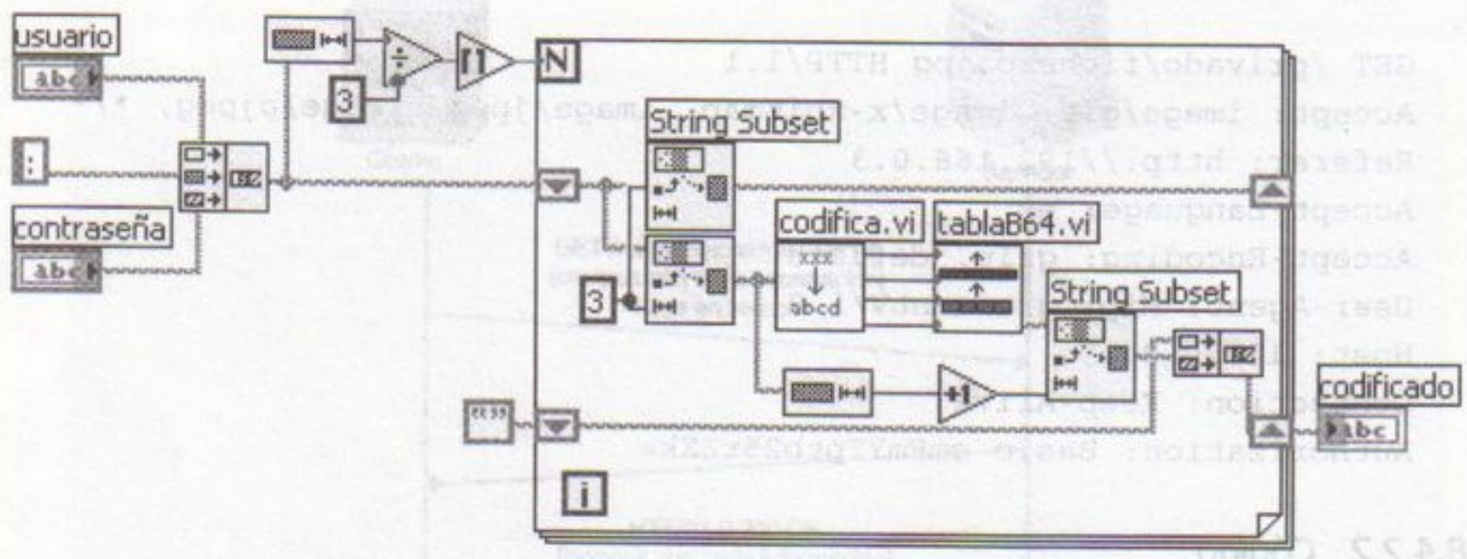


Figura 8-9. base64.vi

El segundo subVI de la fila central de figura 8-8 («lee de web.vi»), como se ha dicho antes, debe construir el comando que será enviado al servidor. Su contenido se muestra en la figura 8-10. Como se aprecia, la forma de construir la petición recuerda a la construcción de comandos SCPI en el capítulo sobre GPIB, usa *Format Into String* para concatenar todas las cadenas de texto. Si todo va bien (se establece la conexión, el usuario y contraseña son correctos y existe el fichero) la respuesta contendrá un campo llamado «Content-Length» que indica la longitud de los datos (la imagen que se pide), usándolo se podrá separar la cabecera HTTP de los datos con *String Subset*.

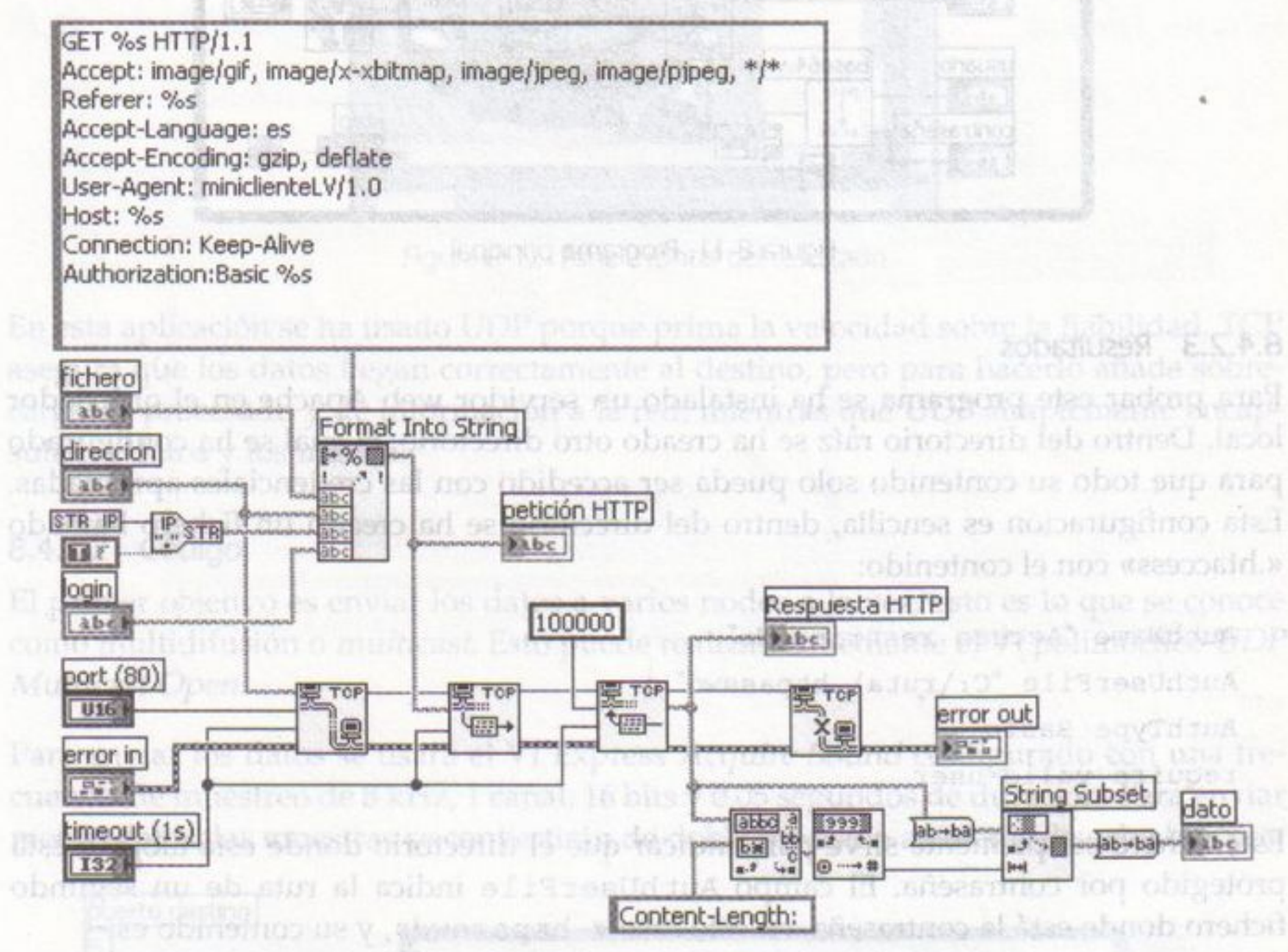


Figura 8-10. Petición HTTP

Como se ve en la figura 8-11, el programa principal usa los dos anteriores para recibir la imagen y, cuando la tiene, la guarda en un fichero además de mostrarla en el Panel Frontal.

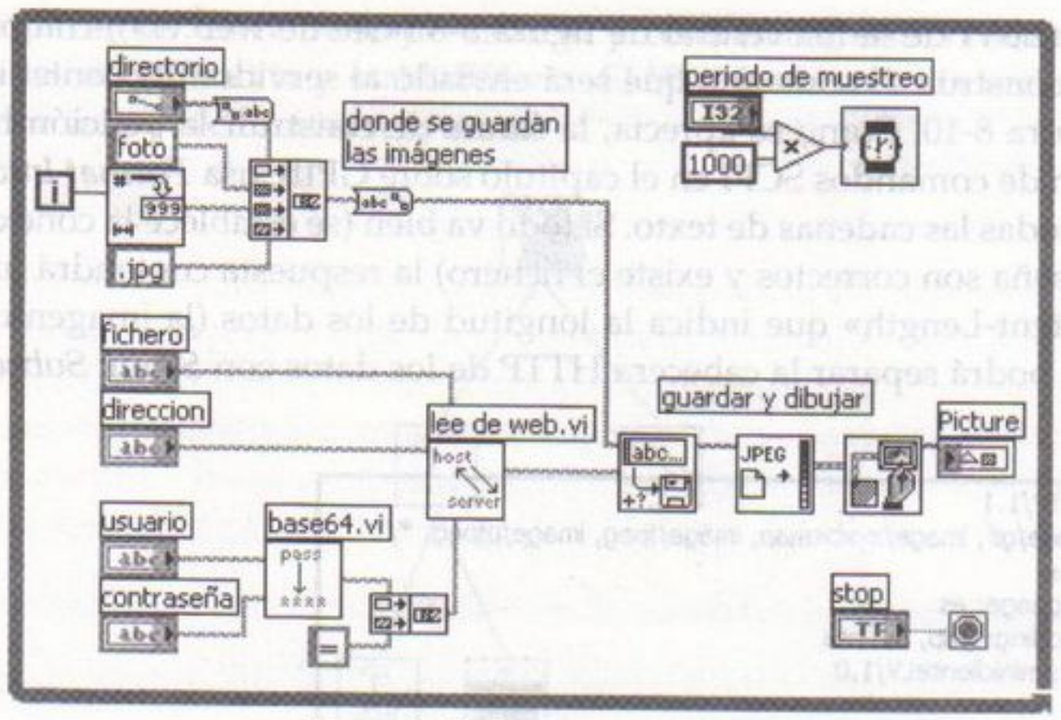


Figura 8-11. Programa principal

8.4.2.3 Resultados

Para probar este programa se ha instalado un servidor web Apache en el ordenador local. Dentro del directorio raíz se ha creado otro directorio, el cual se ha configurado para que todo su contenido solo pueda ser accedido con las credenciales apropiadas. Esta configuración es sencilla, dentro del directorio se ha creado un fichero llamado «.htaccess» con el contenido:

```
AuthName "Acceso restringido"
AuthUserFile "C:\ruta\.htpasswd"
AuthType Basic
require valid-user
```

Este fichero simplemente sirve para indicar que el directorio donde está alojado está protegido por contraseña. El campo `AuthUserFile` indica la ruta de un segundo fichero donde está la contraseña, en este caso «`.htpasswd`», y su contenido es:

```
usuario:contraseña
```

Apache se puede configurar de otras maneras, por ejemplo con la clave cifrada; también se pueden usar otros servidores web, en cualquier caso el resultado deber ser el mismo, el que se muestra en la figura 8-12.

8.4.3 Ejemplo III: Transmisión de voz

8.4.3.1 Explicación teórica

En este ejemplo se creará un pequeño y simple sistema que envíe voz sobre UDP a varios nodos a la vez.

Figura 8-12. Panel Frontal del resultado

En esta aplicación se ha usado UDP porque prima la velocidad sobre la fiabilidad. TCP asegura que los datos llegan correctamente al destino, pero para hacerlo añade sobrecarga de procesado y de información a la red, mientras que UDP simplemente encapsula los datos y los manda.

8.4.3.2 Código

El primer objetivo es enviar los datos a varios nodos a la vez, esto es lo que se conoce como multidifusión o *multicast*. Esto puede realizarse mediante el VI polimórfico *UDP Multicast Open*.

Para enviar los datos se usará el VI Express *Acquire Sound* configurado con una frecuencia de muestreo de 8 kHz, 1 canal, 16 bits y 0,05 segundos de duración. Para enviar menos datos, las muestras se convertirán de doble precisión a enteros de ocho bits pre-

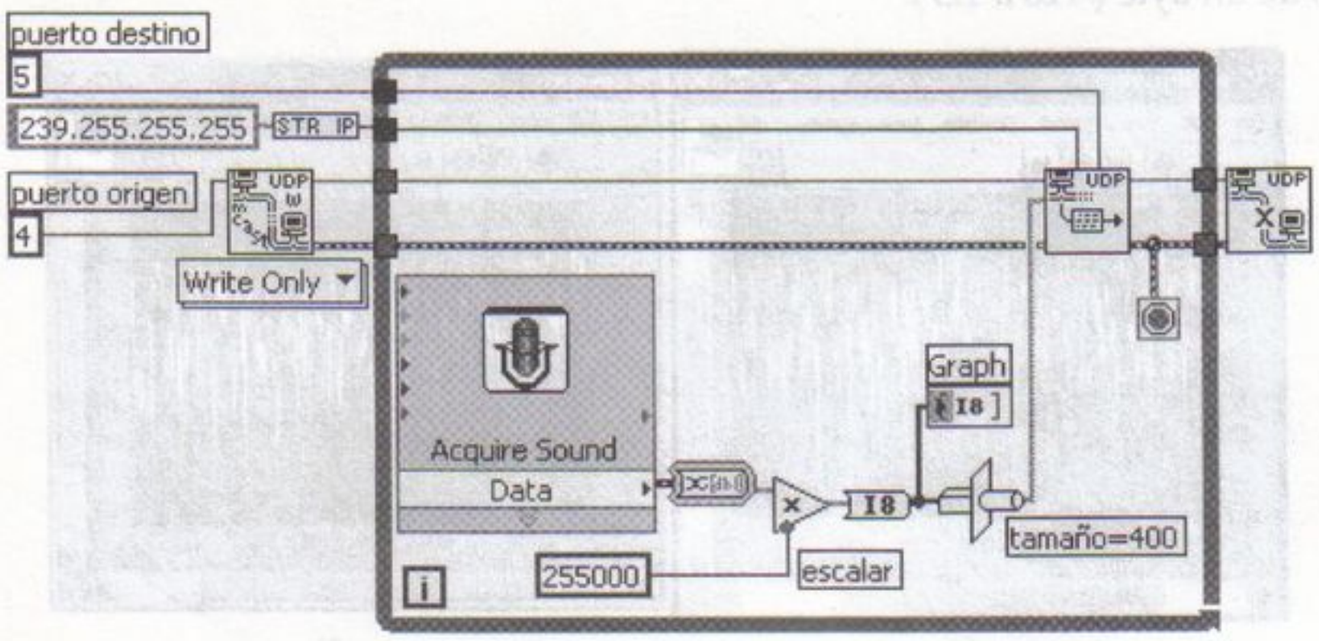


Figura 8-13. Emisor

viamente escalados. El factor de escalado se ha elegido estudiando los máximos que presenta la señal de audio y ajustándolo para aprovechar todo el rango disponible. Este proceso comprime el tamaño de los datos pero hace que el sonido pierda calidad. Finalmente las muestras se convierten a *string* con *Type Cast* y se envían por UDP.

Los receptores se unen al grupo de multidifusión a través de la dirección IP (también llamada dirección del grupo). Después únicamente tienen que leer los datos que les llegan y realizar el proceso inverso: convertir el *string* a *array* de enteros y des-escalarlo por el mismo factor; además también se ha construido un *waveform* indicando el periodo de muestreo (1/8 kHz=0,000125 s). Finalmente, se usa el VI *Play Waveform* para escuchar el sonido.

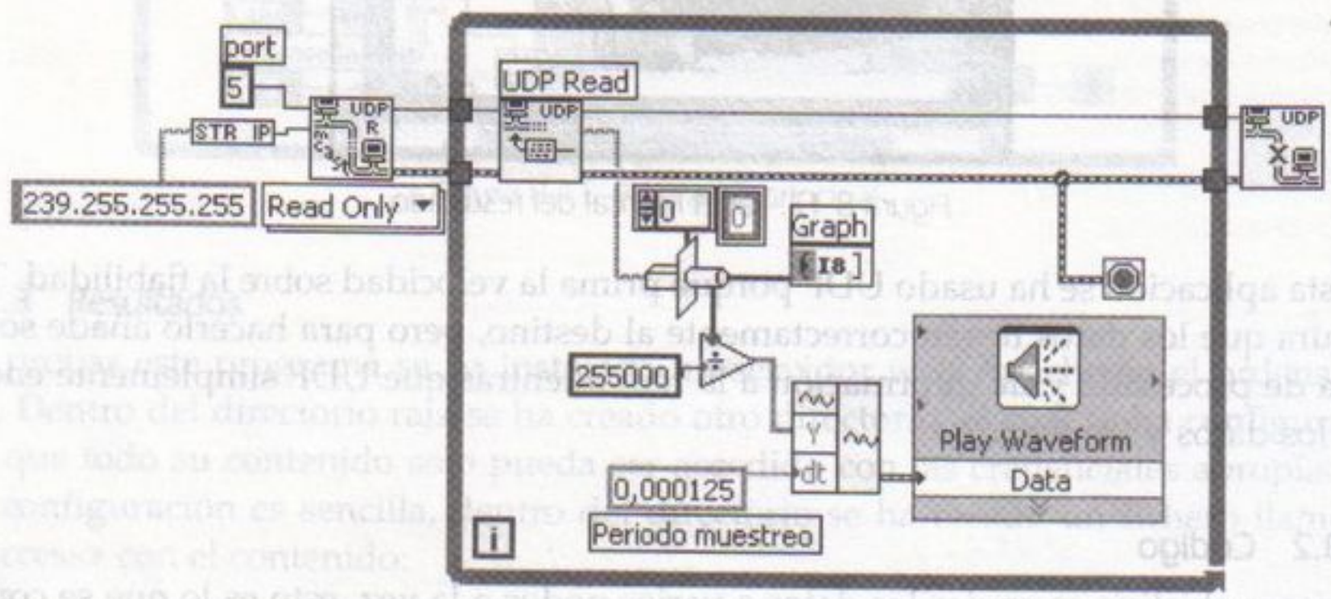


Figura 8-14. Receptor

8.4.3.3 Resultados

En la figura 8-15 se pueden ver los Paneles Frontales del emisor (b) y de un receptor (a). La señal es del tamaño esperado (400 muestras) y no llega a los límites para enteros con signo de un byte (-128 a 127).

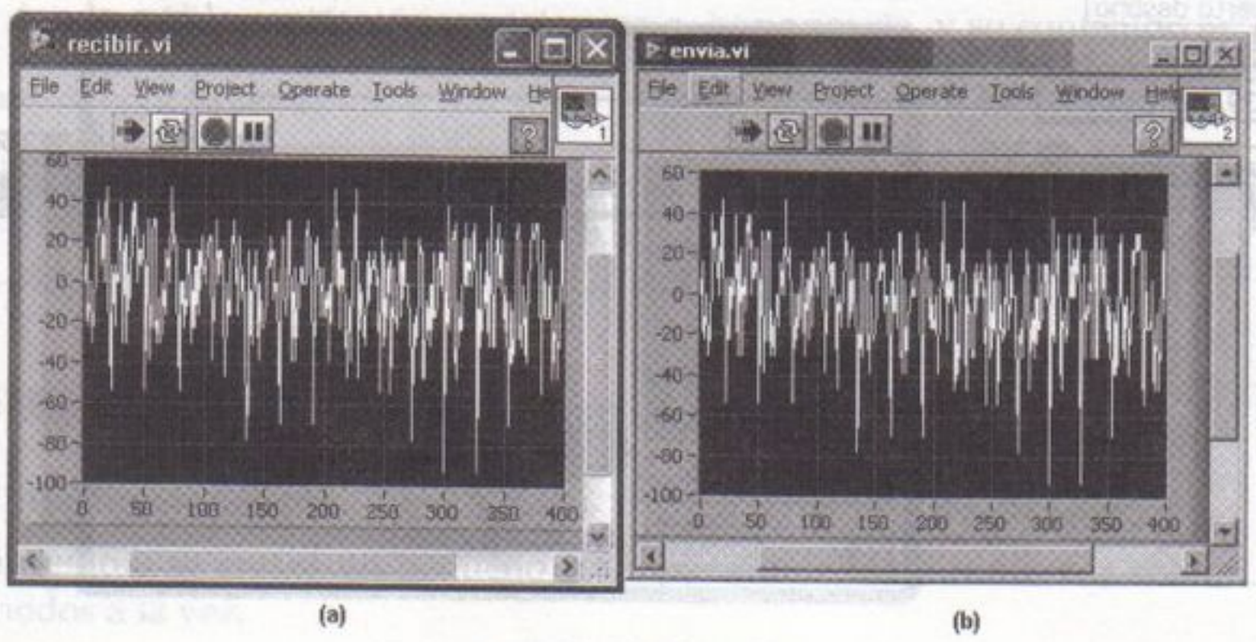


Figura 8-15. Resultado

8.5 Ejercicios

1. Buscar y estudiar ejemplos en la ayuda de LabVIEW y en la web de National Instruments sobre comunicación TCP/IP y UDP.
2. Modificar el ejemplo I para que se pueda modificar el directorio de trabajo.
4. Crear un programa de chat usando TCP/IP. Los nodos deben ser tanto servidores como clientes, según lo requiera la aplicación.

8.6 Bibliografía

Andrew S.Tanenbaum, *Redes de Computadoras*, Prentice Hall PTR, 1998.

Defense Advanced Research Projects Agency, *RFC 791 Internet Protocol*, 1981.

Defense Advanced Research Projects Agency, *RFC 793 Transmission Control Protocol*, 1981.

N. Borenstein y N. Freed, *RFC 1341 Multipurpose Internet Mail Extensions*, 1992.

National Instruments, *AN 160: Using LabVIEW with TCP/IP and UDP*, 2004.

Tim Berners-Lee et alt., *RFC 1945 Hypertext Transfer Protocol*, 1996.

W. Richard Stevens, *TCP/IP Illustrated Vol 1: The Protocols*, Addison-Wesley, 1993.

William Stallings, *Comunicación y Redes de Computadores*, Prentice Hall, 2001.

Capítulo 9

VI Server y comunicaciones avanzadas

9.1 Acceso remoto

Una opción muy interesante de LabVIEW es que permite acceder a un programa de forma remota. Para esto LabVIEW dispone de su propio servidor web y dos mecanismos: paneles remotos y publicación en web.

Estas aplicaciones se basan en el esquema cliente-servidor. El servidor será un servidor web que LabVIEW incorpora, puede habilitarse y configurarse en *Tools > Options > Web Server*, las opciones de configuración incluyen opciones de seguridad para permitir o denegar el acceso a ciertos usuarios, máquinas o archivos. El cliente será o bien el propio LabVIEW o bien un navegador web.

9.1.1 Paneles remotos

El primer método que se estudiará serán los paneles remotos. Para acceder a esta herramienta hay que dirigirse a *Operate > Connect to Remote Panel*, una vez allí se introducirá la dirección de la máquina remota y el nombre del VI que se quiere visualizar, que deberá estar cargado en la memoria de la máquina remota.

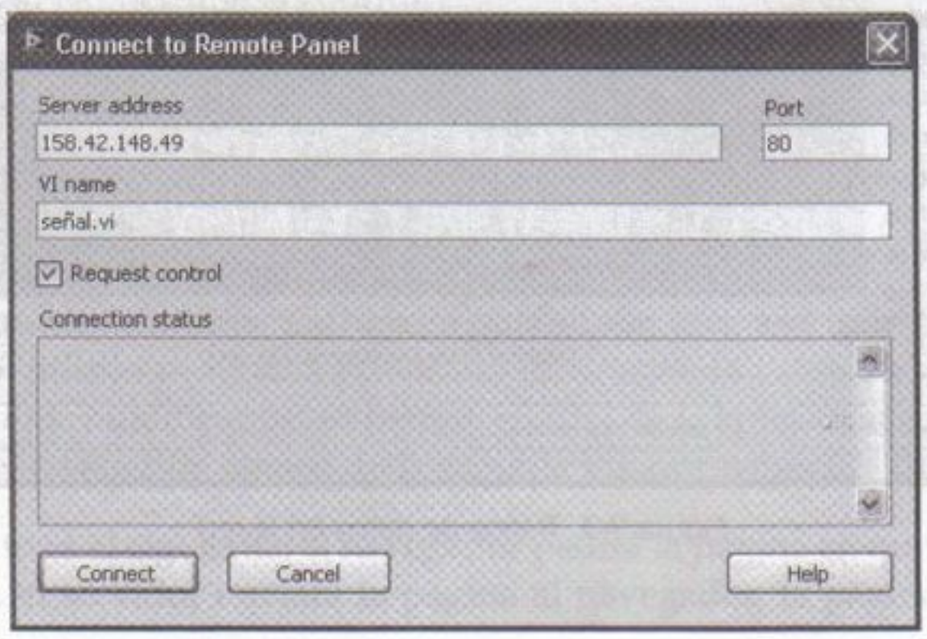


Figura 9-1. Ventana para la conexión a un panel remoto

Si LabVIEW logra conectarse con éxito aparecerá el Panel Frontal del VI remoto, ver figura 9-2. Varios clientes podrán visualizar el mismo VI. Se puede solicitar el control o renunciar a él desde el menú que aparece en el rectángulo blanco de la esquina inferior izquierda del VI o desde el menú contextual del Panel Frontal.

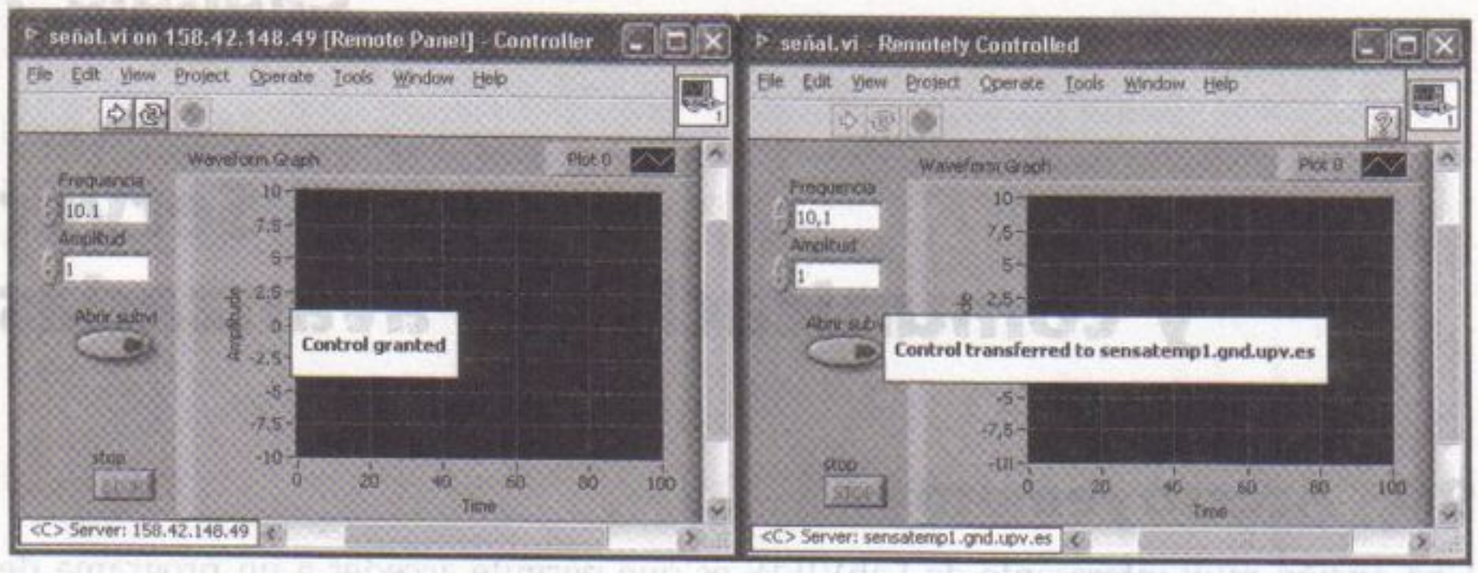


Figura 9-2 Panel local y remoto indicando la conexión

También puede estudiarse el tráfico de datos por la red a través de los paneles remotos mediante *Tools > Remote Panel Connection Manager*.

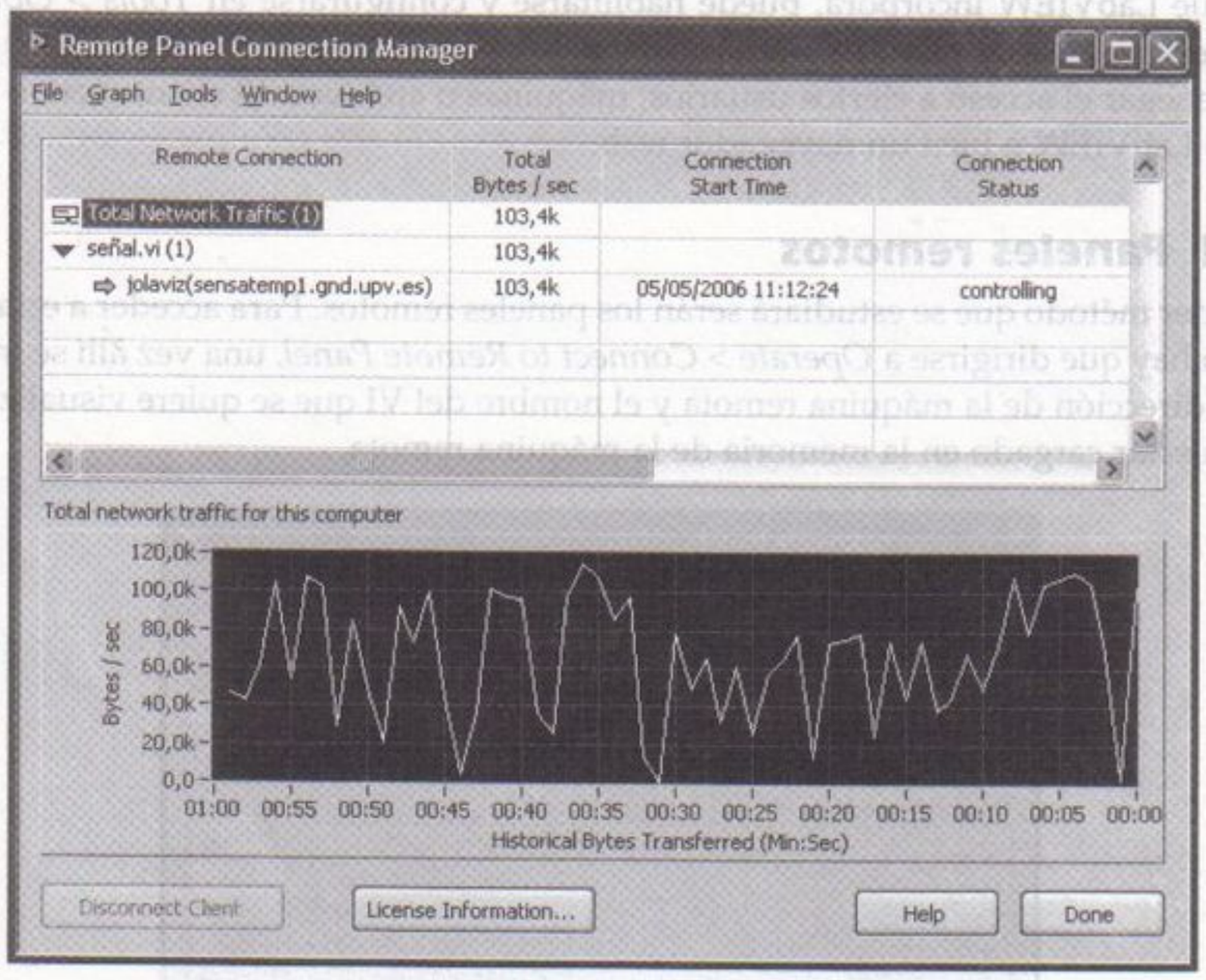


Figura 9-3. Tráfico de datos en la red

9.1.2 Publicación en web

En el siguiente ejemplo se usará un programa muy sencillo, simplemente suma dos números y el resultado se muestra por pantalla. La figura 9-4 muestra su ejecución normal.

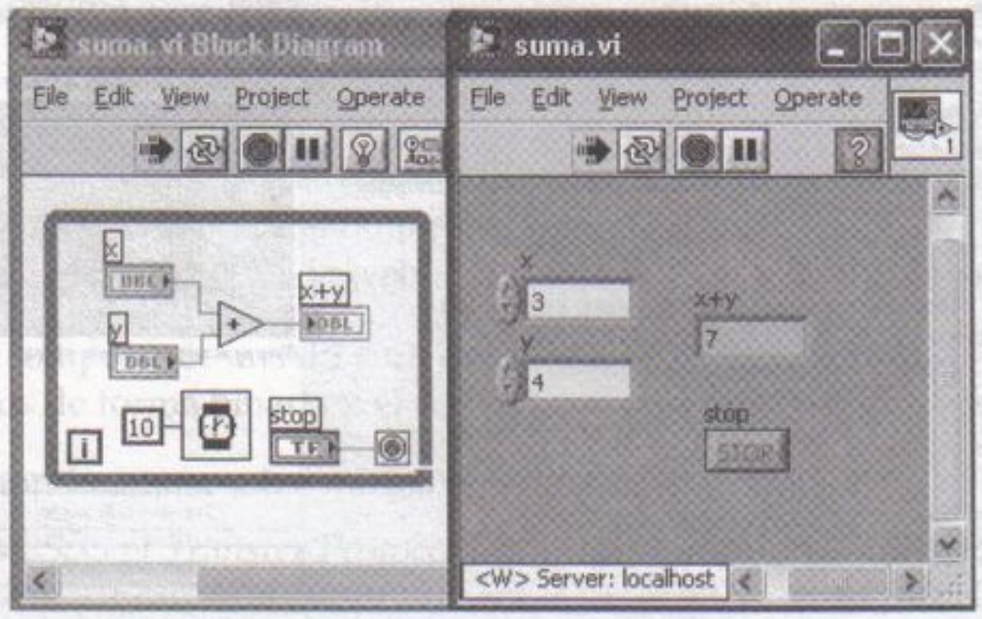


Figura 9-4. VI que suma dos números

Una vez creado el programa se puede acceder a la herramienta de publicación en web en el menú *Tools > Web Publishing Tool.* Esta herramienta consiste en un asistente que guía al usuario en la creación de una página a través de tres pasos. En el primero se elige el VI y el modo de publicación; en el segundo se puede personalizar la página añadiéndole un título y textos que se mostrarán antes y después del Panel Frontal; y finalmente en el tercer paso se guarda el fichero. Dependiendo de la versión de LabVIEW que se disponga habrá más o menos parámetros disponibles. Los modos de publicación para la versión profesional (ver figura 9-5) son:

- *Embedded*: permite el control remoto y la monitorización del VI si está cargado en memoria (abierto).
- *Snapshot*: muestra una imagen PNG del Panel Frontal del VI, es decir, una captura de pantalla. No permite el control.
- *Monitor*: igual que el anterior, pero la imagen se actualiza cada cierto tiempo.

Una vez guardada la página puede personalizarse abriéndola con un editor de texto y usando código HTML estándar, entre otras cosas se pueden insertar imágenes, otros textos, etc.

En la figura 9-6 puede verse una captura de pantalla en la que se muestra el VI anterior en LabVIEW y el mismo programa publicado en una página web vista a través de un navegador. Se puede modificar quien tiene el control de la aplicación a través del menú que aparece en el rectángulo blanco de la esquina inferior izquierda del VI.

El funcionamiento es el siguiente: el navegador pedirá la página en que se publica el VI al servidor web, el servidor enviará la página al navegador, la página web mostrada

requerirá un plug-in que debe estar instalado en el navegador y el plug-in utilizará las funciones de *LabVIEW Run-Time Engine* para mostrar y controlar el Panel Frontal.

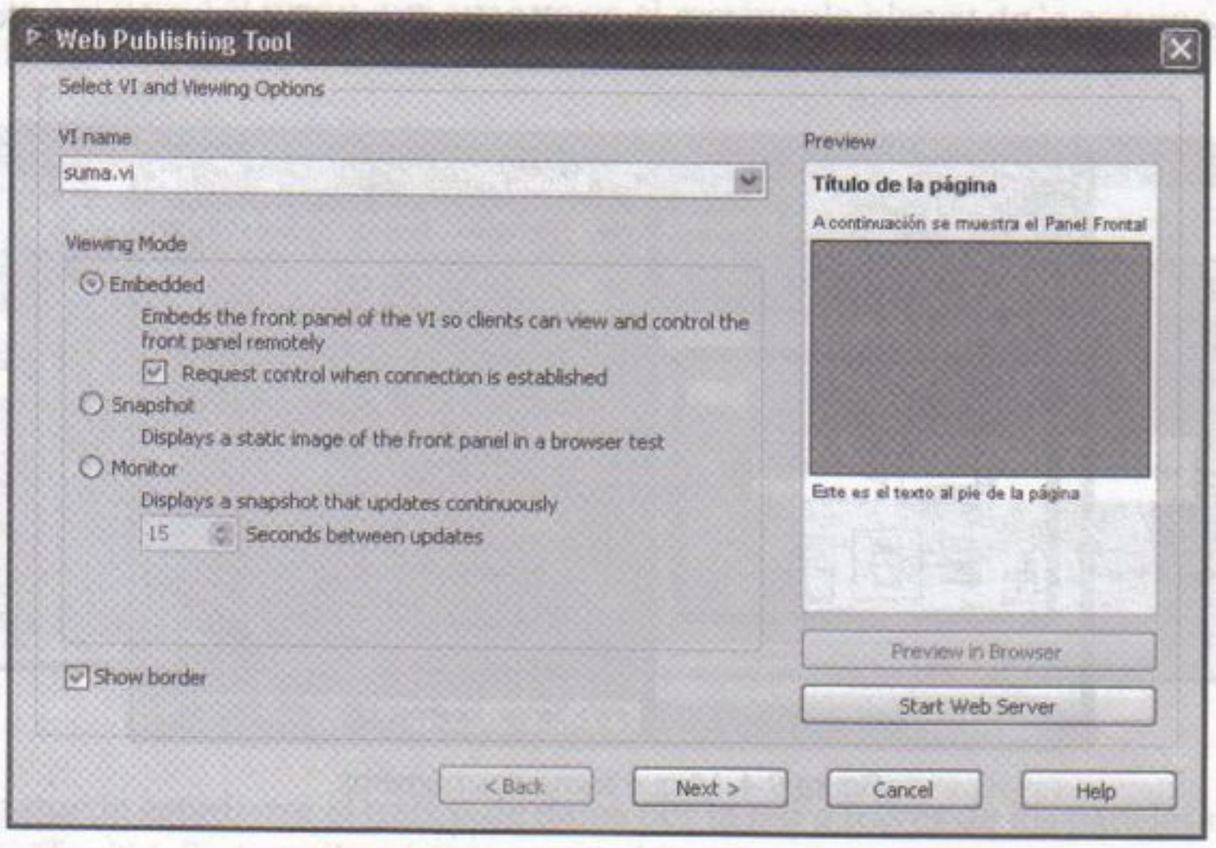


Figura 9-5. Herramienta para la publicación en web

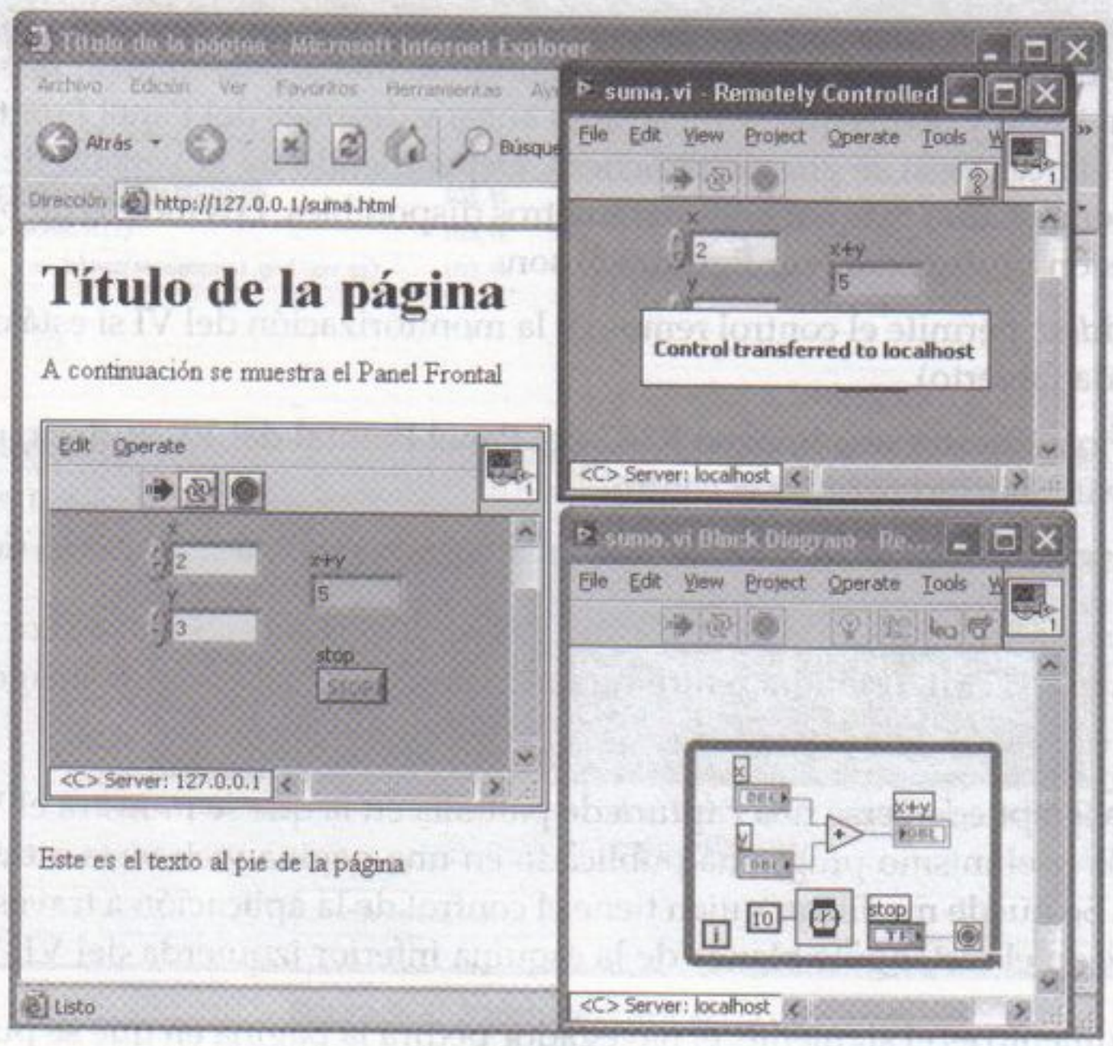


Figura 9-6. Página web publicada con el VI en funcionamiento

9.2 Enlaces de datos

En la sección anterior se ha visto la forma de comunicarse con una aplicación remota. En alguna ocasión puede que solamente interese acceder a unos datos concretos y usarlos en otra aplicación, en este caso los paneles remotos y la publicación en web no serían la solución óptima.

9.2.1 DataSocket

DataSocket es un método que hace que la comunicación entre aplicaciones a través de la red sea más sencilla (de más alto nivel) que con las funciones TCP o UDP y más flexible que los paneles remotos o la web.

DataSocket se compone de un API y un servidor. El API proporciona las funciones para compartir datos de forma binaria y el servidor maneja las conexiones con los clientes.

La comunicación funciona sobre varios protocolos:

- DSTP (*DataSocket Transfer Protocol*): es un protocolo que funciona sobre TCP/IP diseñado específicamente para DataSocket, en él intervienen tres elementos: DataSocket Server (puerto 3015), un publicador (*Publisher*) y un subscriptor (*Subscriber*). Los publicadores envían datos al servidor usando el API y los subscriptores los leen. Tanto los publicadores como los subscriptores son clientes del servidor. Las direcciones del protocolo DSTP son de la forma: `dstp:// nombreMaquina/datos`.
- OPC (*OLE for Process Control*): se usa de forma parecida a DSTP, en lugar de DataSocket Server usa un servidor OPC (el mecanismo para administrar variables compartidas es un servidor OPC). OPC es un estándar de comunicación entre controladores industriales. Las direcciones son `opc://nombreMaquina/nombreServidor/datos`, estas direcciones admiten parámetros para configurar el funcionamiento de la comunicación OPC. Esta comunicación está específicamente diseñada para compartir datos en tiempo real.
- LOOKOUT: este protocolo, al igual que DSTP, fue desarrollado por National Instruments. También trabaja sobre TCP/IP y se usa con SCADAS, el módulo DSC y con dispositivos FieldPoint. Las direcciones dependen de la aplicación.
- HTTP y FTP: son protocolos clásicos y bien conocidos que funcionan sobre TCP/IP y cuyas direcciones son las URLs típicas. Para trabajar con ficheros de texto se puede incluir « `[text]` » al final de la dirección.
- Files: en esta ocasión los datos son leídos directamente de un fichero. Las direcciones son `file:\\nombreMaquina\ruta\fichero`.

Los más usados son los dos primeros protocolos. Para usar el protocolo DSTP hace falta que se esté ejecutando el DataSocket Server, se puede iniciar la aplicación en la carpeta de *National Instruments* > *DataSocket* en el menú de inicio de Windows.

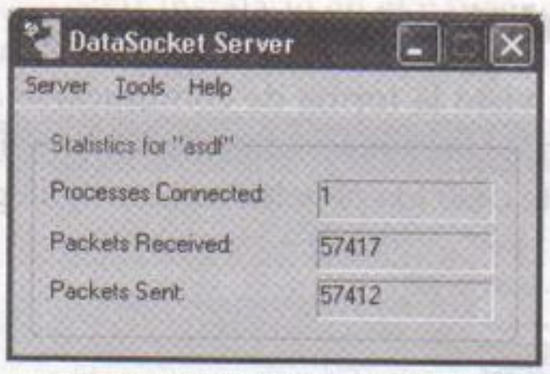


Figura 9-7. Servidor DataSocket

En las versiones 8.x de LabVIEW, National Instruments recomienda usar el *Shared Variable Engine* (servidor OPC) porque es más rápido y confiable (ver apartado 3.4.3).

9.2.1.1 API

En LabVIEW el API de DataSocket lo forman las funciones de *Data Communication > DataSocket*.

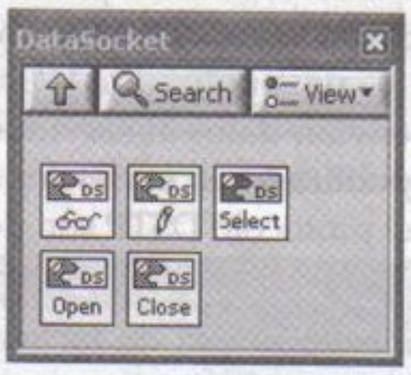
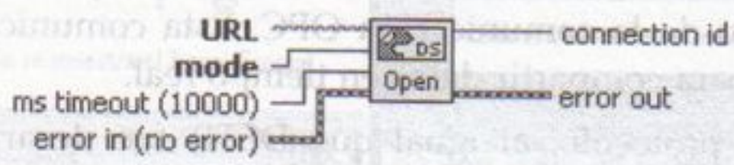


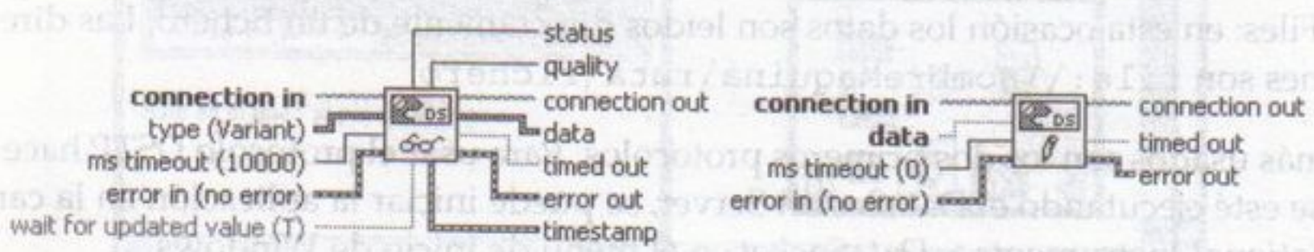
Figura 9-8. Funciones DataSocket

- **DataSocket Open**



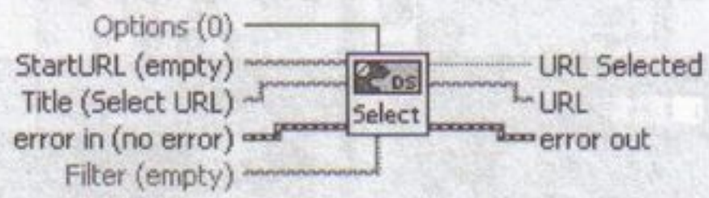
Abre una conexión a un objeto determinado. En el terminal *URL* se escribirá la dirección de acuerdo con el protocolo utilizado, *mode* sirve para indicar si se realizará una lectura, escritura, lectura/escritura o se usarán buffers.

- **DataSocket Read/Write**



Las funciones para leer o escribir datos a través de DataSocket admiten tanto una URL como una referencia a una conexión previamente abierta. Admiten cualquier tipo de datos.

- DataSocket Select URL



Muestra una ventana donde se puede buscar el objeto concreto a que se quiere acceder.

9.2.1.2 Método directo

La forma más fácil de lograr la comunicación con DataSocket no es a través del API, existe otro método más sencillo, donde no hay que realizar ninguna programación.

El primer paso es presionar con el botón derecho sobre el control que contiene los datos a servir, en el menú se elige *Properties > Data Binding*. En la ventana que aparece, figura 9-9, se podrá elegir si los datos se leerán, se escribirán y cuál es la dirección del servidor. Este proceso se realizará tanto en el publicador como en el subscriptor. Cuando se haga aparecerá un pequeño rectángulo junto al control que se coloreará de verde cuando haya conexión con el servidor.

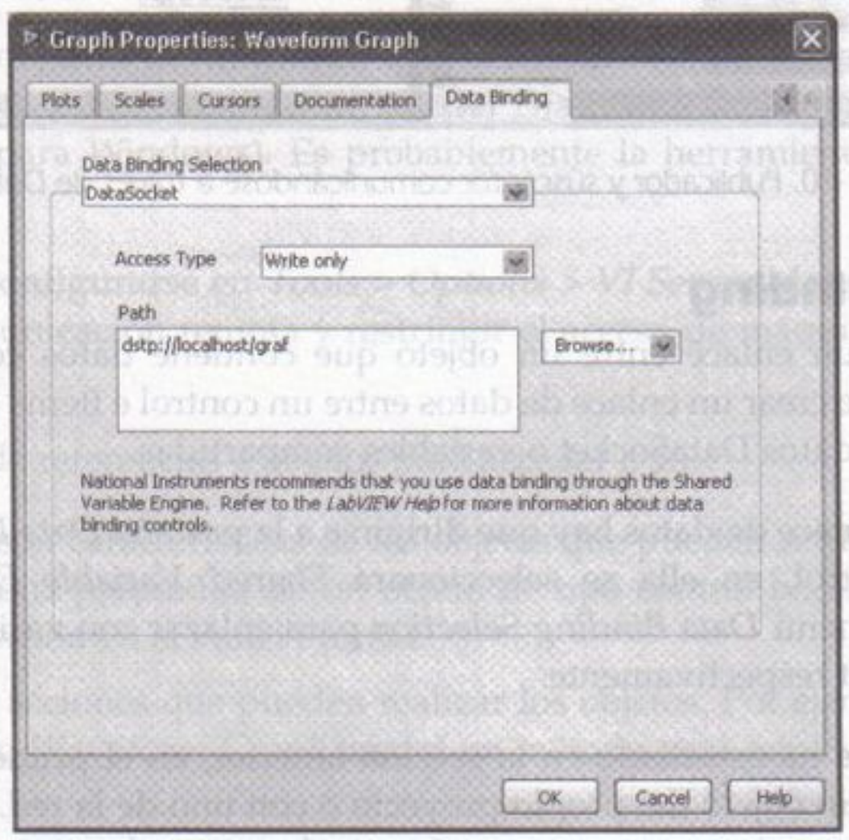


Figura 9-9. Método directo de comunicación con DataSocket

En la siguiente imagen, figura 9-10, se puede ver el publicador a la izquierda y el subscriptor a la derecha compartiendo los datos, en ambos casos el indicador de conexión está en verde.

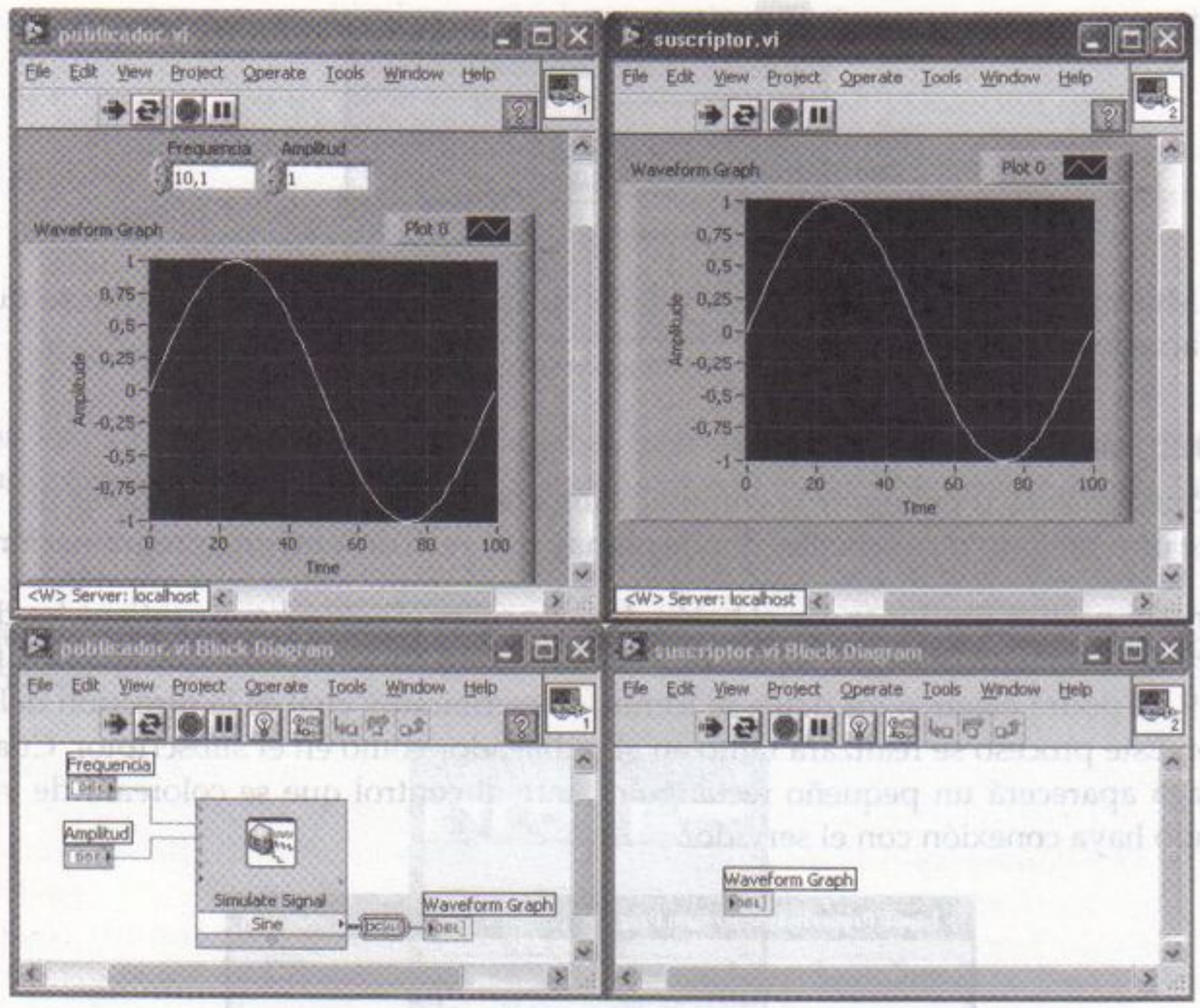


Figura 9-10. Publicador y suscriptor comunicándose a través de DataSocket

9.2.2 Data Binding

Data Binding es un enlace entre un objeto que contiene datos con un control. En LabVIEW se puede crear un enlace de datos entre un control e ítems del propio proyecto o de red, como datos DataSocket o variables compartidas.

Para realizar un enlace de datos hay que dirigirse a la pestaña *Data Binding* de las propiedades del control, en ella se seleccionará *Shared Variable Engine (NI-PSP)* o *DataSocket* en el menú *Data Binding Selection* para enlazar con variables compartidas y datos DataSocket respectivamente.

El segundo caso se ha estudiado en la sección anterior, en el primero hay que elegir entre enlazar con un objeto del propio proyecto o con uno de la red. Una vez seleccionado el objeto aparecerá una marca junto al control que se coloreará cuando haya conexión, en este caso se trata de un triángulo.

Para enlazar con variables compartidas del propio proyecto este método es equivalente a arrastrar la variable desde el explorador del proyecto al Panel Frontal del VI.

En la figura 9-11 puede verse como, sin haber realizado ninguna programación, se pueden compartir datos entre varias aplicaciones.

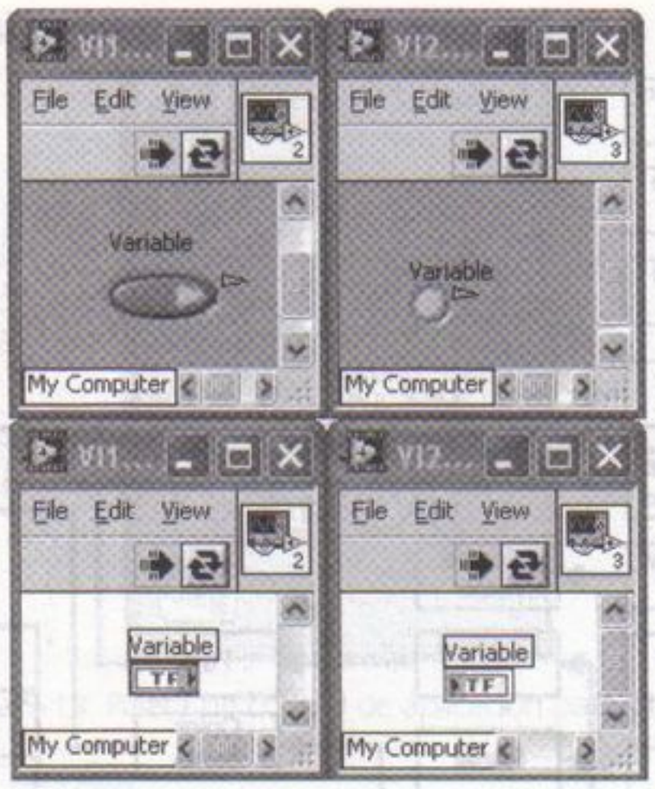


Figura 9-11. Compartir datos entre varias aplicaciones con PSP

9.3 VI Server

VI Server es un conjunto de funciones que desde la versión 5.0 permiten controlar LabVIEW de forma programada a través del Diagrama de Bloques, de TCP/IP o de ActiveX (sólo para Windows). Es probablemente la herramienta más potente de LabVIEW.

VI Server puede configurarse en *Tools > Options > VI Server*, donde se podrá elegir el método de comunicación remota y restringir el acceso de máquinas, usuarios y ficheros.

Se basa en el uso de referencias a objetos para acceder a sus:

- Propiedades: son características de los objetos que pueden ser leídas y/o escritas. Por ejemplo, una propiedad de un objeto de tipo control booleano puede ser su valor o su posición en el Panel Frontal.
- Métodos: son acciones que pueden realizar los objetos. Por ejemplo, un objeto de tipo control de *Waveform Graph* tendrá un método para reiniciarlo a su valor por defecto.

Los objetos de LabVIEW se agrupan en clases, las clases principales de VI Server son *Application* y *VI*, aunque LabVIEW tiene muchas más definidas; se agrupan de forma jerárquica como representa la figura 9-12 (no se muestra XNodes). Para asociar una clase a un nodo hay que seleccionarlo en el menú *Select Class* del menú contextual, una vez hecho se podrá acceder a las propiedades y métodos para los objetos de la clase elegida. Los objetos de las clases "hijas" heredarán las propiedades y métodos de las clases "padre". Se pueden explorar las clases, los métodos y propiedades de cada una de ellas desde LabVIEW con *View > Class Browser*.

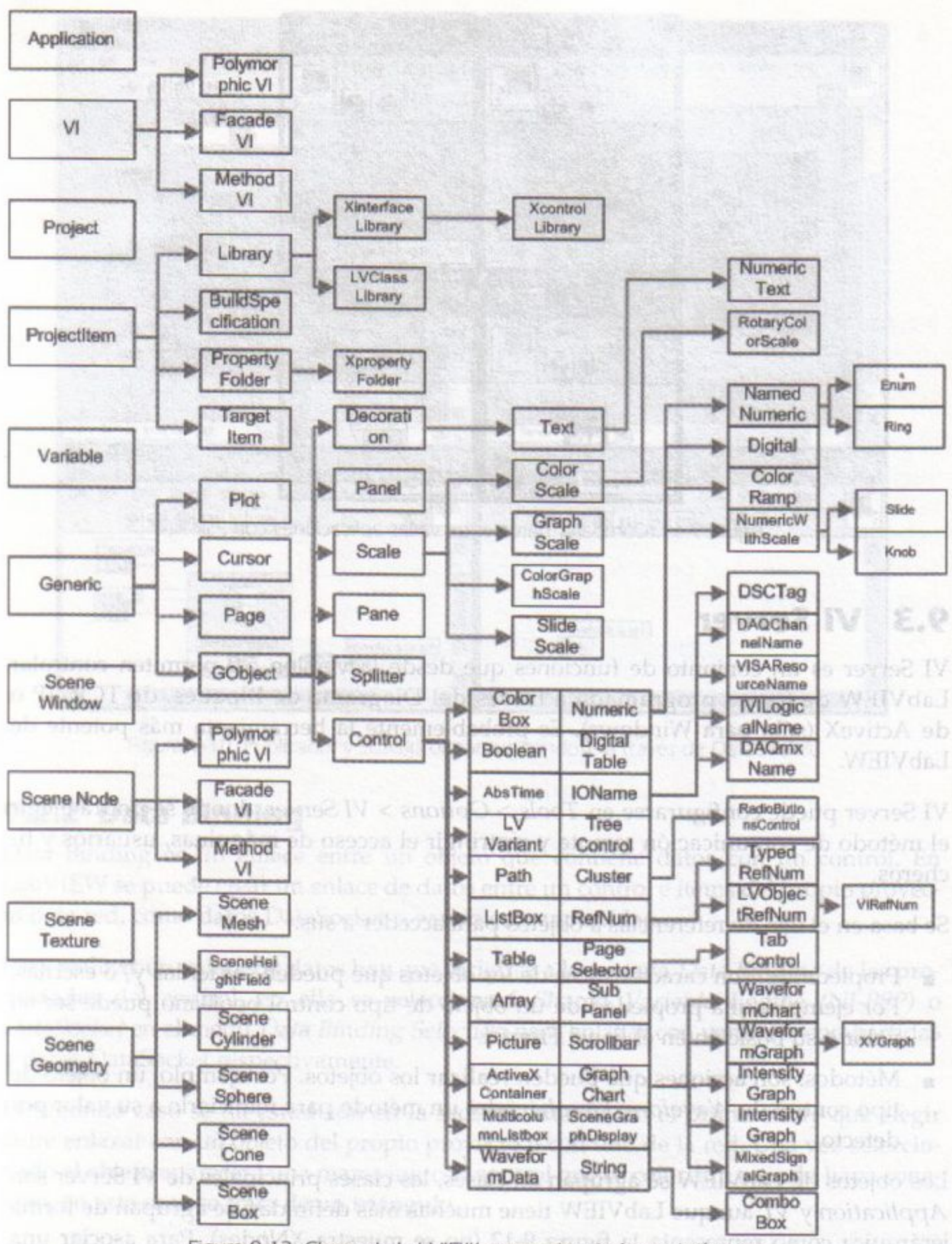
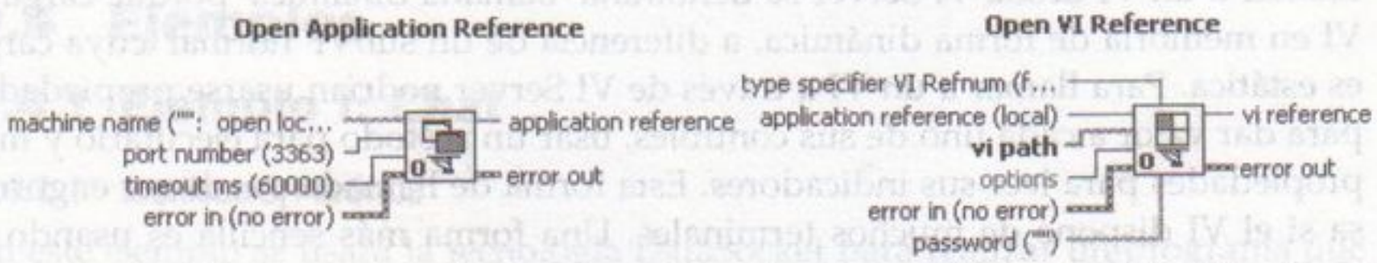


Figura 9-12. Clases de LabVIEW agrupadas de forma jerárquica

Se puede acceder a VI Server a través de las funciones de la paleta *Programming > Application Control*. El proceso básico consistirá en abrir o crear una referencia, leer/escribir propiedades o invocar métodos y finalmente cerrar la referencia.

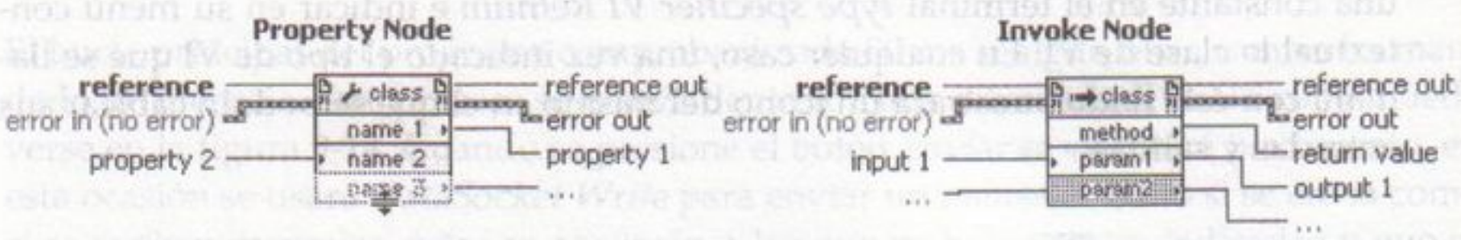
Figura 9-13. Paleta de control de aplicación para VI Server

Abrir referencias



Hay dos funciones para abrir referencias a las clases principales (*Application* y *VI*). En la primera hay que indicar a qué máquina hay que conectarse (si es remota) y la segunda función abre una referencia a un VI en particular. Si se quiere abrir un VI en una máquina remota habría que usar primero *Open Application Reference* y luego *Open VI Reference*, mientras que si el VI está en la máquina local sólo hace falta usar *Open VI Reference*. Si no se usa *Close Reference* las referencias permanecerán abiertas hasta que el VI se detenga.

Propiedades y métodos



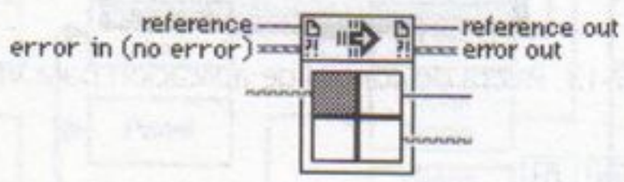
Los nodos de propiedades y métodos son el núcleo de la programación con VI Server. Mediante ellos se accederá a todas las posibilidades que ofrece VI Server. Ambos necesitan como entrada una referencia a un objeto. Una vez hecho se seleccionará automáticamente la clase y mostrarán las propiedades y métodos de que dispone (también puede hacerse desde el menú contextual con *Select Class* y *Link to*). El programador podrá entonces elegir qué propiedad o método en concreto

desea utilizar a través del menú contextual o pinchando directamente sobre ellos con la herramienta *Operate Value*.

Los nodos de propiedad podrán extenderse para que el mismo nodo muestre más de una propiedad, en este caso el orden de ejecución es desde arriba hacia abajo. Algunas propiedades pueden ser tanto de lectura como de escritura, por defecto serán de lectura, pueden cambiarse a través del menú contextual.

Los nodos de métodos mostrarán el nombre del método en el primer terminal, el resto de terminales serán las entradas y salidas de dicho método.

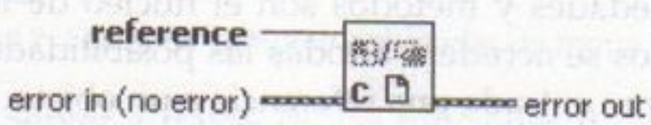
- **Calling By Reference**



Llamar a un VI desde VI Server se denomina llamada dinámica porque carga el VI en memoria de forma dinámica, a diferencia de un subVI normal cuya carga es estática. Para llamar a un VI a través de VI Server podrían usarse propiedades para dar valor a cada uno de sus controles, usar un método para ejecutarlo y más propiedades para leer sus indicadores. Esta forma de llamada puede ser engorrosa si el VI dispone de muchos terminales. Una forma más sencilla es usando el *Calling By Reference Node*.

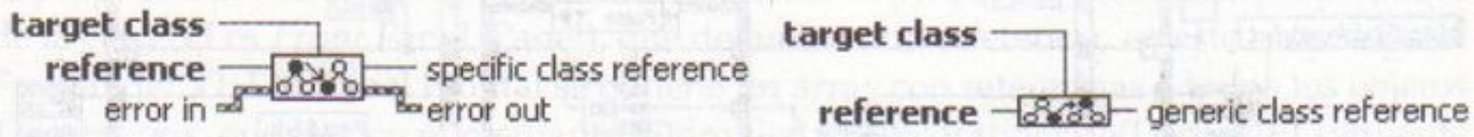
Este nodo actúa de forma parecida a la inclusión tradicional de un subVI, de hecho si en *Call Setup* se selecciona un VI y la opción *Load with callers*, este nodo automáticamente será sustituido por su correspondiente subVI. Además de indicar de forma explicita el VI a llamar desde *Call Setup*, también se puede hacer de forma dinámica, en este caso al nodo se le asociará un tipo estricto de VI (tipo de conector) mediante la opción *VI Server > Browse* de su menú contextual; después se abrirá una referencia al VI en concreto con *Open VI Reference*, el cual debe tener el mismo tipo de conector que el indicado anteriormente, para esto se puede crear una constante en el terminal *type specifier VI Refnum* e indicar en su menú contextual la clase de VI. En cualquier caso, una vez indicado el tipo de VI que se llamará con este nodo, mostrará un icono del mismo en el que se podrán cablear sus entradas y salidas.

- **Close Reference**



Cierra una referencia abierta antes. En caso de que la referencia sea a un VI, éste será descargado de la memoria a menos que su Panel Frontal esté abierto, que sea un subVI de otro VI cargado en memoria o haya otras referencias abiertas.

- To More Specific/Generic Class



La figura 9-12 muestra la jerarquía de objetos en LabVIEW. Para manipular las referencias a esos objetos se dispone de las funciones *To More Specific Class* y *To More Generic Class*. La primera de ellas devuelve una referencia a una clase inferior a la de la entrada, por ejemplo, pasa de una referencia tipo *Control* a una tipo *Numeric*. La segunda hace el proceso inverso.

Por último también hay que comentar que los expertos y gurús de LabVIEW esperan que National Instruments ofrezca dentro de poco tiempo un producto, de momento conocido como XNodes o *VI scripting*, que de una forma parecida a VI Server ofrecería muchas más herramientas.

9.4 Ejemplos

9.4.1 Ejemplo I: Chat

9.4.1.1 Explicación teórica

En este ejemplo se usará la tecnología DataSocket para realizar un programa que gestione un chat. El programa deberá ser capaz de enviar mensajes de texto desde un ordenador a otro y mostrar toda la conversación en una pantalla. El número de nodos que puede haber en el chat será indeterminado.

9.4.1.2 Código

El código en primer lugar abrirá una conexión con DataSocket Server, en la figura 9-14 se ha usado una constante con valor `dstp:\\localhost\chat` porque se va a probar entre dos instancias del programa en el mismo ordenador, para probarlo entre ordenadores distintos el valor de esta constante lógicamente será diferente.

El bucle principal del programa comprobará cada 50 ms si algún otro usuario ha mandado un mensaje. Este proceso se realiza dentro de una estructura EVENT como puede verse en la figura 9-14. Cuando se presione el botón *enviar* se ejecutará otro evento, en esta ocasión se usará *DataSocket Write* para enviar un mensaje. Tanto si se envía como si se reciben mensajes, éstos se anexarán a los que ya había en un indicador y que se desplazará hasta abajo para mostrar los últimos mensaje mediante la propiedad *Text.ScrollPos*.

9.4.1.3 Resultados

Para probar el chat se ha activado DataSocket Server, luego hay que abrir varias instancias del mismo programa, una opción es hacer varias copias del mismo fichero con

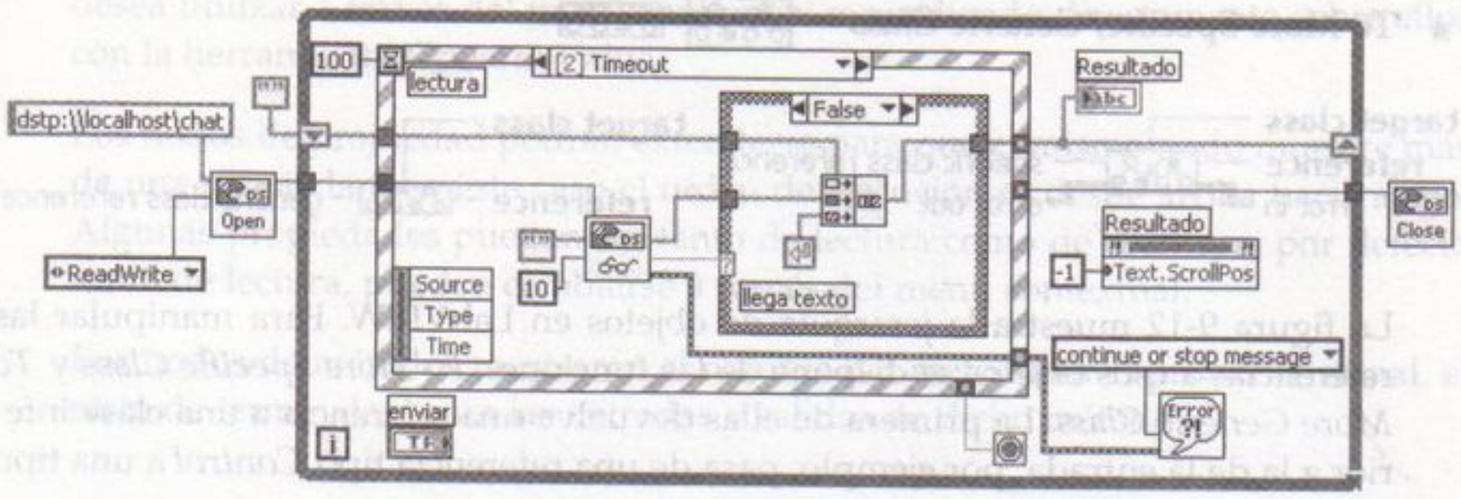


Figura 9-14. Evento de lectura del programa de chat usando DataSocket

nombres diferentes y abrirlas, otra opción es crear una plantilla y abrirla varias veces. El resultado puede verse en la figura 9-15.

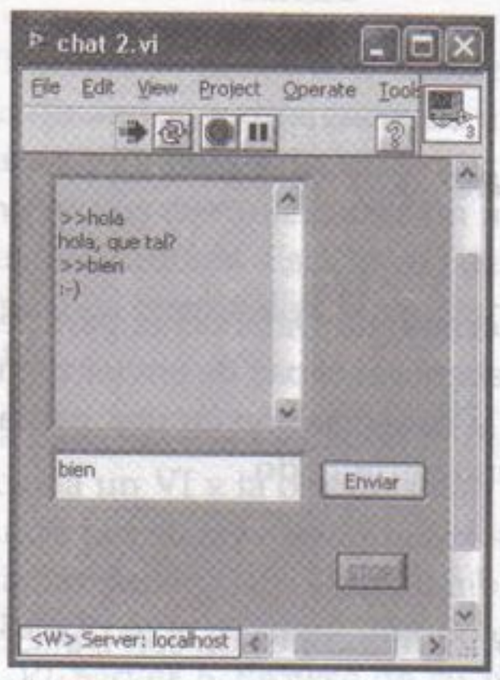


Figura 9-15. Ejecución de chat.vi

9.4.2 Ejemplo II: Controlar Decorations

9.4.2.1 Explicación teórica

Hasta ahora se sabía controlar de forma programada los controles e indicadores colocados en el Panel Frontal de un VI mediante las propiedades asociadas a estos terminales (ver ejemplo III de capítulo 3), pero esto tiene varias limitaciones, una de ellas es que en el Panel Frontal también se pueden colocar *Decorations*, pero éstas no se reflejan en el Diagrama de Bloques, por lo tanto de ellas no se pueden crear referencias, propiedades y métodos como con los controles.

En este ejemplo se trata de mover un objeto *Decoration*, algo que sólo podía hacer el programador mientras está diseñando el VI. Para ello se usará VI Server con el fin de conseguir una referencia a los objetos *Decorations* del Panel Frontal del VI.

9.4.2.2 Código

El programa abrirá en primer lugar una referencia al propio VI, una de las propiedades de la clase VI es *Front Panel* (Panel), que devuelve otra referencia, en este caso al Panel Frontal del VI. Del Panel Frontal se obtiene un *array* con referencias a todos los objetos *Decorations* que hay en él mediante la propiedad *Decorations []* (Decos). El siguiente paso es seleccionar de este *array* la referencia a un objeto en concreto mediante *Index Array*. Ya se tiene una referencia a un objeto *Decoration*, ahora sólo hay que usar sus propiedades *Top* y *Left* para modificar su posición.

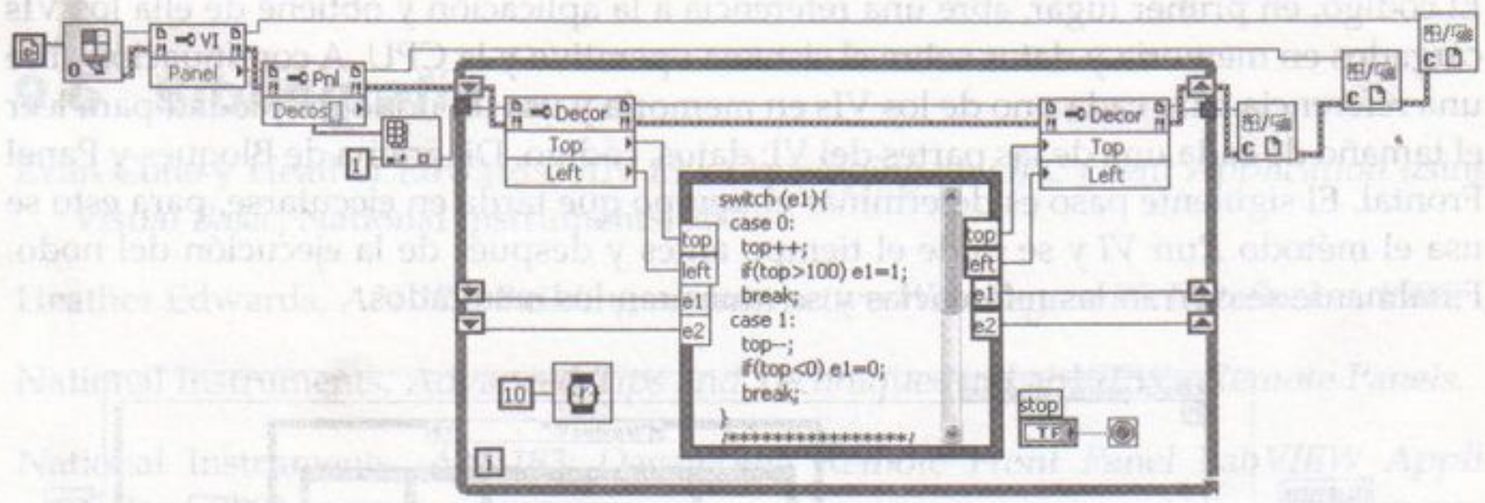


Figura 9-16. VI que mueve un objeto Decoration

El algoritmo para mover el objeto es una máquina de estados para cada una de las dimensiones en que puede moverse, para implementarlo se usará un nodo fórmula cuyo código es:

```
/* Vertical */
switch (e1){
  case 0:
    top++;
    if(top>100) e1=1;
    break;
  case 1:
    top-;
    if(top<0) e1=0;
    break;
}
/* Horizontal */
switch (e2){
  case 0:
    left++;
    if(left>100) e2=1;
    break;
  case 1:
    left-;
    if(left<0) e2=0;
    break;
}
```

9.4.3 Ejemplo III: Rendimiento de VIs

9.4.3.1 Explicación teórica

La siguiente aplicación debe obtener estadísticas sobre el rendimiento de otros programas de LabVIEW. Las estadísticas que debe mostrar son el tamaño que usa (desglosado en cada uno de los componentes del VI) y el tiempo de ejecución.

9.4.3.2 Código

El código, en primer lugar, abre una referencia a la aplicación y obtiene de ella los VIs cargados en memoria y datos sobre el sistema operativo y la CPU. A continuación abre una referencia para cada uno de los VIs en memoria y usa un nodo propiedad para leer el tamaño de cada una de las partes del VI: datos, código, Diagrama de Bloques y Panel Frontal. El siguiente paso es determinar el tiempo que tarda en ejecutarse, para esto se usa el método *Run VI* y se mide el tiempo antes y después de la ejecución del nodo. Finalmente se cierran las referencias y se muestran los resultados.

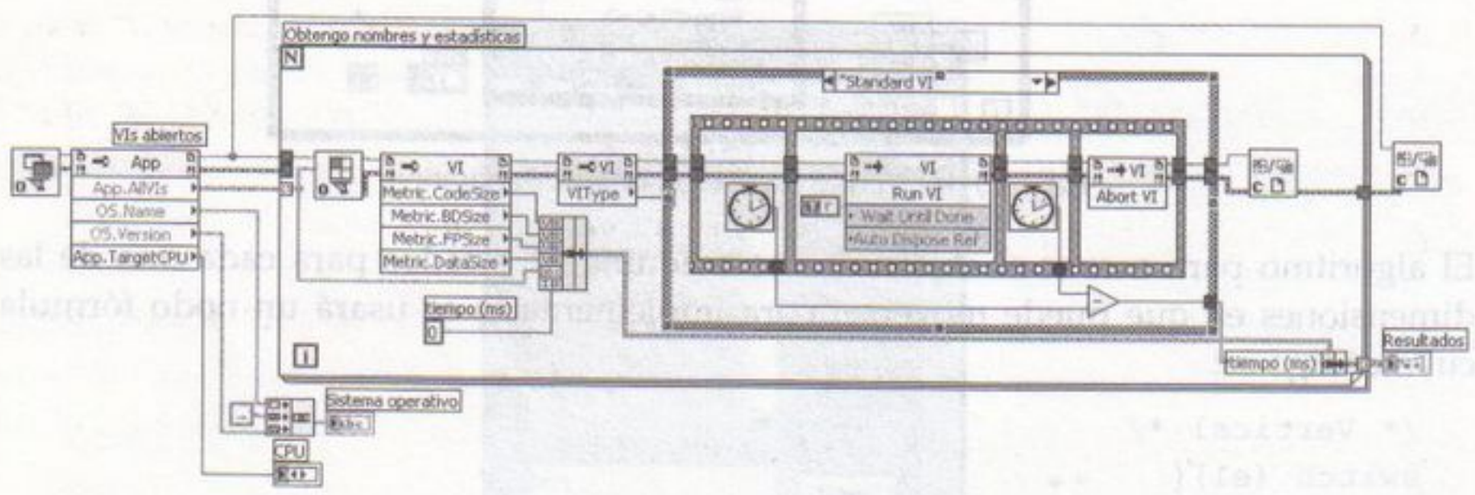


Figura 9-17. Análisis del rendimiento de un VI

9.4.3.3 Resultados

La figura 9-18 muestra una captura de pantalla de la ejecución.

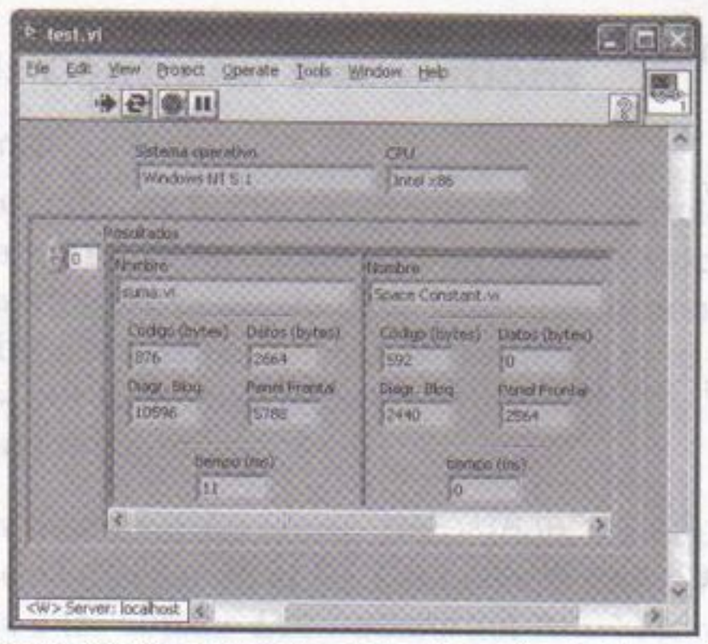


Figura 9-18. Resultados de la ejecución de 'test.vi'

9.5 Ejercicios

1. Publicar en una página web cada uno de los programas de ejemplo de este capítulo.
2. El chat del ejemplo I ¿es una aplicación P2P o cliente/servidor?
3. Repetir el ejemplo del capítulo 3 de variables compartidas usando DataSocket.
4. Implementar el algoritmo del ejemplo II sin emplear el nodo fórmula ni ninguna otra estructura de scripts.

9.6 Bibliografía

Evan Cone y Heather Edwards, *AN 139: Developing an OPC client Application using Visual Basic*, National Instruments, 2000.

Heather Edwards, *AN 127: Building an Interactive Web Page with DataSocket*, 1999.

National Instruments, *Advanced Tips and Techniques in LabVIEW - Remote Panels.*

National Instruments, *AN 183: Developing Remote Front Panel LabVIEW Applications*, 2002.

National Instruments, *LabVIEW Basics II Course Manual*, 2000.

National Instruments, *LabVIEW Communication Techniques for Distributed Applications.*

National Instruments, *LabVIEW User Manual*, 2001.

National Instruments, *Programmatically Controlling LabVIEW.*

National Instruments, *Using Control References and Classes in LabVIEW.*

National Instruments, *WP 1680: Integrating the Internet into Your Measurement System, DataSocket Technical Overview*, 1999.

Sección III

PROGRAMACIÓN AVANZADA

Capítulo 10. Sincronización y Multihilo.

Cuando se realiza una programación más compleja aparecen términos como pueden ser multitarea, procesos y sincronización. Se dice que los sistemas operativos son multitarea porque pueden ejecutar más de un programa a la vez; este concepto puede extenderse también a una sola aplicación y hacer que la aplicación pueda tener varios 'hilos' ejecutándose en paralelo.

Este capítulo comienza explicando los conceptos teóricos y cómo se aplican en LabVIEW, a continuación se describen las técnicas de sincronización y los posibles problemas y soluciones que pueden aparecer cuando se trabaja con múltiples hilos.

Capítulo 11. Modelos de programación.

Continuando con conceptos avanzados, en este capítulo se verán técnicas y modelos para distintos tipos de programas: tratamiento de errores, sistemas de comunicaciones y principalmente máquinas de estados y programación orientada a objetos.

Capítulo 12. Código externo.

Este capítulo abre el código de LabVIEW a otros lenguajes de programación, en él se verá cómo hacer que el código creado con lenguajes como C, C++, C# o Visual Basic se puede usar en LabVIEW y viceversa. Además también se estudiará la forma de usar tecnologías como ActiveX o la reciente .NET en LabVIEW. Una vez más, las explicaciones se basan en ejemplos prácticos.

Capítulo 13. Optimización del interfaz.

Los dos últimos capítulos tratan de optimizaciones, en este caso sobre el interfaz de usuario, para ello se muestran algunos elementos nuevos o no habituales que pueden usarse en los interfaces. El capítulo finaliza con una serie de consejos para dar al programa un aspecto profesional.

Capítulo 14. Optimización del código.

Este capítulo comienza con una breve introducción a la ingeniería del software y a los sistemas de control de versiones para más tarde centrarse en cómo mejorar el rendimiento de los programas, comparando el rendimiento de ejemplos de programas 'buenos' y 'malos' (o mejor dicho, 'no tan buenos').

Capítulo 15. Otras plataformas.

En el último capítulo se verán sencillas aplicaciones del uso de LabVIEW, con algunos módulos extra, para programar sobre distintas plataformas: en ordenadores de mano (PDA), en dispositivos reconfigurables (FPGA) y en sistemas de alto rendimiento (PXI).

Capítulo 10

Sincronización y multihilo

Se dice que los sistemas operativos son multitarea porque pueden ejecutar más de un programa a la vez. Eso lo consiguen compartiendo recursos, principalmente tiempo de la CPU. Si el sistema operativo conmuta con la suficiente rapidez varios programas en una única CPU se consigue un efecto parecido al que habría con varias CPUs.

Un proceso es otro concepto parecido al de programa. El proceso estaría compuesto por instrucciones, espacio de memoria y otra información. La definición exacta depende del sistema operativo al que se haga referencia. Entre la información que el sistema operativo asocia a cada proceso están: una identificación, el estado del proceso, el valor de los registros de la CPU cuando se ejecutó por última vez (contexto) y otros datos. El estado de un proceso indica si puede ejecutarse, está ejecutándose, está esperando a que se le asigne un recurso, etc. Para indicar qué proceso debe ejecutarse y cuál no existe un planificador de procesos. Éste contiene una lista de todos los procesos y su función consiste en cambiar el estado de cada proceso y su contexto. Los planificadores de tareas más básicos simplemente conmutan unos procesos y otros y les asignan un tiempo de proceso fijo. Los más modernos usan un sistema de prioridades bastante complejo para determinar qué proceso pasa a ejecutarse, algunos tienen prioridades fijas, otros se basan en un sistema de créditos.

Un proceso puede contener hilos (uno o varios). Un hilo (del inglés *thread*) es, al igual que un proceso, un flujo de ejecución; es la consecuencia de extender el concepto de multitarea dentro de los procesos.

En los sistemas operativos de la familia Windows NT se pueden crear y ejecutar múltiples hilos dentro de un proceso. De esta forma no sólo se pueden ejecutar varios procesos (programas) en paralelo, sino que dentro de un programa puede haber varios subprogramas (hilos) funcionando también de forma paralela. Se puede estudiar el uso de hilos en los programas con aplicaciones como eXtended Task Manager.

Un ejemplo de proceso con múltiples hilos podría ser un servidor FTP, donde se crearía un nuevo hilo por cada conexión aceptada, de esta forma puede manejar varias conexiones a la vez. Otro ejemplo de aplicación multihilo puede ser MS Word, donde hay un hilo que comprueba la ortografía, otro atento a los menús, otro contando las palabras escritas, etc.

10.1 Multihilo en LabVIEW

En un lenguaje basado en texto donde las instrucciones se ejecutan secuencialmente, es relativamente complejo realizar programas con múltiples hilos. Requiere cierto esfuerzo de imaginación pensar en esas líneas de código ejecutándose en paralelo cuando se están viendo escritas de forma secuencial. Además en muchos de estos lenguajes hay que indicar explícitamente cuándo empieza y acaba un hilo.

En LabVIEW no es necesario indicar nada para trabajar con múltiples hilos dentro de un programa, el propio LabVIEW se encargará de su gestión. Se dice que la programación multihilo es inherente a LabVIEW.

Para habilitar la opción de ejecución multihilo en las versiones anteriores a LabVIEW 8.20 (y siempre que el sistema operativo lo soportase) debía estar activada la casilla *Run with multiple threads* en el menú *Tools > Options > Performance and Disk*.

Como se ha estudiado en temas anteriores, un programa de LabVIEW se compone de su Panel Frontal, su Diagrama de Bloques, un espacio de memoria para los datos y el código compilado para la plataforma donde LabVIEW se está ejecutando. Además también contiene información sobre otros recursos que puede necesitar para ejecutarse, como ficheros dll, subVIs, etc. Cuando se abre un VI se cargan en memoria su espacio de datos y el código compilado, tanto el suyo como el de sus subVIs. El Panel Frontal y el Diagrama de Bloques se cargan cuando es necesario.

Al ejecutar un VI, LabVIEW realiza varias tareas sobre él: comprobar la sintaxis, reagrupar el código en conjuntos de nodos (*clumps*) donde dentro de cada nodo tiene un orden fijo de ejecución y finalmente asignar el espacio de memoria y generar el código. Un paso clave es el de reagrupar el código porque ahí es donde se detectan los paralelismos; dentro de un *clump* no puede haber paralelismo, pero sí entre ellos. Como resultado LabVIEW tiene una serie de secciones de código con un espacio de memoria asignada, cuya ejecución debe ser planificada.

LabVIEW posee un sistema (o varios) que se encarga de la ejecución de un programa. Este sistema de ejecución tiene una lista de *clumps*. El sistema va comprobando la lista de *clumps*, si hay disponibles saca uno de ella, lo pasa a ejecución durante un tiempo y cuando acaba ese tiempo, si el *clump* no ha terminado su tarea, es devuelto a la cola. En el fondo este mecanismo no es más que una rotación de tareas, lo que se conoce como multitarea cooperativa (*cooperative multitasking*).

La primera versión de LabVIEW que introdujo la capacidad multiproceso fue la 5, en ella el sistema de ejecución tenía el control de un hilo del sistema operativo sobre el que iba conmutando las tareas. Desde la versión 7 el sistema de ejecución tiene cuatro hilos del sistema operativo sobre los que conmutar sus tareas (éste es el valor por defecto, pero puede cambiarse). Cuando se carga un VI, LabVIEW pide al sistema operativo que cree los hilos en los que será ejecutado si no estaban ya creados; mientras no se usan pasan a un estado dormido pero no se eliminan, permanecen en espera de más código para ejecutar. Los hilos de LabVIEW pueden comunicarse entre sí a través de una

memoria compartida o por intercambio de mensajes. Estos hilos son manejados por el planificador de hilos del sistema operativo, éste no lo hace de forma rotatoria como el sistema de ejecución de LabVIEW sino por prioridades (*preemptively multitasking*).

10.1.1 Sistemas de ejecución

En la explicación anterior se ha comentado el funcionamiento de un sistema de ejecución. Todos los VIs en ejecución compartirían los hilos asignados a ese sistema. A veces puede ser deseable que varios VIs se puedan ejecutar de forma independiente (sin compartir los hilos con otros VIs), para esto LabVIEW proporciona varios sistemas de ejecución.

Un VI puede asignarse a cualquiera de los sistemas disponibles: *user interface, standard, instrument I/O, data acquisition, other 1* y *other 2*; el nombre de estos sistemas es sólo una sugerencia, por ejemplo un VI de adquisición de datos puede ser asignado al sistema *instrument I/O* y viceversa.

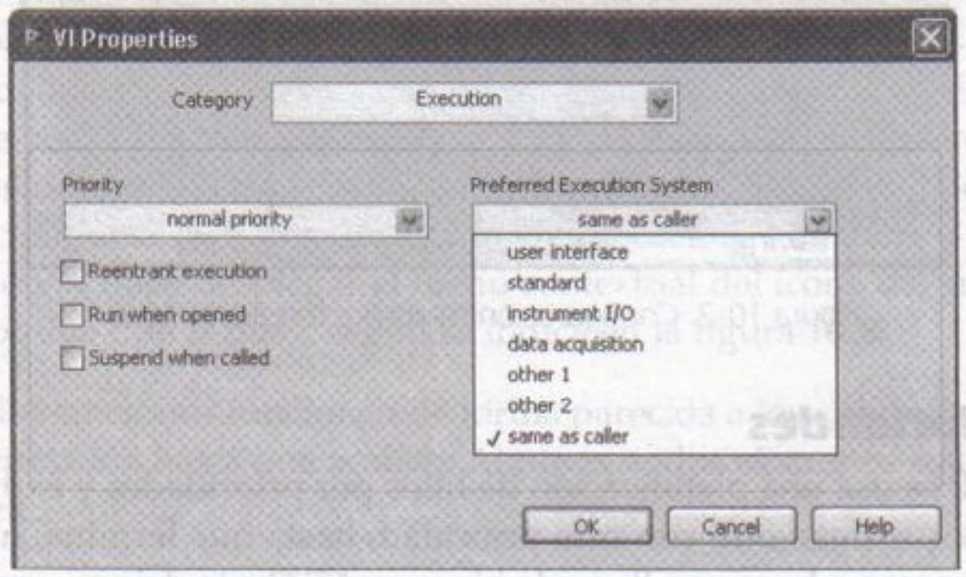


Figura 10-1. Propiedades y sistema de ejecución

El *user interface* es un sistema especial que se encarga de manejar lo que ocurre en el Panel Frontal. Cuando un VI de otro sistema de ejecución quiere cambiar algo de su Panel Frontal (por ejemplo cuando usa la propiedad *visible* de un control) éste pasa la ejecución al sistema *user interface*, por lo tanto es el sistema recomendado para los VIs con muchos nodos de propiedades que modifican aspectos del Panel Frontal. No conviene abusar de este subsistema, ya que el usuario podría percibir una ralentización en el panel que le haría pensar que el programa se ha bloqueado.

El número de hilos máximo es 40, por defecto cuatro por cada prioridad y por cada subsistema, aunque esta configuración puede variarse con *%directorio de instalación de LabVIEW%\vi.lib\Utility\Sysinfo.llb\threadconfig.vi*, ver figura 10-2.

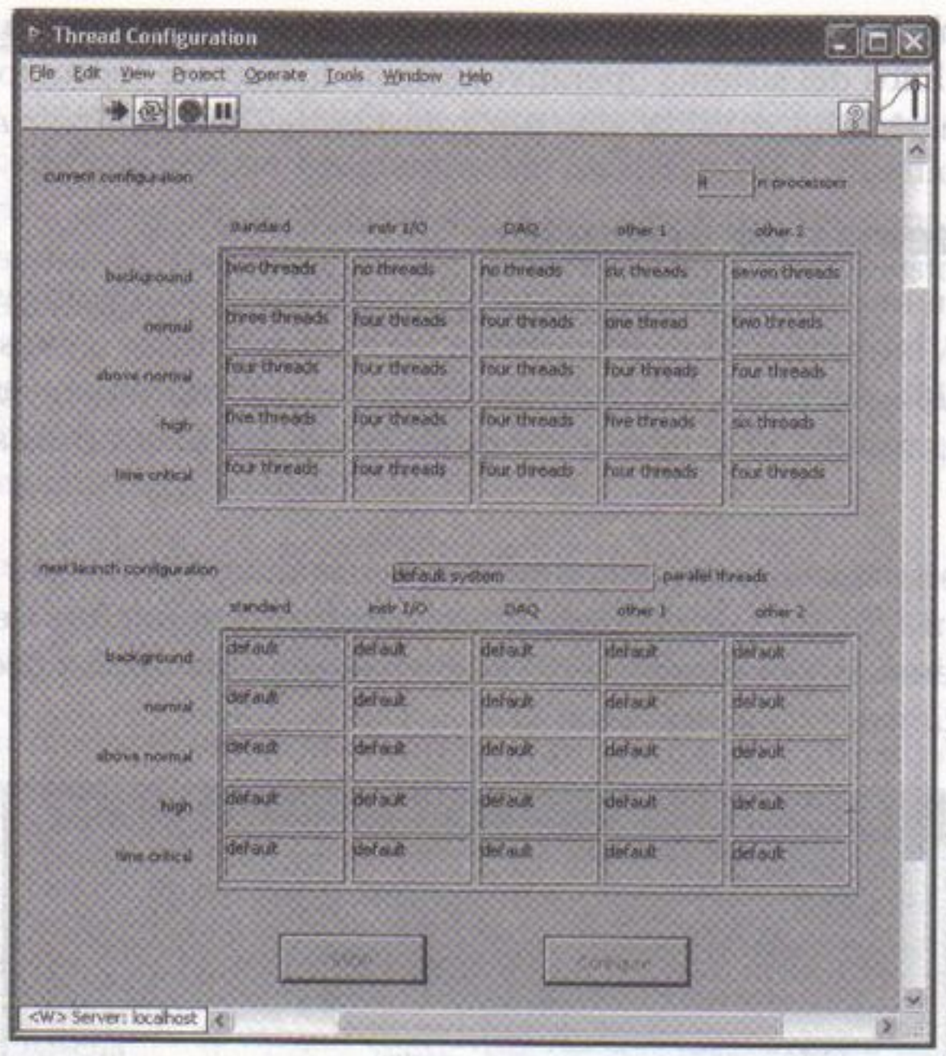

Thread Configuration
File Edit View Project Operate Tools Window Help
current configuration
n processors
standard
instr I/O
DAQ
other 1
other 2
background
normal
above normal
high
time critical
two threads
no threads
no threads
six threads
seven threads
three threads
four threads
four threads
one thread
two threads
four threads
four threads
four threads
four threads
four threads
five threads
four threads
four threads
five threads
six threads
four threads
four threads
four threads
four threads
four threads
next launch configuration
default system
parallel threads
standard
instr I/O
DAQ
other 1
other 2
background
normal
above normal
high
time critical
default
default
default

Tabla 1 - Configuración de ejemplo

	Standard	Other 1	Other 2
Background	2	6	7
Normal	3	1	2
High	5	5	6

En este ejemplo se usarán varios programas, cada uno de ellos está compuesto por dieciséis bucles WHILE en paralelo. Si se ejecutan dos programas asignados a los sistemas *standard* y *other 1* ambos de baja prioridad, se crearán 8 hilos de baja prioridad. Si se ejecuta un programa de prioridad normal en *other 1* y otro de alta prioridad en *other 2* se crearán seis hilos de alta prioridad y uno de prioridad media. Por último si se ejecuta un programa de baja prioridad en *standard*, otro de baja prioridad en *other 1* y otro de alta prioridad en *other 1* se crearán 5 hilos de alta prioridad y ocho de baja prioridad.

La prioridad llamada *subroutine* es especial. Cuando un VI con esa prioridad pasa a ejecución ocupará todo el tiempo de un sistema de ejecución hasta que se complete. Se debe tener cuidado con estos VIs ya que tienen algunas restricciones respecto a los VIs "normales" que conviene consultar en la ayuda. Los VIs marcados con esta prioridad tienen la opción de ser ignorados cuando el flujo de ejecución de un hilo llega a ellos si ya estaban ejecutándose, esto puede ser útil en aplicaciones de tiempo real; para acceder a esta opción se debe desplegar el menú contextual del icono del subVI y seleccionar *Skip Subroutine Call if Busy*, según se indica en la figura 10-3.

Los VIs llamados síncronos funcionan de forma parecida a los *subroutine*, ocupan todo el sistema de ejecución hasta que acaban. Algunos nodos síncronos son los CIN, los de propiedades, etc.

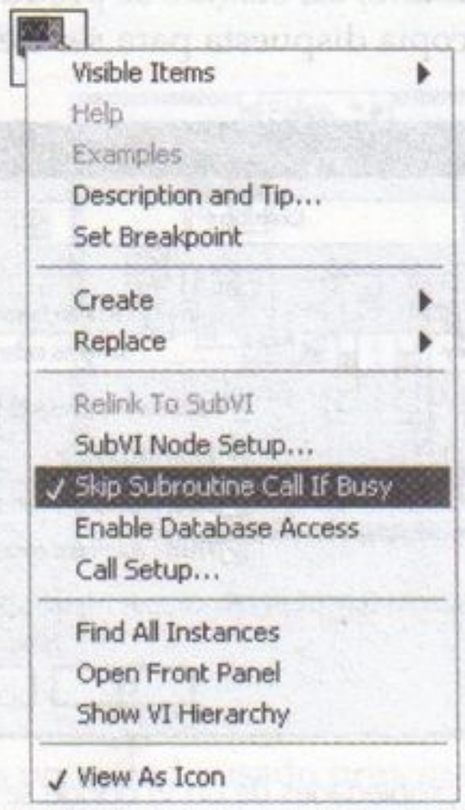


Figura 10-3. Menú contextual del icono de un subVI

Por último, a veces cambiar la prioridad de los VIs cuando no se comprenden los conceptos puede acabar con resultados inesperados o bloqueos, por lo que National Instruments aconseja dejar las prioridades con su valor por defecto y en su lugar usar las funciones de espera para priorizar los VIs.

10.1.3 VIs reentrantes

Como se ha dicho antes, cuando se abre un VI, incluso sin ejecutase se cargará en memoria una copia de su código y espacio de datos, así como del código y espacio de datos de todos los subVIs que contiene su jerarquía. Esto puede comprobarse abriendo un VI cualquiera y ejecutando el código de la figura 10-4.

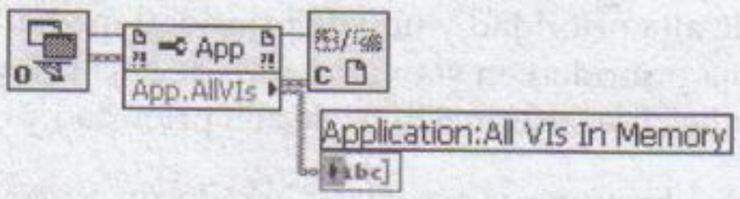


Figura 10-4. VIs cargados en memoria

Cuando se ejecuta un VI y éste llama a otro, LabVIEW buscará primero una referencia al subVI entre los que ya hay cargados en memoria y, si está disponible, lo ejecutará.

Este es el funcionamiento normal, sin embargo puede haber ocasiones en que no sea suficiente. Puede darse el caso que se desee hacer llamadas a un mismo VI de forma simultánea en varios hilos. En estas ocasiones, al llamar al subVI usado por otro programa, LabVIEW respondería que el subVI está ocupado y el VI que lo llama tendría que esperar a que acabe para poder hacer su propia llamada. Para solucionar esto puede crearse lo que se llama un VI reentrante que copiará su código y datos en memoria tantas veces como sea necesario, así cuando se produce una llamada no tendrá que esperar ya que habrá alguna copia dispuesta para ser ejecutada.

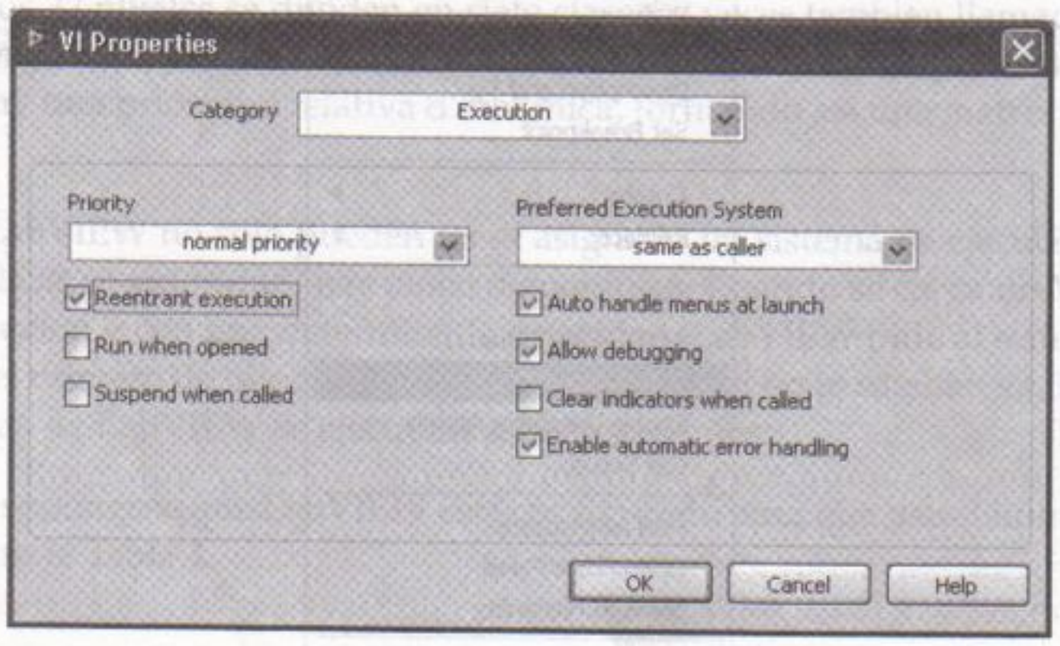


Figura 10-5. Ventana de las propiedades de un VI donde se activa la casilla 'reentrante'

Para indicar que un VI es reentrante se debe activar la casilla correspondiente en *File > VI Properties >Execution*, como muestra la figura 10-5. También hay que decir a usuarios de LabVIEW que usaban VIs reentrantes en versiones anteriores que desde la versión 8.0 se permiten herramientas de depuración dentro de estos programas.

Los VIs reentrantes junto con las llamadas dinámicas dan la posibilidad de realizar funciones recursivas.

Una función recursiva es aquella definida en términos de sí misma, es decir, una función que en su interior se llama a sí misma. Esto implica que habrá una serie de llamadas a una misma función, por cada llamada debe haber una copia de la función, o por lo menos de su contexto, en memoria; de aquí se deduce que debe haber algún momento en el que la auto-invocación se detenga, de lo contrario el programa se expandiría sin fin en la memoria.

Las funciones recursivas tienen su equivalente en forma de bucles. Estos son más eficientes que las funciones recursivas porque consumen menos memoria y tiempo. En general es recomendable realizar la programación por medio de bucles, sólo se debería usar la recursividad cuando con ella se consiga una gran simplificación en la complejidad de la solución (simplificando un problema en subproblemas del mismo tipo).

Para crear un VI recursivo hay que abrir una referencia al mismo VI y como tipo de especificador se debe seleccionar la clase, también, del mismo VI (para esto se crea una constante en el terminal *type specifier VI Refnum* de *Open VI Reference*, se presiona con el botón derecho sobre ella y se elige *Select VI Server Class > Browse*). Si se consulta la ayuda se podrá comprobar que en el campo de opciones se debe indicar el valor ocho. La llamada al VI puede realizarse por medio de *Call by Referente Node*.

En la figura 10-6 se puede apreciar un programa que calcula el factorial de un número por medio de la recursividad.

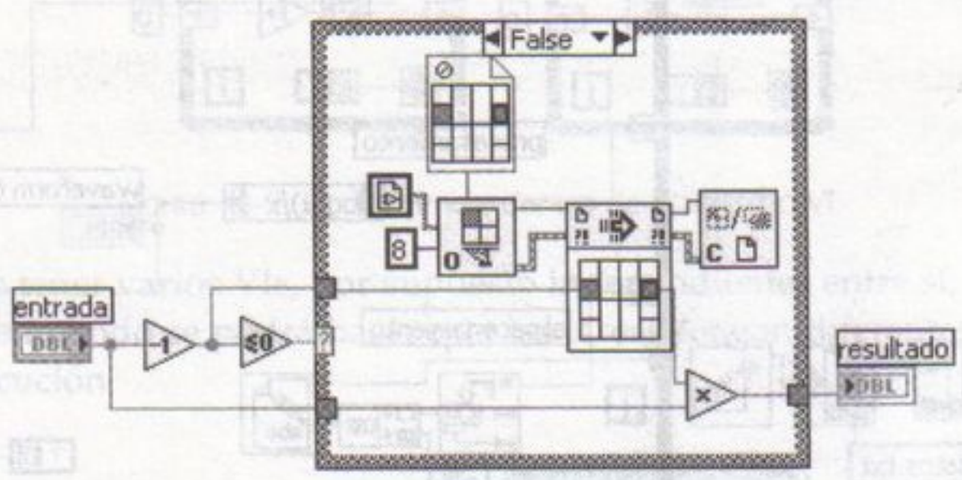


Figura 10-6. VI para el cálculo del factorial utilizando recursividad

10.1.4 Pipeline

El *pipeline* o segmentación es un método usado principalmente en las arquitecturas de microprocesadores y sistemas digitales para aumentar el rendimiento del mismo.

Consiste en descomponer las tareas en varias etapas, éstas funcionarán de forma simultánea, además los resultados de una etapa serán la entrada de la siguiente.

La figura 10-7 muestra de forma esquemática este concepto. En ella se ve una segmentación de cuatro etapas: Adquisición, ProcesadoA, ProcesadoB y Almacenamiento.

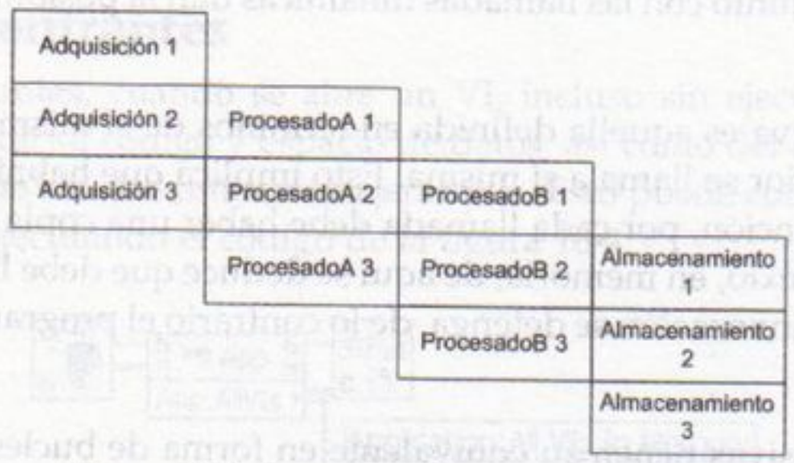


Figura 10-7. Segmentación de cuatro etapas

En LabVIEW se suele usar la segmentación cuando se trabaja con aplicaciones de tiempo real o programando FPGAs. La estructura es realmente sencilla, se basa en un bucle WHILE y varios *Shift Register* usados para que las tareas se pasen los datos entre ellas. En la figura 10-8 se muestra un ejemplo en el que la tarea se ha dividido en tres etapas: generación de datos, procesamiento y almacenamiento. La etapa de generación pasará los datos a través de un *shift register* para que, en la siguiente iteración del bucle sean procesados. Mientras la segunda etapa procesa el dato adquirido en la primera iteración, la primera etapa puede adquirir un nuevo dato; la etapa de procesamiento hará lo propio con la de almacenamiento.

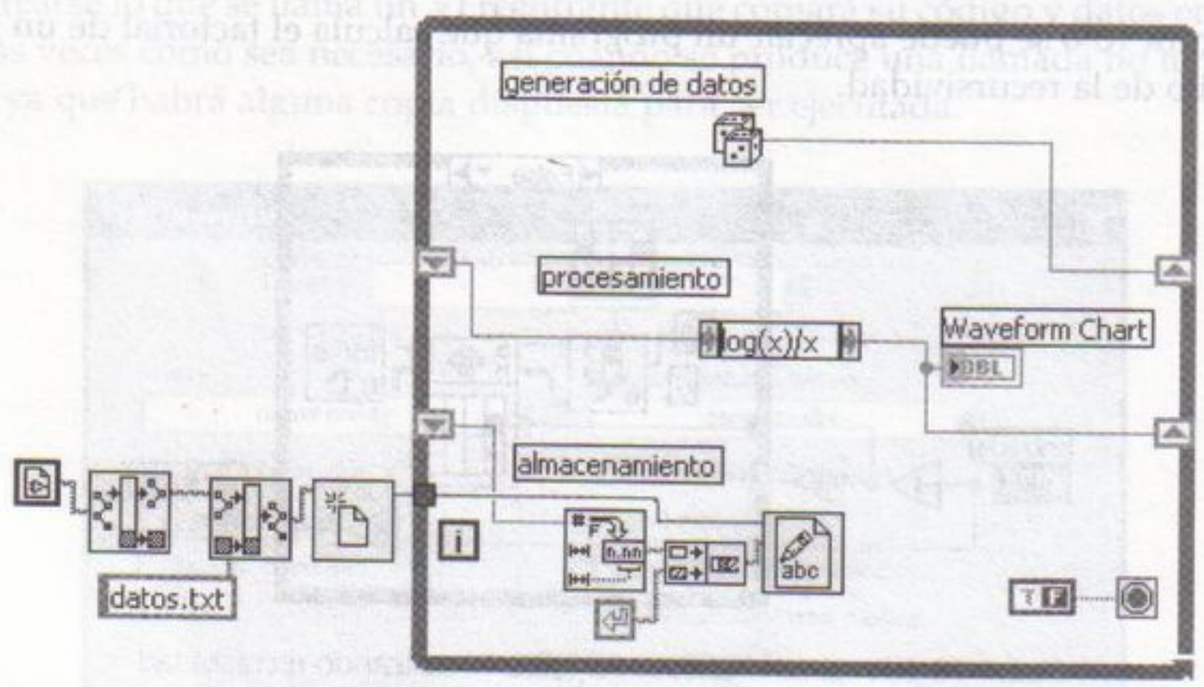


Figura 10-8. Ejemplo de segmentación con tres etapas

Los datos obtenidos para el ejemplo de la figura 10-8 pueden ser:

```
0,000000
-Inf
-26,827066
-0,351195
-0,778569
-0,425704
-0,541356
```

Evidentemente en la primera iteración del bucle se almacenará un dato que es inválido, en general habrá tantos datos inválidos como número de etapas menos uno. También aumentará el tiempo de latencia. Por otra parte la segmentación no es apropiada si las tareas son muy sencillas o una de las subtareas tarda mucho más que las demás (y no puede dividirse más).

10.1.5 Paralelismo

Hasta ahora se ha hablado de VIs que pueden ejecutarse en paralelo y de *clumps* que también pueden hacerlo. Esto significa que el paralelismo puede darse tanto entre VIs como entre porciones de código dentro de un mismo VI.

Dentro del mismo VI se ejecutarán de forma concurrente las partes de código que no sean dependientes entre sí. El ejemplo más claro son dos bucles WHILE en paralelo. La principal ventaja de este método es la sencillez. Una de las desventajas es que no se pueden modificar las prioridades (excepto con nodos de espera), pero en las nuevas versiones de LabVIEW esto se ha paliado con los TIMED LOOP, que sitúan sus prioridades entre *Time Critical* y *High Priority*.

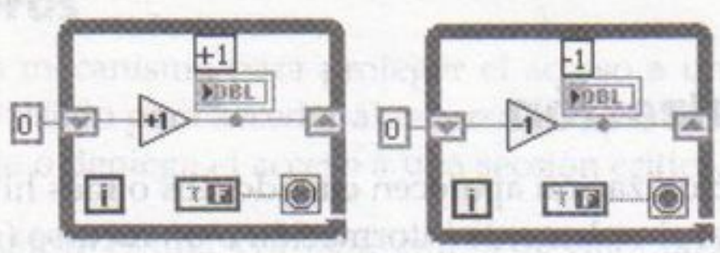


Figura 10-9. Concurrencia dentro de un mismo VI

Otro método es tener varios VIs, por supuesto independientes entre sí, funcionando a la vez. Con este método se podrá hacer que las tareas tengan diferentes prioridades y sistemas de ejecución.

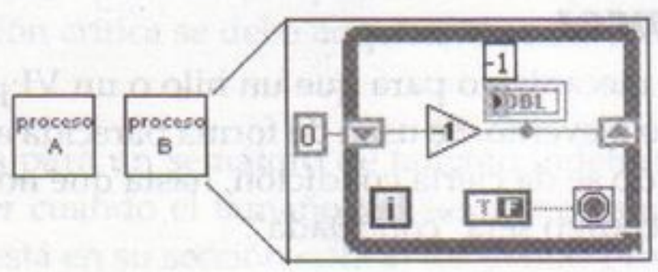


Figura 10-10. Concurrencia entre varios VIs

Otra forma sería usar VI Server para abrir, ejecutar y cerrar VIs. Este método tiene la ventaja de que la creación y finalización de las tareas se puede realizar en cualquier momento y de forma abrupta. En la figura 10-11 se muestran dos procesos que son lanzados antes de los bucles WHILE y son finalizados después, el código que aparece en las ventanas a ambos lados de los bucles es el correspondiente a los subVIs que lanzan y finalizan tareas.

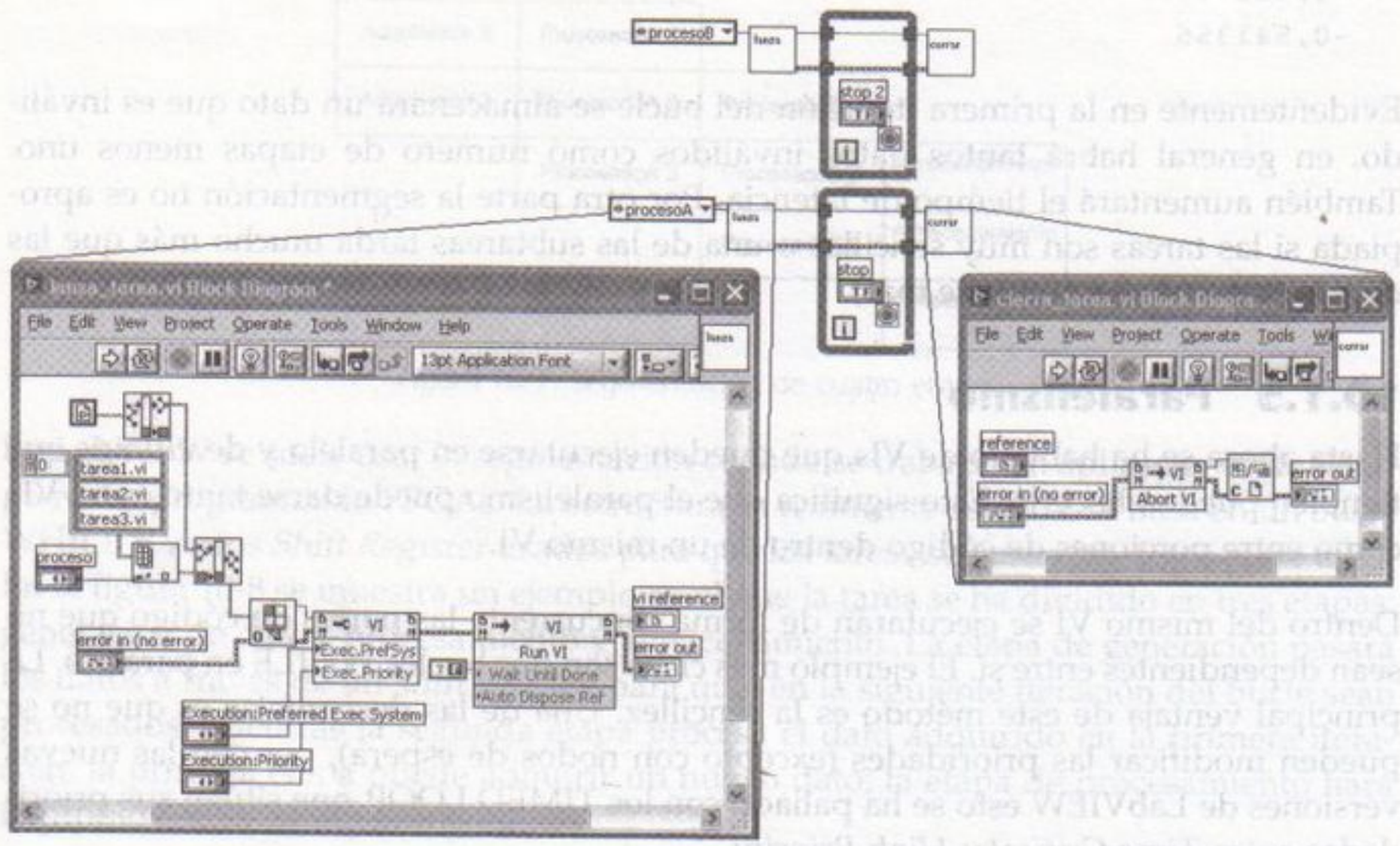


Figura 10-11. Concurrencia usando el VI Server

10.2 Sincronización

Los problemas de sincronización aparecen cuando dos o más hilos intentan compartir algo, ese algo puede ser simplemente información o un recurso (una variable, un fichero, una impresora...).

Durante esta sección se estudiarán los mecanismos que ofrece LabVIEW para sincronizar tareas. Se encuentran todos bajo el menú *Functions > Programming > Synchronization.*

10.2.1 Occurrences

Las *occurrences* son un mecanismo para que un hilo o un VI pueda indicar a otro que ha ocurrido cierto suceso o evento. Se usan de forma parecida a los eventos, sirven para ejecutar un código cuando se da cierta condición, hasta que no sea cierta esa condición la tarea que contiene el código será "congelada".

En este ejemplo lo que se desea es controlar desde una tarea la ejecución de la otra. Para ello habrá un hilo que generará una señal, pero esta acción sólo debe llevarse a cabo cuando en el segundo hilo se pulse un botón.

En el código mostrado en la figura 10-12 se crea una referencia a una *occurrence* que será pasada a todos los hilos del programa; en el hilo inferior se activa el evento cuando se presiona el botón *generar*. En ese momento el hilo superior, que estaba "congelado", se reactivará y generará una señal con la forma de onda indicada.

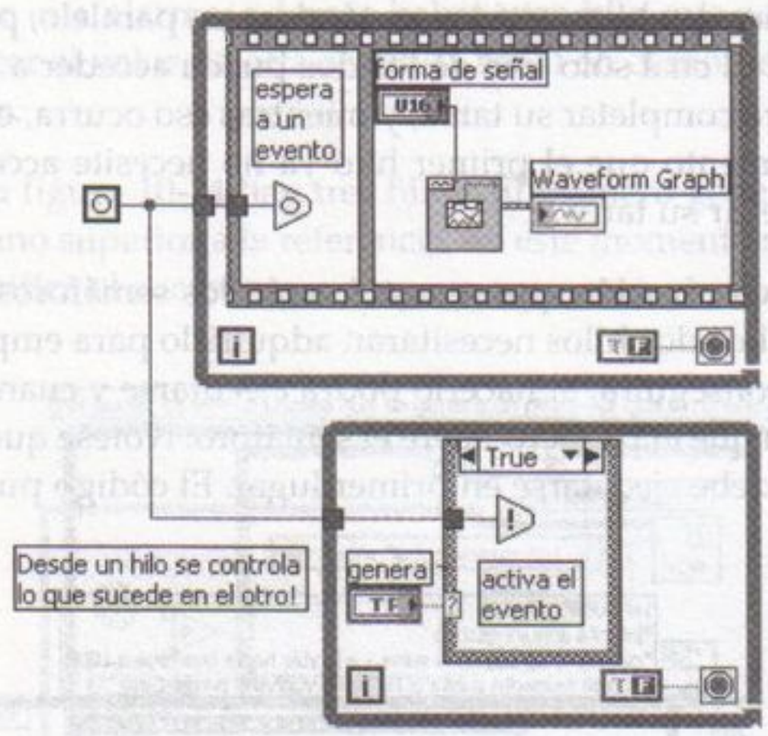


Figura 10-12. Ejemplo de alarma con *occurrence*

10.2.2 Semáforos

Los semáforos son un mecanismo para proteger el acceso a un determinado recurso compartido. El código usado para acceder al recurso compartido se llama sección crítica, el semáforo permite o deniega el acceso a una sección crítica.

Según Dijkstra un semáforo es una «*variable entera de valor no negativo, sobre las que se definen las operaciones P y V*». Las operaciones P y V vienen del holandés *Proberen*, probar y *Verhogen*, incrementar. El valor inicial del semáforo es la cantidad de unidades que se disponen del recurso. La operación P detiene la ejecución hasta que hay un recurso disponible, en ese momento lo adquiere; la operación V hace que un recurso que había sido adquirido quede liberado. Las operaciones P y V no pueden ejecutarse concurrentemente ni pueden ser interrumpidas, se dice que son atómicas. Cuando se quiera ejecutar una sección crítica se debe adquirir el semáforo y cuando se termina se libera.

La definición anterior es para un semáforo de tamaño indeterminado, sin embargo es más fácil de comprender cuando el tamaño del semáforo es uno: si algún proceso ha adquirido el semáforo (está en su sección crítica) los demás procesos tendrán que esperar a que el primero lo libere para que ellos puedan adquirirlo. Es lo que se conoce

como paso de testigo. Cuando el tamaño del semáforo es uno también se llaman *mutex* (exclusión mutua).

El siguiente es un ejemplo que se basa en el concepto de "paso de testigo". Dentro del programa habrá una parte que, a partir de un control numérico, le irá sumando uno hasta que sea mayor o igual que 1.000; otra parte del programa irá restando uno al mismo control hasta que el número sea menor o igual que -1.000.

En este programa se tiene un dato (el control numérico) que comparten dos hilos a través de variables locales, los hilos están ejecutándose en paralelo, por lo tanto se necesita un mecanismo por el cual sólo uno de los dos pueda acceder a la variable el tiempo que sea necesario para completar su tarea, y mientras eso ocurra, el otro hilo debe estar esperando; en el momento que el primer hilo ya no necesite acceder a la variable, el segundo podrá empezar su tarea.

El mecanismo de sincronización que se usará serán los semáforos. Se creará un semáforo de tamaño uno. Los dos hilos necesitarán adquirirlo para empezar a trabajar, pero sólo uno de ellos lo conseguirá, al hacerlo podrá ejecutarse y cuando acabe lo liberará; el otro hilo esperará a que el primero libere el semáforo. Nótese que en ningún momento se indica qué hilo debe ejecutarse en primer lugar. El código puede verse en la figura 10-13.

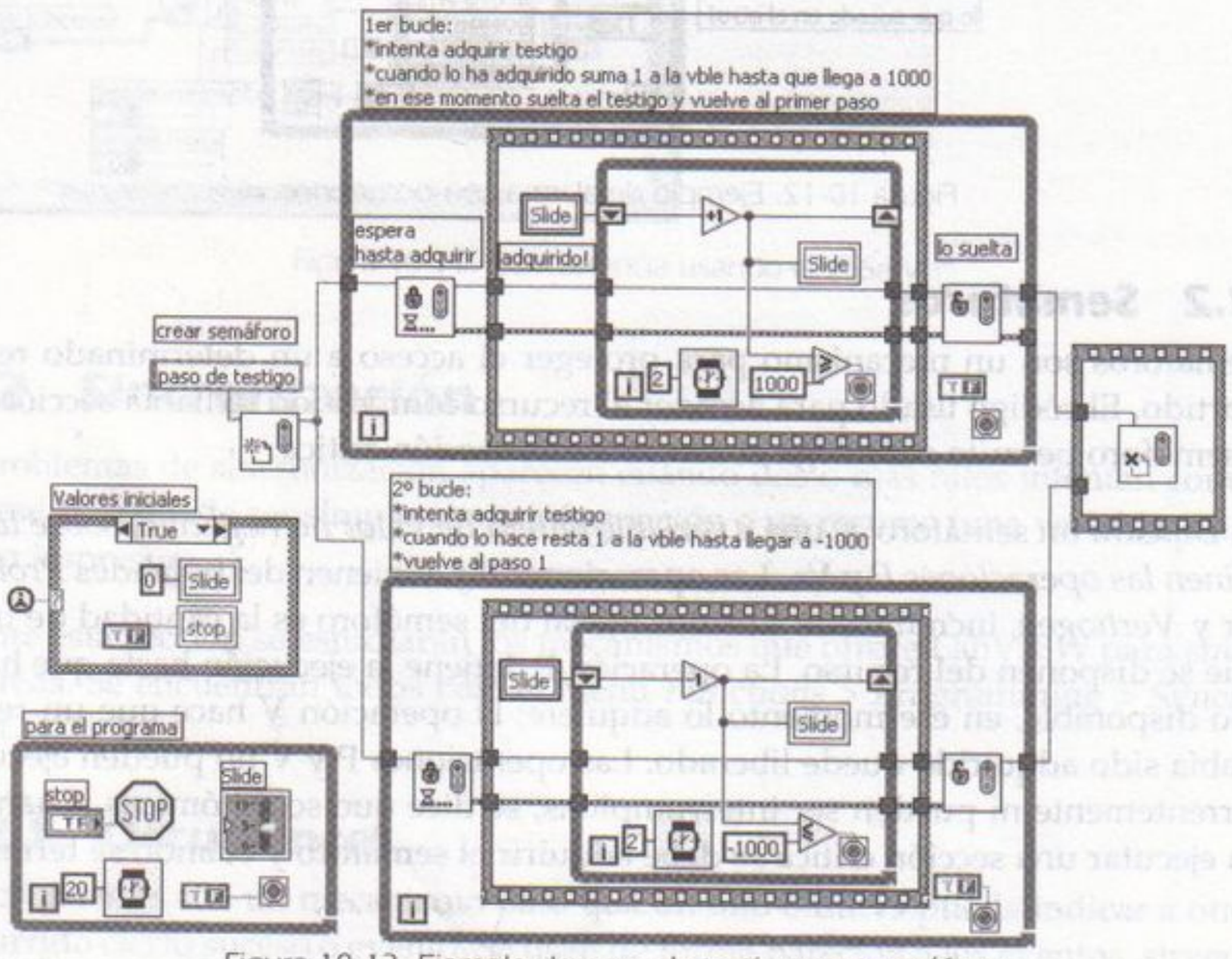


Figura 10-13. Ejemplo de paso de testigo con semáforos

10.2.3 Notificaciones

Son muy parecidos a las *occurrences* en el sentido de que proveen un mecanismo de espera hasta que ocurra un suceso o evento. Las diferencias entre *notifier* y *occurrences* es que las primeras pueden cancelarse y además pueden enviarse mensajes junto con la señal de evento.

En este ejemplo se creará un sistema de alarma. Se tienen una serie de datos que se van generando uno a uno, cuando alguno de estos datos sea mayor que una cantidad prefijada se debe activar una alarma; además habrá otro hilo que adquiere una segunda señal, se desea conocer el valor de esta segunda señal en el momento en que la primera sobrepasó el límite.

En el programa de la figura 10-14 hay tres hilos. El primero generará números aleatorios hasta que haya uno superior a la referencia, en este momento se construye un mensaje de aviso y se notifica el suceso.

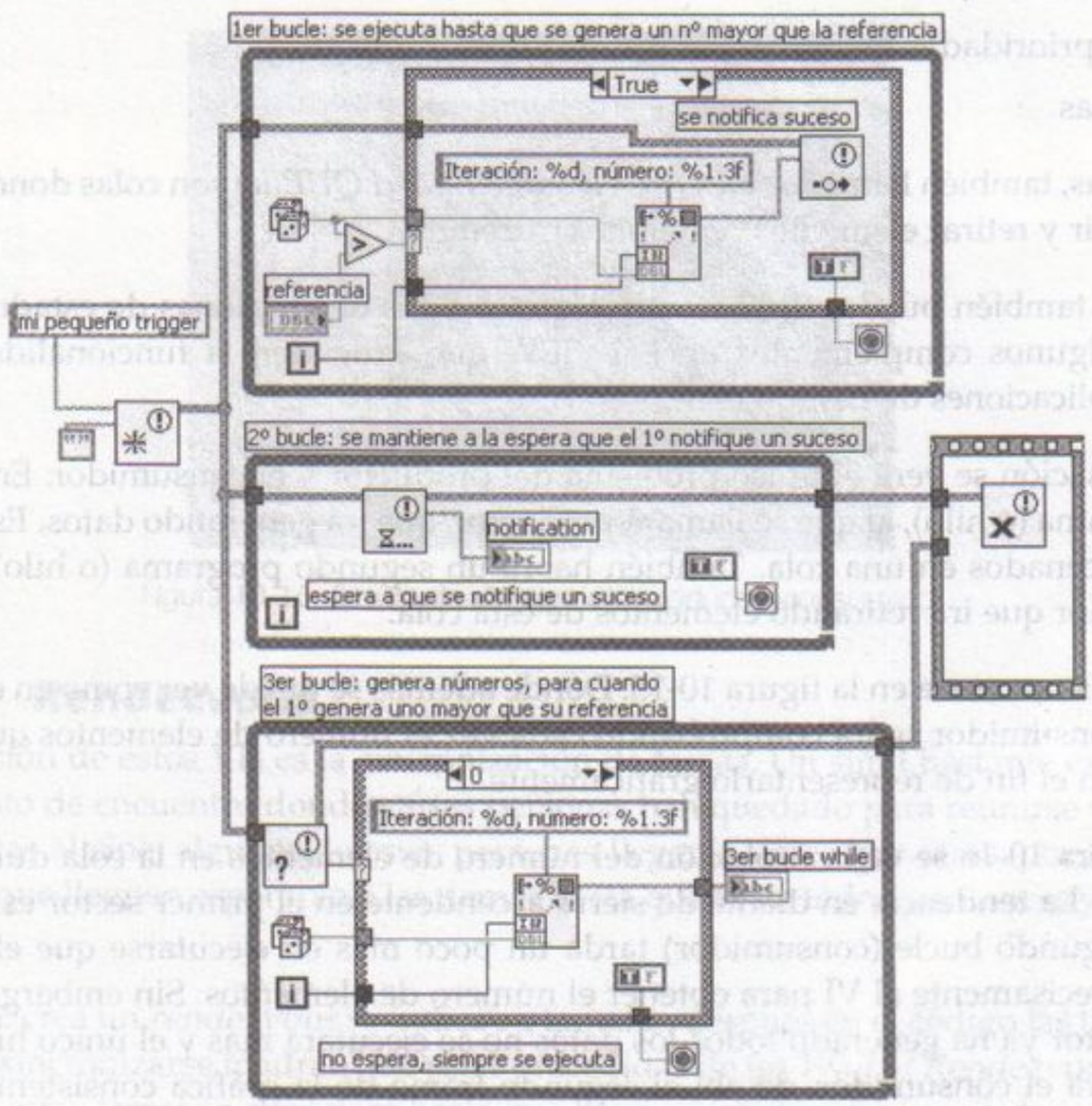


Figura 10-14. Ejemplo de alarma basado en notificaciones

El último hilo se ejecuta en paralelo al primero y, al igual que éste, genera números aleatorios. Cuando el primer hilo notifica el suceso, este último hilo también construye un mensaje con el valor del número aleatorio generado y el número de iteración del bucle.

El segundo ilustra la espera de sucesos y el intercambio de información entre los nodos *notifier*, cuando sucede el evento muestra el mensaje creado por el primer bucle.

Para comprender mejor el funcionamiento de este programa se aconseja ejecutarlo con la opción *Highlight Execution* activada. También se puede encontrar otro ejemplo abriendo la plantilla *Master/Slave*.

10.2.4 Colas

Las colas es un mecanismo de compartir información. Normalmente las colas se comportan como memorias FIFO, es decir, el primer elemento que se almacena es el primero que se sacará, no obstante en LabVIEW también pueden añadirse y sacar elementos en el orden contrario, como una pila de memoria, en este caso funcionaría como una LIFO.

Hay varios tipos de colas:

- Sin prioridad
- Con prioridad
- Bicolas

Las bicolas, también llamadas DEQUE (*Double Ended QUEue*) son colas donde se pueden añadir y retirar elementos por ambos extremos.

Las colas también pueden aplicarse en algunos tipos de máquinas de estado, además existen algunos complementos de LabVIEW que extienden la funcionalidad de las colas a aplicaciones de tiempo real.

A continuación se verá el típico problema del productor y el consumidor. En él habrá un programa (o hilo), al que se llamará productor, que irá generando datos. Estos datos son almacenados en una cola. También habrá un segundo programa (o hilo) llamado consumidor que irá retirando elementos de esta cola.

El código se muestra en la figura 10-15. Donde además se puede ver como en el programa del consumidor se irá comprobando cada vez el número de elementos que hay en la cola con el fin de representarlo gráficamente.

En la figura 10-16 se ve la evolución del número de elementos en la cola durante una ejecución. La tendencia en diente de sierra ascendente en el primer sector es debido a que el segundo bucle (consumidor) tarda un poco más en ejecutarse que el primero debido precisamente al VI para obtener el número de elementos. Sin embargo cuando el productor ya ha generado todos los datos no se ejecutará más y el único hilo en ejecución será el consumidor, de ahí el segundo tramo de la gráfica consistente en una recta con pendiente negativa.

También es importante conocer que el menú inicial de LabVIEW en el que hay diversas plantillas de programas existen dos ejemplos del programa del productor-consumidor con colas.

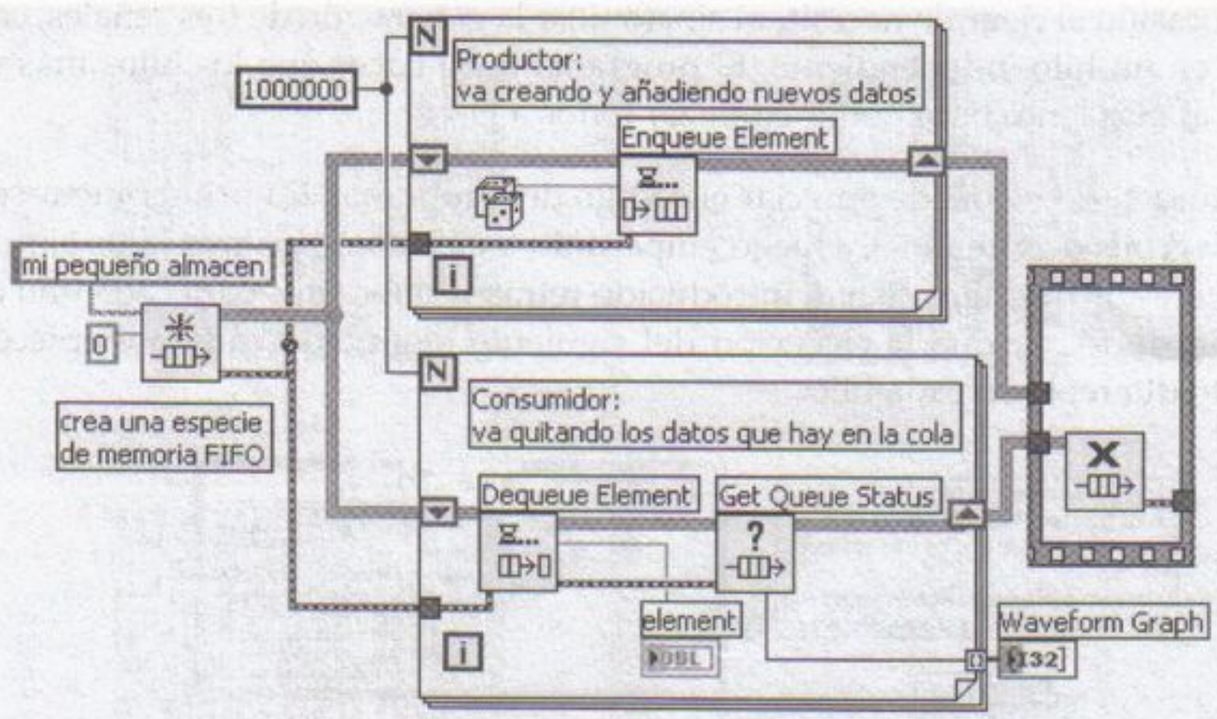


Figura 10-15. Ejemplo productor-consumidor

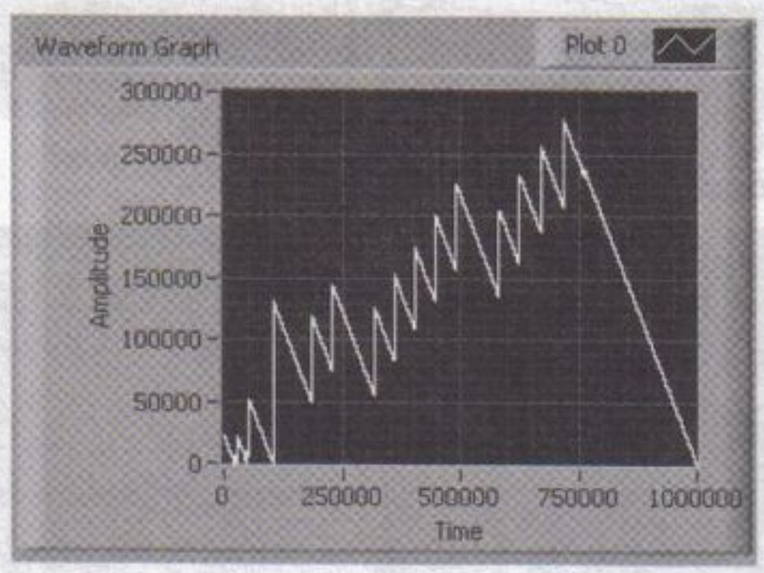


Figura 10-16. Resultado del ejemplo productor-consumidor

10.2.5 Rendezvous

La aplicación de estos VIs es la sincronización de tareas. Un símil bastante explicativo es un punto de encuentro donde varias personas han quedado para reunirse y luego ir todas juntas al cine, algunas de estas personas llegarán antes que otras, por lo que las primeras que lleguen esperarán a las demás hasta que estén todos y entonces ya podrán ir al cine.

Cuando se crea un *rendezvous* se indica su tamaño, después en el código las tareas que necesiten sincronizarse tendrán que estar precedidas de un *Wait at Rendezvous*. La ejecución de estas tareas se congelará hasta que haya esperado un número de tareas igual al número indicado al crear el *rendezvous*, en ese momento todas las tareas empezarán simultáneamente.

El tamaño del *rendezvous* puede cambiarse durante la ejecución del programa.

En esta ocasión el ejemplo necesitará sincronizar la generación de tres señales, cada una de ellas en un hilo independiente. El programa debe hacer que los hilos más rápidos esperen al más lento para poder empezar todos a la vez.

En la figura 10-17 se puede apreciar el código del programa. En primer lugar se creará un *rendezvous* de tamaño 3 y luego empezarán a ejecutarse los tres hilos simultáneamente. En estos tres hilos se han introducido retrasos diferentes para cada uno de ellos con el fin de forzar que la ejecución del siguiente bloque de código empiece en un momento diferente en cada hilo.

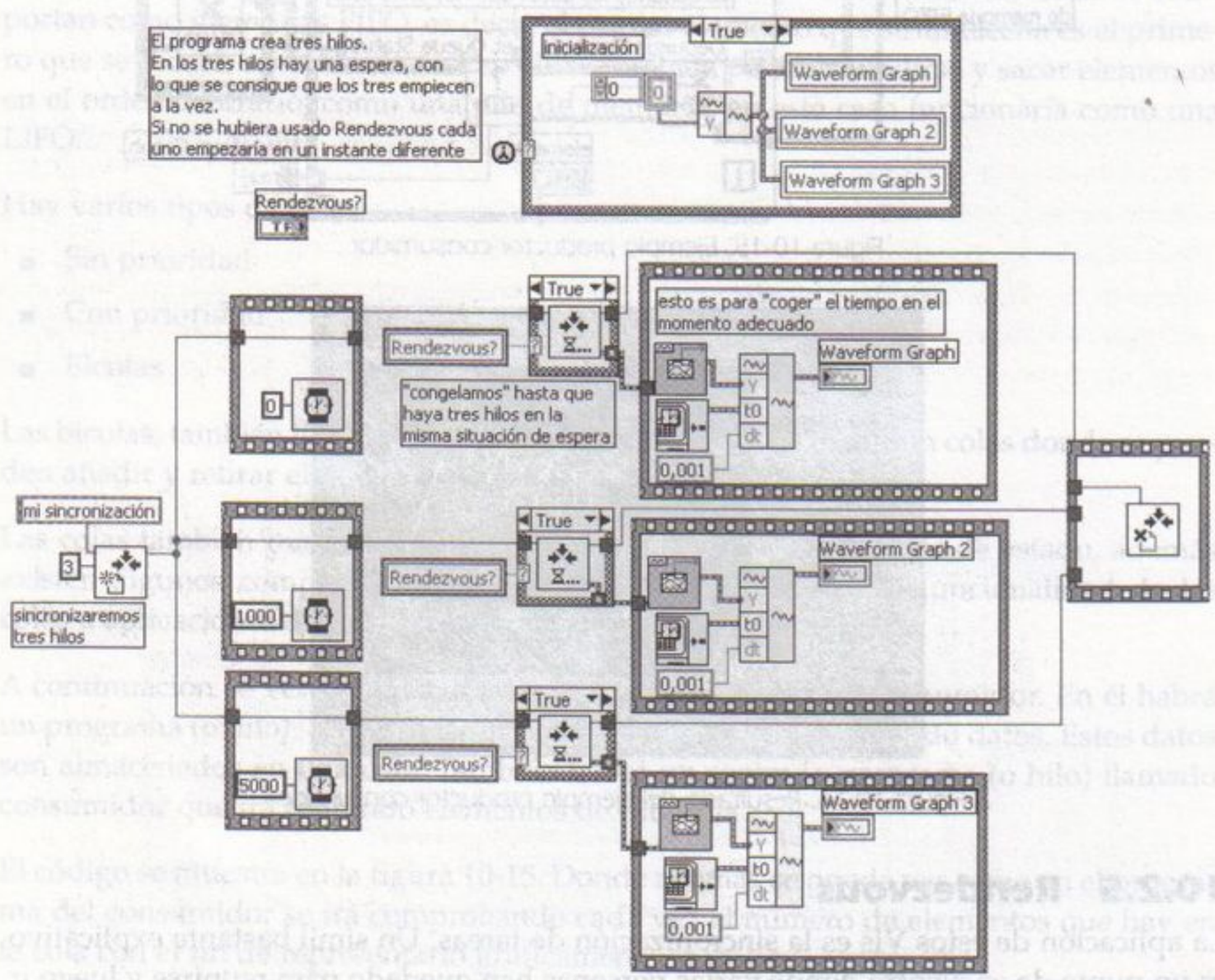


Figura 10-17. Ejemplo Rendezvous

Si no se activan los *rendezvous* se obtiene la gráfica de la figura 10-18. En ella se ven las tres señales, además si se mira más detalladamente se observa que el eje horizontal de las gráficas empieza en momentos diferentes, además la diferencia entre el comienzo de una gráfica y otra es justamente el retraso introducido en el código (un segundo entre la primera y la segunda y cinco segundos entre la primera y la tercera). Sin embargo si se activan las esperas *rendezvous*, la generación de las señales no empezará hasta que haya tres hilos esperando, por lo tanto las tres señales generadas empezarán exactamente en el mismo instante, el resultado se puede ver en las gráficas de la figura 10-19.

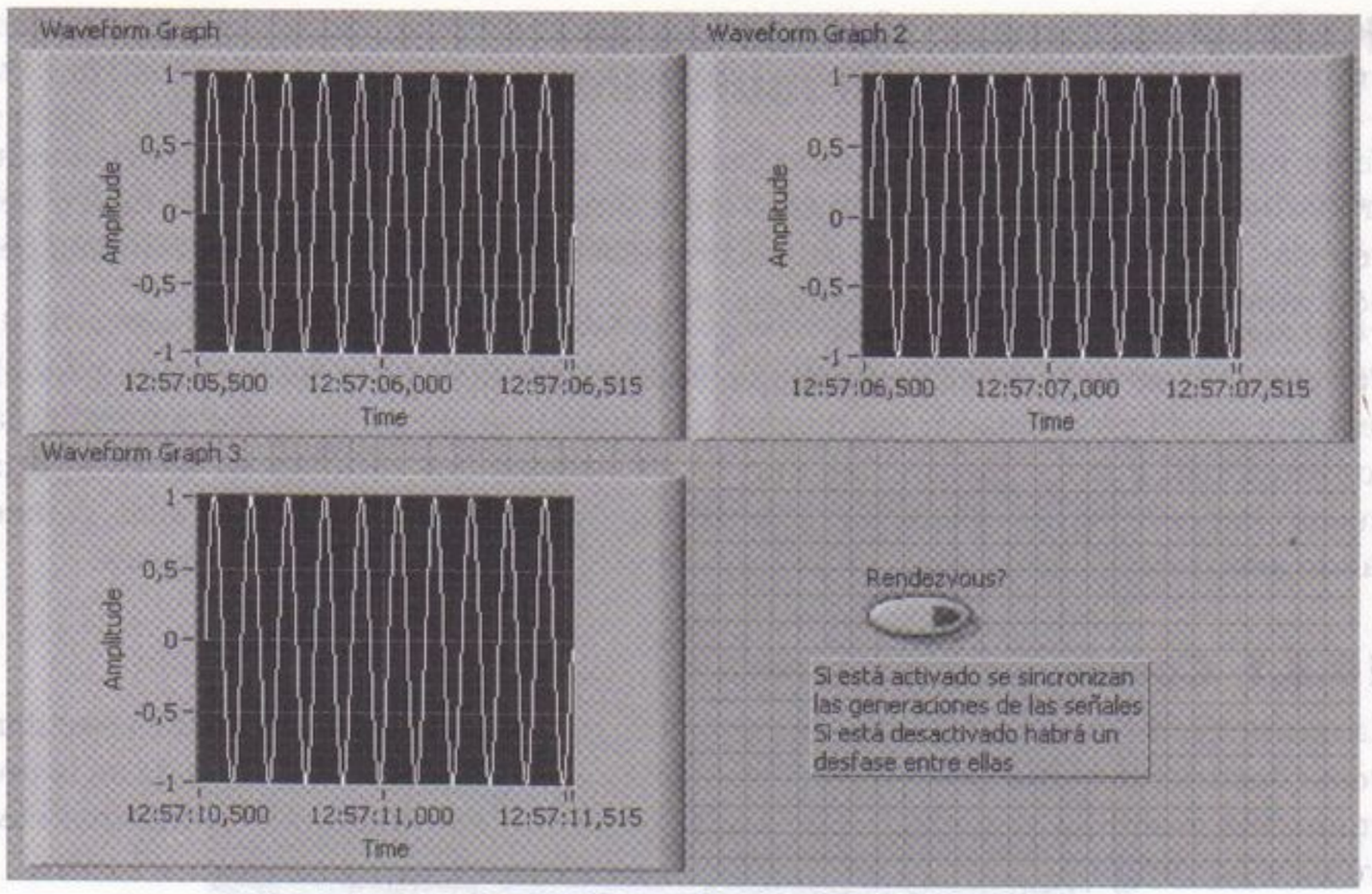


Figura 10-18. Resultado sin activar el Rendezvous

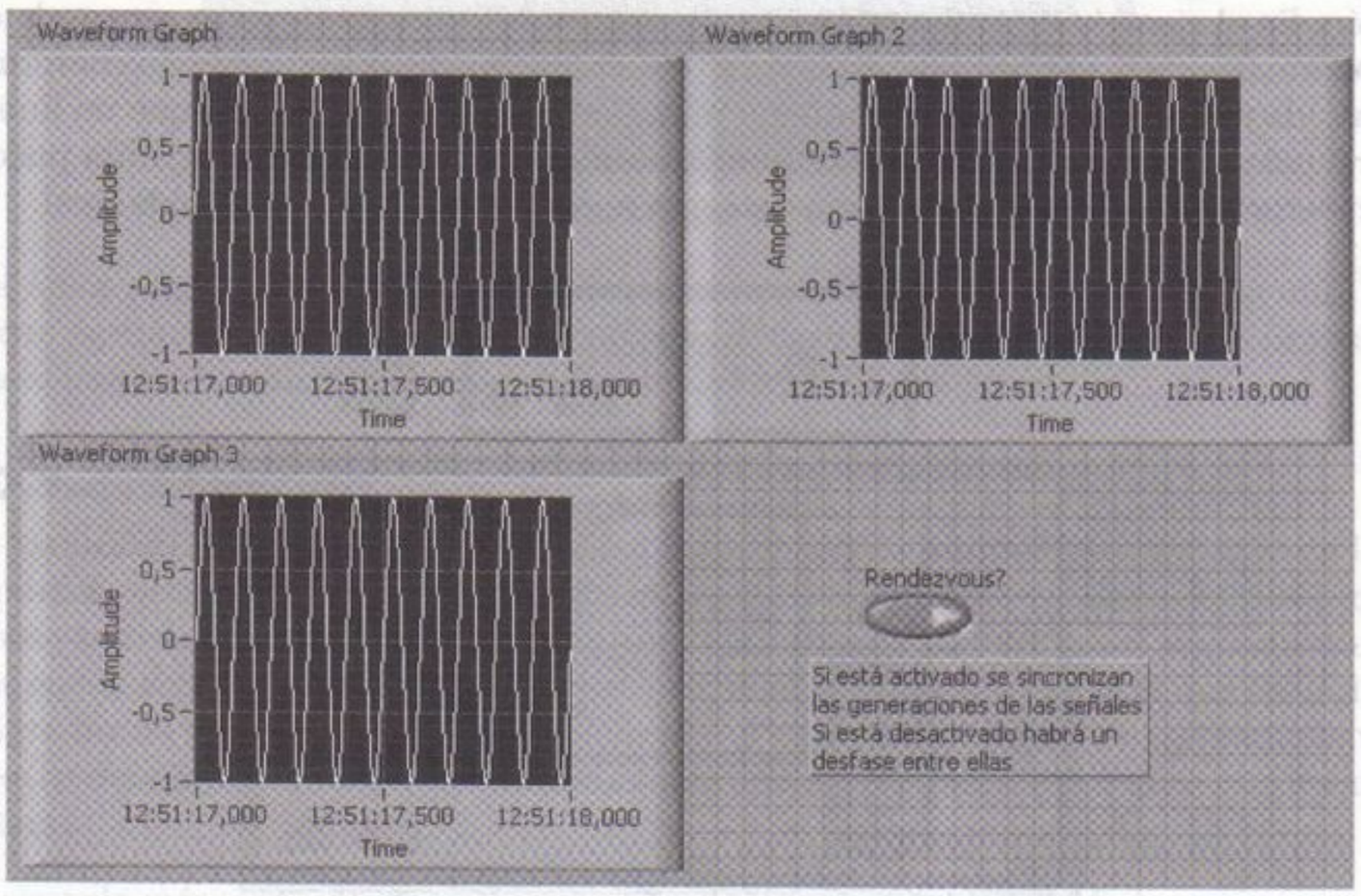


Figura 10-19. Resultado activando el Rendezvous

10.3 Problemas típicos de la programación multihilo

Tener múltiples hilos en un programa puede ser una buena idea porque aporta modularidad y la sensación de multiproceso entre otras ventajas. Pero también tienen desventajas, como un aumento de la carga computacional debido a que necesitan una gestión, aumento del consumo de memoria, etc.

Por otro lado se debe tener presente que la programación de varias tareas en paralelo requiere que el programador sea mucho más cuidadoso que en una programación más convencional. Pueden presentarse nuevas situaciones que resulten en un comportamiento imprevisto del sistema. En esta sección se hablará de algunas de ellas, comentando anécdotas famosas provocadas por estos fallos y sus posibles soluciones.

10.3.1 Condición de carrera

La condición de carrera es un efecto que ocurre cuando se tienen varios hilos ejecutándose concurrentemente y éstos comparten datos. Este problema no sólo se da en la programación multihilo, también se puede encontrar en algunos diseños de sistemas electrónicos hardware.

Un factor determinante para producirse la condición de carrera es el tiempo de ejecución. En el ejemplo de la figura 10-20 se muestran dos hilos, el primero genera datos continuamente y además a la máxima velocidad que puede; el segundo hilo comprueba cada milisegundo el valor actual del dato generado, si éste es mayor que un límite prefijado (0,5 en la imagen) se escribe en un *string*. Obsérvese que la escritura puede provenir de dos fuentes: del mismo dato que se comprobó y de una variable local.

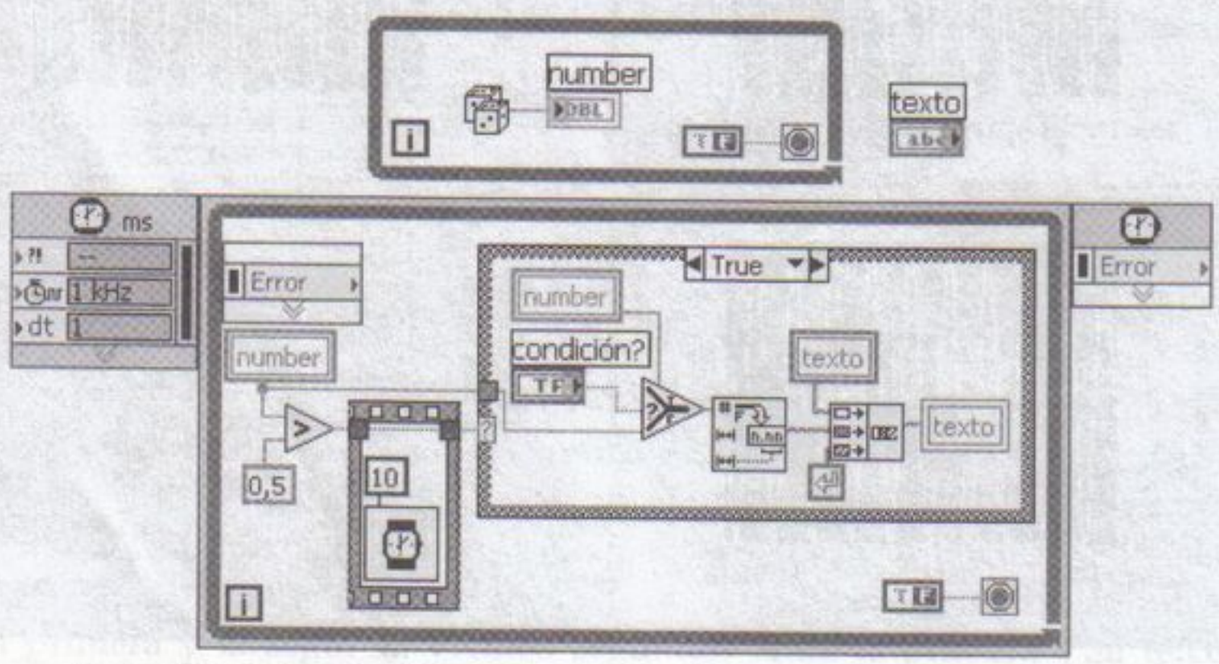


Figura 10-20. La condición de carrera: problema en programación multihilo

La condición de carrera se da cuando el dato que se almacena es el que proviene de la variable local situada dentro del CASE; en esta ocasión el tiempo que tarda el segundo hilo en pasar de la comprobación a la parte de escritura en el interior del CASE es sufi-

ciente para que el primer hilo genere otro dato, por lo que el valor que hay en la variable local dentro del CASE no es el mismo que el que se usó para evaluar la condición de dicho CASE.

Para comprobar el error se debe tener activo el botón *condición?* para seleccionar, como dato a mostrar, la variable local del interior del CASE.

Si no hubiese fallo sólo se almacenarían valores superiores a 0,5. Puede verse en la figura 10-21 como cuando se produce la condición de carrera se almacenan valores erróneos (menores a 0,5).

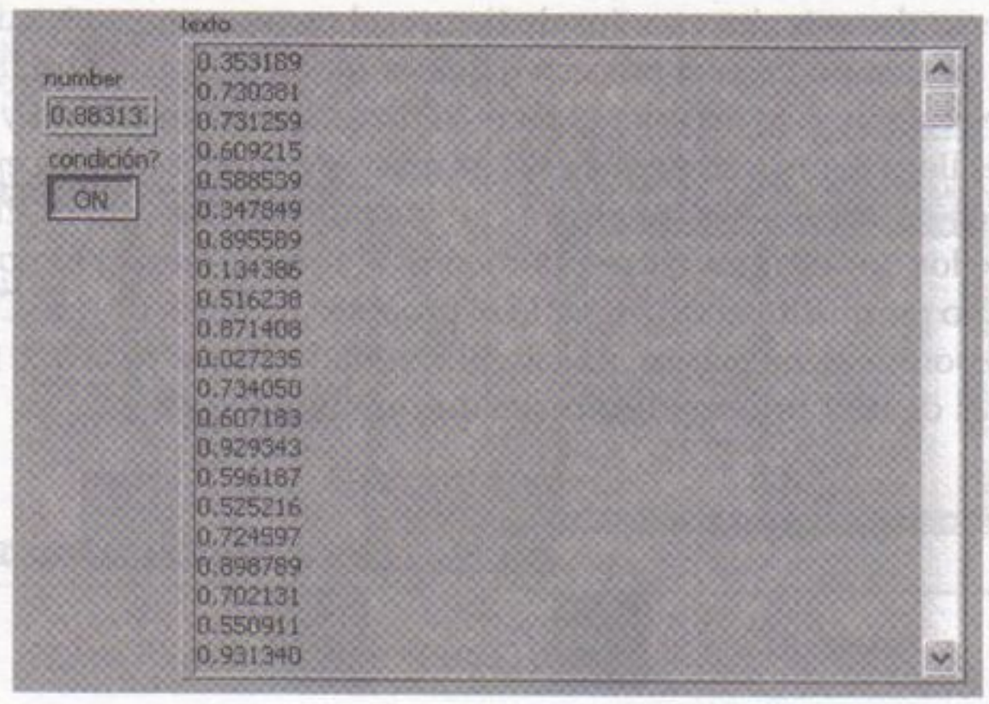


Figura 10-21. Resultado de la ejecución del ejemplo de condición de carrera

Los fallos producidos en la máquina de radioterapia Therac-25 que costaron la vida a varias personas fueron debidos a la condición de carrera. La solución para este problema es una buena sincronización, fundamentalmente con semáforos.

10.3.2 Inanición

La inanición o *starvation* es lo contrario a la inversión de prioridad (que se verá en la siguiente sección). Si hay un recurso que es accedido muy frecuentemente por dos procesos y uno de ellos tiene más prioridad que el otro, puede suceder que el segundo apenas pueda acceder al recurso. Si, por ejemplo, este segundo proceso es el que controla el interfaz de usuario, éste podría percibir que el sistema no responde o su rendimiento se ha deteriorado cuando en realidad es sólo uno de los procesos el que no puede ejecutarse.

El ejemplo más obvio de *starvation* es un bucle WHILE sin ningún VI de espera en su interior, al ejecutarse esto provoca que este bucle consuma la mayoría del tiempo de la CPU, quedando menos tiempo de procesado para otras tareas del mismo programa y ralentizando todo el sistema.

10.3.3 Inversión de prioridad

Este problema ocurre cuando dos hilos con diferente prioridad intentan acceder al mismo recurso. Si el que tiene la prioridad más baja obtiene primero un recurso y el que tiene más prioridad quiere acceder después a ese mismo recurso, debe esperarse a que el primero lo libere, de esta forma el que tenía menos prioridad ha hecho esperarse al de más prioridad, justo lo contrario a lo que debería haber ocurrido.

El esquema de la figura 10-22 muestra el efecto de la inversión de prioridad con varias tareas, en él hay tres procesos: A, B y C, ordenados de mayor a menor prioridad, también hay tres recursos: el uno, el dos y el tres. El primer proceso que comienza es el C, que necesita tener durante dos cuantos de tiempo el recurso uno; antes de que acabe el primer cuanto de tiempo llega el proceso B que, como tiene más prioridad, pasa inmediatamente a ejecución, éste necesita el recurso dos durante un cuanto y el tres durante otro, después llega el proceso A, que es el de más prioridad, éste para ejecutarse necesita durante dos cuantos el recurso tres y un cuanto el recurso uno. El proceso A se ejecuta durante los dos cuantos correspondientes al recurso tres, pero no puede adquirir el recurso uno porque lo tenía asignado el proceso C y antes de que C lo libere tiene que finalizar B porque tiene más prioridad. El resultado es que para que finalice el proceso A, que es el de más prioridad, necesita que antes acaben B y C.

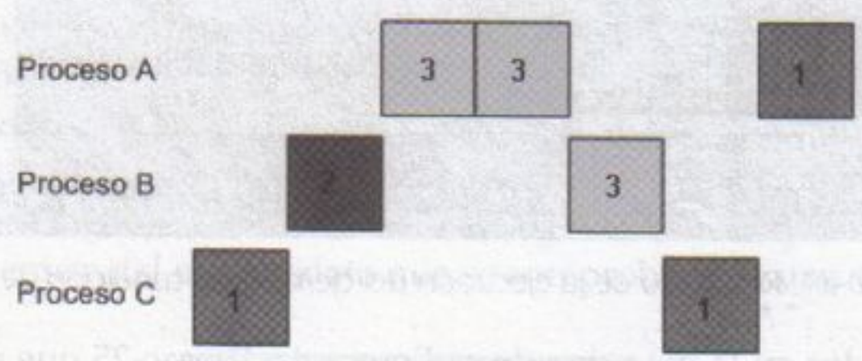


Figura 10.22. Inversión de prioridad para tres procesos

En LabVIEW no debería suceder este problema a menos que se modifiquen los niveles de prioridad de los VIs. Pero si sucede y además se combina con el de inanición puede ser fatal, sobre todo en sistemas de tiempo real.

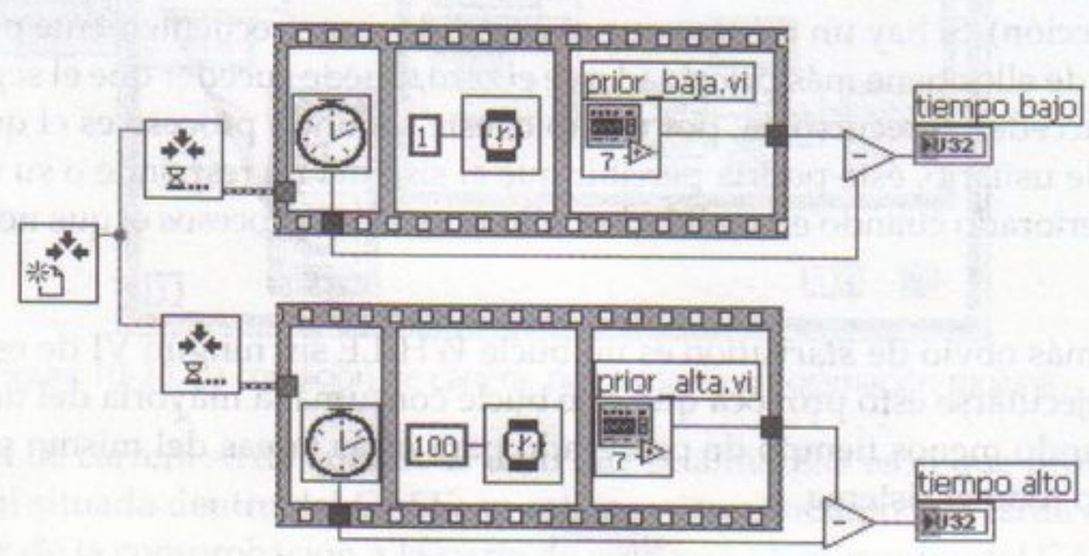


Figura 10-23. Lanzador.vi

A continuación se estudiará un programa de LabVIEW. Se tiene un VI llamado *lanzador.vi* que servirá para medir el tiempo que tardan en ejecutarse otros dos VIs, estos dos VIs se llaman *prior_alta.vi* y *prior_baja.vi* con una prioridad muy alta y muy baja respectivamente.

Como se ve en la figura 10-23 el VI de prioridad menor empezará a ejecutarse mucho antes que el de más prioridad, además este VI obtendrá un recurso que no es más que otro VI no reentrante llamado *recurso compartivo.vi*, el VI de menor prioridad no liberará el recurso hasta pasado un tiempo relativamente alto, durante ese tiempo el VI de mayor prioridad tendrá que estar esperando a que acabe el de menor prioridad.

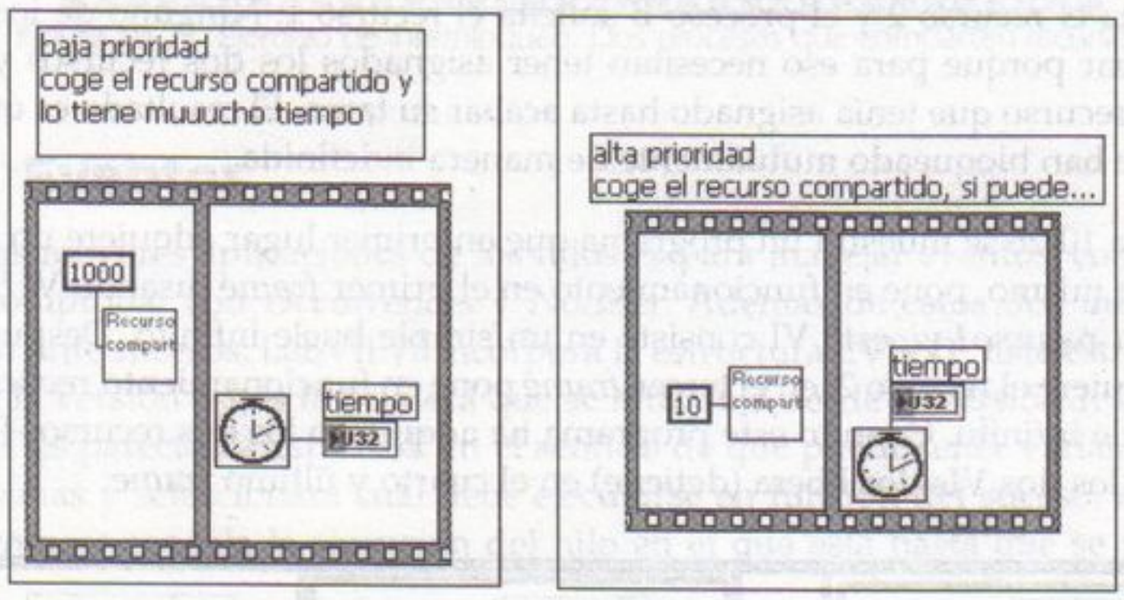


Figura 10-24. Recurso compartido para las diferentes prioridades

El VI del recurso compartido tardará en ejecutarse 10 ms por el valor que tienen como entrada, en el caso del VI de baja prioridad tardaría 10 segundos, mientras que en el de alta prioridad tardaría 100 ms. Si no hubiera problemas, en el programa del lanzador el primer hilo (el que contiene el VI de menor prioridad) debería tardar 10,001 segundos y el segundo (el de más prioridad) 200 ms. Sin embargo el resultado es el mostrado en la figura 10-25.

Figura 10-25. Resultado de la ejecución de lanzador.vi

Los métodos más usados para evitar o, más bien, paliar este problema son los de herencia de prioridad.

Como curiosidad durante la misión *Mars Pathfinder* a Marte, la sonda sufría un error que consistía en la pérdida de la información metereológica que recopilaba. Esta pérdida de información se debía a unos continuos *resets* que sufría el sistema y estos *resets* eran debidos a que se producía una inversión de prioridad entre varios hilos y al tar-

dar los hilos de más prioridad más tiempo de lo esperado, la nave pensaba que se habían bloqueado y para solucionarlo se reseteaba.

10.3.4 Interbloqueo

El interbloqueo, también llamado *deadlock* es un problema bastante complejo y difícil de solucionar, aunque también es poco frecuente. Para explicarlo de una forma clara se particularizará para dos procesos y dos recursos.

La situación de partida es la siguiente: hay dos procesos, A y B; el proceso A tiene asignado el recurso 1 y el proceso B tiene asignado el recurso 2. Ahora ocurre que el proceso A solicita el recurso 2 y el proceso B solicita el recurso 1. Ninguno de los procesos podrá acabar porque para eso necesitan tener asignados los dos recursos y ninguno liberará el recurso que tenía asignado hasta acabar su tarea. El resultado es que los dos procesos se han bloqueado mutuamente de manera indefinida.

En la figura 10-26 se muestra un programa que en primer lugar adquiere un recurso, o lo que es lo mismo, pone en funcionamiento en el primer *frame* (usando VI Server) un VI llamado *recurso1.vi*, este VI consiste en un simple bucle infinito. Después de una espera adquiere el recurso 2, en el tercer *frame* pone en funcionamiento *recurso2.vi* que es otro bucle infinito. Cuando este programa ha adquirido los dos recursos (ha puesto en marcha los dos VIs) los libera (detiene) en el cuarto y último *frame*.

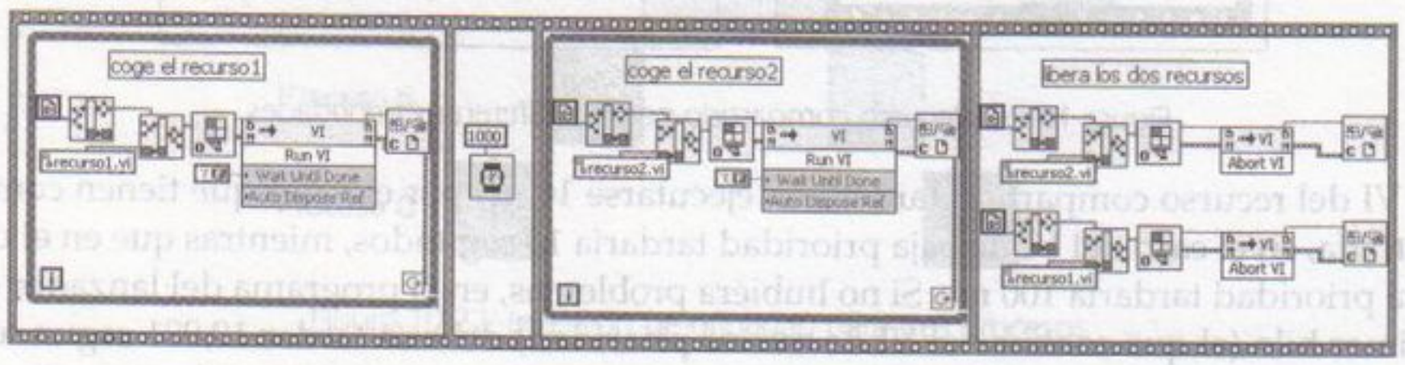


Figura 10-26. Ejemplo que utiliza dos recursos, proceso A

Si ahora se tiene otro VI con el mismo código pero cambiando el orden de los recursos (*frames* 1 y 3) y ambos se ejecutan a la vez, los dos se quedarán bloqueados en el bucle WHILE del tercer *frame* porque no pueden adquirir el segundo recurso debido a que ya lo había adquirido el otro VI.

En este caso el programa lanzador parará los dos VIs que sirven como recursos en caso que estuvieran ejecutándose antes y después abre su Panel Frontal para cargar una única copia de ellos en memoria. Después lanza los dos procesos que son los VIs *procesoA.vi* y *procesoB.vi*.

El resultado de la ejecución de estos dos procesos es que ambos se bloquean mutuamente. Puede comprobarse que si se ejecuta completamente primero un proceso y luego el otro no ocurre interbloqueo.

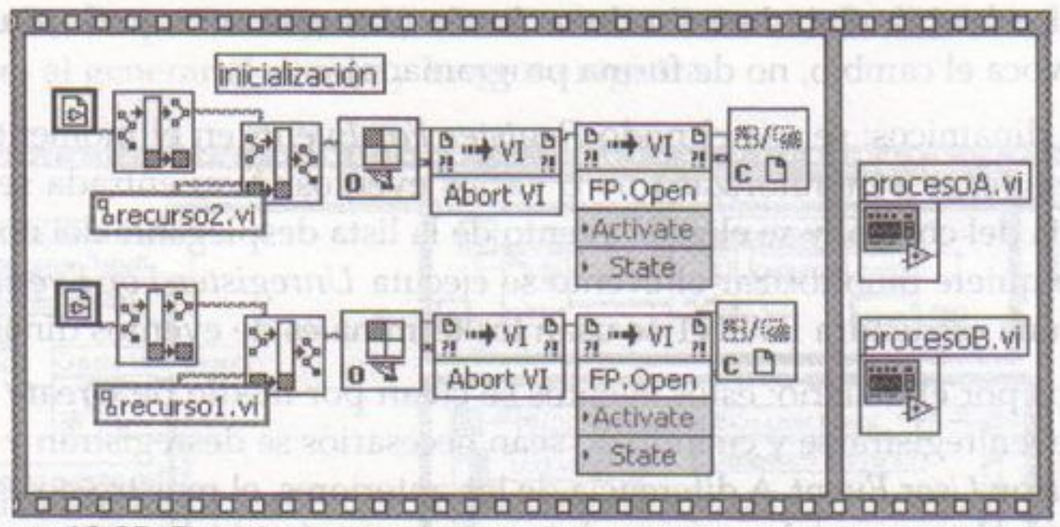


Figura 10-27. Ejemplo de interbloqueo. Dos procesos que comparten recursos

10.4 Eventos

Una de las mayores aplicaciones de los hilos es para manejar eventos, como ya se ha podido comprobar con *Occurrences* y *Notifier*. Además de estos dos métodos para reaccionar ante sucesos, LabVIEW incorpora la estructura EVENT. Esta estructura aparecida en la versión 6.1 es la primera que se introdujo desde la versión 1. La estructura EVENT es parecida a un CASE en el sentido de que puede tener varias páginas o subdiagramas y seleccionará cuál debe ejecutarse en función del suceso, al igual que las *occurrences* congela la ejecución del hilo en el que está hasta que se produce un evento.

La estructura EVENT ya se estudió con detalle al principio del libro (capítulo 2), ahora se verá el resto de VIs de la paleta *Dialog & User Interface > Events*.

Figura 10-28. Menú de eventos

Cuando ocurre un evento el sistema operativo lo lleva a una cola. Al llegar la ejecución a la estructura EVENT, LabVIEW busca en la cola algún evento de los que tiene registrados y si lo encuentra ejecuta el subdiagrama asociado, luego se repite el proceso para el siguiente evento y así sucesivamente.

Hay tres tipos de eventos dependiendo de la forma de registrarse:

- Eventos estáticos: son los que se configuran en el menú de la estructura EVENT; estos eventos permanecen registrados durante toda la ejecución del programa. Un

ejemplo sería el cambio de valor de un botón (únicamente cuando el usuario es el que provoca el cambio, no de forma programada).

- Eventos dinámicos: se usa el nodo *Register For Events* en el momento en que se quiere empezar a monitorizar uno o varios eventos. A su entrada se cablea una referencia del control y se elige el evento de la lista desplegable del nodo. Cuando ya no se quiere monitorizar el evento se ejecuta *Unregister For Events*. Para asociarlo a una estructura EVENT se usan los terminales de eventos dinámicos.
- Definidos por el usuario: estos eventos se crean por medio de *Create User Event*, luego deben registrarse y cuando no sean necesarios se desregistran y se eliminan con *Destroy User Event*. A diferencia de los anteriores, el registro no es a través de una referencia porque el evento no lo produce ningún objeto en concreto, el evento es producido cuando se ejecuta *Generate User Event*.

La principal aplicación de los dos primeros tipos de eventos es manejar el interfaz de usuario, mientras que la tercera es similar a *Occurrences* y *Notifier*.

10.4.1 Ejemplo I: Eventos dinámicos

Este evento es parte del código de un programa mayor; en él, primero se ejecutan ciertas tareas, pero en un momento determinado se desea que el usuario introduzca una contraseña para que la ejecución continúe.

La tarea que se monitorizará será el cambio de un control tipo *string*. Cuando la contraseña introducida sea la correcta, se desregistrará el evento y continuará la ejecución. Si se hubiera hecho de forma estática, el evento estaría registrado durante todo el tiempo que el VI estuviera ejecutándose.

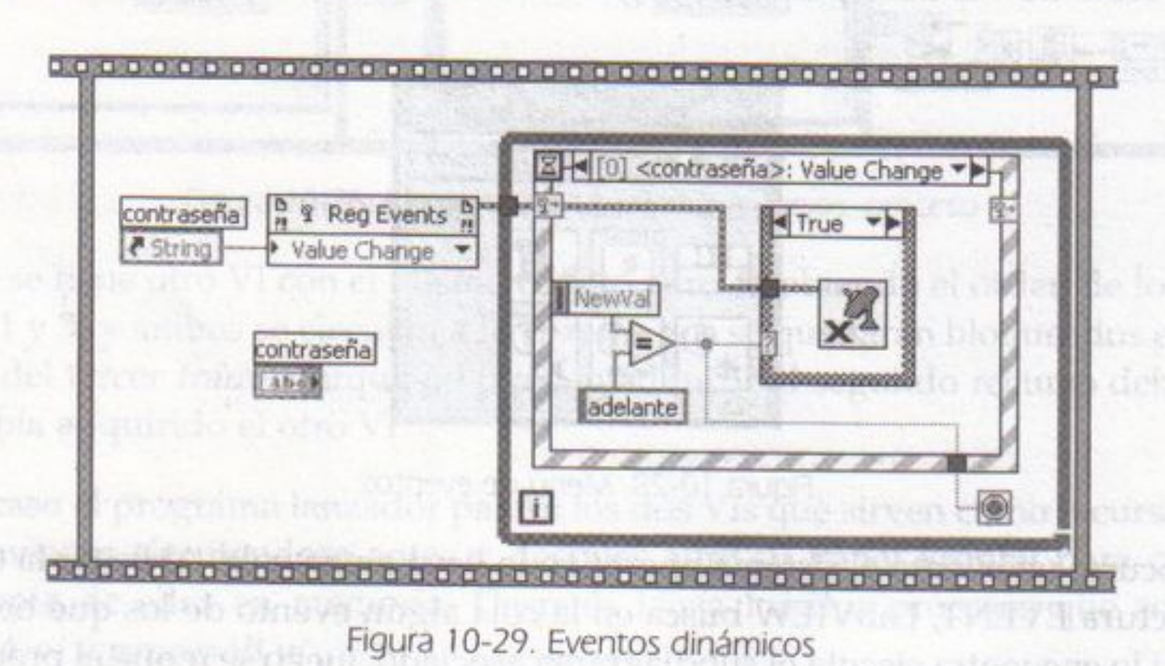


Figura 10-29. Eventos dinámicos

10.4.2 Ejemplo II: Eventos de usuario

En este ejemplo se desea generar un evento cada vez que se recibe una conexión TCP en el puerto 13. Cuando se genere este evento otro hilo responderá indicando la fecha y la hora al *host* que abrió la conexión.

Obsérvese como en primer lugar se crea el evento de usuario indicando el tipo de dato que se pasará al generar el evento y luego se registra.

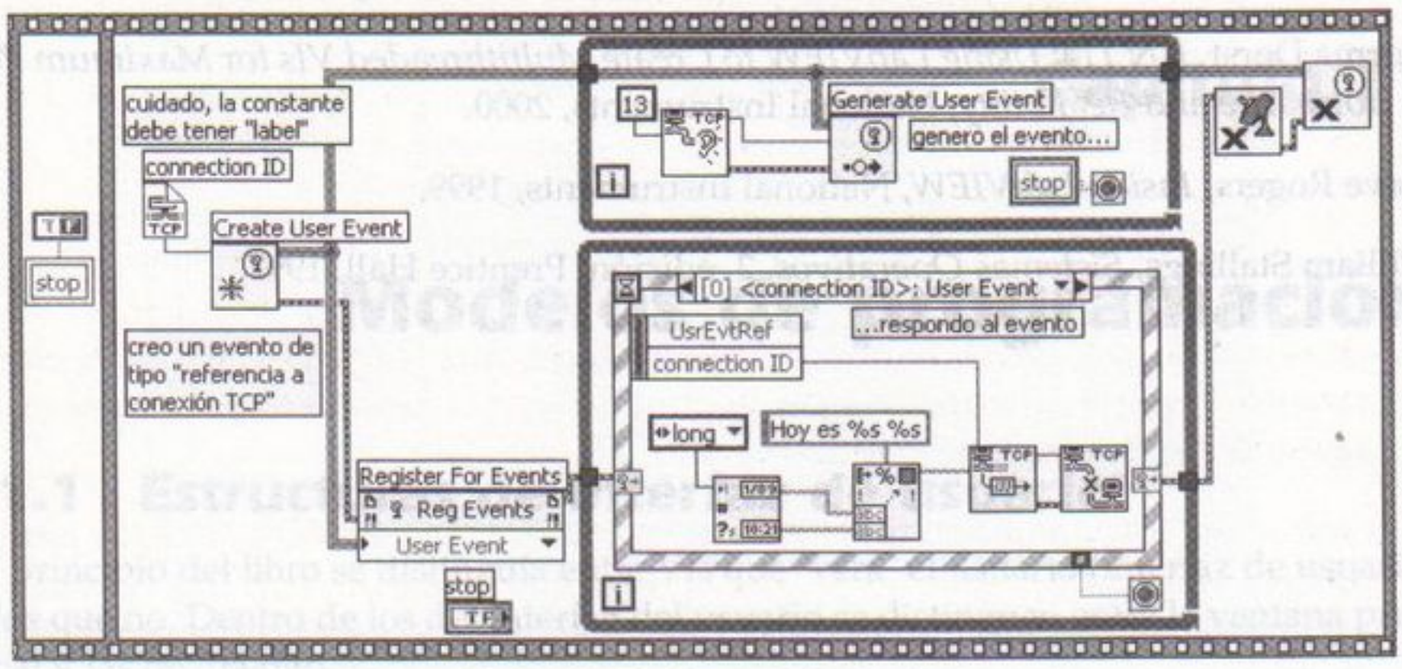


Figura 10-30. Eventos de usuario

10.5 Ejercicios

1. Sustituir el CASE del programa que calcula el factorial de forma recursiva por un *Select*. ¿Qué ocurre? ¿Por qué?
2. Realizar mediante occurrences lo mismo que en el ejemplo de los semáforos.
3. En el ejemplo con semáforos comprobar qué ocurre si quitan las estructuras SEQUENCE de los dos hilos. Justificarlo.
4. Hacer un programa cliente-servidor El cliente simplemente debe abrir una conexión TCP con el servidor y esperar su respuesta. Cuando el servidor acepte una conexión debe escanear el bus GPIB para identificar los equipos conectados a él, cuando acabe el escáner devolverá los resultados al cliente. Nótese que el tiempo que necesitará el servidor para atender una conexión será elevado; si mientras está atendiendo una conexión llega otra de un segundo cliente puede que este cliente aborte la conexión debido a que el tiempo que tarda el servidor en aceptar la conexión es tan grande que el cliente interpreta que no hay servidor. Para solucionar este problema el servidor debe tener dos hilos: uno que únicamente acepte conexiones y otro que las trate, como mecanismo de comunicación entre estos dos hilos usar las colas.
5. Solucionar el problema de la condición de carrera.
6. Hacer que el VI utilizado como recurso compartido en el problema de la inversión de prioridad sea reentrante. ¿Qué ocurre? ¿Por qué?

10.6 Bibliografía

National Instruments, *AN 199: LabVIEW and Hiper-Threading*, 2004.

Norma Dorst, *AN 114: Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Realibility*, National Instruments, 2000.

Steve Rogers, *Inside LabVIEW*, National Instruments, 1999.

William Stallings, *Sistemas Operativos*, 2 edición, Prentice Hall, 1997.

Capítulo 11

Modelos de programación

11.1 Estructuras de interfaz de usuario

Al principio del libro se distinguía entre VIs que "verá" el usuario (interfaz de usuario) y los que no. Dentro de los del interfaz del usuario se distinguen entre la ventana principal y las de diálogo.

En versiones antiguas de LabVIEW, antes de que se introdujesen las nuevas estructuras, un VI de interfaz de usuario no era más que un bucle WHILE que debía estar continuamente comprobando si se había seleccionado alguna opción, normalmente un botón con *Latch When Released* por acción mecánica. Al activarse una opción se ejecutaba su código asociado que estaría dentro de un CASE conectado al botón. Todo el código podría estar en distintos CASE y éstos dentro del mismo WHILE. Para no consumir una cantidad excesiva de tiempo de CPU se añadía una pequeña espera al bucle. También se añadía un botón en la condición del WHILE para detener el programa.

En la figura 11-1 se muestra esta estructura en la que se permanece a la espera de que el usuario pulse el botón RUN.

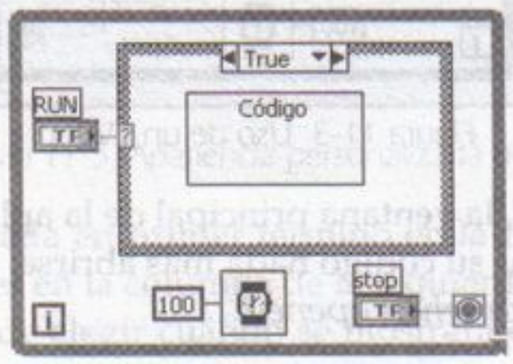


Figura 11-1. VI de interfaz de usuario clásico

A veces también es necesario ejecutar una acción cuanto cambia uno o varios valores, para conseguirlo se podría realizar una pequeña modificación al programa anterior y quedará como en la figura 11-2.

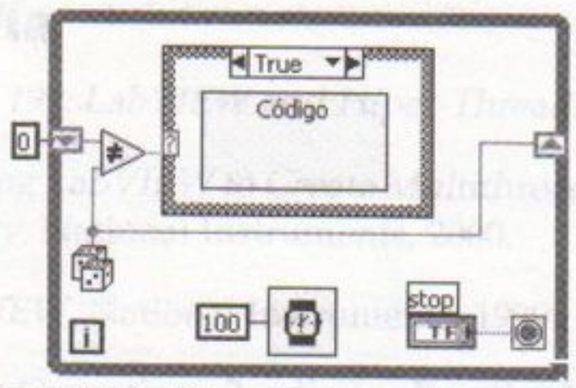


Figura 11-2. Detección de cambio de valor

La ventaja de este programa es que se pueden asociar acciones a las distintas opciones del usuario. El principal inconveniente es que aunque se use una función de espera, el programa seguirá ejecutándose continuamente.

Una estructura más apropiada que CASE para el interfaz de usuario es EVENT porque puede detener la ejecución de un programa hasta que el usuario elija una opción, por lo tanto puede aprovechar mejor los ciclos de reloj del procesador. Además mediante los eventos de usuario también pueden generarse eventos de forma programada. Cuando se emplee, hay que tener la precaución de que EVENT se ejecute antes que la condición de parada del bucle, esto puede conseguirse incluyendo un evento asociado al cambio de valor del botón de *stop*, este botón se sitúa dentro del subdiagrama correspondiente y se cablea al terminal del WHILE; el túnel creado puede dejarse sin valor o usar una constante booleana FALSE en el resto de subdiagramas.

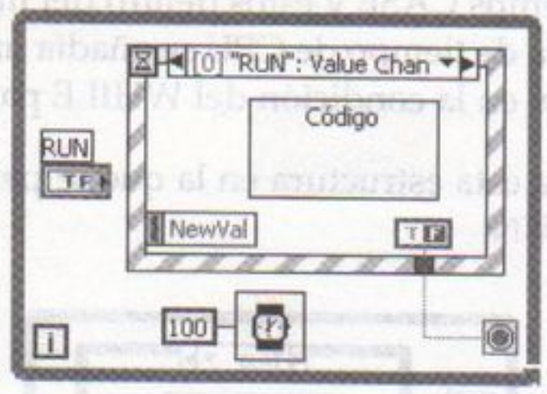


Figura 11-3. Uso de un EVENT

Ya se utilice CASE o EVENT, la ventana principal de la aplicación o cualquier otra ventana de diálogo debe ejecutar su código nada más abrirse, esto se consigue en *File > VI Properties > Execution > Run when opened.*

En el caso de las ventanas de diálogo, además de ejecutar el código también necesitará mostrar su Panel Frontal, ésta y otras opciones relacionadas con el aspecto pueden encontrarse en *File > VI Properties > Window Appearance*. Las tres opciones de esta ventana (*Top-level application window, Dialog* y *Default*) son las predeterminadas, además de ellas también se puede personalizar el aspecto en *Custom*.

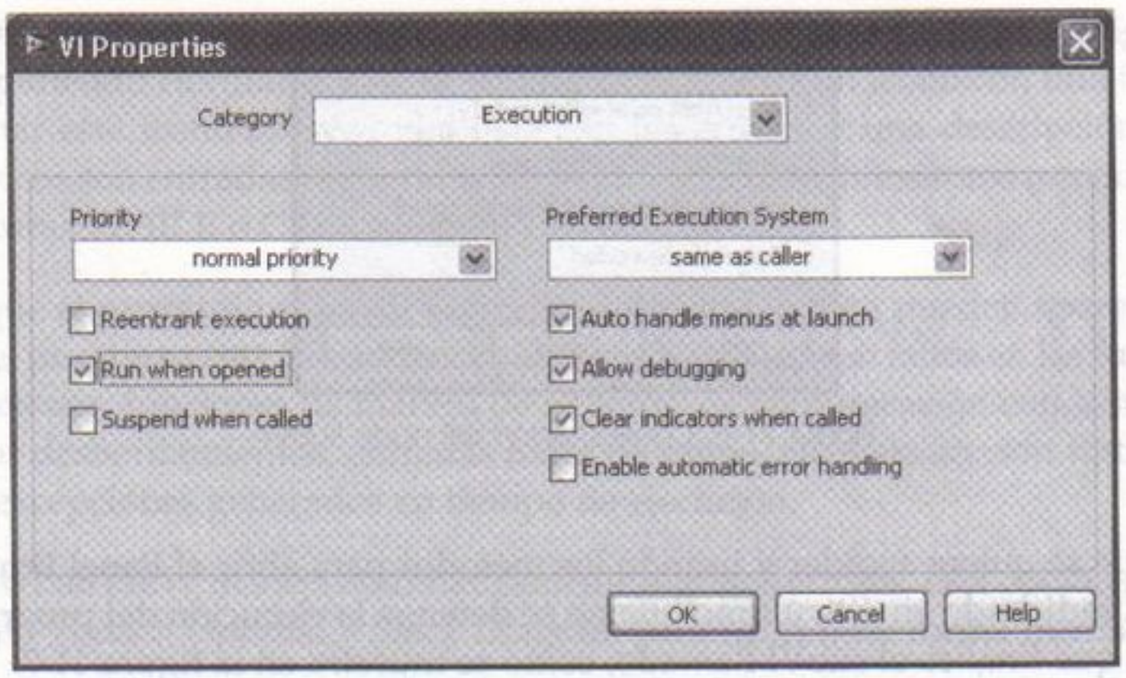


Figura 11-4. Configuración de las propiedades del VI para ejecutarse al abrirse

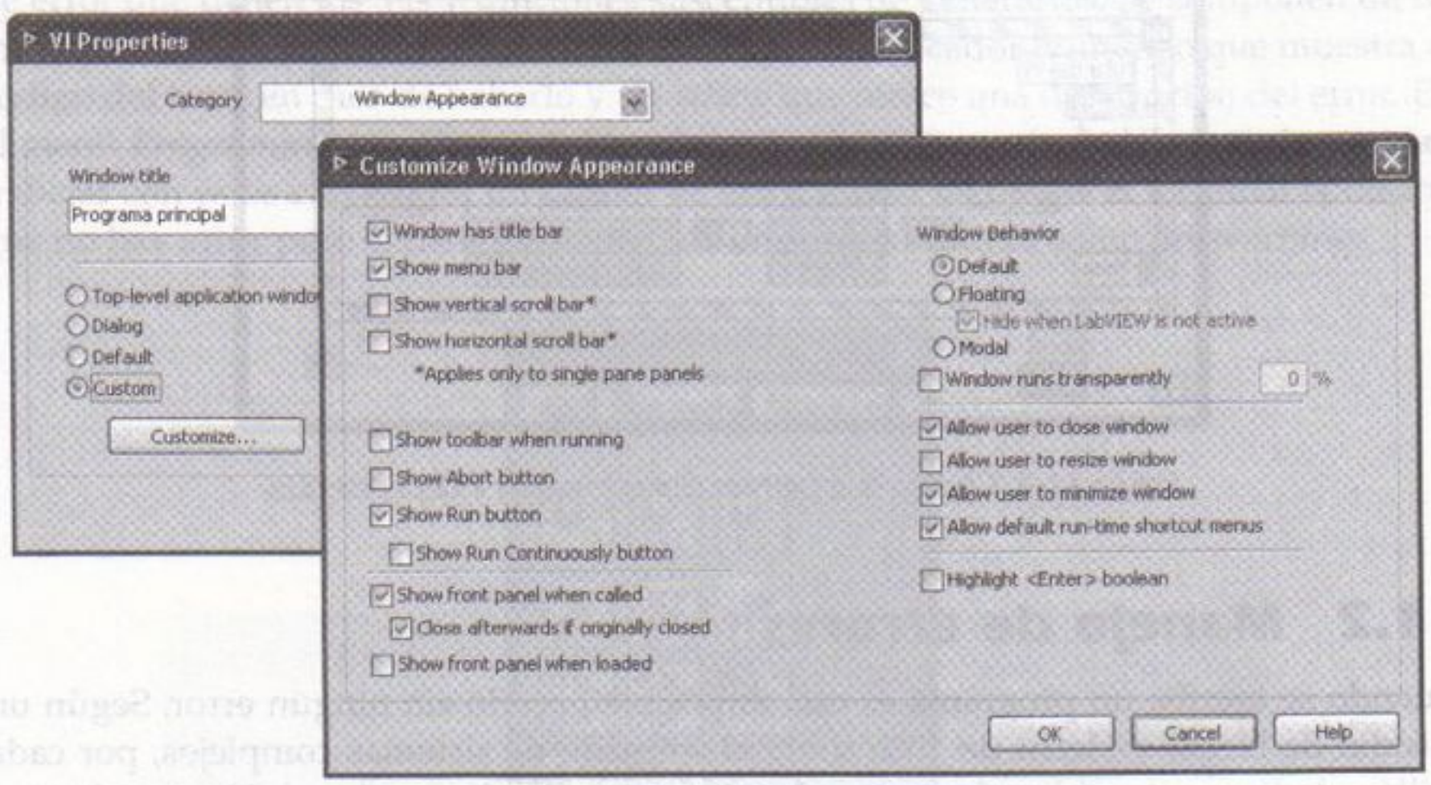


Figura 11-5. Apariencia personalizada del VI

Dentro de la ventana mostrada en primer término en la figura 11-5, las opciones más interesantes son las inferiores en la columna de la izquierda y las superiores en la derecha. En las primeras se puede elegir cuándo se mostrará el Panel Frontal, estas opciones son equivalentes a las que se tienen yendo al menú contextual de un VI y pulsando en *SubVI Node Setup*. Las segundas determinan la forma de mostrarse la ventana: normal (*Default*), en primer plano (*Floating*) o impidiendo acceder a otras ventanas de LabVIEW hasta cerrar esta (*Modal*).

Debe tenerse siempre en cuenta que al llamar a un subVI el hilo de ejecución del programa padre se quedará esperando a que termine la ejecución del "hijo"; a veces será deseable que aparezca una ventana pero que la anterior siga ejecutándose. Para conseguir este efecto se debe realizar la llamada a través de VI Server abriendo una referen-

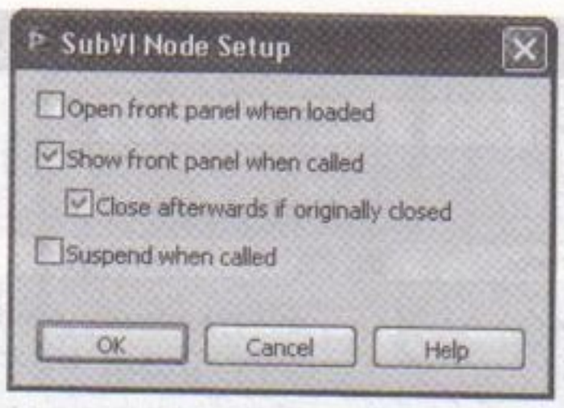


Figura 11-6. Configuración del nodo SubVI

cia al VI que se quiere acceder y usando los métodos para abrir el Panel Frontal y ejecutar el VI, indicando en este último que no se detenga la ejecución del programa principal hasta que finalice el subVI llamado, como se muestra en la figura 11-7.

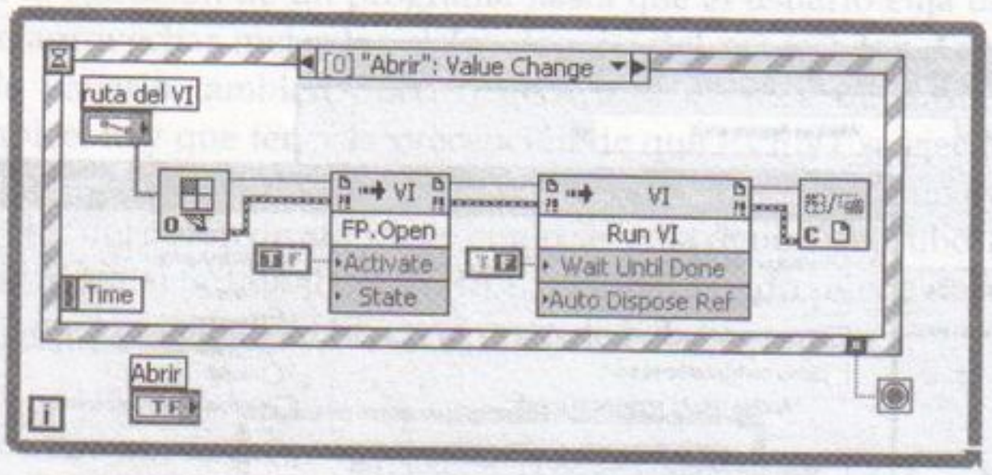


Figura 11-7. Código que permite abrir ventanas independientes

11.2 Manejo de errores

Cuando se escribe un programa es casi imposible hacerlo sin ningún error. Según un estudio de Hecht y Hecht de 1986 sobre el software de sistemas complejos, por cada millón de líneas de código hay unos 20.000 errores, de los cuales el 90% pueden ser detectados con sistemas de comprobación y 200 más durante el primer año, los 1.800 fallos restantes no son detectados.

Los errores pueden tener varios orígenes:

- Malas especificaciones.
- Errores de diseño.
- Avería en el hardware.
- Fallos en la comunicación.

Los fallos afectan a la fiabilidad y seguridad del sistema, por lo tanto se hace necesario detectar, manejar y si es posible corregir adecuadamente estos fallos.

Como se vió en el primer tema, LabVIEW indica si hay algún error en el código del programa impidiendo ejecutarlo, esto se simboliza mediante un icono con una flecha rota. Al presionar este icono aparece una lista con los errores y una descripción. Algunos errores típicos son entradas requeridas sin valor conectado, hilos rotos o condiciones de las estructuras WHILE o CASE también sin conectar.

Una vez que el programa es sintácticamente correcto también pueden aparecer errores mientras el programa está funcionando, algunos ejemplos son fallos en las funciones de comunicaciones, funciones de manejo de ficheros, de comunicación con instrumentos, errores por desbordamientos, entradas fuera de rango, etc. Esta sección trata sobre los errores o excepciones generados en tiempo de ejecución.

Por defecto LabVIEW maneja automáticamente algunos errores, esta opción puede desactivarse en *Tools > Options > Block Diagram* para hacerlo de forma manual.

La principal herramienta que proporciona LabVIEW para detectar fallos son los *cluster* de error que tienen los VIs y funciones susceptibles de generarlos. Se componen de un indicador booleano que indica si hay error o no, un indicador numérico que muestra el código del error en caso de haberlo y un *string* que ofrece una descripción del error. En el menú *Programming > Dialog & User Interface* hay una serie de VIs que sirven para trabajar con estos *clusters*. Si un *cluster* de error indica mediante el terminal booleano que no hay error pero sí tiene un código de error se considera como un *warning*.

Figura 11-8. Menú de diálogos e interfaz de usuario

Los VIs más usados son: *Simple Error Handler* y *General Error Handler*. Su funcionamiento es idéntico, de hecho el primero contiene al segundo; pueden mostrar un mensaje al usuario cada vez que a su entrada *Error In* le llega un error (o *warning*); es conveniente que en caso de mostrar el mensaje, éste tenga un botón de parada porque si la ejecución es cíclica podría llegar a impedir usar el interfaz al usuario, como muestra el programa de la figura 11-9.

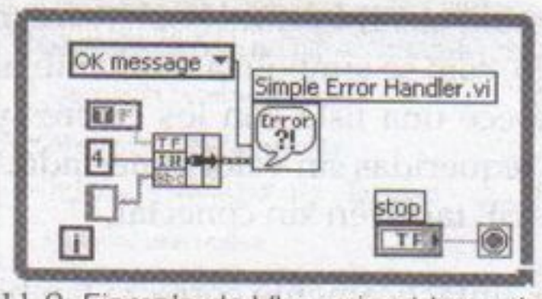


Figura 11-9. Ejemplo de VI que impide su detención

El VI *General Error Handler* permite personalizar el error asociando una descripción a cada código. Entre otras opciones también permite ignorar el error.

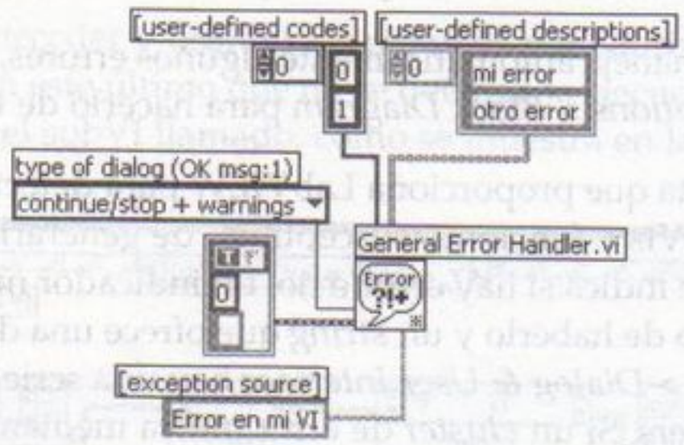


Figura 11-10. Personalización del error asociando una descripción a cada código

Si no se especifican las descripciones de los errores, éstas se toman de las definidas en LabVIEW por defecto. Cuando el número de errores sea elevado es más conveniente agruparlos todos en un fichero externo en lugar de en *arrays* dentro del código del programa. El fichero puede crearse y editarse desde *Tools > Advanced > Edit Error Codes*, por defecto se almacenará en *user.lib\errors* del directorio de instalación de LabVIEW.

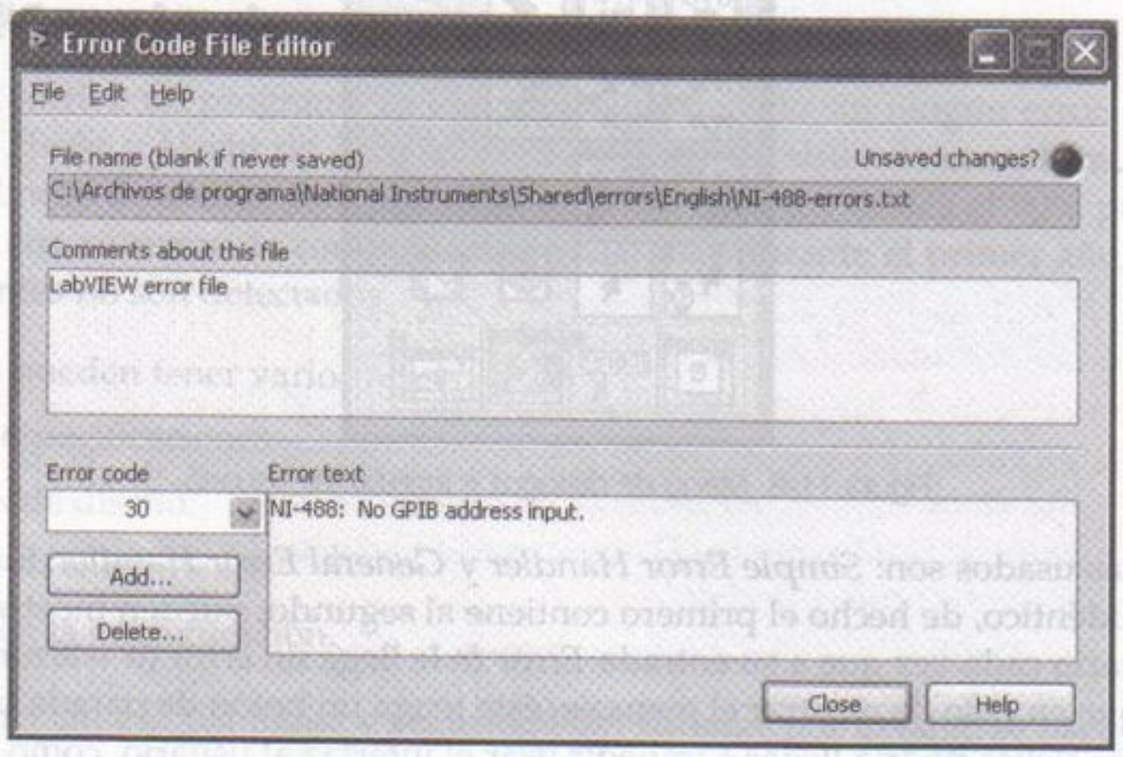


Figura 11-11. Editor de ficheros de error

Para manejar los errores pueden emplearse las estructuras CASE y WHILE y el VI *Select*, cableadas de la forma indicada en la figura 11-12.

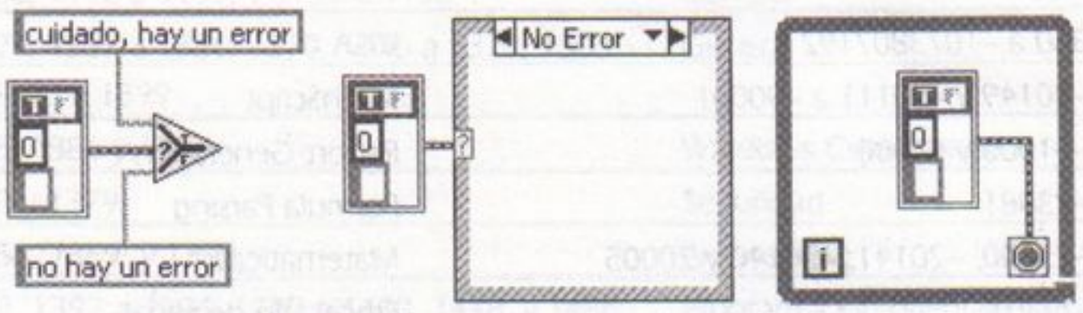


Figura 11-12. Manejo de errores con Select, CASE y WHILE

El VI *Select* puede considerarse como una simplificación del CASE cuando se quiere obtener un valor u otro dependiendo de si ha habido error o no. La estructura CASE puede ejecutar un subdiagrama u otro dependiendo de si ha habido un error o no y el WHILE puede repetir el código de su interior mientras haya un error o detenerse cuando ocurra. Cuando se construye un *driver* para comunicarse con dispositivos externos o con ficheros es conveniente crear los VIs de forma que todos tengan una entrada y una salida de error, el código en su interior debería colocarse dentro de un CASE de manera que sólo se ejecute cuando no haya error.

Por el contrario, para ejecutar un código como respuesta a un error producido también puede usarse la estructura CASE de forma equivalente a TRY. Una opción interesante es ejecutar un determinado código dependiendo del rango en que se encuentre el código numérico del error. La figura 11-13 muestra el código para manejar una excepción del puerto serie.

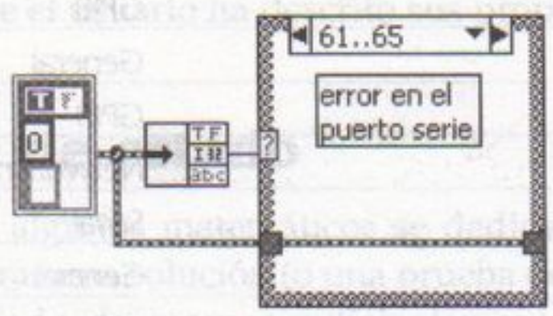


Figura 11-13. Manejo de una excepción del puerto serie

Los rangos de los errores típicos y la familia de funciones que los generan vienen dados por la Tabla 1.

Tabla 1 - Rango de los errores típico y familia de funciones

Rango	Función
–2147467263 a –1967390460	Networking
–1950679040 a –1950678977	Variables compartidas
–1967362045, –1967362022 a –1967361997 y –1967345663 a –1967345609	Seguridad

Rango	Función
–1074003967 a –1074003950, –1074000001 y –1074000000	Instrument Driver
–1073807360 a –1073807192	VISA
–90165 a –90149 y –90111 a –90001	MathScript
–41005 a –41000 y 41000	Report Generation
–23096 a –23081	Formula Parsing
–23096 a –23000, –20141, –20140 y 20005	Matemáticas
–20337 a -20301	Procesado de señal
–20207 a –20201	Point By Point
–20119 a –20001	Procesado de señal
–4644 a –4600	Expresiones regulares
–2983 a –2970, –2964 a –2960, –2955 a –2950, y –2929 a –2901	Control de código
–2586, –2585, –2583 a –2581, –2578, –2575, –2574, –2572 a –2550, –2526 a –2501 y 2552	Storage
–1821, –1820, –1817 a –1800	Waveform
–1719 a –1700	Apple Event
–1300 a –1210	Instrument Driver
–823 a –800 y 824	TIMED LOOP
–620 a –600	Windows Registry Access
0	Procesado de señal, GPIB, Instrument Driver, Formula Parsing, VISA
1 a 20	GPIB
1 a 52	General
30 a 32, 40 a 41	GPIB
53 a 66	Networking
61 a 65	Serial
67 a 91	General
92 a 96	Windows Connectivity
97 a 100	General
102 y 103	Instrument Driver
108 a 121	Networking
116 a 118 y 122	General
1000 a 1045	General
1046 a 1050 y 1053	Scripts
1051 a 1091	General
1101, 1114, 1115, 1132 a 1134, 1139 a 1143 y 1178 a 1185	Networking
1094 a 1157	General
1158 a 1169, 1318, 1404 y 1437	Run-Time Menu

Rango	Función
1172, 1173, 1189 y 1199	Windows Connectivity
1174 a 1188, 1190 a 1194, y 1196 a 1198	General
1301 a 1321, 1357 a 1366, 1370, 1376 a 1378, 1380 a 1391 y 1395 a 1399	General
1325, 1375, 1386 y 1387	Windows Connectivity
1367, 1368 y 1379	Seguridad
1337 a 1356, 1369, y 1383	Networking
1371, 1373, 1392 a 1394, 1400 a 1403, 1448, y 1486	Programación orientada a objetos
1430 a 1455, 1468 a 1470, 1483 a 1485, 1487 a 1493, y 1497 a 1500	General
1456 a 1467 y 1471 a 1482	3D Picture Control
1800 a 1809 y 1814	Waveform
4800 a 4806, 4810, 4811, y 4820 a 4823	General
16211 a 16554	SMTP
20001 a 20030, 20307, 20334, y 20351 a 20353	Procesado de señal
56000 a 56005	General
180121602 a 180121604	Variables compartidas
1073479937 a 1073479940	Instrument Driver
1073676290 a 1073676457	VISA

Además los rangos desde 5.000 a 9.999 y -8.999 a -8.000 están reservados para mensajes definidos por el usuario. Cuando una aplicación se distribuye también deben adjuntarse los ficheros en los que el usuario ha descrito sus propios errores.

11.3 Máquinas de estado

A principios del siglo XX algunos matemáticos se dedicaban a buscar un método o algoritmo capaz de encontrar una solución (o una prueba de que no existe) ante un problema matemático enunciado de manera suficientemente precisa. En primer lugar debían definir qué es un algoritmo, las propuestas básicamente fueron:

- Una máquina computadora abstracta pero definida de modo preciso (por ejemplo un autómata).
- Una construcción formal de procedimientos de cómputo (por ejemplo un sistema de Thue).
- Una construcción formal productora de clases de funciones (por ejemplo funciones recursivas).

Lamentablemente se demostró que lo que buscaban estos matemáticos no es posible, pero sirvió de base a la teoría de la computación que abarca temas como la teoría de autómatas, la teoría de los lenguajes formales, la teoría de la computabilidad y la teoría de la complejidad algorítmica. En aquella época todavía no existían ordenadores y

los trabajos se basaban en modelos abstractos de cálculo, máquinas imaginarias que ejecutaban un algoritmo. Estas máquinas eran sorprendentemente parecidas a los ordenadores de hoy en día pues manejaban conceptos como máquinas de propósito general, programación, dualidad hardware-software y lenguajes formales.

Los autómatas son modelos de comportamiento compuestos por estados, transiciones y acciones. Suelen representarse como en la figura 11-14. El mecanismo de control está situado en la parte inferior, este mecanismo mostrará el estado en que se encuentra la máquina en cada momento. El control tiene un cabezal lector sobre el cual va avanzando una cinta que está dividida en celdas, cada una de las cuales contendrá un símbolo; el símbolo que se sitúe sobre el cabezal será la entrada de la máquina en ese instante. En el siguiente paso la cinta avanzará una celda y un nuevo símbolo será leído.

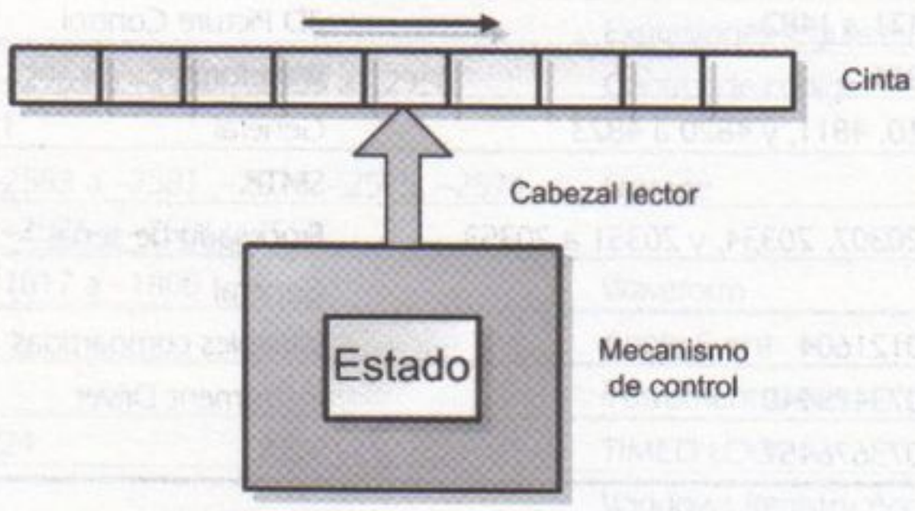


Figura 11-14. Ejemplo de un autómata

Un autómata puede usarse para modelar una construcción hardware (por ejemplo un circuito secuencial) o construcciones software (por ejemplo un analizador léxico).

Un autómata finito, también conocido como máquina de estados finita, se dice que es determinista cuando su mecanismo de control, estando en un estado conocido *q* y recibiendo una entrada *a*, en el siguiente instante pasará a otro estado también conocido, esto es: *estado siguiente=f(estado actual, entrada actual)*, siendo *f()* la función de transición, que puede representarse mediante una tabla o un diagrama.

La Tabla 2 tiene una fila para cada estado y una columna para cada entrada. El contenido de cada celda será el siguiente estado al que tiene que pasar la máquina cuando encontrándose en el estado indicado por su fila, reciba la entrada indicada por su columna.

Tabla 2 - Estados de un autómata finito

	Entrada 1	Entrada 2	Entrada 3
➡ Estado 1	Estado 1	Estado 2	Estado 2
Estado 2	Estado 2	Estado 3	Estado 1
# Estado 3	Estado 1	Estado 2	Estado 1

Para indicar los estados iniciales se dibuja una flecha en la tabla y un nodo con un arco entrante en el diagrama. Para los estados finales, también conocidos como de aceptación, se usa el símbolo # en la tabla y un nodo con doble círculo en el diagrama.

Los diagramas de estados representan a cada estado con un círculo y las transiciones entre ellos son flechas que van de unos a otros, sobre cada flecha se indica la entrada o entradas que produce esa transición, como se puede ver en la figura 11-15.

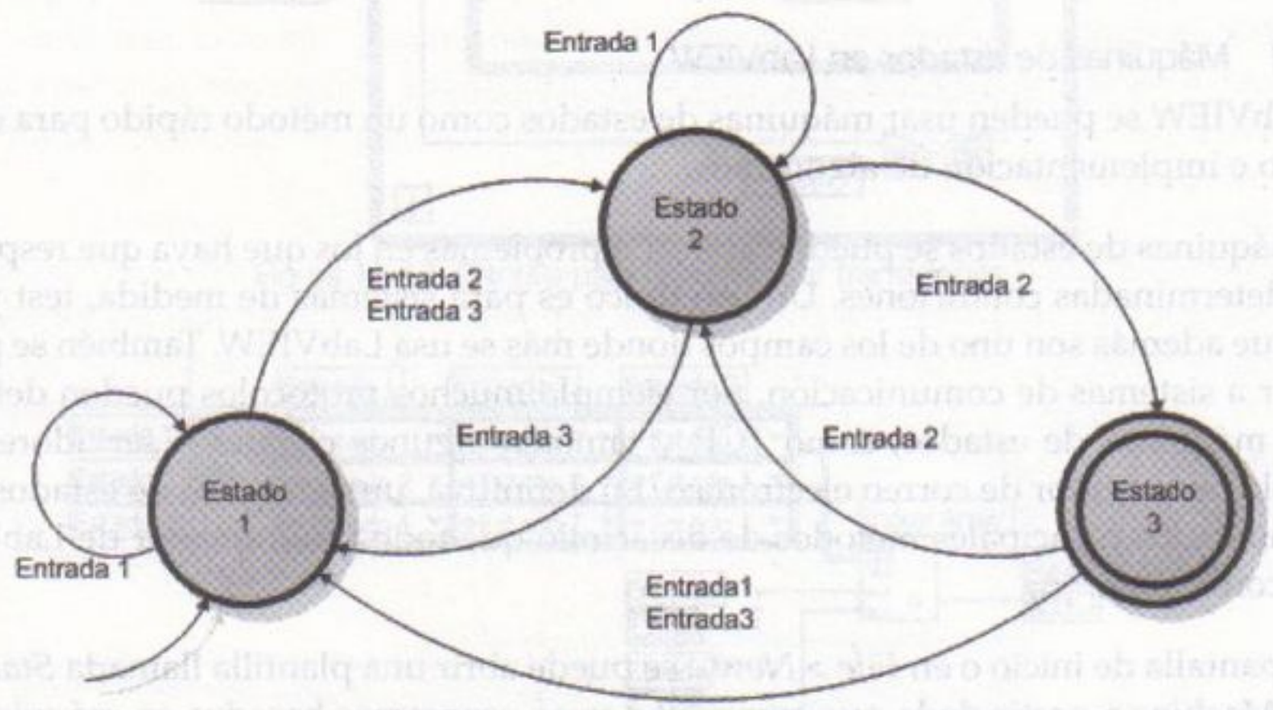


Figura 11-15. Diagrama de estados

Los autómatas también pueden ejecutar una acción que puede ser simplemente modificar el valor de una variable de salida u otras más complejas. La acción puede ejecutarse cuando se entra en un nuevo estado, cuando se sale o cuando se realiza una transición.

Hay varios tipos de autómatas finitos, los más destacables son los de Moore y Mealy. En los primeros la acción ocurre al entrar en un estado, en los segundos la acción es ejecutada al recibir una entrada y depende tanto de ésta como del estado del que proviene, estas máquinas pueden conseguir una reducción del número de estados.

En los autómatas finitos no deterministas no se puede establecer el estado siguiente a partir únicamente del estado actual y la entrada actual. Estos autómatas tendrán algún nodo del diagrama con arcos de salida con el mismo símbolo o algún nodo al que le falte algún arco.

Los autómatas de pila añaden a los anteriores un elemento más al esquema de la figura 11-14: una memoria en forma de pila. En cada transición se podrán leer y/o escribir datos de la memoria, así las transiciones no dependen únicamente de la entrada actual, sinó también de condiciones anteriores.

Un autómata todavía más general es la máquina de Turing, propuesta por Alan M. Turing en 1936. En ella la cabeza lectora puede moverse y además puede escribir sobre la cinta, de esta manera puede usar la propia cinta como memoria. Estas máquinas tie-

nen un estado final en el que se parará el cálculo, a diferencia de los estados de aceptación de las anteriores en los que se podía continuar el cálculo. La importancia de la máquina de Turing es que mediante ella se puede realizar cualquier operación computable, es lo que se conoce como tesis de Church-Turing.

Las aplicaciones de los autómatas son muy variadas, se aplican en campos como la lingüística, biología, matemáticas, electrónica e informática.

11.3.1 Máquinas de estados en LabVIEW

En LabVIEW se pueden usar máquinas de estados como un método rápido para el desarrollo e implementación de algoritmos.

Las máquinas de estados se pueden aplicar a problemas en los que haya que responder ante determinadas condiciones. Un uso típico es para sistemas de medida, test y control, que además son uno de los campos donde más se usa LabVIEW. También se puede aplicar a sistemas de comunicación, por ejemplo muchos protocolos pueden definirse como máquinas de estados, como TCP, y también algunos clientes o servidores, por ejemplo un servidor de correo electrónico. En definitiva, una máquina de estados debe ser uno de los principales métodos de desarrollo que todo programador de LabVIEW debe considerar.

En la pantalla de inicio o en *File > New...* se puede abrir una plantilla llamada *Standard State Machine* a partir de la que empezar a crear programas basados en máquinas de estados.

Todos los estados de la máquina deben ser definidos en una constante de tipo *Enum*. Si además se crea un tipo personalizado (*Type Def.* ó *Strict Type Def.*) será más sencillo modificar los estados a posteriori (ver sección sobre controles en el capítulo 1).

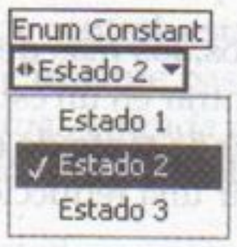


Figura 11-16. Definición de estados en una constante tipo Enum

El código consta de un bucle WHILE y dentro de él un CASE, el bucle tiene un *Shift register* por el que circula el siguiente estado al que tiene que pasar la máquina; la acción asociada al estado estará contenida en el subdiagrama correspondiente del CASE y también la decisión del siguiente estado al que debe pasar la máquina.

Para implementar las transiciones de la máquina de estados se puede usar como modelo el código de la figura 11-17 o el de la figura 11-18, donde se ha introducido directamente la tabla de transiciones, ver Tabla 2, a un *array* de dos dimensiones. Para obtener el estado siguiente simplemente hay que acceder al elemento correspondiente mediante *Index Array* usando un valor numérico que represente a la entrada y el estado como índices.

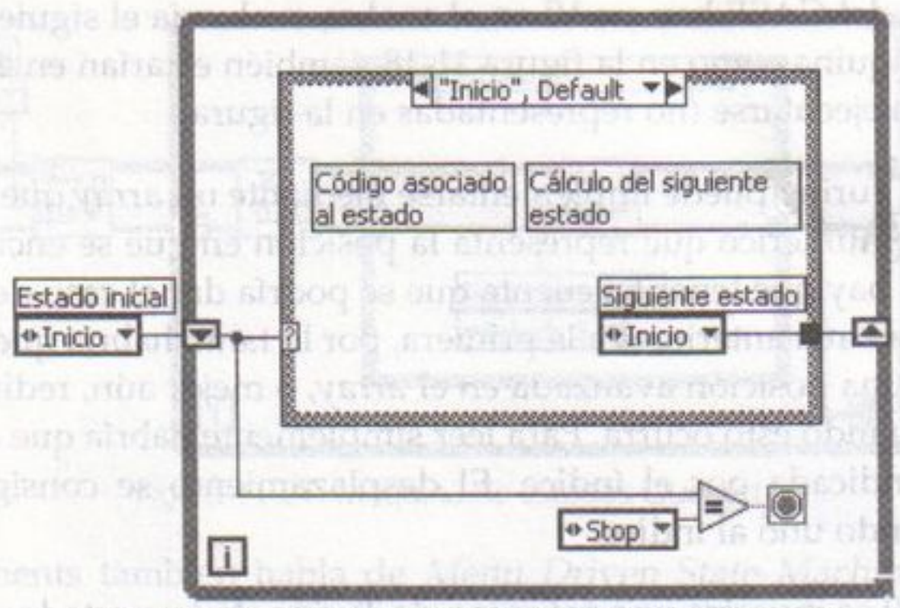


Figura 11-17. Plantilla para implementar transiciones I

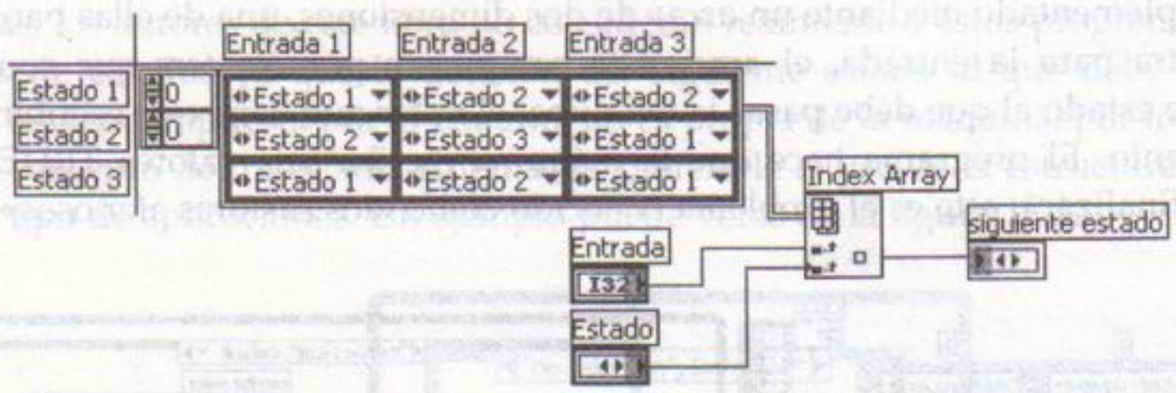


Figura 11-18. Plantilla para implementar transiciones II

Los autómatas finitos pueden implementarse mediante una máquina de estados estándar. Para los autómatas de pila hay que introducir una cola. En el capítulo sobre multihilo se vieron algunos VIs que permiten crear y trabajar con colas, éstos se hallan en *Programming > Syncronization > Queue Operations*. En la figura 11-19 puede apreciarse el uso de estos VIs, en primer lugar se crea una referencia a la pila eligiendo el tipo

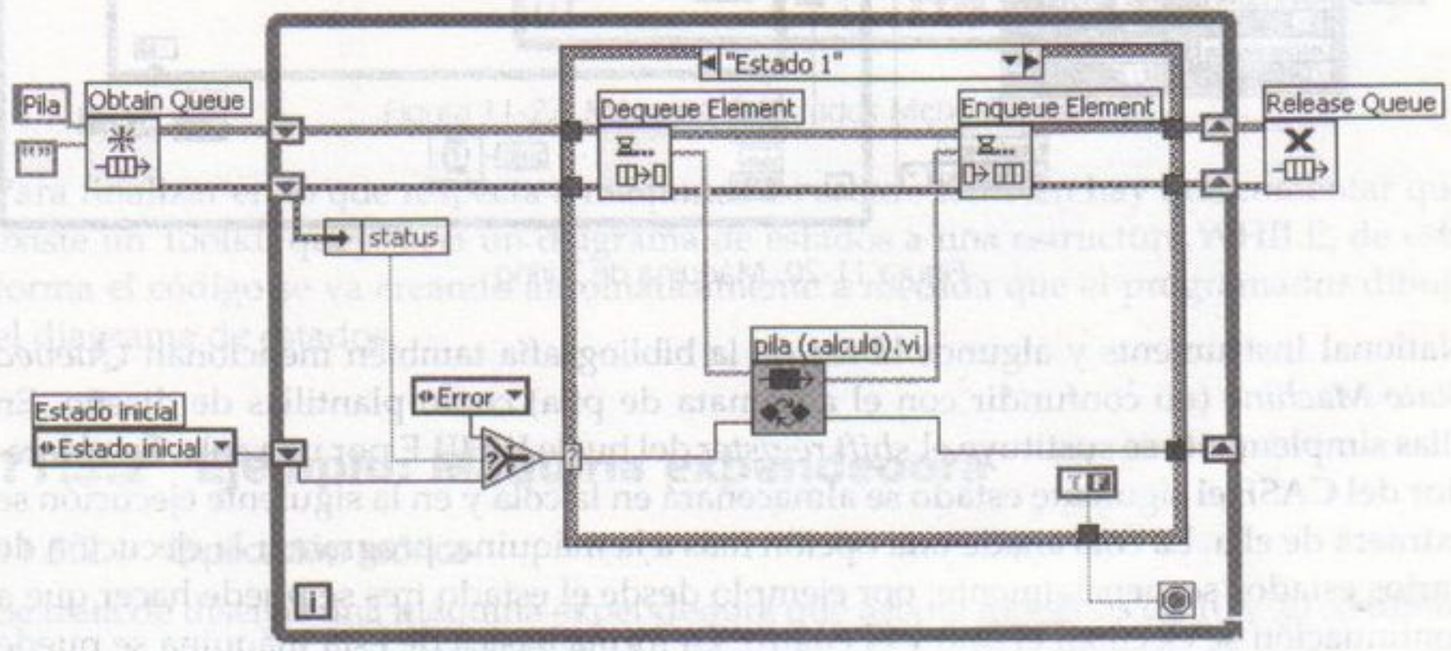


Figura 11-19. Máquina de estados con pila

de datos; dentro del CASE hay un VI en el cual se evaluaría el siguiente estado al que debe pasar la máquina como en la figura 11-18, también estarían en el CASE las acciones que deberían ejecutarse (no representadas en la figura).

Una máquina de Turing puede implementarse mediante un *array* que hace las veces de cinta y un índice numérico que representa la posición en que se encuentra el cabezal. En esta máquina hay que tener en cuenta que se podría dar el caso de intentar acceder a posiciones de la cinta anteriores a la primera, por lo tanto habría que hacer que el primer dato ocupe una posición avanzada en el *array*, o mejor aún, redimensionar y desplazar el *array* cuando esto ocurra. Para leer simplemente habría que acceder a la posición del *array* indicada por el índice. El desplazamiento se consigue simplemente sumando o restando uno al índice.

En la figura 11-20 se muestra una máquina de Turing de tres estados con una entrada de datos booleana con los datos inicialmente con valor FALSE. La tabla de transiciones se ha implementado mediante un *array* de dos dimensiones, una de ellas para los estados y otra para la entrada, el *array* está compuesto por *clusters* que contienen el siguiente estado al que debe pasar la máquina, el valor que tiene que escribir y el desplazamiento. El programa hace que la máquina escriba seis valores TRUE en doce pasos y finalizará, este es el problema conocido como «los castores afanosos».

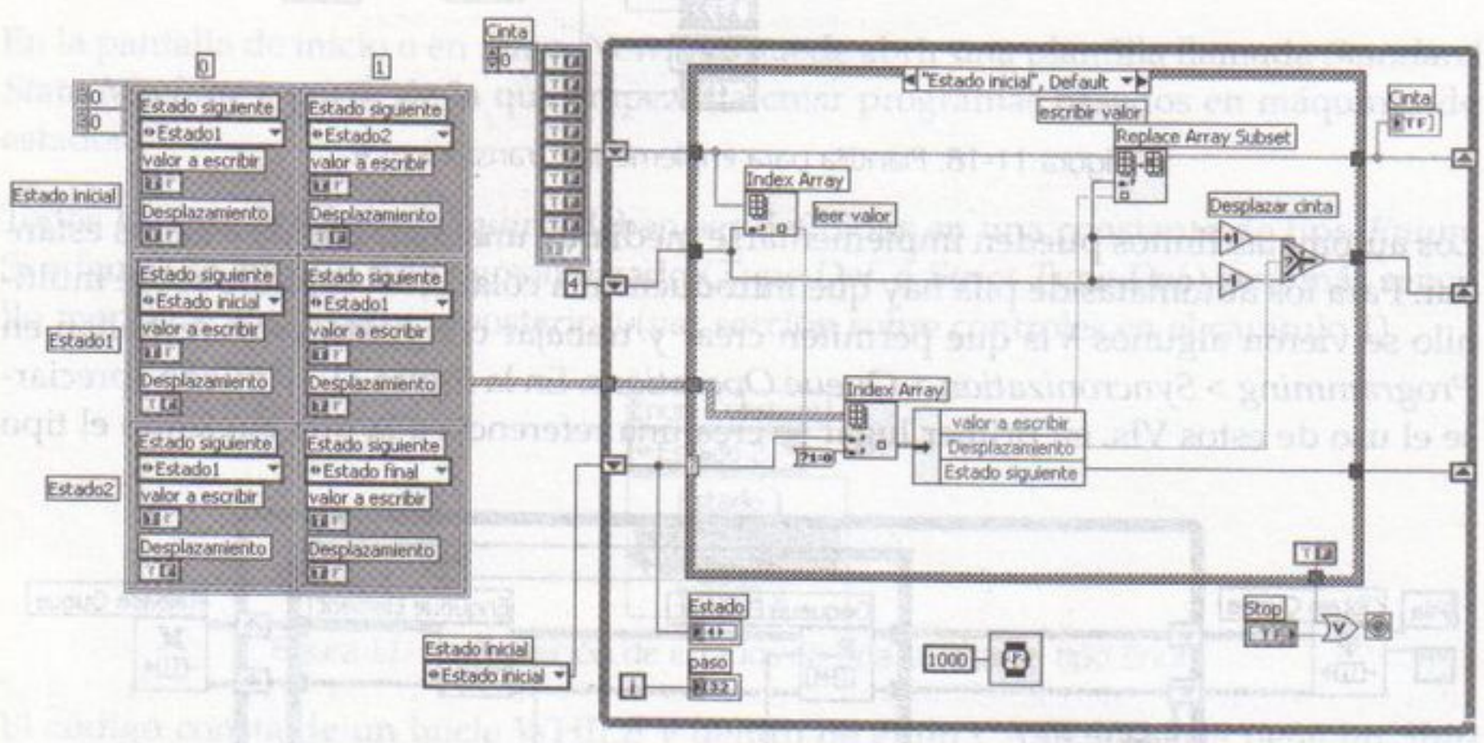


Figura 11-20. Máquina de Turing

National Instruments y algunos libros de la bibliografía también mencionan *Queued State Machine* (no confundir con el autómata de pila) como plantillas de diseño. En ellas simplemente se sustituye el *shift register* del bucle WHILE por una cola. En el interior del CASE el siguiente estado se almacenará en la cola y en la siguiente ejecución se extraerá de ella. La cola añade una opción más a la máquina: programar la ejecución de varios estados secuencialmente; por ejemplo desde el estado tres se puede hacer que a continuación se ejecuten el uno y el cuatro. La forma básica de esta máquina se puede ver en la figura 11-21.

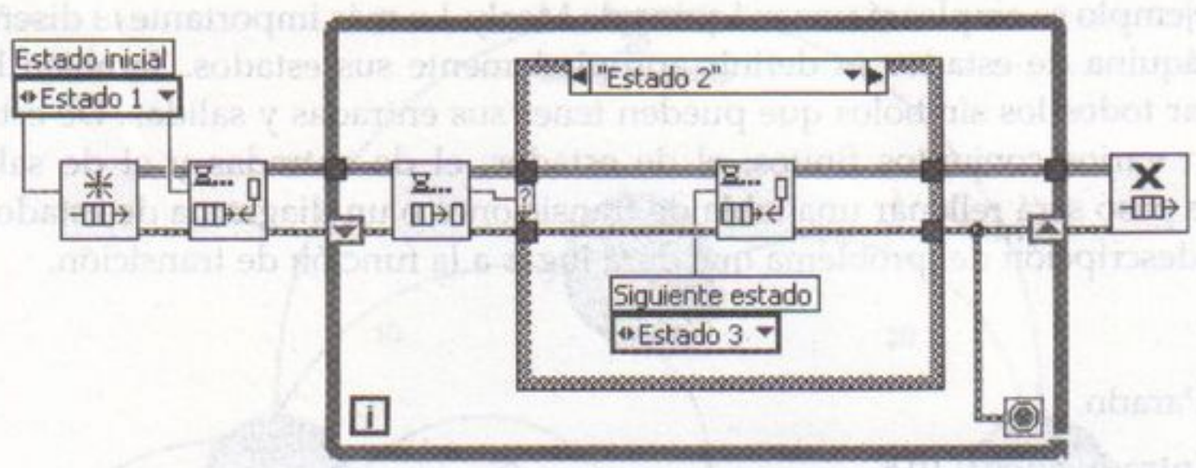


Figura 11-21. Máquina de estados Queued

National Instruments también habla de *Menu Driven State Machine*. Se trata de la misma estructura de la máquina estándar pero sustituyendo el control *Enum* de los estados por las opciones de un menú del interfaz de usuario. Sin negar la utilidad de las mismas, los autores de este libro no consideran realmente a estos programas como máquinas de estados ya que la decisión del siguiente estado al que debe pasar la máquina le viene impuesta por el usuario, no es propia de la máquina, por lo tanto se pierde el concepto de transición. Además es preferible el uso de la estructura EVENT para este tipo de aplicaciones. Un ejemplo puede verse en la figura 11-22.

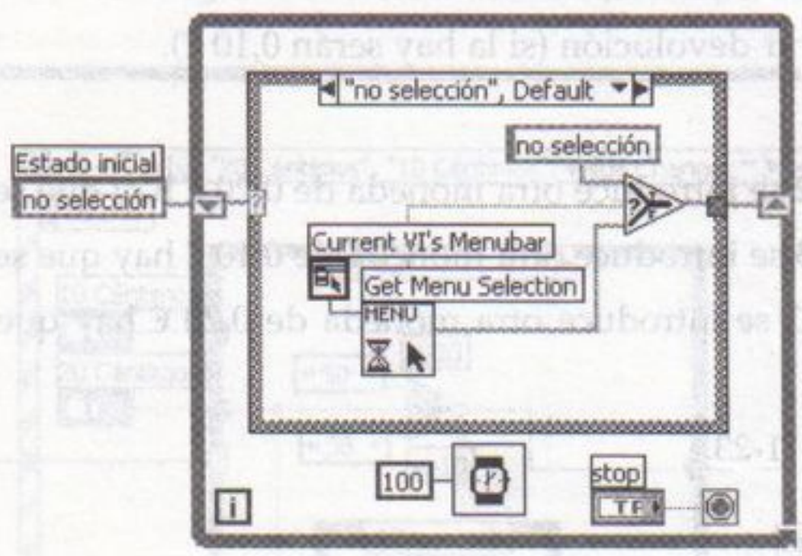


Figura 11-22. Máquina de estados Menu Driven

Para finalizar en lo que respecta a máquinas de estado también hay que comentar que existe un Toolkit que asocia un diagrama de estados a una estructura WHILE, de esta forma el código se va creando automáticamente a medida que el programador dibuja el diagrama de estados.

11.3.2 Ejemplo: Máquina expendedora

11.3.2.1 Explicación teórica

Se trata de diseñar una máquina expendedora que acepta monedas de 10 y 20 céntimos de euro y sirve refrescos que valen 40 céntimos.

En este ejemplo se empleará una máquina de Mealy. Lo más importante al diseñar cualquier máquina de estados es definir apropiadamente sus estados. También hay que identificar todos los símbolos que pueden tener sus entradas y salidas. De esta forma se tienen varios conjuntos finitos: el de estados, el de entradas y el de salidas. El siguiente paso será rellenar una tabla de transiciones o un diagrama de estados a partir de la descripción del problema que dará lugar a la función de transición.

Estados

- S0: Parado.
- S1: Introducidos 0,10 €.
- S2: Introducidos 0,20 €.
- S3: Introducidos 0,30 €. (una moneda de 0,10 y otra de 0,20).

Entradas

- Moneda de 0,10 €.
- Moneda de 0,20 €.

Salidas

- Señal para obtener la bebida.
- Señal para indicar devolución (si la hay serán 0,10 €).

Acciones

- Si estando en S2 se introduce otra moneda de 0,20 € hay que servir la bebida.
- Si estando en S3 se introduce otra moneda de 0,10 € hay que servir la bebida.
- Si estando en S3 se introduce otra moneda de 0,20 € hay que servir la bebida y devolver 0,10 €.

Transiciones: figura 11-23.

11.3.2.2 Código

Como la máquina de estados depende del usuario se implementará mediante eventos. Hasta que no se introduce una nueva moneda no se pasa al siguiente estado, esta es la razón por la que se ha elegido una máquina de Mealy, si se hubiese hecho de Moore se necesitarían dos estados más para las salidas.

Cada vez que se presionen unos botones que simbolizan las dos posibles monedas que se introducirán en la máquina, se ejecutará la máquina de estados. Para saber qué moneda se ha introducido sólo hay que comprobar si es la de 20 céntimos, si no lo es será la de 10, esto sirve para calcular el siguiente estado al que pasará la máquina mediante un nodo *Select*.

En los estados S2 y S3 se ha añadido un segundo CASE que sirve para calcular las salidas.

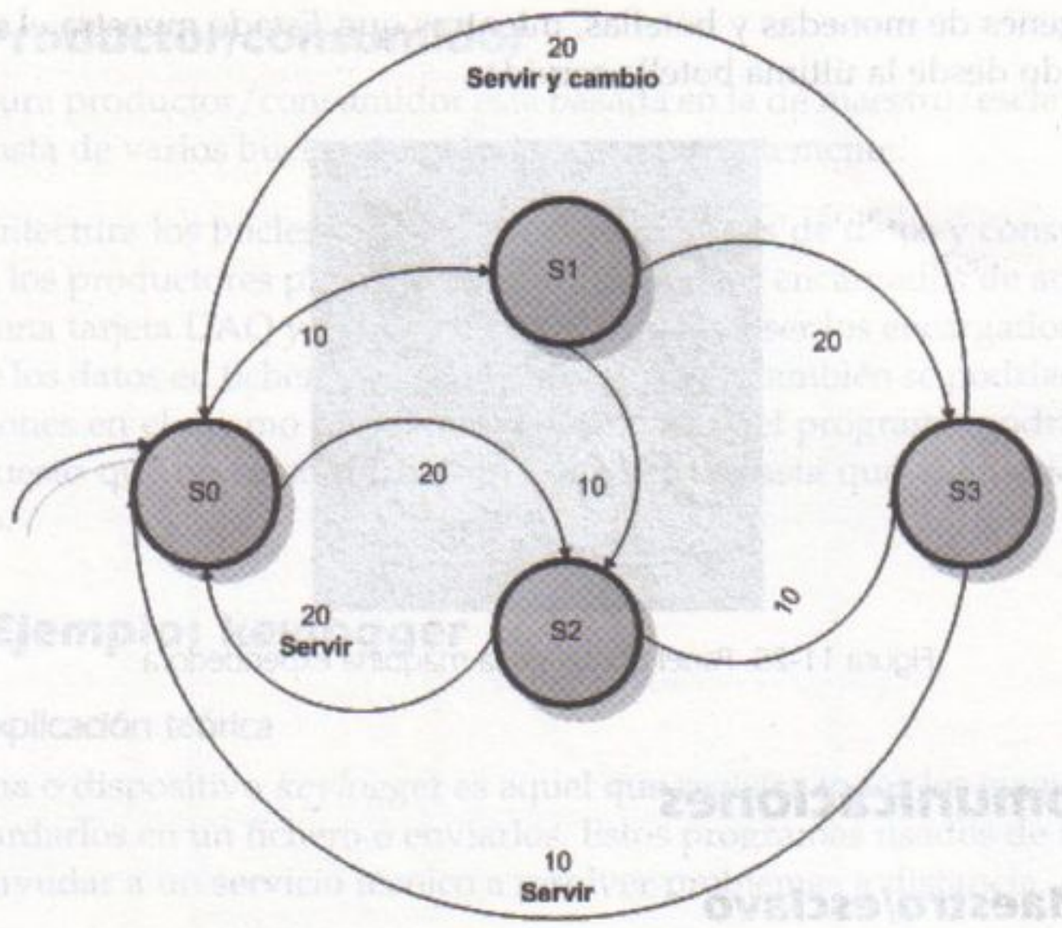


Figura 11-23. Diagrama de estados de la máquina expendedora

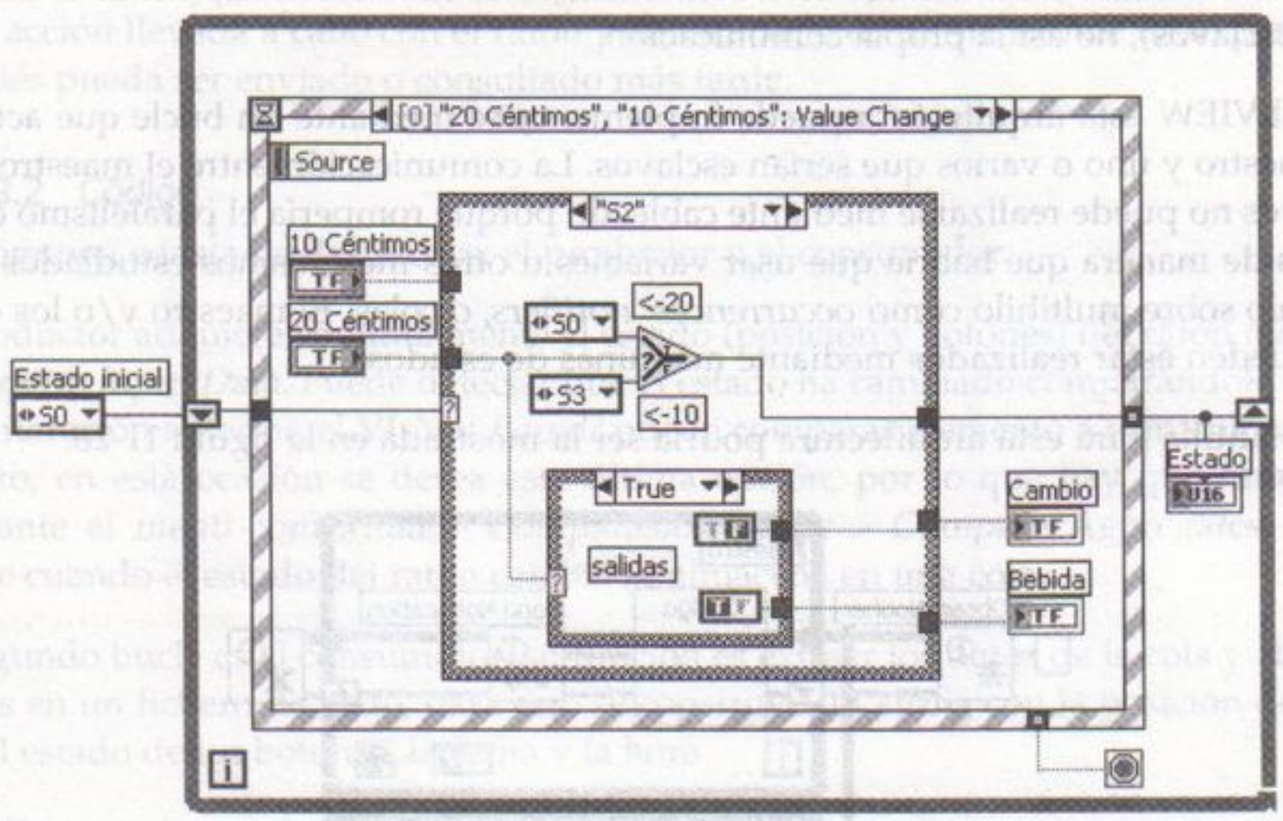


Figura 11-24. Código que implementa la máquina expendedora

11.3.2.3 Resultados

En la figura 11-25 se aprecia una captura de pantalla del Panel Frontal después de haber introducido 0,50 €. Los botones e indicadores booleanos que se pueden ver en el código de la figura 11-24 (*10 Céntimos, 20 Céntimos, Cambio* y *Bebida*) se han personaliza-

do con imágenes de monedas y botellas, mientras que *Estado* muestra el valor que se ha introducido desde la última botella servida.

Figura 11-25. Panel frontal de la máquina expendedora

11.4 Comunicaciones

11.4.1 Maestro/esclavo

La estructura de maestro/esclavo (*master/slave*) se suele usar para ejecutar tareas de una forma asíncrona. El control de la comunicación es unidireccional (desde el maestro a los esclavos), no así la propia comunicación.

En LabVIEW esta arquitectura puede implementarse mediante un bucle que actuaría de maestro y uno o varios que serían esclavos. La comunicación entre el maestro y los esclavos no puede realizarse mediante cableado porque rompería el paralelismo de los bucles de manera que habría que usar variables u otros mecanismos estudiados en el capítulo sobre multihilo como *occurrences, notifiers,* o colas. El maestro y/o los esclavos pueden estar realizados mediante máquinas de estados.

Una plantilla para esta arquitectura podría ser la mostrada en la figura 11-26.

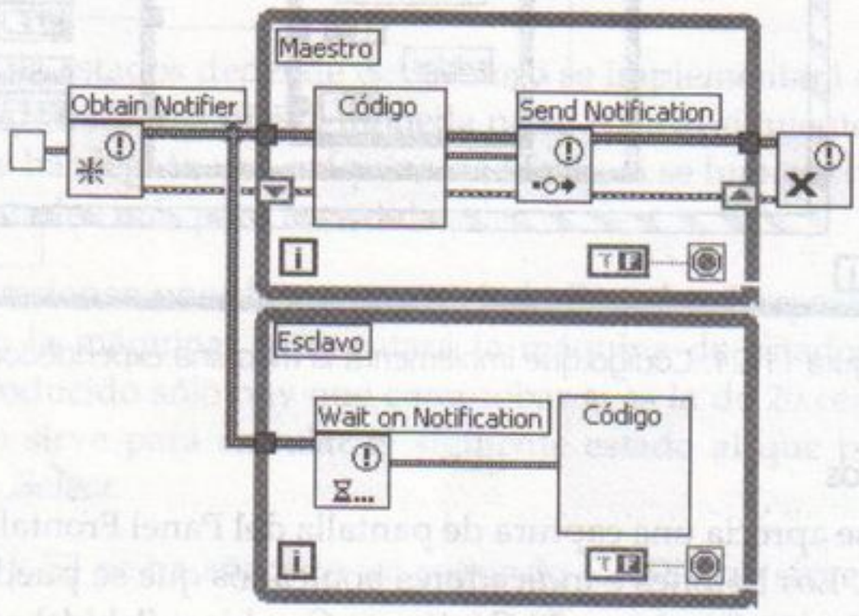


Figura 11-26. Plantilla de una comunicación maestro/esclavo

11.4.2 Productor/consumidor

La arquitectura productor/consumidor está basada en la de maestro/esclavo y, al igual que ella, consta de varios bucles ejecutándose concurrentemente.

En esta arquitectura los bucles se dividen en productores de datos y consumidores de los mismos; los productores pueden ser, por ejemplo, los encargados de adquirir datos a través de una tarjeta DAQ y los consumidores pueden ser los encargados de guardar cada uno de los datos en ficheros. En el ejemplo anterior también se podrían realizar las dos operaciones en el mismo bucle, pero la eficiencia del programa podría caer enormemente puesto que no se adquiriría un nuevo dato hasta que el anterior haya sido almacenado.

11.4.3 Ejemplo: keylogger

11.4.3.1 Explicación teórica

Un programa o dispositivo *keylogger* es aquel que registra todos los eventos del teclado para guardarlos en un fichero o enviarlos. Estos programas usados de forma benigna pueden ayudar a un servicio técnico a resolver problemas a distancia.

El programa de este ejemplo sigue la misma filosofía que un *keylogger*, pero con eventos de ratón en lugar de teclado. Lo que debe hacer es almacenar en un fichero cualquier acción llevada a cabo con el ratón y almacenarlo en un fichero de texto para que después pueda ser enviado o consultado más tarde.

11.4.3.2 Código

El programa consta de dos bucles: el productor y el consumidor.

El productor adquiere continuamente el estado (posición y botones) del ratón mediante *Acquire Input Data*. Puede detectar que el estado ha cambiado comparándolo con el de la iteración anterior, el VI *Not Equal?* puede comparar elemento a elemento o todo el dato, en esta ocasión se desea esta última opción, por lo que hay que cambiarla mediante el *menú contextual > Comparasion Mode > Compare Aggregates*. Finalmente cuando el estado del ratón cambia se almacena en una cola.

El segundo bucle es el consumidor, su función es extraer los datos de la cola y almacenarlos en un fichero de texto, para esto se construye un *string* con la posición del cursor, el estado de los botones, la fecha y la hora.

El código puede verse en la figura 11-27.

11.4.3.3 Resultado

Después de ejecutar el programa, para ver el resultado debe abrirse el fichero generado, un posible resultado es:

```
Coord: 298 324 0 Bot: FALSE FALSE FALSE FALSE Hora 03/04/2006 12:23:00
```

```
Coord: 298 324 0 Bot: FALSE TRUE FALSE FALSE Hora 03/04/2006 12:23:00
Coord: 296 324 0 Bot: FALSE TRUE FALSE FALSE Hora 03/04/2006 12:23:00
Coord: 295 323 0 Bot: FALSE TRUE FALSE FALSE Hora 03/04/2006 12:23:00
Coord: 295 324 0 Bot: FALSE FALSE FALSE FALSE Hora 03/04/2006 12:23:00
Coord: 284 325 0 Bot: FALSE FALSE FALSE FALSE Hora 03/04/2006 12:23:00
Coord: 266 319 0 Bot: FALSE FALSE FALSE FALSE Hora 03/04/2006 12:23:00
```

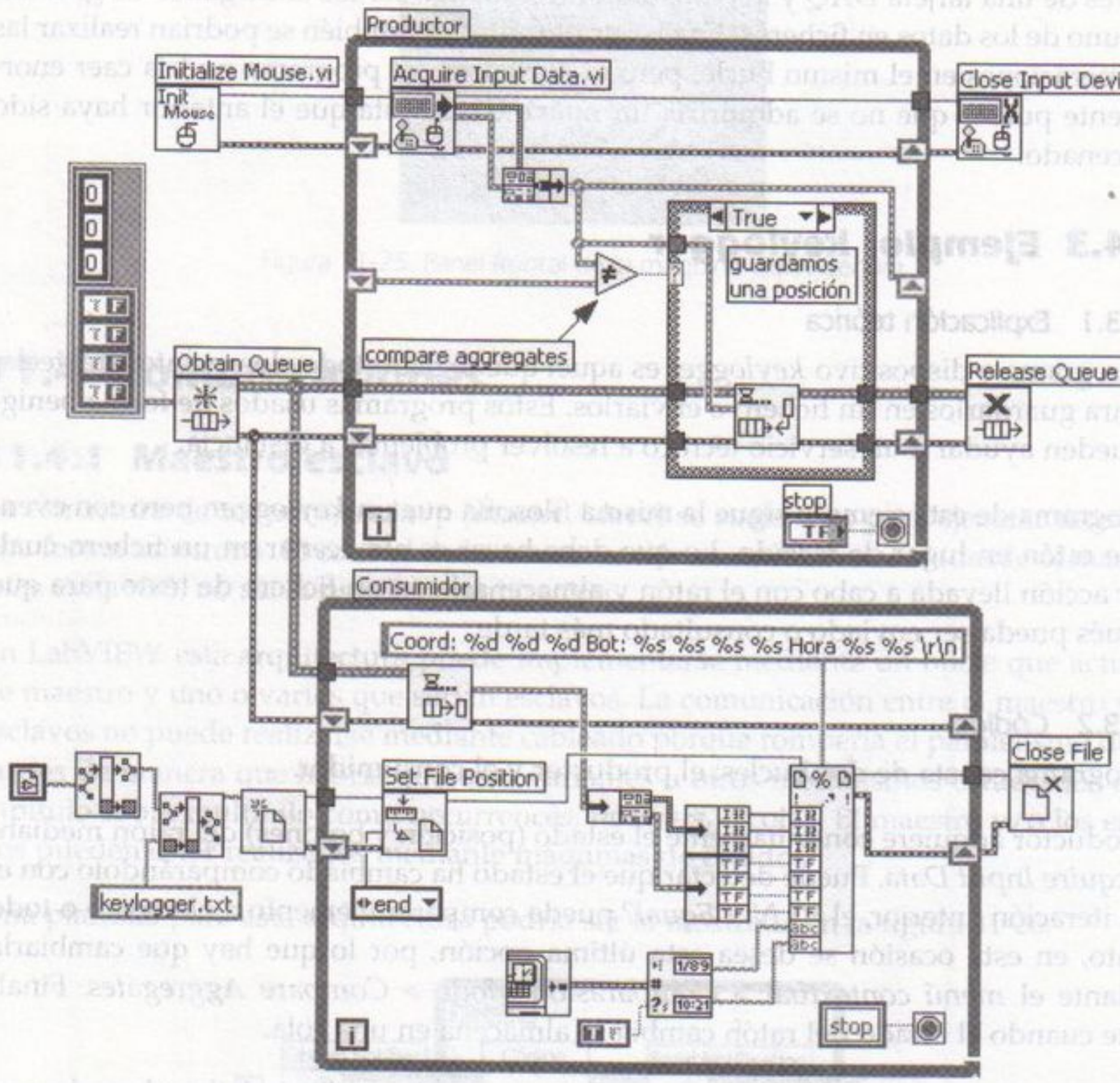


Figura 11-27. *Keylogger*

11.5 Orientado a objetos

Existen diversos paradigmas en la programación. En la programación "clásica", también es llamada procedural o imperativo, se basa el comportamiento del programa en llamadas a funciones y los datos en variables, ambos independientes entre sí. En la programación por flujo de datos se hace hincapié en las transformaciones que sufren los datos desde la entrada a la salida.

La programación orientada a objetos (OOP) es otro paradigma de programación que gira en torno al concepto de objeto, que es una entidad que agrupa tanto el estado (datos) como el comportamiento (procedimientos).

El primer lenguaje orientado a objetos fue *Simula 67* y, basado en él, Xerox creó más tarde *Smalltalk*. En los años ochenta se popularizó este método de programación debido principalmente al uso de las interfaces de usuario, en los que la OOP es especialmente apropiada, y a una extensión de C para soportar a la OOP: C++.

Los principales conceptos que se manejan al trabajar con OOP se pueden resumir en:

- **Abstracción**: se pueden usar los objetos sin entrar en detalles de cómo están hechos. Cada generación de lenguajes de programación ha aumentado el nivel de abstracción.
- **Encapsulación**: cada objeto es un conjunto de otros componentes (datos y procedimientos) y se puede considerar como un ente en sí mismo. Algunos componentes pueden estar "ocultos" dentro del objeto, de manera que sólo se podría acceder a ellos a través de los procedimientos que el objeto posee a tal efecto.
- **Herencia**: un objeto está descrito por medio de una clase. Puede haber clases basadas en otras formando así una estructura jerárquica. Un objeto hereda las propiedades y comportamiento de las clases en las que se basa.
- **Polimorfismo**: es el equivalente a la sobrecarga de funciones o de operadores en lenguajes procedurales. Se trata de asociar el mismo nombre a comportamientos diferentes y, dependiendo del tipo de datos con que se invoquen, se seleccionará uno u otro.

Por otra parte, al diseñar un programa basado en OOP los componentes fundamentales serán:

- **Clases**: es la descripción de un objeto, una plantilla que se aplicará a sus instancias. En su definición agrupa los datos y procedimientos que tendrán los objetos de esa clase. Como es una mera descripción se trata de un concepto abstracto que describe cómo son los datos, pero al no ser una entidad concreta, éstos no pueden tener ningún valor.
- **Objetos**: es una instancia de una clase. Los datos que contiene un objeto sí pueden tener valores. Nótese la diferencia entre clase y objeto: la clase es simplemente una descripción y el objeto es una entidad definida de la forma que indica la clase. Un ejemplo podría ser la idea «coche» (clase) y un coche real (objeto).
- **Métodos**: son los procedimientos que contiene el objeto y determinan su comportamiento. Es el equivalente a las funciones de los lenguajes procedurales. La llamada o invocación de un método se llama mensaje, los mensajes se pueden pasar de un objeto a otro o internamente dentro del mismo objeto. Siguiendo con el ejemplo anterior, la clase «coche» tendría los métodos «acelerar» o «frenar».
- **Datos**: los datos que componen los objetos se clasifican dependiendo de si se puede acceder a ellos desde el exterior del objeto o no (únicamente los pueden modificar

los procedimientos del objeto). Los que pueden ser accedidos desde el exterior se llaman propiedades o atributos. Los que no pueden ser accedidos desde el exterior describen el estado del objeto. Son el equivalente a las variables en la programación procedural. En el ejemplo del «coche» podría ser el «color».

En la figura 11-28 se ha tratado de representar el concepto de OOP. A la izquierda se puede ver la definición de una clase, que está compuesta de varios datos y varios procedimientos, a la derecha hay tres objetos de esa misma clase que se están comunicando entre sí por medio de mensajes.

Para crear un objeto éste debe inicializarse, para ello se emplea un método especial llamado constructor cuya función es inicializar los datos de ese objeto. Otro método especial es el destructor, que libera la memoria ocupada por un objeto cuando ya no es necesario. En algunos casos es el propio lenguaje el que crea el constructor de forma automática.

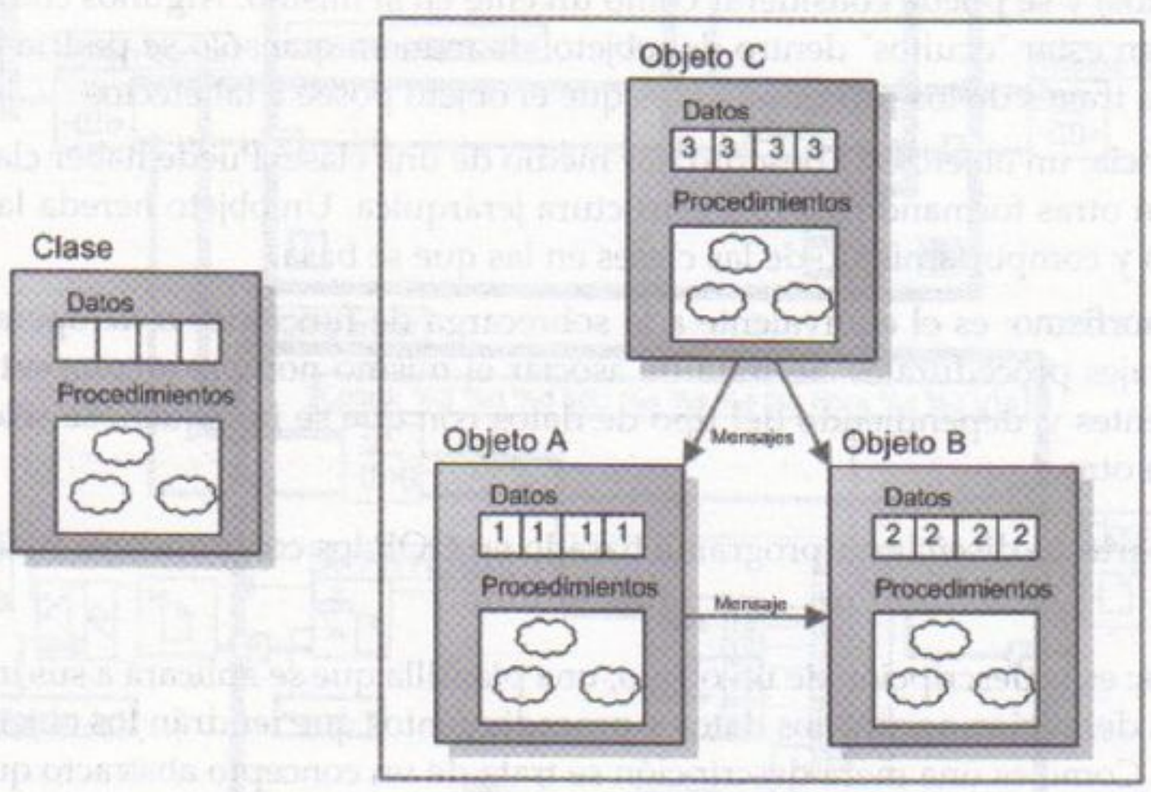


Figura 11-28. Concepto de programación orientada a objetos

Al aprender programación orientada a objetos se requiere del programador una nueva forma de pensar a la que no está habituado, algo costoso al principio. Para entender mejor todos estos conceptos se partirá del siguiente ejemplo:

En una empresa se quiere desarrollar un programa que gestione las fichas de los empleados, para empezar se puede crear una clase genérica llamada empleado que como atributos tendría `nombre`, `departamento` y `sueldo` y como procedimientos puede tener `subir_sueldo`, `bajar_sueldo`, `trasladar` y `nomina`, por simplicidad las subidas y bajadas de sueldo son del 10%. En seudo-código podría ser:

```
CLASE empleado:
    nombre, departamento, sueldo
    subir_sueldo:
```

```
        sueldo = sueldo * 1.1
    bajar_sueldo:
        sueldo = sueldo * 0.9
    trasladar (nuevoDep):
        departamento = nuevoDep
    nomina
        RETURN sueldo
FIN CLASE
```

Para crear una ficha de un empleado sólo habría que instanciar la clase y crear el objeto correspondiente. En el siguiente seudo-código se crea un objeto mediante la palabra reservada NUEVO, el nuevo objeto se llama `empleado123`, además se pasan los parámetros de inicialización que el constructor usará para dar valor a las variables en el momento de crear el objeto:

```
empleado123 = NUEVO empleado ("Juan", "Marketing", 1000)
```

Una vez creado el objeto ya se pueden emplear sus métodos para trabajar con él, por ejemplo si se quiere aumentar el sueldo y trasladar de departamento al empleado anterior se podría hacer:

```
empleado123->subir_sueldo
empleado123->trasladar ("RR. HH.")
```

Para sustituir al empleado anterior la empresa puede contratar a uno nuevo (empleado321) que sería instanciado de la misma forma. Con esto se consigue tener dos objetos de la misma clase pero con valores diferentes:

```
empleado321 = NUEVO empleado ("Pepe", "Marketing", 500)
```

Además se puede partir en la clase anterior para crear otras nuevas. Por ejemplo se podría definir la clase directivo que heredará las propiedades y métodos de la clase empleado, también puede añadir la propiedad años para indicar el tiempo de antigüedad, añadir un método para modificar la propiedad anterior y redefinir el método subir_sueldo para que sus subidas de sueldo sean superiores a la del resto de empleados y que además dependan de la antigüedad en la empresa:

```
CLASE directivo BASADA_EN empleado:
    años
    subir_sueldo:
        sueldo = sueldo * 1.2 + 100 * años
    setAños (nuevosAños):
        años = nuevosAños;
FIN CLASE
```

Un lenguaje orientado a objetos permite implementar algunas o todas estas características. Muchos lenguajes como C++ o Java proporcionan herramientas expresamente

diseñadas para esta tarea, otros lenguajes no lo hacen y, si se quieren usar, debe ser el programador el que adapte su programa a los planteamientos de la OOP.

La OOP es un método de programación que presenta una serie de beneficios, como son:

- **Mantenimiento**: todos los objetos de una misma clase son creados a partir de la misma descripción, por lo que los cambios realizados en la clase se reflejan inmediatamente en todos sus objetos.
- **Reutilización**: los programas orientados a objetos son muy modulares, los objetos creados para un programa, si están bien definidos, pueden usarse en otros programas sin necesidad de alterar el código. Además dentro del mismo programa la herencia también proporciona un alto nivel de reutilización del código.
- **Trabajo colaborativo**: al ser una forma de programar más modular es más apropiado cuando se tiene un proyecto en el que trabajan varios programadores, de esta forma se pueden repartir mejor las tareas.

Cuando se va a empezar un proyecto orientado a objetos el primer y más importante paso es identificar correctamente las clases de objetos que se van a necesitar. El siguiente paso es el diseño de las propiedades y métodos que tendrán los objetos y que les permitirán comunicarse unos con otros. A continuación se empezarían a implementar los métodos y finalmente se combinarían todos los objetos para formar el programa. Hay lenguajes como UML (*Unified Modeling Language*) que ayudan en este proceso de análisis y creación de programas.

Por último también hay que decir que la OOP también tiene desventajas como una depuración más complicada y una mayor curva de aprendizaje al principio.

11.5.1 Programación orientada a objetos en LabVIEW

Realmente se puede considerar que ya se ha usado la OOP en LabVIEW desde el principio de este libro, no hay más que recordar los nodos de propiedades de controles e indicadores y el capítulo sobre VI Server. Por otra parte también se verán conceptos de OOP aplicados en LabVIEW en los próximos capítulos sobre ActiveX y .NET. Además se ha aplicado la filosofía de OOP en otras varias situaciones: comunicaciones TCP, manejo de ficheros, adquisición de datos, etc.

Por ejemplo las funciones para manejar ficheros se pueden considerar como métodos para los objetos de una hipotética clase fichero. En el ejemplo de la figura 11-29 se puede suponer la primera función como un constructor de un objeto tipo fichero, esta función devuelve el objeto encapsulado en un dato de tipo *refnum*, las funciones *Get File Size*, *Set File Position* y *Read from Text File* serían los métodos del objeto, el primero devolvería el valor de la propiedad *Size*, el segundo escribiría sobre la propiedad *Position* y el tercero devolvería el valor de otra propiedad, en este caso el propio contenido del fichero, en este ejemplo *Close File* sería el método destructor.

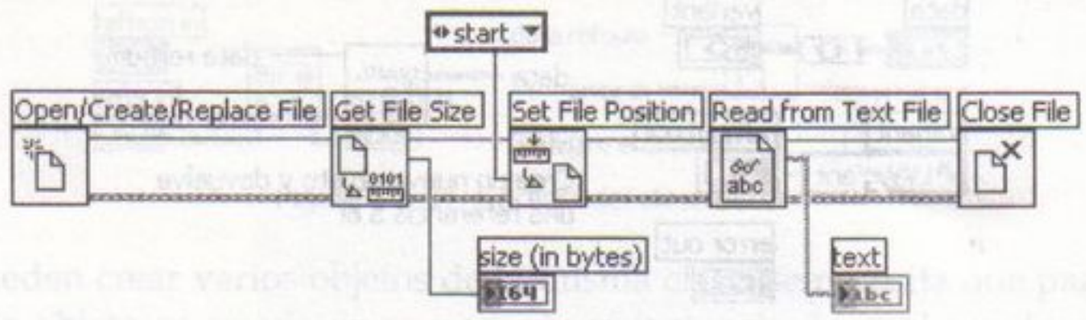


Figura 11-29. Clase fichero

En versiones anteriores a la 8.20, LabVIEW no proporcionaba facilidades para que los programadores pudiesen usar el paradigma de la programación orientada a objetos; el programador de LabVIEW no podía crear clases y objetos a su antojo (excepto con XControls en la versión 8.0). No obstante a pesar de que el lenguaje no ofrecía estas facilidades, un programador podía seguir una serie de reglas para aplicar la metodología de OOP; evidentemente se perdían algunas de las características mencionadas antes, pero en algunos casos merecía la pena el esfuerzo. A continuación se presenta un método manual básico que puede usarse en versiones antiguas de LabVIEW.

En primer lugar se debe crear la clase, ésta es una descripción del conjunto de propiedades y métodos que tendrán los objetos. En OOP las propiedades son el equivalente de las variables y los métodos de las funciones, de esta forma los métodos pueden implementarse mediante VIs y las propiedades mediante variables agrupadas en un *cluster*, por ejemplo puede crearse el control de la figura 11-30.

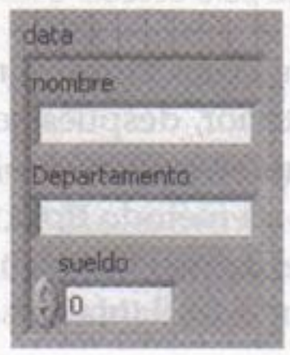


Figura 11-30. Cluster para construir una clase

Para crear un nuevo objeto se invocará al constructor, este método es un VI que contendrá un control de tipo *cluster* como el descrito antes, el cual tiene los datos del objeto; para la salida de este VI, con el fin de proporcionar encapsulación, sólo se devolverá una referencia al control. El código de un constructor podría ser el mostrado en la figura 11-31. Lo más destacado es el empleo de datos tipo *Variant*, con esto se consigue que el constructor sea lo más genérico posible, así aunque los datos de la clase cambien, el tipo de la referencia siempre permanecerá constante.

Los métodos necesitarán leer y/o escribir los datos del objeto, pero no podrán hacerlo directamente al estar encapsulado, por lo tanto habrá que crear un interfaz que proporcione acceso a los datos; para ello se dispone de la referencia al *cluster* situado en el VI del constructor. Para leer y escribir un dato de un VI desde otro a través de esta referencia basta con usar un *Property Node* con la propiedad *Value*, como se indica en la

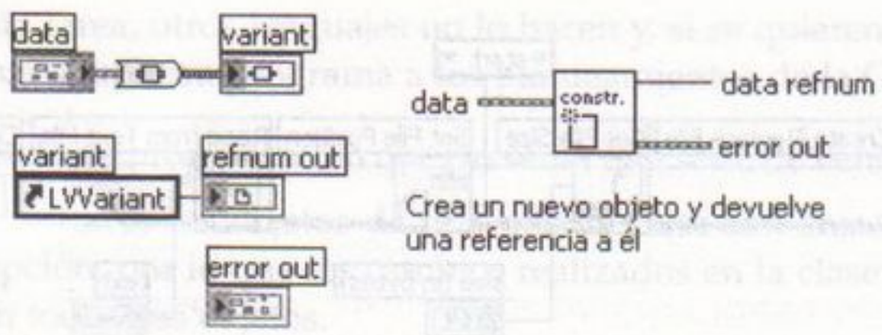


Figura 11-31. Código de un constructor

figura 11-32. Este VI sólo debe usarse dentro de los métodos del objeto y nunca fuera de ellos.

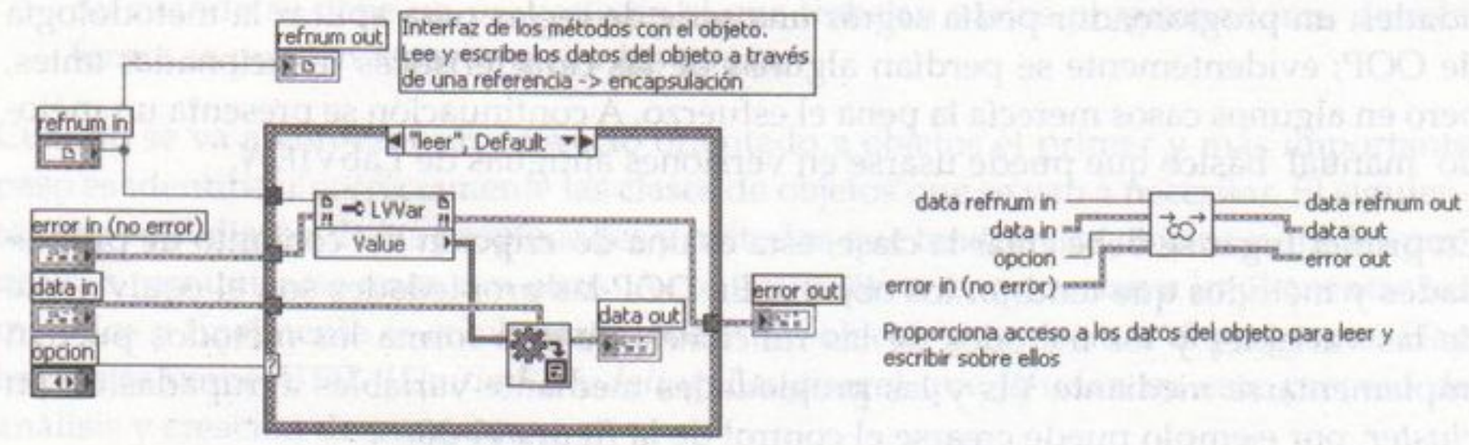


Figura 11-32. Interfaz para acceder a los datos de un objeto

Todos los métodos tendrán la misma estructura: si necesitan leer los datos se accederá al objeto al principio con el VI anterior, después se ejecutará el código del método y finalmente se guardarán los datos nuevamente en el objeto si es necesario. En la figura 11-33 se puede ver la plantilla para un método típico, en este programa habría que añadir las entradas y salidas con el exterior que fuesen necesarias, también habría que crear el código que debe ejecutar el método y adaptar los nodos *Bundle By Name* al objeto.

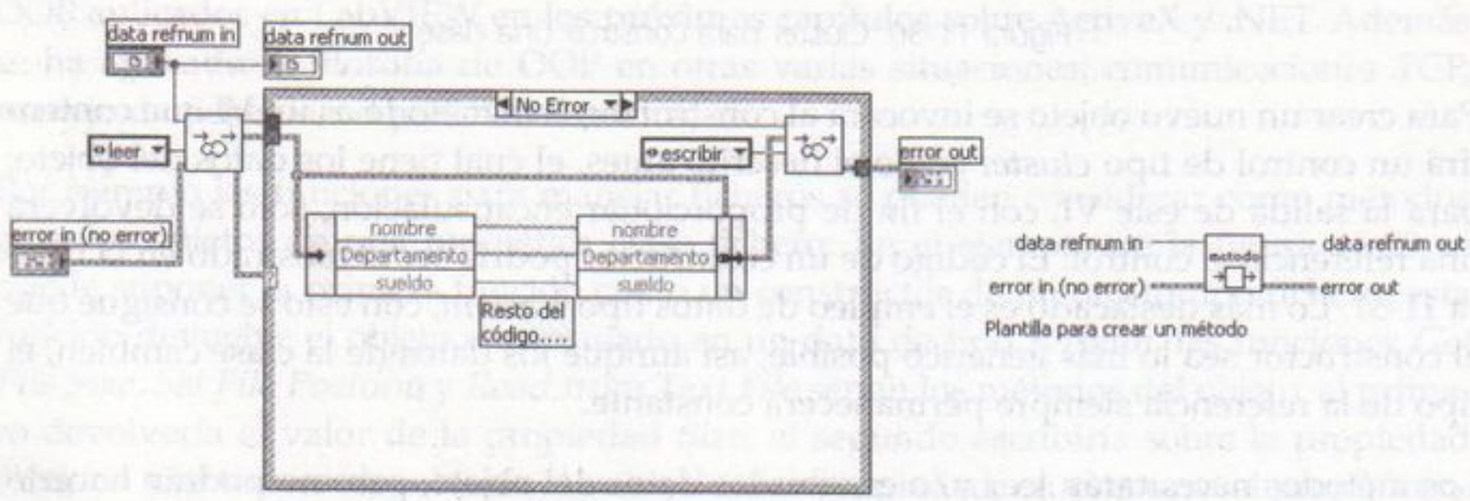


Figura 11-33. Plantilla para crear un método

El método del destructor sería incluso más sencillo que el del constructor, su código se muestra en la figura 11-34.

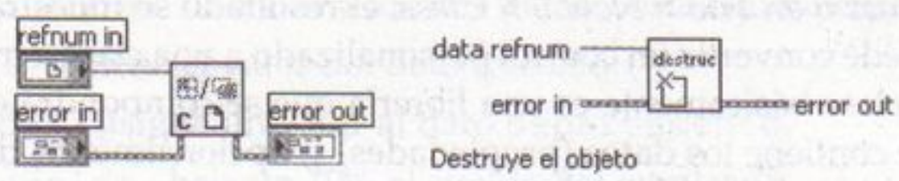


Figura 11-34. Código del destructor

Como se pueden crear varios objetos de la misma clase, se necesita que para cada llamada a cada objeto se pueda crear una nueva instancia, lo cual puede conseguirse haciendo que estos VIs sean reentrantes. Finalmente se puede crear una librería con todos los métodos del objeto. Un ejemplo del uso de esta metodología puede ser el mostrado en la figura 11-35.

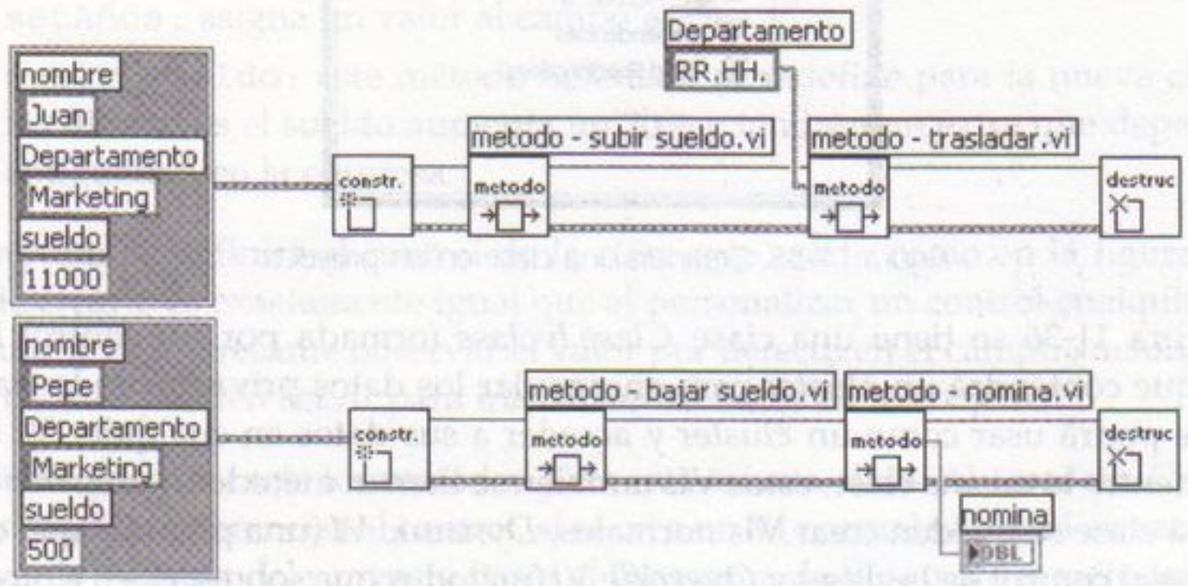


Figura 11-35. Ejemplo del uso de clases

Mucho antes de la versión 8.x aparecieron herramientas que extendían la capacidad de LabVIEW para aproximarse a la OOP. Básicamente son asistentes y plantillas que ayudan a crear programas siguiendo filosofías muy parecidas a la anterior. A continuación se verá una breve descripción de dos de ellas: OpenGOOP y GOOP.

Las siglas GOOP significan *Graphical Object Oriented Programming*. La empresa sueca ENDEVO junto con National Instruments desarrollaron hace años *GOOP Toolkit*. Este Toolkit añade un asistente al que se puede acceder desde el menú *Tools > LabVIEW GOOP Wizard*. Este asistente permite crear clases nuevas y métodos a través de una serie de pasos guiados.

OpenGOOP es una colección de plantillas de código libre y gratuito creada por la comunidad de desarrolladores de OpenG. Estos VIs se basan en llamadas dinámicas a un VI reentrante que sería el equivalente a nuestro constructor, también proporciona mecanismos de sincronización para leer y escribir los datos basados en semáforos. Además cuenta con una serie de plantillas para facilitar la creación del código de los métodos.

La principal mejora de la versión 8.20 respecto a la anterior es el soporte a la OOP. Dentro de un proyecto se puede crear una nueva clase desde la opción del *New > Class*

del menú contextual o en *File > New... > Class*, el resultado se muestra en la figura 11-36 (también se puede convertir un control personalizado a una clase a través de su menú contextual). Una clase básicamente es una librería que se compondrá obligatoriamente de un control que contiene los datos (propiedades) y, opcionalmente, de VIs (métodos).

Figura 11-36. Creando una clase en un proyecto

En la figura 11-36 se tiene una clase *Clase.lvclass* formada por un control llamado *clase.ctl* que contendrá un *cluster* para encapsular los datos privados de la clase. Este control se podrá usar como un *cluster* y acceder a sus datos en cualquier VI siempre que esté dentro la misma clase, estos VIs también se llaman métodos o *member VI*; dentro de una clase se pueden crear VIs normales, *Dynamic VI* (una plantilla que contiene referencias al control de la clase) y *Override VI* (métodos que sobrescriben a otros cuando hay herencia). En VIs exteriores a la clase, el control que contiene los datos funcionará de forma parecida a una referencia.

Desde las propiedades de la clase (*.lvclass) se podrá definir su icono, documentación, etc. También se definirán qué VIs son privados o públicos en la clase (*Item Settings*), el color y aspecto de los cables (*Wire Apperance*), y la herencia entre clases (*Inheritance*).

Al igual que con la jerarquía de los subVIs, también se puede ver la jerarquía de clases en *View > LabVIEW Class Hierarchy*.

11.5.1.1 Ejemplo: Fichas de empleados

Este ejemplo consistirá básicamente en hacer lo mismo que se describió con seudocódigo al principio de esta sección. El programa estará compuesto de una clase llamada empleado y, a partir de ella, se creará otra llamada directivo.

La primera clase estará compuesta de:

- Datos: `nombre`, `departamento` y `sueldo`. Los dos primeros son *strings* y el tercero numérico.
- Métodos:
  - `setAll`: asigna el valor de los tres datos.

- `getAll`: obtiene un valor a los tres datos. Sería equivalente a un constructor.
- `nomina`: obtiene el valor del dato `sueldo`.
- `trasladar`: asigna un valor al dato `departamento`.
- `bajar_sueldo`: baja un 10% el sueldo del empleado.
- `subir_sueldo`: aumenta un 10% el sueldo del empleado.

La clase `directivo` heredará los atributos y métodos de la clase `empleado`, por lo tanto se reutilizarán los anteriores componentes, además habrá los siguientes cambios:

- Datos: se añade el atributo años, que es el tiempo que se lleva en la empresa.
- Métodos:
  - `setAños`: asigna un valor al campo años.
  - `subir_sueldo`: este método heredado se redefine para la nueva clase; para los directivos el sueldo aumenta un 20% y tendrán un extra que depende de la antigüedad en la empresa.

Para empezar se definirá el control de la clase `empleado` como en la figura 11-37, la forma de crearlo es exactamente igual que al personalizar un control cualquiera, véase el capítulo 1. Es interesante observar el valor por defecto en el campo *sueldo*, los valores por defecto pueden servir para inicializar los datos de los objetos.

El siguiente paso es crear los métodos para esta misma clase. Al crear un *Dynamic VI* ya se tendrán las referencias al control, si se crea un VI normal se podrá arrastrar el control desde la ventana del proyecto hasta el Diagrama de Bloques del método. Como se ha dicho antes, el control de una clase actúa como un *cluster* normal cuando se usa con VIs de la misma clase, por lo tanto se podrán usar las funciones para construir y desglosar *clusters* (*Bundle* y *Unbundle* respectivamente). En la figura 11-38 se puede ver el código a la izquierda y el terminal a la derecha de tres de los métodos descritos antes: (a) `getAll`, (b) `subir_sueldo` y (c) `trasladar`; el resto de métodos tienen un código muy parecido a éstos.

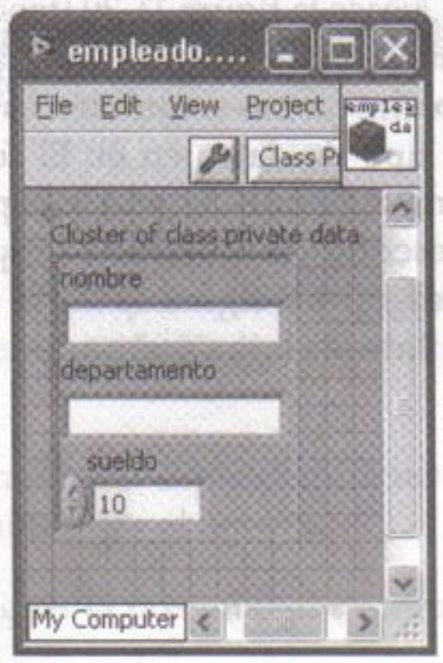


Figura 11-37. Control de la clase 'empleado'

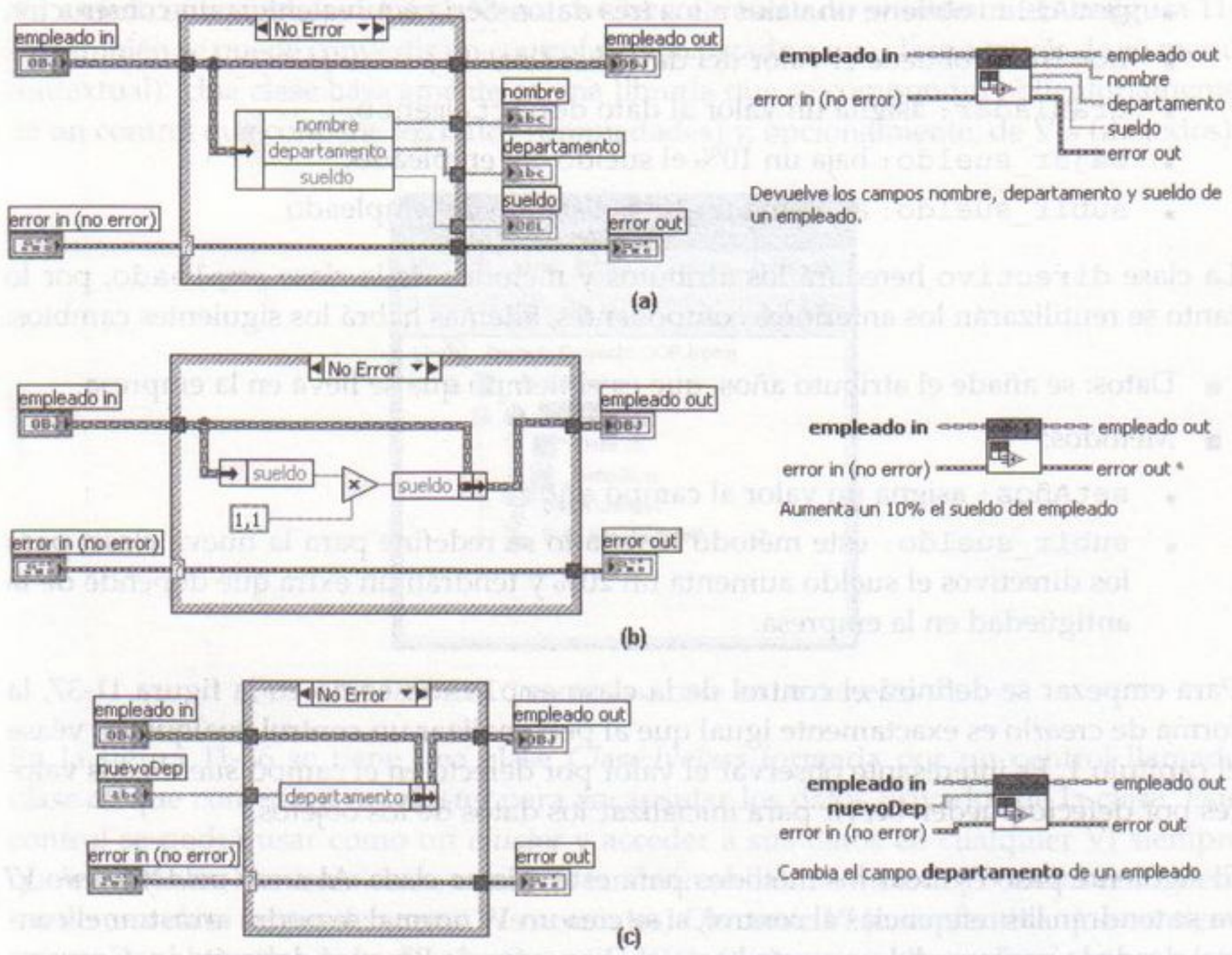


Figura 11-38. Métodos de la clase empleado: (a) getAll, (b) subir_sueldo y (c) trasladar

Una vez creados todos los métodos ya está acabada la clase `empleado`, después se procederá de igual manera con la clase `directivo`. En el control que define los datos de la clase únicamente se añadirá un campo numérico llamado *años*. El método `setAños` se construye de forma parecida a los anteriores. Para indicar la herencia hay que acudir a las propiedades de la clase y en *Inheritance* se cambiará la herencia para que dependa de la clase `empleado`, véase la figura 11-39 (tener en cuenta que una clase sólo puede tener herencia de una única clase). Cuando una clase depende de otra se podrán usar los métodos de la clase "padre" como si también estuvieran en la "hija" porque el tipo de cable no determina el tipo de datos en él. El último paso para completar la segunda clase será redefinir el método `subir_sueldo`, para ello se creará un *Override VI* indicando el método a sobrescribir (el conector y propiedades de los dos VIs deben ser iguales), puede verse su código en la figura 11-40 en el que se han usado los métodos `nomina`, `getAll` y `setAll` de la clase `empleado`.

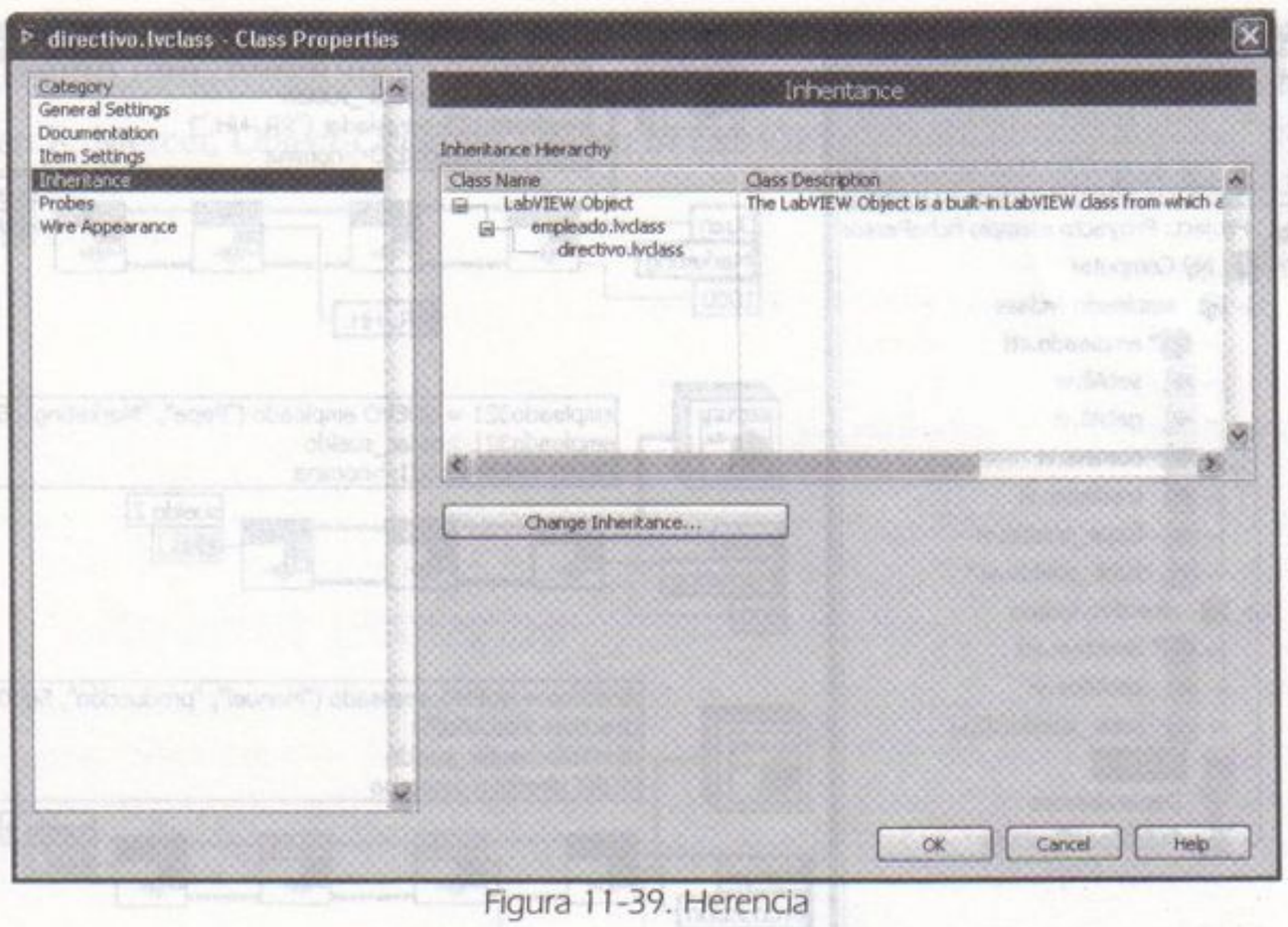


Figura 11-39. Herencia

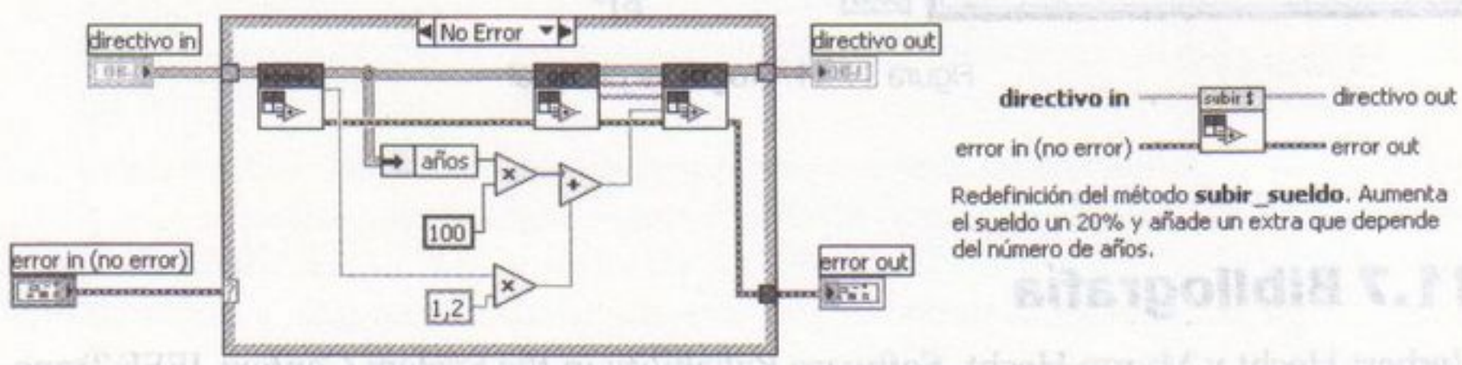


Figura 11-40. Método subir_sueldo en la clase directivo

Una vez creadas totalmente las dos clases, por fin se podrá realizar el programa principal en el que se construirán dos objetos de la clase empleado y uno de la clase `directivo`, sobre estos objetos se aplicarán diversos métodos como se puede ver en la figura 11-41 (en la página siguiente).

11.6 Ejercicios

1. Modificar el programa de la máquina de Turing para que si se intenta acceder a posiciones que no existen en el *array* (índice con valor -1), éste se redimensione adecuadamente.

2. Realizar el ejemplo de la máquina expendedora mediante una máquina de Moore (ayuda: usar eventos definidos por el usuario).

3. Añadir al programa *keylogger* la posibilidad de almacenar también pulsaciones de teclado.

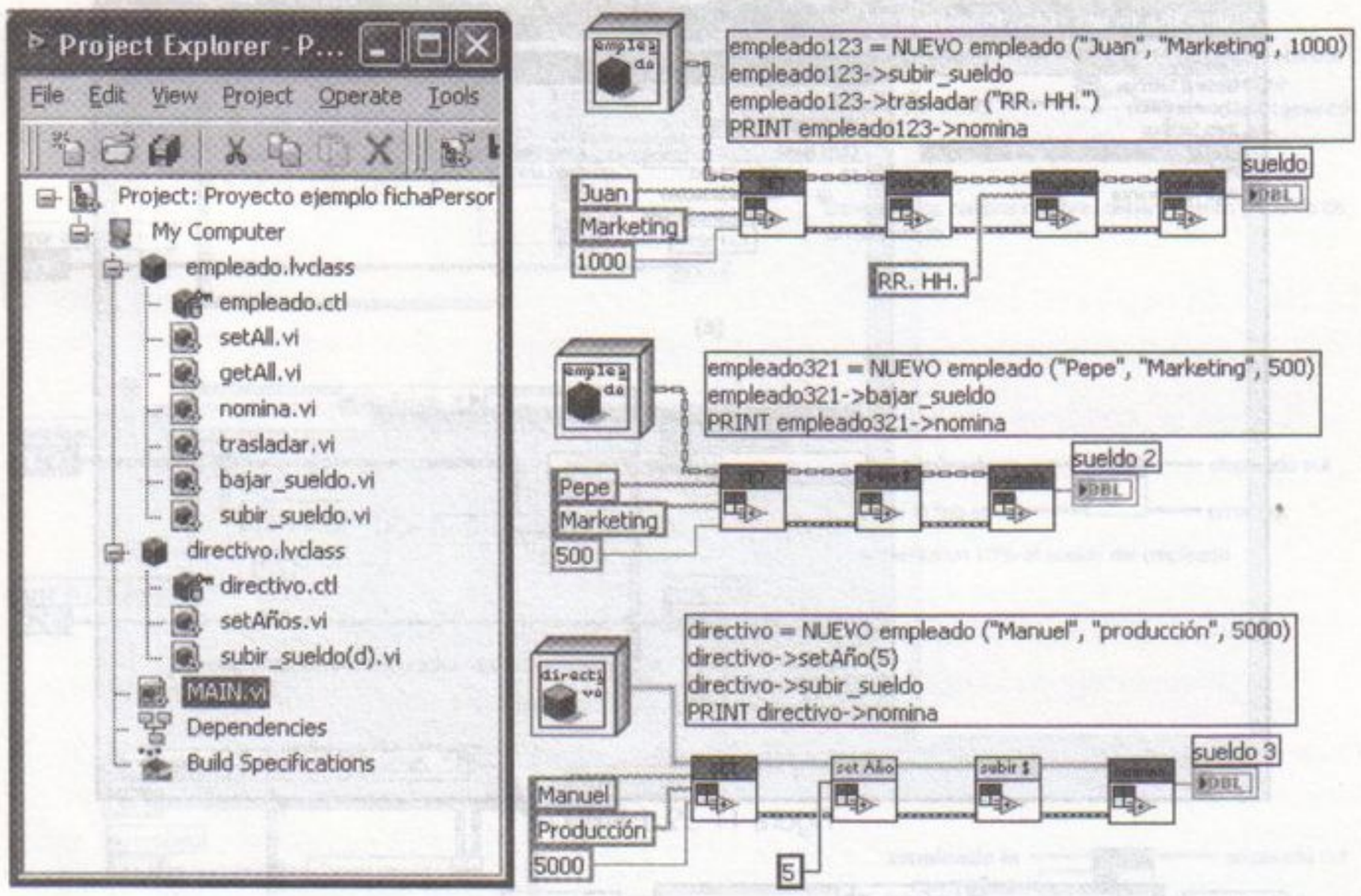


Figura 11-41. Programa principal

11.7 Bibliografía

Herbert Hecht y Myron Hecht, *Software Reliability in the System Context*, IEEE Trans. Software Eng., 1986.

Isabel Navarrete et alt., *Teoría de Autómatas y Lenguajes Formales*, Universidad de Murcia, 2003.

J. Glenn Brookshear, *Teoría de la computación*, Addison-Wesley, 2000.

Jim Kring, *OpenGOOP - a component framework*, 2003.

Jörgen Jehander y Stepan Riha, *Graphical Object-Oriented Programming*, ENDEVO y National Instruments, 1999.

Kevin Hogan, *Architectures for Designing*, National Instruments, 2002.

National Instruments, *LabVIEW Object-Oriented Programming: The Decisions Behind the Design*, 2006.

National Instruments, *LabVIEW State Diagram Toolkit User Guide*, 2003.

Robbie Gehbauer, *Introduction to Object-Oriented Development*, National Instruments, 2006.

Serafín Moral, *Teoría de Autómatas y Lenguajes Formales*, Universidad de Granada, 2001.

Stephen R. Mercer, *Object-Oriented Design in LabVIEW*, National Instruments, 2006.

Capítulo 12

Código externo

12.1 Librerías externas

Una librería o biblioteca es un conjunto de programas, subprogramas y otros recursos. Otros programas pueden hacer uso del código que contienen las librerías, para ello estos programas contendrán en su código referencias a las librerías, también llamadas enlaces (*link*). Las tipos de librerías, de acuerdo con la forma de enlazar su código, se clasifican en:

- **Estáticas:** las librerías estáticas son enlazadas al programa durante la compilación del mismo por el *linker*, que une el código de las librerías con el del programa que las llama.
- **Compartidas:** los programas incluyen referencias a las funciones de las librerías, que se mantienen en ficheros separados. Estas funciones se insertan en el programa cuando se ejecuta; la carga de las librerías puede ser cuando el programa comienza a ejecutarse o en el momento de producirse la llamada.

Las librerías contienen funciones, el proceso de llamada a las funciones se muestra en el siguiente ejemplo, en él se puede ver como en un momento determinado la función1 llama a función2 pasándole dos valores. Los valores de las variables de cada función se almacenan en un *buffer* de memoria o pila. Cada función tendrá un ámbito o alcance, es decir, una porción de pila con la cual podrá trabajar, el ámbito es definido por las flechas número 1 (base) y número 2 (límite superior) de la figura 12-1. Por simplificación se supone que la pila únicamente almacena las variables mostradas en el código de alto nivel (no se considera la dirección de retorno u otra información).

a) Inicialmente se puede asumir que las dos flechas están en la posición inferior.

b) Cuando función1 llama a función2 se insertan los valores de los dos parámetros en la pila, en primer lugar se almacena el valor 1 y luego el 2.

c) Ahora en función2 se produce un cambio de ámbito, es decir, se modifican los punteros de la pila (ver el punto 1 en la figura), ya que únicamente los valores pasados en la llamada y variables locales a función2 están disponibles (ámbito de la función).

d) Función2 finaliza, por lo que libera el espacio ocupado por sus variables locales.

Capítulo 12

Código externo

12.1 Librerías externas

Una librería o biblioteca es un conjunto de programas, subprogramas y otros recursos. Otros programas pueden hacer uso del código que contienen las librerías, para ello estos programas contendrán en su código referencias a las librerías, también llamados enlaces (*link*). Los tipos de librerías, de acuerdo con la forma de enlazar su código, se clasifican en:

- **Estáticas:** las librerías estáticas son enlazadas al programa durante la compilación del mismo por el *linker*, que une el código de las librerías con el del programa que las llama.
- **Compartidas:** los programas incluyen referencias a las funciones de las librerías, que se mantienen en ficheros separados. Estas funciones se insertan en el programa cuando se ejecuta; la carga de las librerías puede ser cuando el programa comienza a ejecutarse o en el momento de producirse la llamada.

Las librerías contienen funciones, el proceso de llamada a las funciones se muestra en el siguiente ejemplo, en él se puede ver como en un momento determinado la `funcionA` llama a `funcionB` pasándole dos valores. Los valores de las variables de cada función se almacenan en un *buffer* de memoria o pila. Cada función tendrá un ámbito o alcance, es decir, una porción de pila con la cual podrá trabajar, este ámbito es definido por las flechas número 1 (base) y número 2 (límite superior) de la figura 12-1. Por simplificación se supone que la pila únicamente almacena las variables mostradas en el código de alto nivel (no se considera la dirección de retorno u otra información).

a) Inicialmente se puede asumir que las dos flechas están en la posición inferior.

b) Cuando `funcionA` llama a `funcionB` se insertan los valores de los dos parámetros en la pila, en primer lugar se almacena el valor 1 y luego el 2.

c) Al entrar en `funcionB` se produce un cambio de ámbito, es decir, se modifican los punteros de la pila (ver el puntero base), esto provoca que solamente los valores pasados en la llamada y variables locales a funcionB estén disponibles (alcance de la función).

d) `funcionB` finaliza, por lo que libera el espacio de memoria de sus variables locales.

a) También se libera el espacio que ocupaban los parámetros, esto lo puede hacer `funcionB` o `funcionA`. Finalmente funcionA vuelve a tener el mismo ámbito que tenía antes de la llamada y puede seguir su ejecución.

Cuando hay varias funciones anidadas este proceso se repetiría por cada llamada. Cada hilo de ejecución tendría su propia pila. Para el valor del parámetro de retorno, en caso que exista, podría utilizarse un registro del procesador.

Cuando el programa es compilado, el compilador introducirá las instrucciones necesarias para gestionar todo el proceso del paso de parámetros. Otra tarea que pueden realizar algunos compiladores es el *name managling* o *name decoration*, esto consiste en cambiar el nombre de las funciones para asegurarse que sea único o proporcionar información sobre la misma función; algunos motivos por los que un nombre de función puede no ser único son la sobrecarga o los espacios de nombres.

```
1:   void funcionA (void){
2:         ...
3:         funcionB(1,2);
4:         ...
5:   }
6:   void funcionB (int a, int b){
7:         int c;
8:         ...
9:   }
```

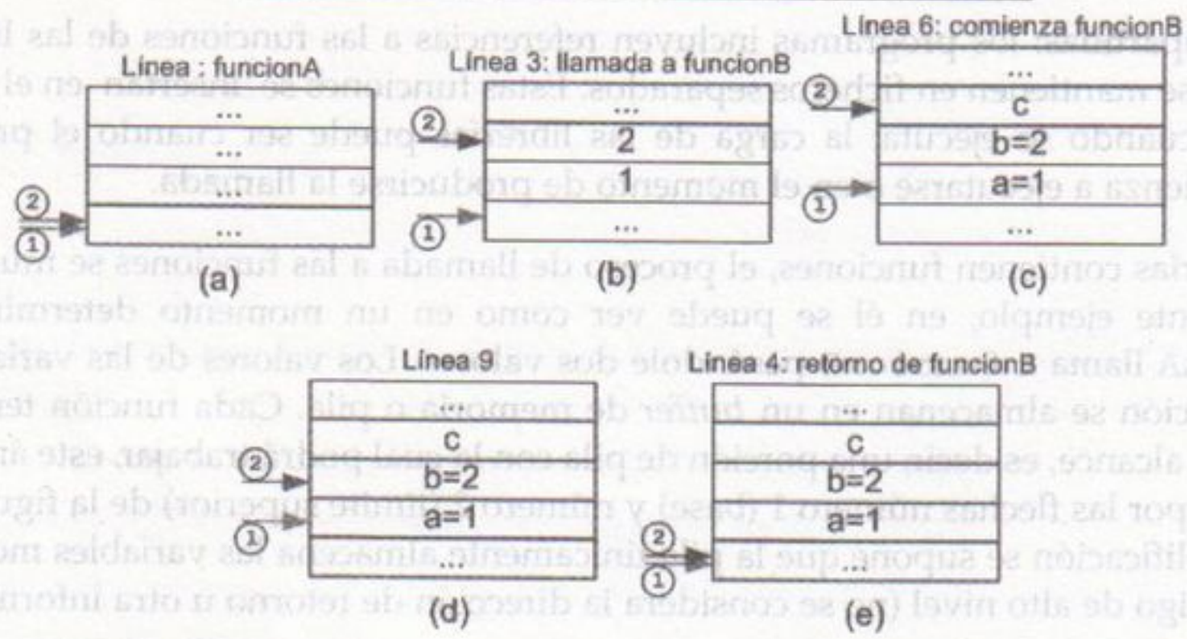


Figura 12-1. Evolución de la pila cuando se realizan llamadas a funciones

En el ejemplo anterior se ha visto cómo se pasan parámetros de una función a otra, pero el paso de esos parámetros debe seguir unas reglas llamadas convenio de llamada o *calling convention*. Básicamente se trata de definir el paso de parámetros y el *name decoration*. Hay diversos *calling convention*, los más populares son:

- *Cdecl*: es la estándar de ANSI C/C++. Los argumentos se introducen en la pila de derecha a izquierda (al contrario que en la figura 12-1), la función que realiza la llamada limpia la pila y la decoración del nombre se realiza con el prefijo «_».

- *Pascal*: es el que suelen usar los compiladores de Pascal. Los parámetros se introducen de izquierda a derecha, la función llamada debe sacar los parámetros de la pila antes de terminar.
- *Stdcall*: es el convenio usado en las funciones del API de Windows. Es muy parecido al estándar de C, los parámetros se introducen de derecha a izquierda, la función que es llamada es la que debe sacar los parámetros antes de retornar y la decoración es con el prefijo «_», el sufijo «@» y el número de bytes de la pila.
- *Safecall*: es una convención usada por el compilador Borland Delphi en Windows en programación COM. Los parámetros se introducen de izquierda a derecha, la función llamada es la encargada de sacarlos de la pila.
- *Fastcall*: es específico de algunos compiladores, como C++ Builder. Los parámetros se introducen de derecha a izquierda excepto los dos o tres primeros que, si pueden, usarán registros de la CPU (en x86 son EAX, EDX y ECX), la función llamada es la que saca los parámetros antes de volver y la decoración es con el prefijo «@» y el sufijo «@» con número de bytes de la pila. Los valores de retorno también usan los registros de la CPU (AL, AX o EAX).
- *Thiscall*: es la que se usa por defecto al llamar a *member functions* en C++. Los argumentos se pasan de derecha a izquierda excepto el argumento "this" que usa un registro de la CPU (ECX), la pila es limpiada por la función llamada y los nombres de las funciones hacen referencia a la clase y la función.

El conocido error *buffer overflow* aprovechado en muchos *exploits* es el que se produce cuando al escribir el valor de una variable no se respetan los límites de éstas en la pila, lo que provocaría que otra variable u otra información de la pila tuviera un valor erróneo. Suele darse con variables de tamaño alterable, como *strings* o *arrays*. En LabVIEW este error no puede darse ya que LabVIEW puede redimensionar el espacio de las variables dinámicamente.

Las librerías compartidas tienen varias ventajas, como que el mismo código puede ser usado por varias aplicaciones, se pueden actualizar sólo algunas partes del programa (por ejemplo mediante un *service pack*), añadir nuevas funcionalidades (por ejemplo los *plug-ins*), etc.

Las librerías compartidas en Windows se llaman DLL o *Dynamic Linking Library* y tienen extensión *.dll, en Linux suelen tener extensión *.so y en MacOS se llaman *Framework*.

12.1.1 CIN

Los nodos *Code Interface Node* (CIN) albergan en su interior código externo ya compilado que queda incrustado en el propio VI por lo que se tiene un fichero totalmente autocontenido.

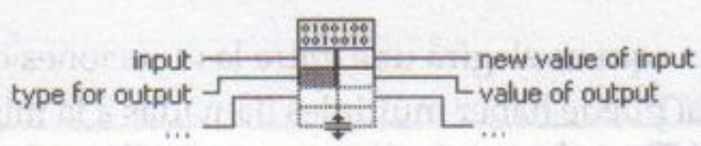


Figura 12-2. Code Interface Node

Para usar un nodo CIN:

1) En primer lugar debe adaptarse su tamaño al número de parámetros deseado.
2) Luego conectar los terminales a las entradas y salidas, indicando explícitamente los terminales que son sólo salida mediante el menú contextual. Con este paso el nodo sabrá de qué tipo es cada parámetro.
3) A continuación se creará, también con el menú contextual, un fichero con la plantilla del programa que se insertará en el nodo. El programa creado tendrá una función llamada *CINRun* que es el punto de entrada del programa, es decir, la función por donde comienza la ejecución, equivalente a la función *main* de C/C++.
4) Cuando el programa esté creado y compilado se deberá cargar en el CIN el fichero del código objeto (*.lsb) resultante con la opción *Load Code Resource*.

Además de funciones normales, un CIN también puede tener otras rutinas que son llamadas cuando el VI es cargado, abortado, guardado, cerrado, compilado... las funciones son: CINRun, CINInit, CINAbort, CINLoad, CINSave, CINDispose y CINProperties.

Usar los nodos CIN tiene algunas desventajas respecto a las librerías compartidas que se verán a continuación, como una menor modularidad (si se cambia el código de la librería se debe modificar el VI) y que no todos los entornos de programación pueden crear código para CIN. Por todo ello National Instruments recomienda usar librerías compartidas en lugar de CIN.

12.1.2 Librerías compartidas en LabVIEW

En el menú *Connectivity > Libraries & Executables* se encuentra el nodo *Call Library Function Node* que permite enlazar funciones de librerías compartidas.

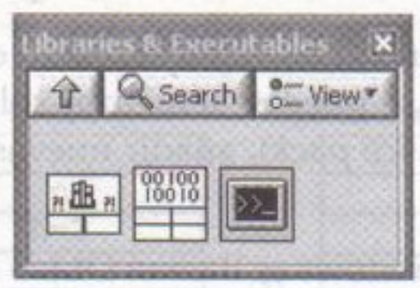


Figura 12-3. Menú Libraries & Executables

Al colocar el nodo en el Diagrama de Bloques y configurarlo aparece la pantalla mostrada en la figura 12-4. Sus campos son:

- *Functions*:
  - *Library Name or Path*: en este campo se indicará la ruta de la librería. Activando la casilla inferior la ruta podrá indicarse desde el Diagrama de Bloques.
  - *Function Name*: aquí se elegirá una entre las funciones que contiene la librería.
  - *Thread*: indica si puede haber múltiples llamadas a la misma librería (*reentrant*) o no (*Run In UI Thread*), en este último caso las llamadas se realizarán en el sis-

tema de ejecución del interfaz para asegurarse que solamente haya un hilo realizando la llamada. Los nodos reentrantes se dibujarán con una tonalidad amarilla en el Diagrama de Bloques y los no reentrantes de color naranja. En la figura 12-5 se pueden ver los primeros a la izquierda y los segundos a la derecha.

- *Calling Conventions*: es la forma en que se pasan datos a la función.

- *Parameters*: en esta pestaña hay varios campos en los que se indicarán los parámetros de la función. Habrá un parámetro de retorno (*return type*) y varios parámetros a los que se le asignará un nombre y un tipo.
- *Callbacks*: permite configurar llamadas a otras funciones en ciertos instantes claves de la ejecución del nodo. *Reserve* especifica la función a llamar cuando se reserva un espacio de tiempo y memoria para cada llamada en un VI reentrante; *Unreserve* es lo contrario a la anterior, puede ser útil en tareas de depuración, y *Abort* permite configurar una función cuando se intenta abortar la ejecución del VI. También se muestra en *Prototype for these procedures* una cabecera genérica en C para estas funciones.
- *Function Prototype*: va mostrando una reconstrucción de la cabecera de la función en estilo C de acuerdo con lo que el programador indique en los campos anteriores, puede servir para comparar el resultado con la cabecera real de la función.
  - *Type, Data Type, Pass*: indican el tipo de datos del parámetro seleccionado. LabVIEW no puede reconocer todos los tipos de datos de todos los lenguajes de programación existentes, pero hay equivalencias entre la mayoría de tipos de otros lenguajes y los de LabVIEW. Si un tipo de datos no es soportado por LabVIEW se pueden pasar/recibir los datos "planos" como un *string* o *array* y hacer una conversión, usar *Adapt to Type* a/desde un *cluster* o también se puede escribir una librería compartida o CIN que sirva de puente (*wrapper*).

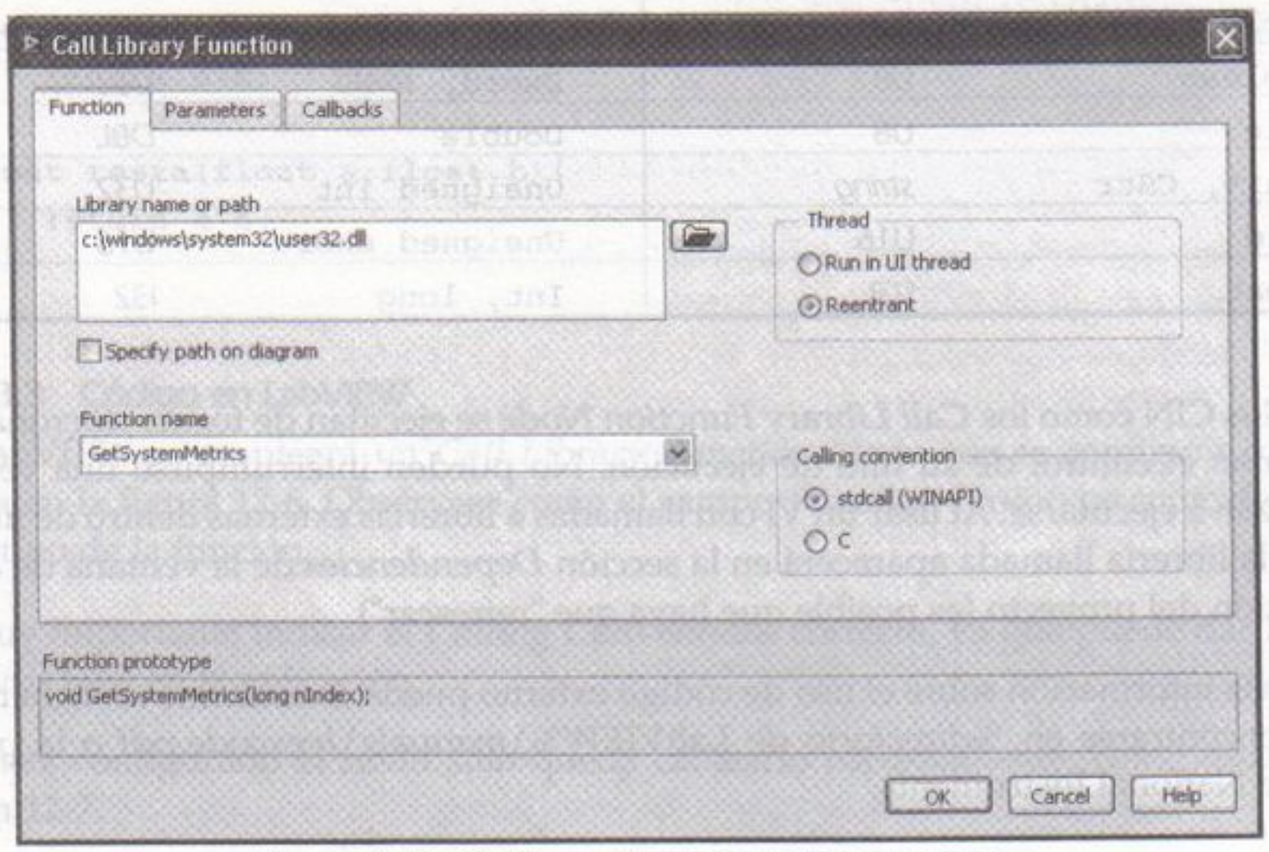


Figura 12-4. Configuración de Call Library Function

En la versión 8.20 se ha introducido un asistente que, a partir de la librería compartida y un fichero de cabecera, permite crear una librería con nodos en los que se llama a las distintas funciones. Este asistente se encuentra en *Tools > Import > Shared library (.dll)...*

En la figura 12-5 puede verse el aspecto de este nodo después de ser configurado como muestra la figura 12-4. A la izquierda se muestran nodos reentrantes y a la derecha no reentrantes; los superiores aceptan un *path* desde el Diagrama de Bloques y a los de abajo hay que indicárselo desde la ventana de configuración. En cada nodo el terminal superior es de sólo lectura y corresponde al valor de retorno, el resto de terminales son los otros parámetros y se leerán o escribirán de acuerdo con la cabecera de la función.

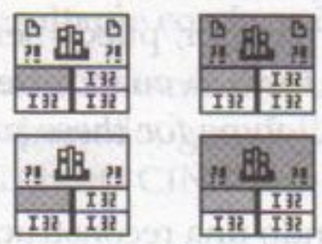


Figura 12-5. Aspecto del nodo ya configurado

En la Tabla 12-1 puede verse una relación de equivalencias entre diferentes tipos de datos en C/C++ y en LabVIEW. Puede obtenerse más información ejecutando *Call DLL.vi* en *%directorio de LabVIEW%\examples\dll\data passing\Call Native Code.llb*.

Tabla 1 - Equivalencias entre tipos de datos

C	LabVIEW	C	LabVIEW
Bool	I32	short	I16
Boolean	U8	Dword, HWND	U32
Byte	U8	Double	DBL
Char*, CStr	*string*	Unsigned int	U32
Word	U16	Unsigned short	U16
float	SGL	Int, long	I32

Tanto los CIN como los *Call Library Function Node* se ejecutan de forma síncrona, esto es, toman el control de su hilo de ejecución. No pueden interrumpirse una vez que empiezan a ejecutarse. Al usar un VI con llamadas a librerías externas dentro de un proyecto, la librería llamada aparecerá en la sección *Dependencies* de la ventana de administración del proyecto (es posible que haya que "refrescar").

Para más información sobre el uso de código externo puede consultarse un fichero que suele encontrarse en *%directorio de LabVIEW%\manuals\lvexcode.pdf* o la página web de National Instruments.

12.1.3 Ejemplo I: Creación de una DLL y su uso en LabVIEW

12.1.3.1 Explicación teórica

En este ejemplo se creará una librería compartida en Windows para LabVIEW. Se usará el compilador de C++ de Visual Studio, de Microsoft.

Antes de entrar a estudiar el código hay que comentar algunas peculiaridades de Visual C++:

- Para evitar el *name decoration* se usa la cláusula `extern "C"`.
- Para indicar que la función será accesible desde el exterior hay que escribir `__declspec(dllexport)` antes de la declaración de la función.
- Para cada *calling convention* existen directivas como `__cdecl` ó `__stdcall`.

La librería de este ejemplo contendrá dos funciones que realizarán una suma y una resta de dos números flotantes que se le pasan como parámetros.

12.1.3.2 Código en C++

```
//includes
#include "windows.h"

//cabeceras
extern "C"{
    __declspec(dllexport) float suma(float a,float b);
    __declspec(dllexport) float resta(float a,float b);
}
//funciones
float suma(float a,float b){
    return a+b;
}
float resta(float a,float b){
    return a-b;
}
```

12.1.3.3 Código en LabVIEW

En LabVIEW se empleará un *Call Library Function Node* que se configura como se indica en la figura 12-6. Obsérvese como el campo *Function Prototype* coincide con el prototipo de la función.

Es muy importante indicar el *Calling Conventions* correcto, ya que puede dar lugar a un error fatal en la aplicación.

Una vez configurado el nodo sólo queda cablearlo correctamente, como muestra la figura 12-7.

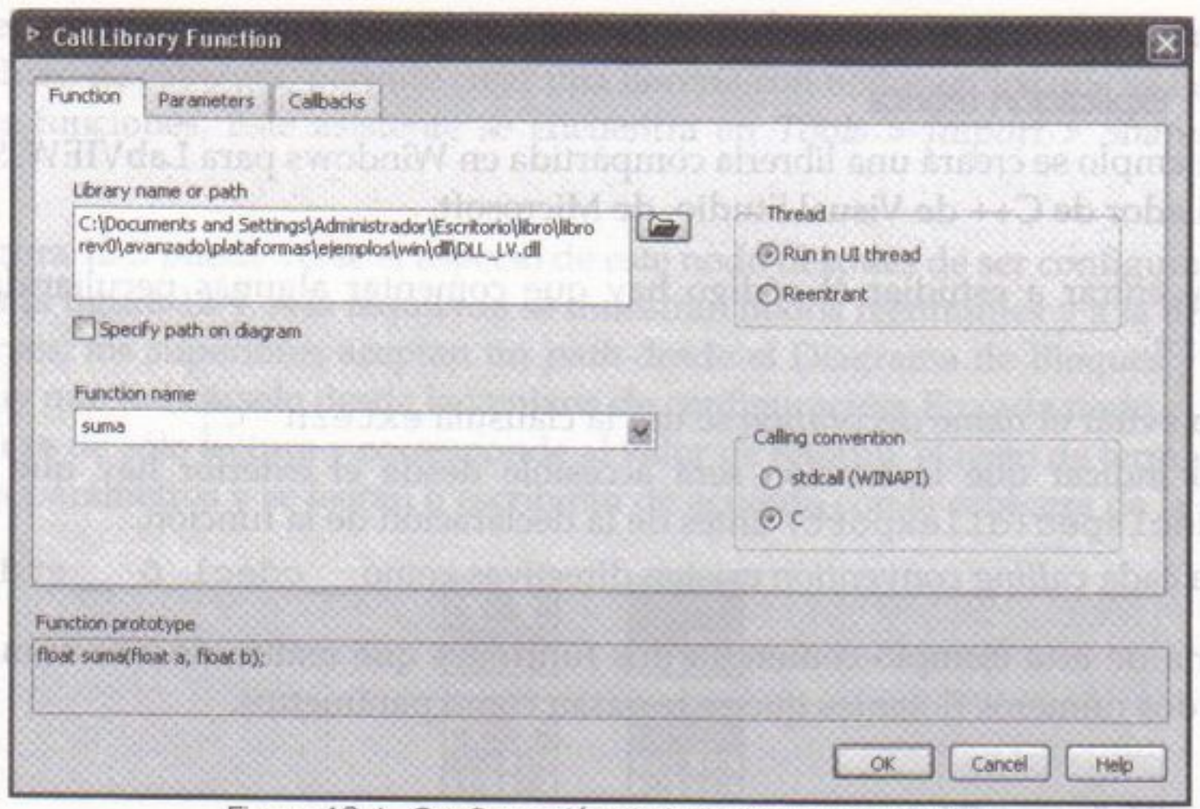


Figura 12-6. Configuración de la llamada a la librería

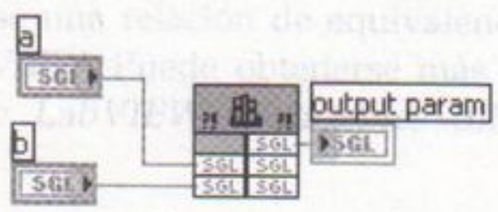


Figura 12-7. Ejemplo VI de una llamada a la librería creada

12.1.4 Ejemplo II: Wait (μs)

12.1.4.1 Explicación teórica

Si el programa se ejecuta sobre otras plataformas, como por ejemplo sobre un dispositivo NI-RIO, se puede usar como referencia de tiempo los relojes de estas plataformas y tener una resolución de microsegundos. Sin embargo cuando LabVIEW se ejecuta de forma normal sobre un PC, todas las funciones de retardos y medida de tiempos tienen una resolución de milisegundos.

En este ejemplo se diseñará un programa que al ejecutarse en un ordenador con Windows inserte un ciclo de espera con una precisión del orden de microsegundos. A diferencia de *Wait (ms)*, en este caso la espera será activa, es decir, este programa no se podría usar dentro de un bucle para evitar que éste ocupara todo el tiempo la CPU.

Para crear este programa se emplearán dos funciones del API de Windows. La primera es `QueryPerformanceCounter`, que devuelve el valor de un contador de tics del procesador. La segunda función usada es `QueryPerformanceFrecuency` que devuelve el número de tics por segundo del procesador.

12.1.4.2 Código

La documentación sobre las funciones del API de Windows puede encontrarse en el sitio MSDN de Microsoft. Buscando las dos funciones usadas en este ejemplo se puede ver que ambas están en el fichero *Kernel32.dll* y sus declaraciones son:

```
BOOL QueryPerformanceCounter(
LARGE_INTEGER *lpPerformanceCount
);

BOOL QueryPerformanceFrequency(
LARGE_INTEGER *lpFrequency
);

typedef union _LARGE_INTEGER {
struct {
    DWORD LowPart;
    LONG HighPart;
};
struct {
    DWORD LowPart;
    LONG HighPart;
} u;
LONGLONG QuadPart;
} LARGE_INTEGER, *PLARGE_INTEGER;
```

El *calling convention* de estas funciones es el típico del API de Windows.

El tipo de datos que devuelven es un puntero a un `LARGE_INTEGER`, que es un dato de 64 bits (un `union` es como un `struct` pero en un instante concreto sólo puede contener el valor de uno de sus miembros). Con toda esta información ya se pueden hacer las llamadas a ambas funciones, tal como se puede ver en la figura 12-8.

En primer lugar, el programa adquiere la cuenta de tics nada más entrar a la función con `QueryPerformanceCounter`. A continuación se calculan los tics que deben pasar en el periodo de tiempo indicado y se suma esta cantidad a la anterior para obtener la cuenta final. En el último paso se entra en un bucle del que sólo se saldrá cuando la cuenta de tics sea igual o superior a la calculada antes. El programa completo se muestra en la figura 12-9.

12.1.4.3 Resultados

Para poner a prueba el programa anterior se programará un retraso de 10 ms (10.000 µs) y se medirá el tiempo antes y después de la llamada al VI anterior. El resultado puede verse en la figura 12-10.

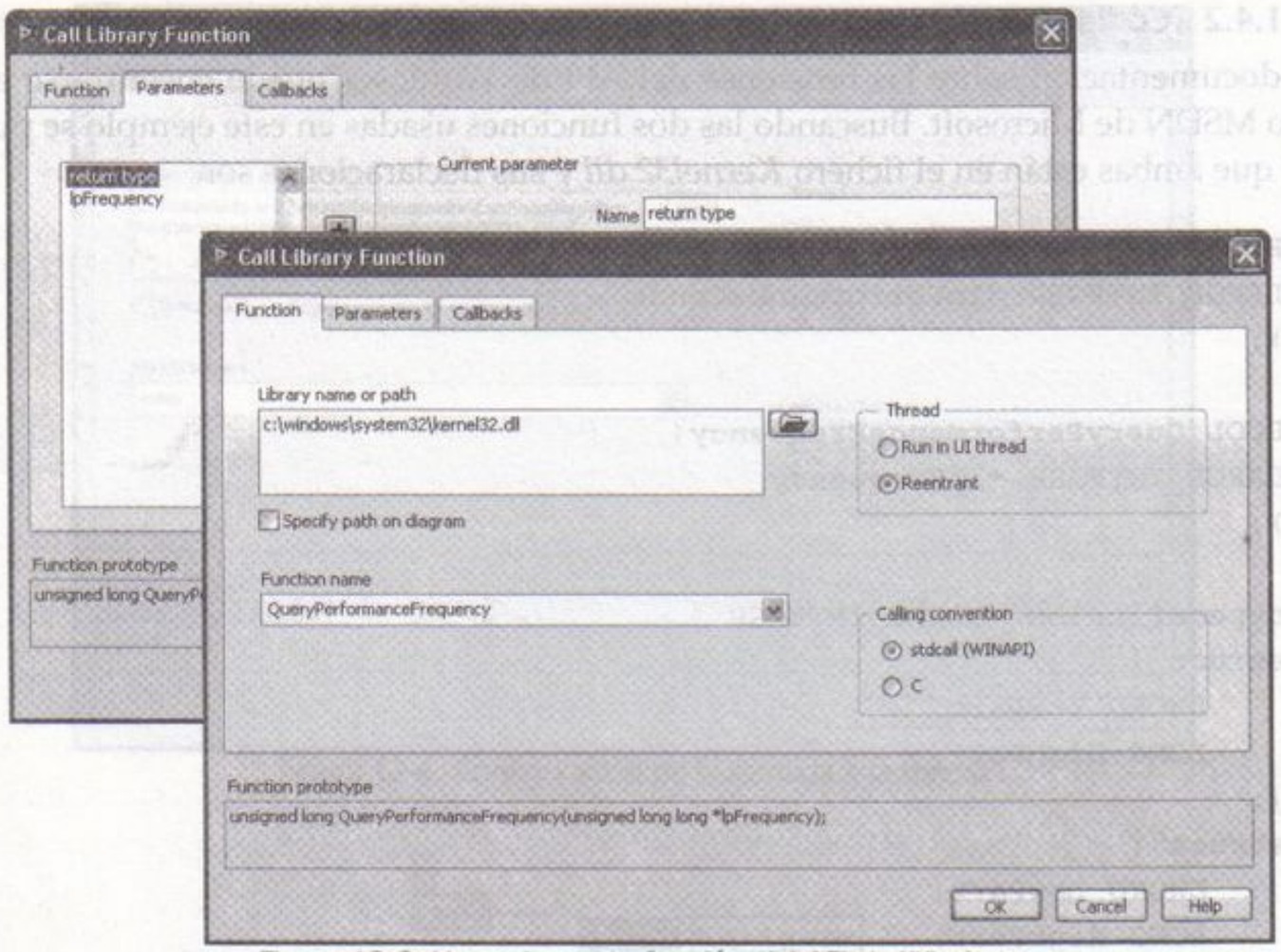


Figura 12-8. Llamada a una función del API de Windows

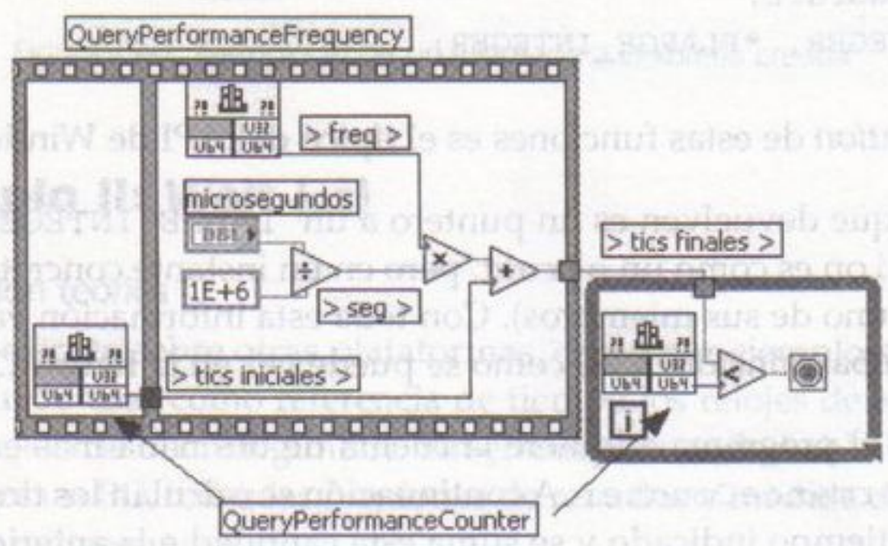


Figura 12-9. Espera activa en microsegundos

En este ejemplo se han usado dos funciones del API de Windows y se ha visto la importancia de disponer de la documentación apropiada. Es interesante ver el efecto que tendría cambiar el tipo de datos y el *Calling Convention* en las llamadas a las funciones.

También hay que hacer un pequeño comentario sobre el algoritmo del programa. Hay muchas formas de crear el programa, pero en esta ocasión se ha hecho de manera que se minimicen las tareas que hay que realizar dentro del bucle para que la ejecución del mismo sea lo más rápida posible y así aumentar la precisión de la función.

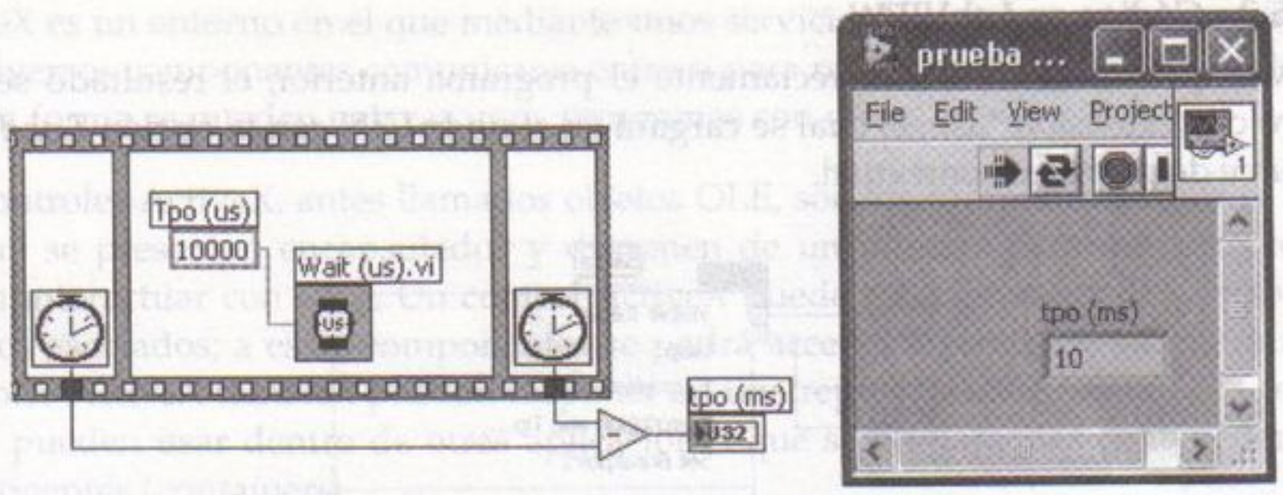


Figura 12-10. Ejemplo de ejecución del VI WAIT

12.1.5 Ejemplo III: CIN

12.1.5.1 Explicación teórica

Para crear el código de un nodo CIN se seguirá el método descrito antes: colocar el nodo CIN en el Diagrama de Bloques, ajustar su tamaño, cablearlo correctamente y crear un fichero *.c que contendrá una plantilla de la función.

El código generado por el nodo CIN contendrá un *include* de un fichero llamado *extcode.h*, este fichero contiene definiciones de los tipos de datos de LabVIEW y algunas funciones; éste y otros ficheros que también son necesarios para compilar el código se encuentran en *%directorio de LabVIEW%\cintools*.

12.1.5.2 Código en C++

El código está basado en la plantilla generada por LabVIEW, simplemente se ha añadido la línea `*Salida = *a+*b;`.

```
#include "extcode.h"
MgErr CINRun(float32 *Salida, float32 *a, float32 *b);

MgErr CINRun(float32 *Salida, float32 *a, float32 *b)

{
/* Insert code here */
*Salida = *a+*b;
return noErr;
}
```

El compilador debe configurarse convenientemente, como indica el fichero *%directorio de instalación de LabVIEW%\manuals\lvexcode.pdf*.

12.1.5.3 Código en LabVIEW

Una vez que se ha creado correctamente el programa anterior, el resultado será un fichero con extensión *.lsb, el cual se cargará en el nodo CIN con la opción *Load Code Resource* de su menú contextual.

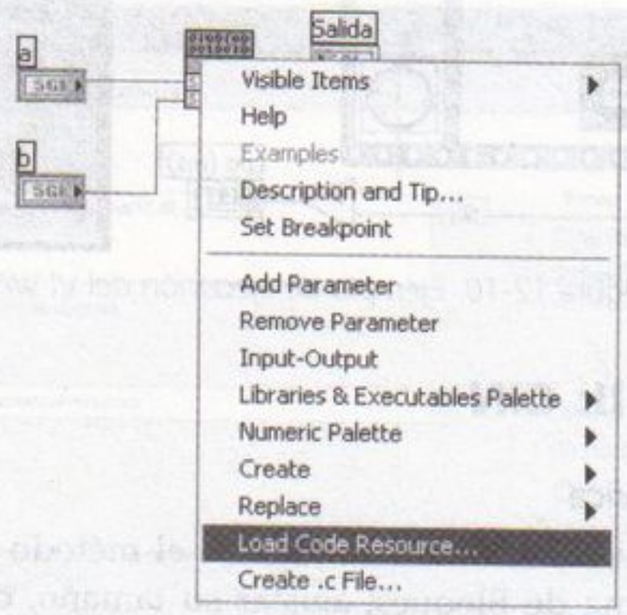


Figura 12-11. Cargar el programa en el nodo CIN

Una vez cargado el programa en el CIN ya se podrá ejecutar el programa. Es interesante comprobar como el tamaño del VI aumenta como resultado de incluir el código creado antes en el nodo CIN, el tamaño puede comprobarse en *File > Properties > Memory Usage*.

12.2 Comunicaciones entre aplicaciones

Cada sistema operativo tiene sus particularidades, una de ella es la forma de comunicación entre programas: MacOS tiene los *Apple events*, UNIX las *pipes* y Windows los ActiveX.

En sistemas MacOS, los *Apple events* son uno de los mecanismos de comunicación entre aplicaciones. Está basado en intercambio de mensajes. Son una parte esencial del *AppleScript system*.

En Linux y otros sistemas las *pipes* o tuberías son un mecanismo para encadenar programas en el que la salida de uno se redirige a la entrada de otro. El concepto es parecido al de *pipeline*.

12.2.1 ActiveX

Microsoft ha desarrollado varias tecnologías para compartir recursos basadas en objetos, la primera de ellas fue DDE, los primeros programas en usarla fueron Word y Excel para permitir embeber un documento dentro de otro. OLE 1.0 apareció en 1991 basado en DDE. Dos años después se presentó la segunda versión de OLE y otros tres años después (con mínimos cambios respecto a OLE 2.0) nació ActiveX.

ActiveX es un entorno en el que mediante unos servicios basados en objetos se permite a diversos componentes comunicarse entre sí para reutilizar el código de los demás, de esta forma se pueden enlazar unos programas con otros.

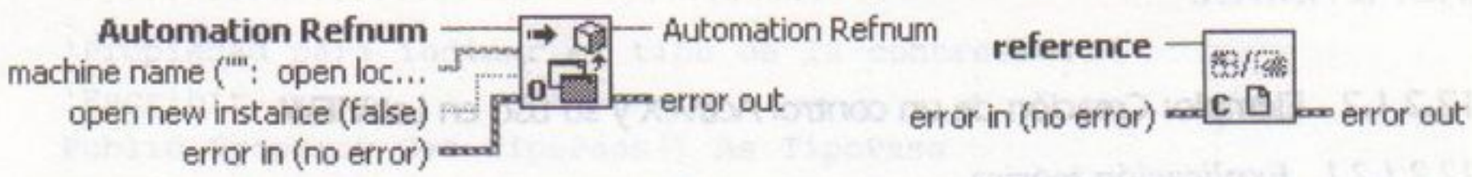
Los controles ActiveX, antes llamados objetos OLE, son los componentes que se reutilizarán, se presentan encapsulados y disponen de un interfaz para que otro código pueda interactuar con ellos. Un control ActiveX puede tener propiedades, métodos y eventos asociados; a estos componentes se podrá acceder a través de su interfaz. Los controles ActiveX también pueden disponer de una representación visual. Los controles se pueden usar dentro de otras aplicaciones que serán llamadas contenedores de componentes (*containers*).

12.2.1.1 ActiveX en LabVIEW

LabVIEW puede actuar como programa contenedor de un control ActiveX. Además también puede actuar como servidor ActiveX habilitando esta opción en *Tools > Options > VI Server: Configuration > ActiveX*.

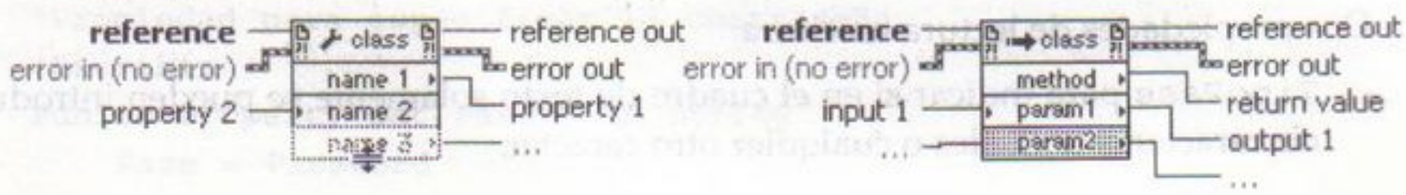
Figura 12-12. Menú ActiveX

- **Automation Open/Close Reference**



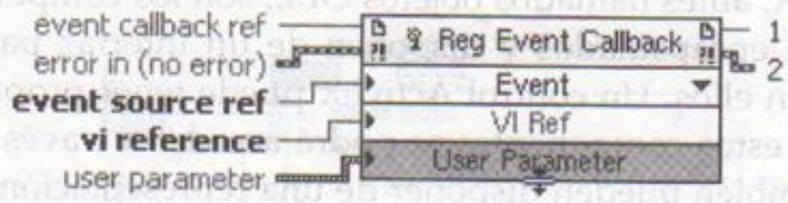
Sirven para abrir y cerrar referencias a controles ActiveX. En el primer caso, el terminal *Automation Refnum* puede venir de un control (*Paleta de controles > Modern > Refnum > Automation Refnum*), una constante o un *Container* (*Paleta de controles > Modern > Containers > ActiveX Container*). Se usará un *Container* cuando el control tenga una interfaz gráfica y se desee mostrar.

- **Property Node/Invoke Node**



Al cablear una referencia sobre estos nodos aparecerán las propiedades y métodos del control ActiveX.

- Register Event Callback



Un control también puede tener asociados ciertos eventos. En este caso cuando se dé una determinada condición, el control ActiveX disparará un código en respuesta al evento (si ha sido registrado).

En este nodo se cableará una referencia al control ActiveX en el terminal *event source ref.* En *vi reference* se conectará una referencia a un VI, el código de este VI se ejecutará cuando ocurra el evento. La referencia debe ser a un tipo estricto (marcado con una estrella naranja) y tendrá unos terminales prefijados. Para crear automáticamente la referencia lo más sencillo es acudir al menú contextual de este terminal y seleccionar la opción *Create Callback VI,* esto creará una referencia y haciendo doble clic sobre ella se abrirá el VI reentrante en el que puede colocarse el código del evento.

También puede ser útil para trabajar con ActiveX el explorador de clases disponible en *View > Class Browser,* mediante él se podrán explorar las propiedades y métodos de cada clase, es equivalente a la opción *Select Property/Method* del menú contextual de los nodos de propiedades y métodos respectivamente. Los *containers* en el Panel Frontal también tienen la opción *Property Browser* que muestra una ventana donde se puede dar valor a las propiedades del objeto.

Por último hay que decir que se puede convertir un ActiveX a un control (*.ctl) con *Tools > Import > ActiveX Controls to Palette...* El control se encuentra en *Controls > .NET & ActiveX.*

12.2.1.2 Ejemplo: Creación de un control ActiveX y su uso en LabVIEW

12.2.1.2.1 Explicación teórica

Para comprender como usar controles ActiveX en LabVIEW se empezará por conocer cómo están hechos y cómo funcionan. Con este propósito se diseñará un sencillo control ActiveX en Visual Basic para más tarde usarlo en LabVIEW.

El control ActiveX tendrá un interfaz que consistirá en un cuadro de texto cuya finalidad es servir de entrada para que un usuario escriba una contraseña. Constará de:

- Dos propiedades de lectura-escritura:
  - `TipoPass` para indicar si en el cuadro de texto solamente se pueden introducir caracteres textuales o cualquier otro carácter.

- Pass para indicar cuál es la contraseña.
- Un método, A_mayusculas, que pasa lo escrito en el control a mayúsculas.
- El evento correcto que se activará cuando el texto introducido en el control sea igual a la contraseña.

El resto de funciones, eventos y variables que se puedan usar en el programa no son accesibles desde el exterior.

12.2.1.2.2 Código en Visual Basic

El control consistirá en un control de tipo TextBox llamado *passTextBox*. El código asociado a este control es:

```
''''''
```

```
End Property
'Leer
Public Property Let Pass(ByVal sPass As String)
Password = sPass
End Property

''''''
```

12.2.1.2.3 Código en LabVIEW

El código en LabVIEW es extremadamente sencillo. En primer lugar se debe colocar un *ActiveX Container* en el Panel Frontal, en él se insertará el objeto diseñado previamente registrado. El *ActiveX Container* debe entonces mostrar el cuadro de texto.

Nótese que, aun estando el programa de LabVIEW sin ejecutarse, el control ActiveX sí está funcionando: se puede escribir en él y puede dar lugar a la ejecución de métodos y disparo de eventos. Si se quiere evitar esto hay que ir al menú contextual del container y elegir *Advanced > Design Mode*.

Figura 12-13. Menú de selección del objeto ActiveX

También en el *menú contextual > Property Browser* se pueden asignar valores a las propiedades del control. En este ejemplo no se modificarán desde esta ventana sino que las propiedades se asignarán con nodos desde el código del programa, el cual puede verse en la figura 12-14.

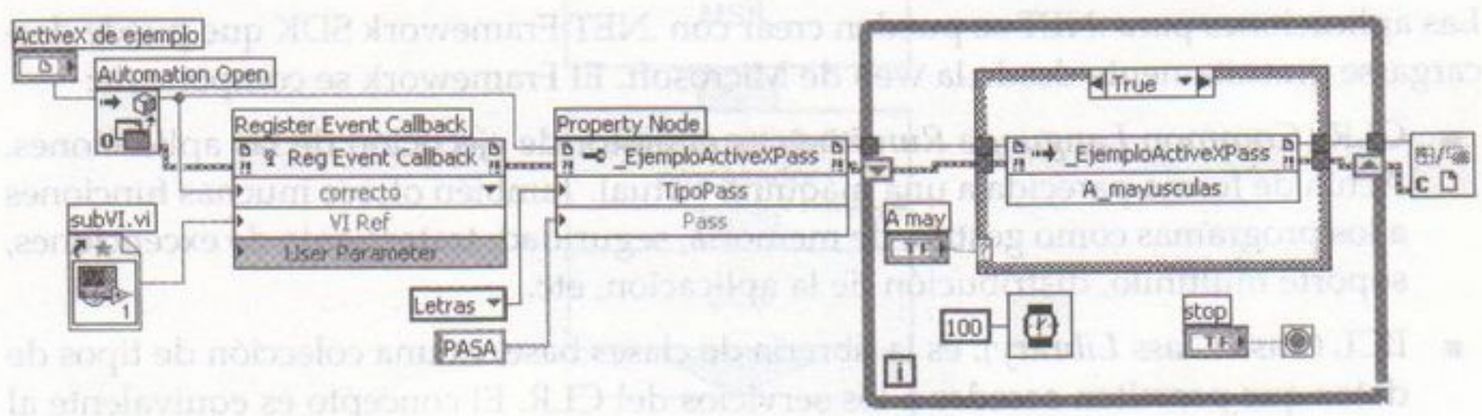


Figura 12-14. Código del programa ejemplo de ActiveX

Este programa empieza registrando el evento `correcto` definido en el código en Visual Basic, este evento se activará cuando se introduzca en el cuadro de texto la contraseña. En el momento de activarse este evento se ejecutará *subVI.vi*, que simplemente muestra un mensaje por pantalla.

El siguiente paso es asignar un valor mediante *Property Node* a las dos propiedades del control: `TipoPass` y `Pass`.

Finalmente, cuando se presiona el botón *A may* se ejecutará el método `A_mayusculas` que convierte el texto introducido en el control a mayúsculas.

12.2.1.2.4 Resultados

A continuación se muestra una captura de pantalla del resultado. En el control ActiveX se ha introducido la contraseña correcta e inmediatamente se ha ejecutado la rutina `passTextBox_KeyUp` que comprueba la contraseña y muestra una ventana (`MsgBox`). A continuación se ejecutaría *subVI.vi* al dispararse el evento `correcto`.

Figura 12-15. Panel Frontal de la ejecución del VI con ActiveX

12.2.2 .NET

.NET es un conjunto de nuevas tecnologías de Microsoft que permite distribuir software de forma potente y sencilla. El código se distribuye en forma de servicios que pueden ser accedidos de forma remota, es independiente del lenguaje de programación y de la de la plataforma.

Las aplicaciones para .NET se pueden crear con .NET Framework SDK que puede descargarse gratuitamente desde la web de Microsoft. El Framework se compone de:

- CLR (*Common Language Runtime*): es el motor de ejecución de las aplicaciones. Actúa de forma parecida a una máquina virtual. También ofrece muchas funciones a los programas como gestión de memoria, seguridad, tratamiento de excepciones, soporte multihilo, distribución de la aplicación, etc.
- BCL (*Base Class Library*): es la librería de clases base, es una colección de tipos de datos que permiten acceder a los servicios del CLR. El concepto es equivalente al API de Windows o al MFC de Visual C++. Debido a su tamaño se ha dividido de forma jerárquica en espacios de nombres, los cuales contienen clases, tipos de datos, otros espacios de nombre, etc.
- Lenguajes de programación: C#, C++, Visual Basic... También existen otros lenguajes que disponen de compiladores para .NET aunque no están dentro del Framework, por ejemplo COBOL, Python, Perl, etc.

- ASP.NET: es un entorno que proporciona herramientas para aplicaciones basadas en navegadores web.

En la figura 12-16 se ha representado la estructura de los componentes de .NET Framework.

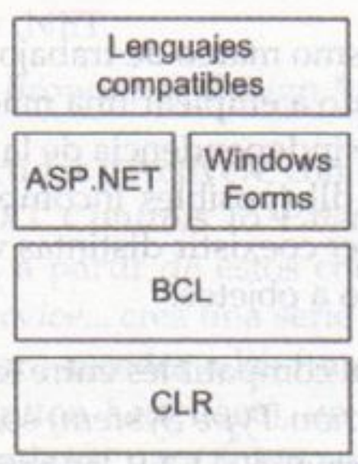


Figura 12-16. Estructura .NET

El funcionamiento, a grandes rasgos, es el siguiente: se escribe un programa en cualquier lenguaje compatible con .NET, gracias al BCL y CLR todos los lenguajes tendrán prácticamente los mismos recursos. Una vez creado el programa se transforma a otro lenguaje llamado «Lenguaje Intermedio de MicroSoft» (MSIL), también se crea lo denominado *Metadata*, que es información sobre los campos, propiedades y otras características del programa. Finalmente el programa en lenguaje intermedio es convertido al código nativo del procesador (este paso puede ser *Just-In-Time* si la conversión es gradual y se produce durante la ejecución o *Native Image Generador Compilation* si es anterior a la ejecución) y ejecutado por el CLR.

Figura 12-17. Pasos para ejecutar una aplicación .NET

Los ensamblados o *assemblies* son básicamente los programas hechos en .NET. Se trata de un conjunto de recursos agrupados en uno o varios ficheros (módulos) con extensión *.dll o *.exe. Incluyen un *Manifesto* y metadatos con información sobre el nombre

y la versión, dependencias, permisos, información sobre lo que contiene el *assembly*, etc. También incluyen el código MSIL resultado de la compilación. El GAC (*Global Assembly Cache*) es una lista de los *assemblies* públicos presentes en el sistema, en Windows puede verse con el explorador, normalmente se haya en el directorio *C:\WINDOWS\assembly*.

Al tener todos los lenguajes el mismo marco de trabajo, .NET aporta independencia del lenguaje de programación y debido a emplear una máquina virtual que ejecuta un lenguaje intermedio también se tiene independencia de la plataforma; .NET elimina lo que se conocía como infierno de las dll (posibles incompatibilidades al instalar distintas versiones de la misma dll) al poder coexistir distintas versiones de las mismas librerías y también es totalmente orientado a objetos.

Para que el código y los datos sean compatibles entre los distintos programas, éste debe seguir algunas reglas. CTS (*Common Type System*) son las reglas que deben seguir las definiciones de tipos de datos para que el CLR las acepte, es decir, define cómo deben ser los tipos comunes a todos lenguajes .NET. CLS (*Common Language Specification*) son las reglas que deben cumplir las definiciones de tipos para que los tipos y sus miembros sean accedidos desde el exterior.

Otros conceptos relacionados con .NET son:

- *Windows Form*: se utiliza en las aplicaciones tradicionales de Windows para crear el interfaz de la aplicación, es parecido a los formularios de Visual Basic o Visual C++.
- *Web Forms*: es el equivalente a *Windows Forms* para ASP.NET.
- ADO.NET: arquitectura para acceder a bases de datos.

12.2.2.1 .NET en LabVIEW

La forma de trabajar con .NET en LabVIEW es parecida a la vista anteriormente con ActiveX, se puede acceder a objetos, propiedades, métodos y eventos asociados a un servidor .NET.

El menú con funciones relacionadas con .NET puede encontrarse en *Connectivity > .NET*. También puede agregarse un *Container* en el Panel Frontal yendo a *Modern > Containers > .NET Container*.

Figura 12-18. Menú de funciones .NET

Las referencias a los objetos .NET se pueden crear desde el Panel Frontal usando un *Container* o también desde el Diagrama de Bloques con un *Constructor Node*. El resto de funciones del menú .NET se usan del mismo modo que las funciones de ActiveX.

Algunas diferencias entre .NET y ActiveX en LabVIEW son:

- LabVIEW no es un servidor .NET.
- No se puede usar *Property Browser* ni *Design Mode*.

Al igual que las librerías compartidas y ActiveX, con .NET también hay asistentes en el menú *Tools > Import*. Con *.NET Controls to Palette...* se importan controles de la misma manera que en ActiveX, a partir de estos controles se podrán crear nodos de métodos y propiedades. *Web Service...* crea una serie de VIs a modo de *wrapper* (como con las librerías compartidas) para acceder a las funciones de un servicio web basado en WSDL (*Web Services Description Language*), un servicio web es básicamente una aplicación que sigue unos estándares para intercambiar información con otras aplicaciones.

12.2.2.2 Ejemplo I: Creación de una librería de clases .NET y su uso en LabVIEW

12.2.2.2.1 Explicación teórica

En este ejemplo se diseñará en C# una pequeña clase que sirva de ejemplo para ver su posterior uso en un programa en LabVIEW.

La clase realizará cálculos matemáticos básicos con dos operadores. Estará compuesta por:

- Dos propiedades:
  - `Operador1`.
  - `Operador2`.
- Un método con el original nombre `Metodo` que realizará la operación indicada con los dos operadores. Las operaciones serán la suma, resta, multiplicación o división.
- No será necesario ningún control, eventos u otras funciones privadas.

12.2.2.2.2 Código en C#

El código en C# estará compuesto por un *Namespace* que contendrá la clase `ClaseCalcula` descrita anteriormente. Al compilarse se creará el fichero *ejemplo.dll*.

```
using System;
using System.Collections.Generic;

namespace ejemplo
{
    /// <summary>
    /// Descripción de ClaseCalcula
    /// </summary>
```

```
public class ClaseCalcula
{
        private static int valor1;
        private static int valor2;

        /// <summary>
        /// Propiedad 1: primer operador
        /// </summary>
        public int Operador1{
                set{valor1=value;}
                get{return valor1;}
        }

        /// <summary>
        /// Propiedad: segundo operador
        /// </summary>
        public int Operador2{
                set{valor2=value;}
                get{return valor2;}
        }

        /// <summary>
        /// Método que realiza una operación numérica.
        /// </summary>
        public static float Metodo( int op){
                switch (op){
                        case 0:
                                return (float)valor1+valor2;
                                break;
                        case 1:
                                return (float)valor1-valor2;
                                break;
                        case 2:
                                return (float)valor1*valor2;
                                break;
                        case 3:
                                return (float)valor1/valor2;
                                break;
                        default:
                                return 0;
                                break;
                }
        }
    }
}
```

12.2.2.2.3 Código en LabVIEW

Como en este ejemplo no hace falta tener ningún control en el Panel Frontal, ni la clase diseñada lo tiene, no será necesario un *Container*, en su lugar se empleará *Constructor Node* para crear una instancia de la clase. Al colocar esta función en el Diagrama de Bloques aparecerá un menú en el que se puede elegir un *assembly* y un objeto en él, como muestra la figura 12-19.

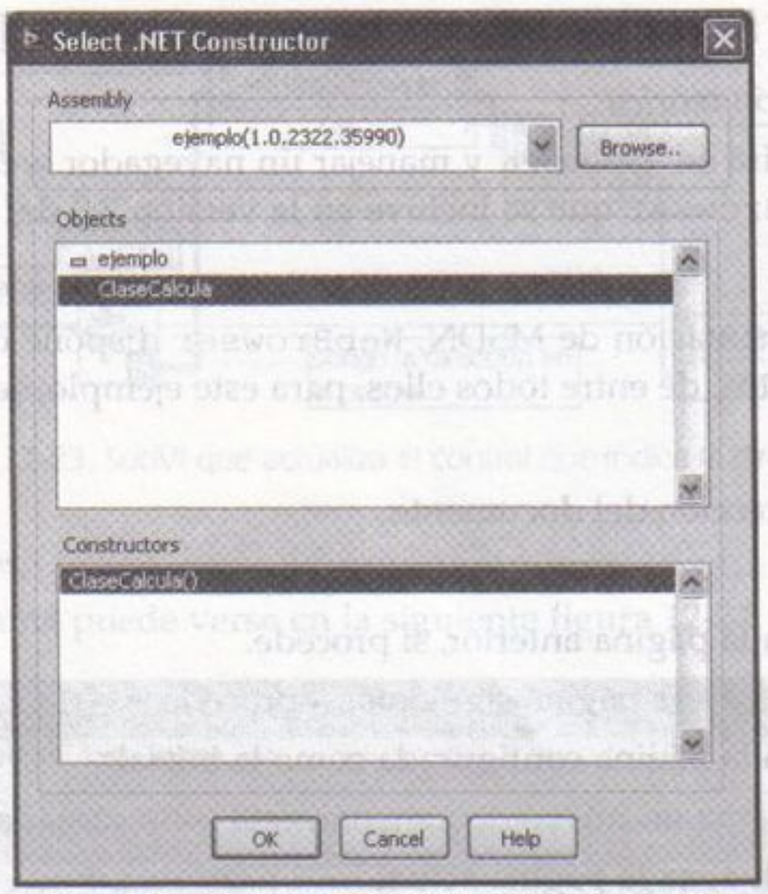


Figura 12-19. Selección de un constructor .NET

Una vez creado el objeto de la clase deseada se podrán usar los nodos de propiedades y métodos sobre la referencia. En este ejemplo se usará un nodo propiedad en el que se escribirán las dos propiedades para dar valor a los dos operadores y después se invocará al método para realizar el cálculo.

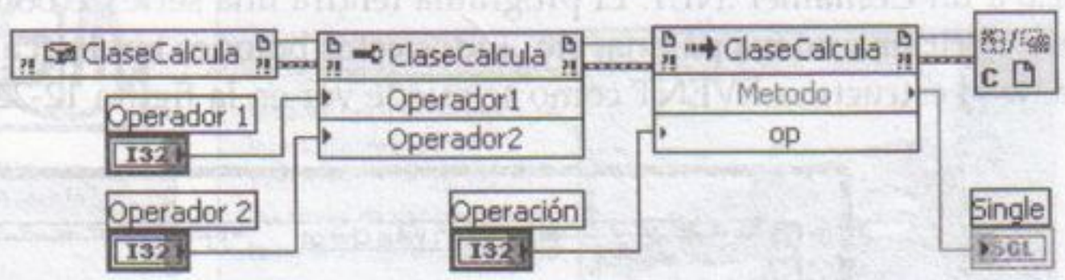


Figura 12-20. Código del ejemplo en .NET

12.2.2.2.4 Resultados

En la figura 12-21 puede verse la ejecución del código anterior.

Figura 12-21. Panel Frontal del resultado

12.2.2.3 Ejemplo II: Navegador Web

12.2.2.3.1 Explicación teórica

Este ejemplo consistirá en embeber y manejar un navegador web a través de un *container* llamado `WebBrowser` que se incluye en la versión 2.0 del .NET Framework de Microsoft.

Como dice la documentación de MSDN, `WebBrowser` dispone de muchas propiedades, métodos y eventos, de entre todos ellos, para este ejemplo, se usarán solamente:

- Propiedades:
  - `Url`: es la dirección del documento.
- Métodos
  - `GoBack`: va a la página anterior, si procede.
  - `GoForward`: va a la página siguiente, si procede.
  - `GoHome`: va a la página configurada como la inicial.
  - `Navigate`: carga un nuevo documento, está sobrecargada.
  - `Refresh`: actualiza la página actual.
- Eventos
  - `Navigated`: evento que se dispara cuando se accede a un nuevo documento.

12.2.2.3.2 Código en LabVIEW

En primer lugar se debe asociar el control System.Window.Forms.WebBrowser de .NET Framework 2.0 a un *Container* .NET. El programa tendrá una serie de botones para ir atrás, adelante, abrir una nueva página, etc. Todos estos botones tendrán asociados un subdiagrama de la estructura EVENT como se puede ver en la figura 12-22.

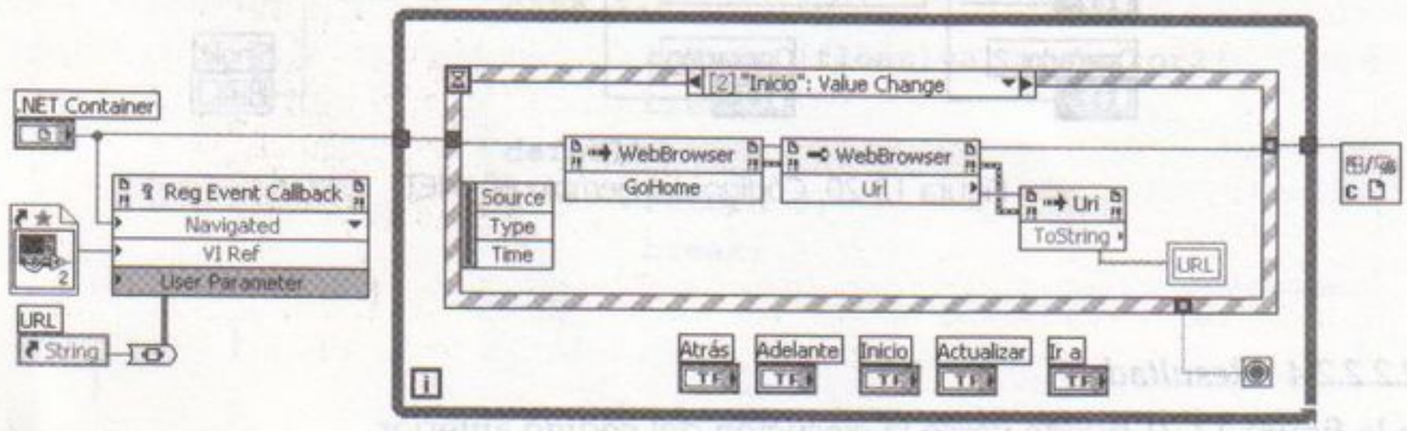


Figura 12-22. Navegador web

Al principio del programa también se debe registrar el evento Navigated. El subVI que atiende el evento Navigated se ejecuta cada vez que cambia la dirección de la página mostrada, su función será actualizar el control que indica la dirección, por eso como parámetro recibe una referencia a ese control. Su código se muestra en la figura 12-23.

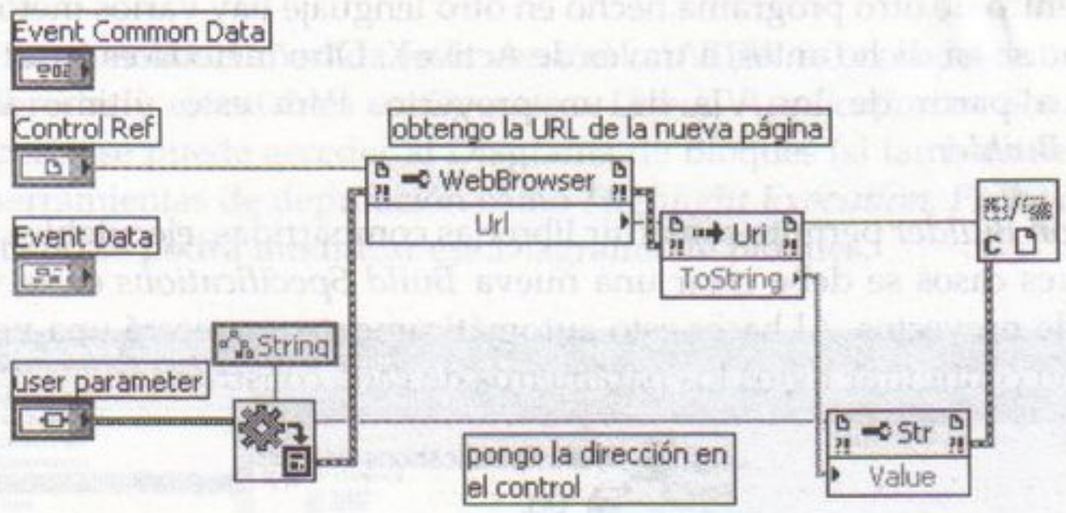


Figura 12-23. SubVI que actualiza el control que indica la dirección

12.2.2.3.3 Resultados

El aspecto del programa puede verse en la siguiente figura 12-24.

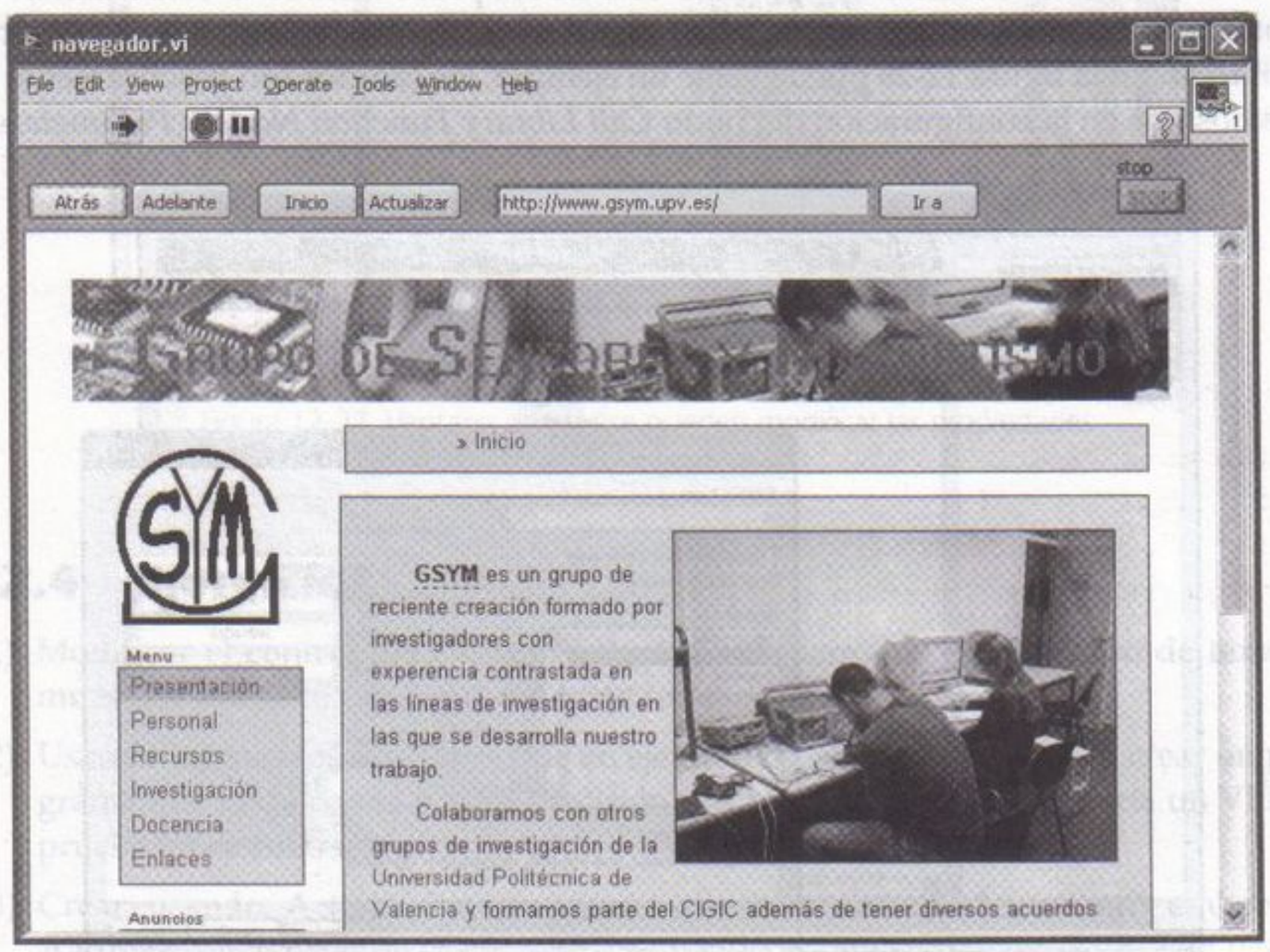


Figura 12-24. Ejecución del Navegador Web sobre LabVIEW

12.3 Creación de librerías y ejecutables con LabVIEW

En este capítulo hasta ahora se han visto métodos para comunicar otros programas con nuestros VIs. Si lo que se desea es el proceso contrario, es decir, usar un programa de LabVIEW dentro de otro programa hecho en otro lenguaje hay varios métodos, uno de ellos es, como se ha dicho antes, a través de ActiveX. Otro método es crear una librería compartida a partir de los VIs de un proyecto. Para esto último se empleará *Application Builder*.

El *Application Builder* permite construir librerías compartidas, ejecutables e instaladores. En los tres casos se debe crear una nueva *Build Specifications* en la ventana del explorador de proyectos. Al hacer esto automáticamente aparecerá una ventana en la que se podrán configurar todos los parámetros de cada construcción.

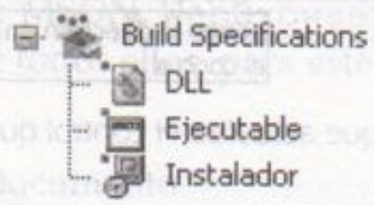


Figura 12-25. Build Specifications

En el caso de crear librerías compartidas uno de los pasos incluirá la construcción del prototipo de las funciones a partir de los terminales definidos en los VIs que contenga la librería. En la figura 12-26 puede verse un ejemplo, los campos son prácticamente los mismos que en la configuración del nodo *Call Library Function Node > Parameters*.

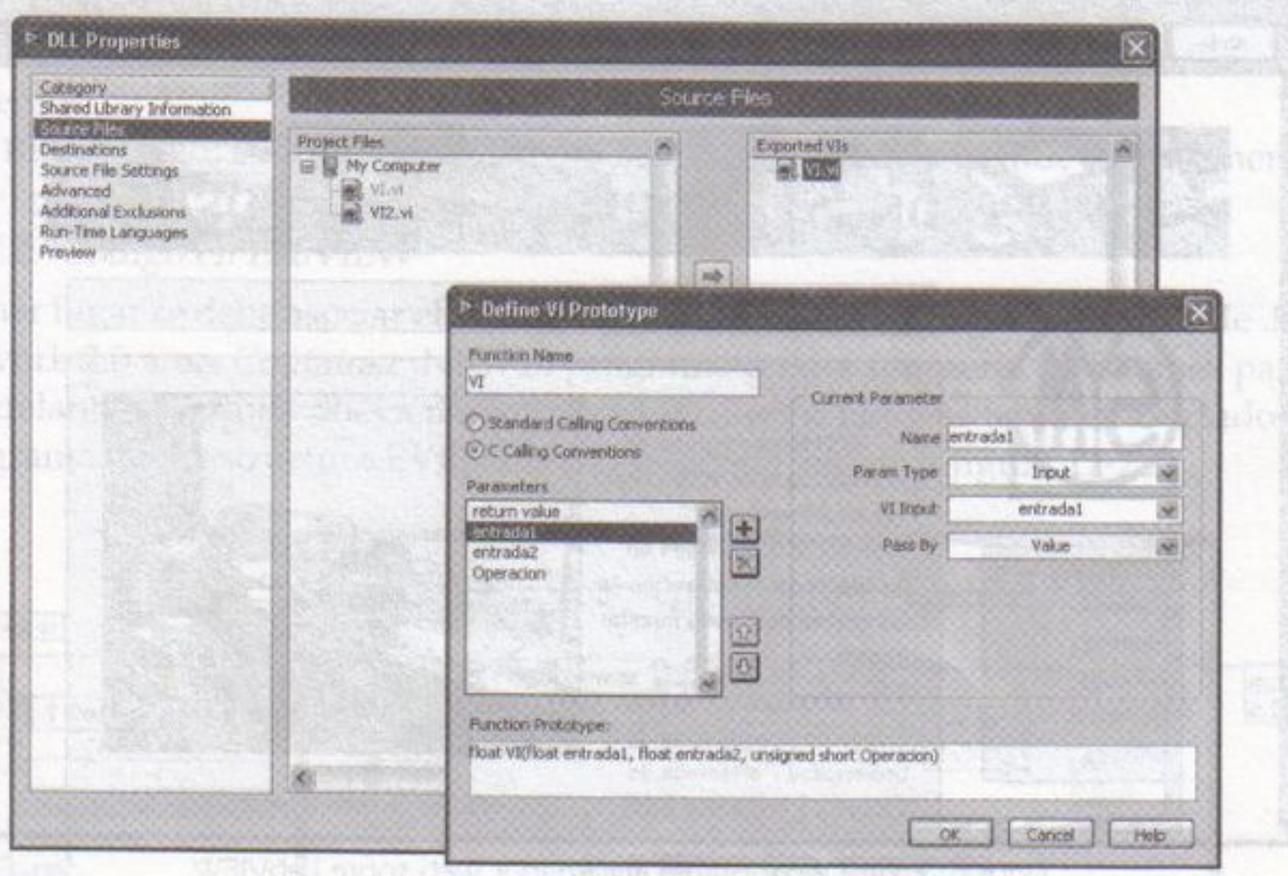


Figura 12-26. Definición de prototipo

La mayor aplicación de *Application Builder* es crear ejecutables. En este caso los principales parámetros a configurar son el VI principal, los VIs llamados dinámicamente (a través de VI Server) y las propiedades de cada VI. En la figura 12-27 puede verse la ventana en la que se pueden sobrescribir las propiedades de los VIs (*File >VI Properties*) al crear un ejecutable.

Una nueva funcionalidad añadida en la versión LabVIEW 8.0 es la posibilidad de depuración de librerías y ejecutables en *Operate > Debug Application or Shared Library...* Con esta opción se puede acceder al Diagrama de Bloques (si también se ha incluido) y usar las herramientas de depuración como *Highlight Execution, Probe* o *Breakpoint*. Lógicamente no se podrá modificar ese Diagrama de Bloques.

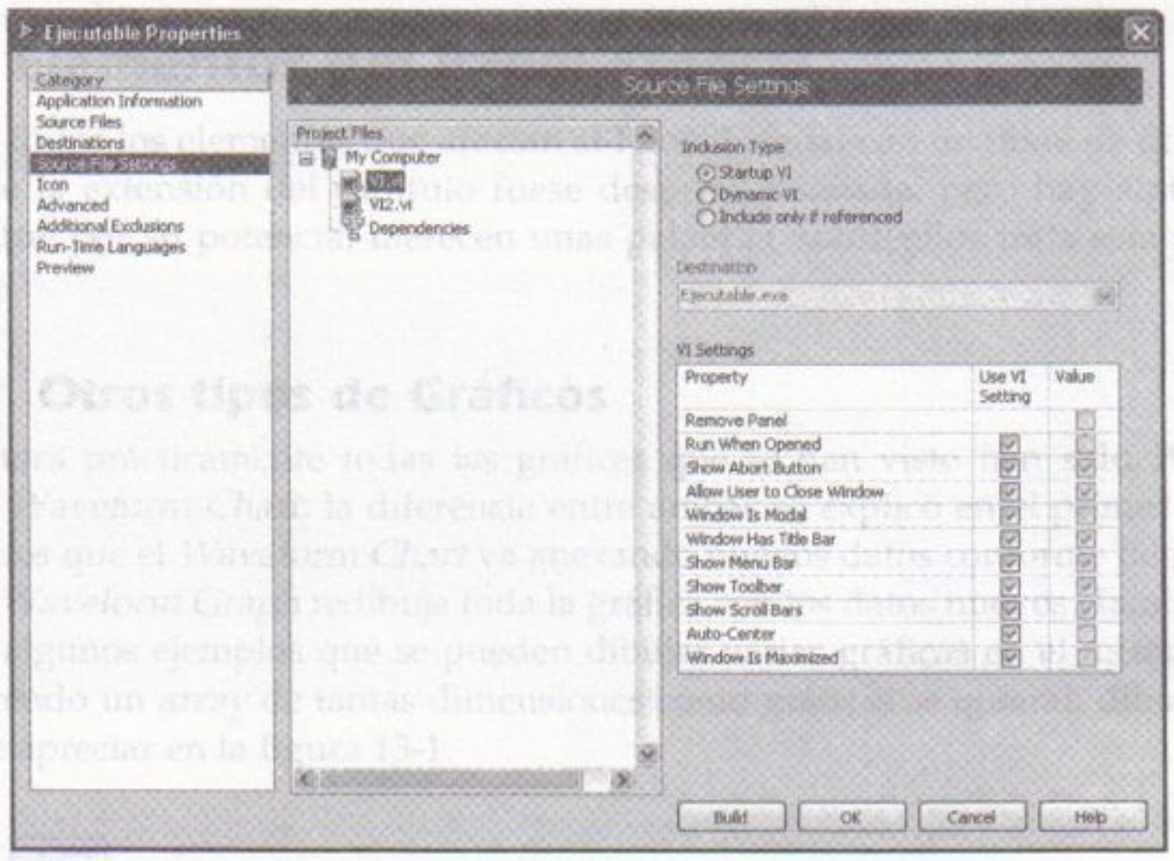


Figura 12-27. Ventana en que se pueden modificar las propiedades

12.4 Ejercicios

1) Modificar el control del ejemplo I de ActiveX para que en el campo de texto se muestren asteriscos en lugar del texto tecleado.
2) Usando las mismas funciones que en ejemplo II de librerías externas, crear un programa que mida con la mayor precisión posible el tiempo que tardaría un VI bajo prueba en ejecutarse.
3) Crear usando ActiveX un programa que muestre un documento embebido. Algunas sugerencias de documentos son: videos, una animación flash, una página web, ficheros de office, ficheros PDF...

12.5 Bibliografía

Apple, *Developer Connection.*

Archana Shrotriya, *AN087: Writing Win32 Dynamic Link Libraries (DLLs) and Calling Them from LabVIEW*, 1996.

Jeff Ferguson et alt., *C# Bible*, Wiley Publishing, 2002.

Microsoft, *MSDN.*

National Instruments, *LabVIEW, Using External Code in LabVIEW*, 2003.

Capítulo 13

Optimización del interfaz

13.1 Elementos del Panel Frontal

Estudiar todos los elementos que afectan al Panel Frontal con un nivel de detalle alto haría que la extensión del capítulo fuese desproporcionada, pero hay algunos elementos que por su potencial merecen unas palabras, sobre ellos trata esta parte del capítulo.

13.1.1 Otros tipos de Gráficos

Hasta ahora prácticamente todas las gráficas que se han visto han sido *Waveform Graph* o *Waveform Chart*, la diferencia entre ambas se explicó en el primer capítulo, recordemos que el *Waveform Chart* va anexando nuevos datos conforme llegan mientras que *Waveform Graph* redibuja toda la gráfica con los datos nuevos. También se ha visto en algunos ejemplos que se pueden dibujar varias gráficas en el mismo control construyendo un *array* de tantas dimensiones como gráficas se quieran dibujar, como se puede apreciar en la figura 13-1.

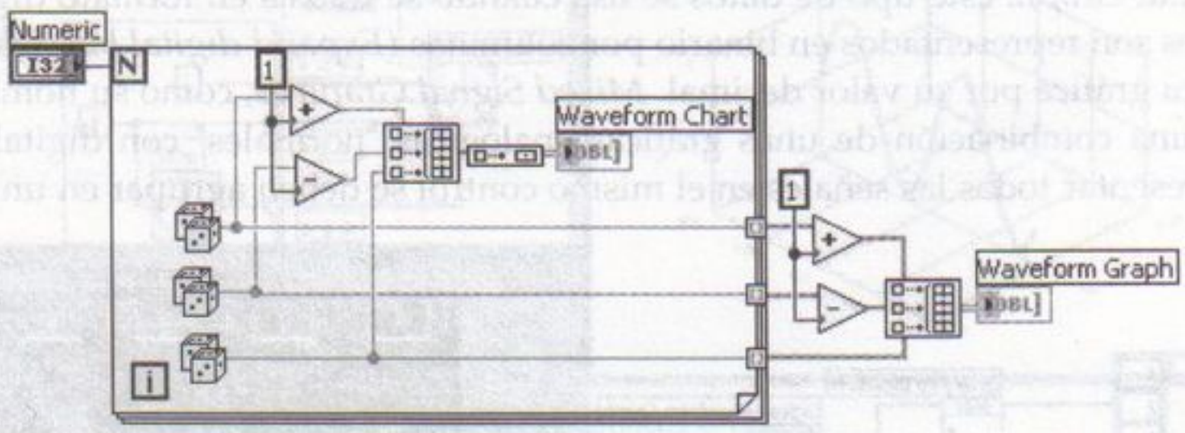


Figura 13-1. Representación de varios gráficos con el mismo control

Además de los dos tipos de gráficas básicos se dispone de otros, por ejemplo:

- *XY Graph*: son un tipo de gráfico muy genérico. El tipo de datos que aceptan consiste en un *cluster* de dos componentes, cada uno es un *array* que contiene los valores de la gráfica para los ejes X e Y (aunque también pueden aceptar un *array* de *clusters* compuestos por dos puntos). Para mostrar más de una gráfica en el mismo

control se debe construir un *array* en el que cada elemento será una gráfica con el formato anterior. En la figura 13-2 se muestra un ejemplo.

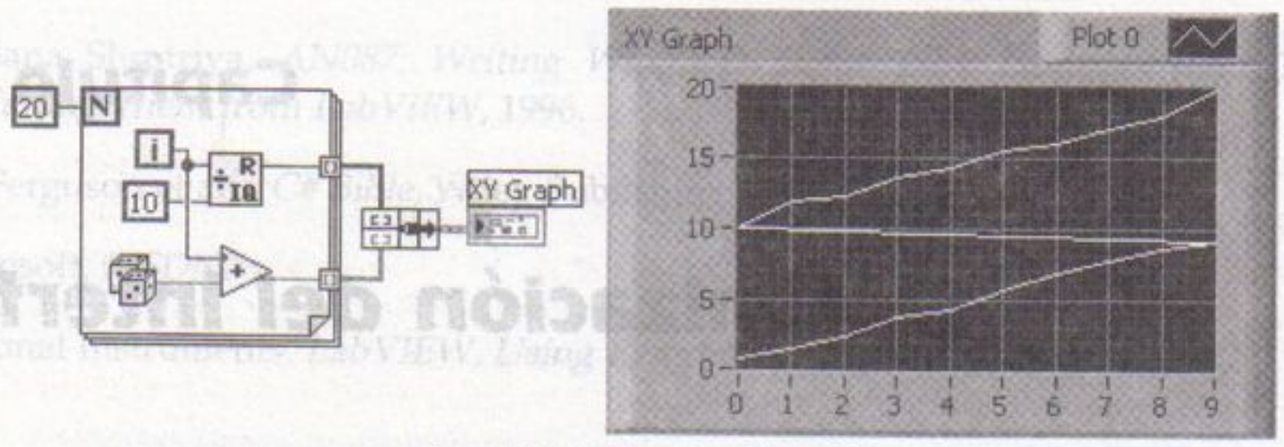


Figura 13-2. Tipo de gráfico XY

- *Intensity* (*Graph* y *Chart*): se usan para mostrar datos de tres dimensiones en una gráfica de dos dimensiones, la tercera dimensión se sustituye por la intensidad de un color.

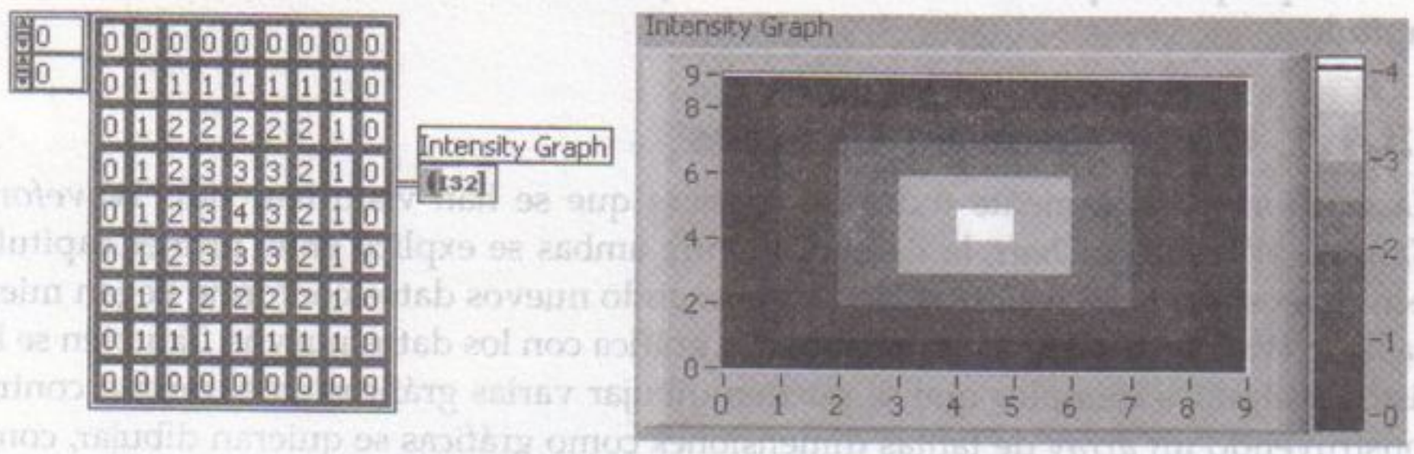


Figura 13-3. Gráfico tipo *Intensity*

- *Digital Graph*: este tipo de datos se usa cuando se trabaja en formato digital. Los datos son representados en binario por columnas (*Expand digital buses*) o en una única gráfica por su valor decimal. *Mixed Signal Graph* es, como su nombre indica, una combinación de unas gráficas analógicas "normales" con digitales, para representar todas las señales en el mismo control se deben agrupar en un *cluster*.

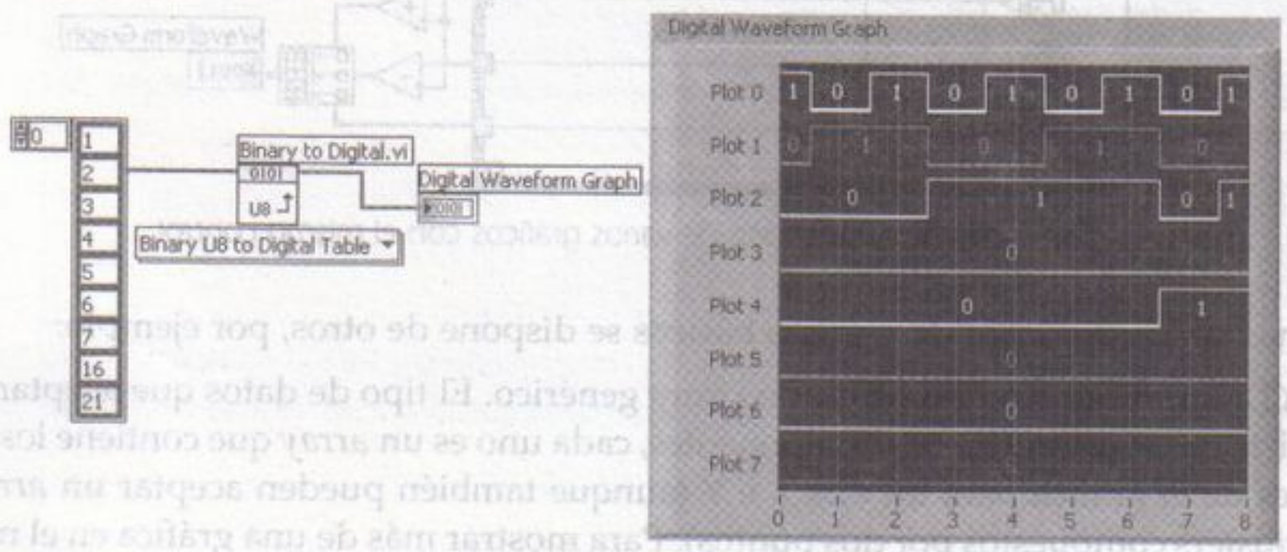


Figura 13-4. *Digital Graph*

Pictures: además de lo mencionado hasta ahora, también se pueden representar gráficos mediante los controles *Picture*. Para emplearlos se hará uso de los VIs del menú *Programming > Graphics & Sound > Picture Plots*, que incluyen funciones para representar, entre otros, diagramas polares y de Smith, como se observa en la figura 13-5.

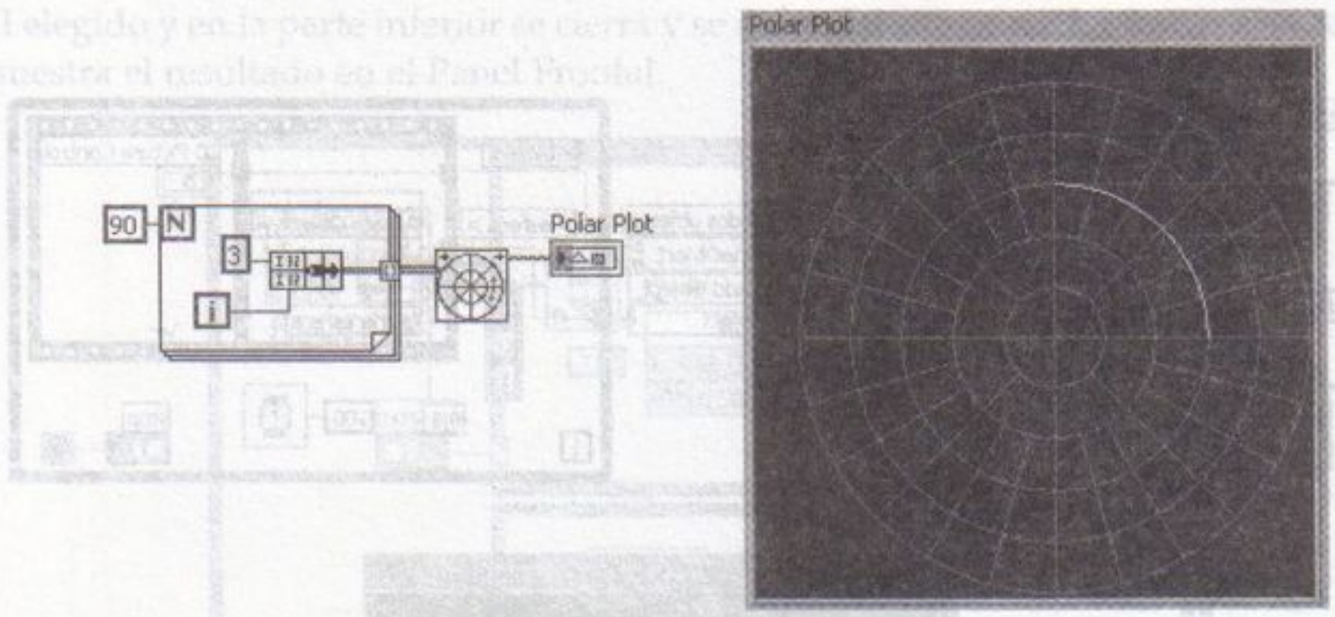


Figura 13-5. Utilización de controles *Pictures* para representar gráficos

- *3D Graph*: estas gráficas incluyen las de curvas, superficie y paramétricas. No están disponibles en todas las versiones ni en todas las plataformas. Se basan en la tecnología ActiveX y se manejan gracias a los VIs de la paleta *Programming > Graphics & Sound > 3D Graph Properties*. Para personalizar estos gráficos hay que acceder a sus propiedades desde el menú contextual.

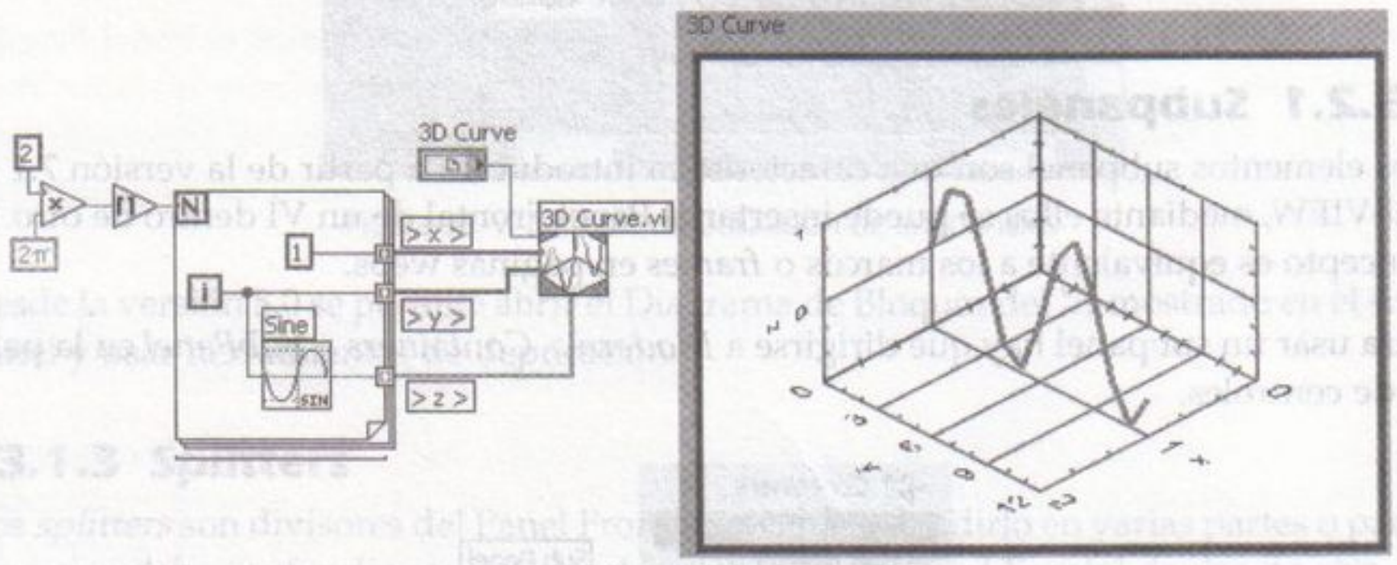


Figura 13-6. Gráficos en 3D

- *3D Picture Control*: este control es una de las novedades de la versión 8.20, permite crear varios objetos 3D en un control (escena) y manipularlos de forma individual, bien a través de los VIs del menú *Programming > Graphics & Sound > 3D Picture Control* o de las propiedades y métodos del control. En la figura 13-7 se ve un programa en el que se crean dos objetos llamados *cubo* y *esfera*, la esfera se mueve a la posición (0,5 0,5 0,5) y se añaden los dos a la misma escena con el método *Object > Add Object*; dentro del bucle simplemente se va rotando la esfera

mientras se visualizan los dos objetos; el resultado es un control que muestra ambos objetos, el cubo fijo y la esfera desplazándose; en la parte inferior de la imagen se puede ver una captura de pantalla. Desde el Panel Frontal, mediante el menú contextual del control también se puede cambiar la perspectiva y el control de cámara.

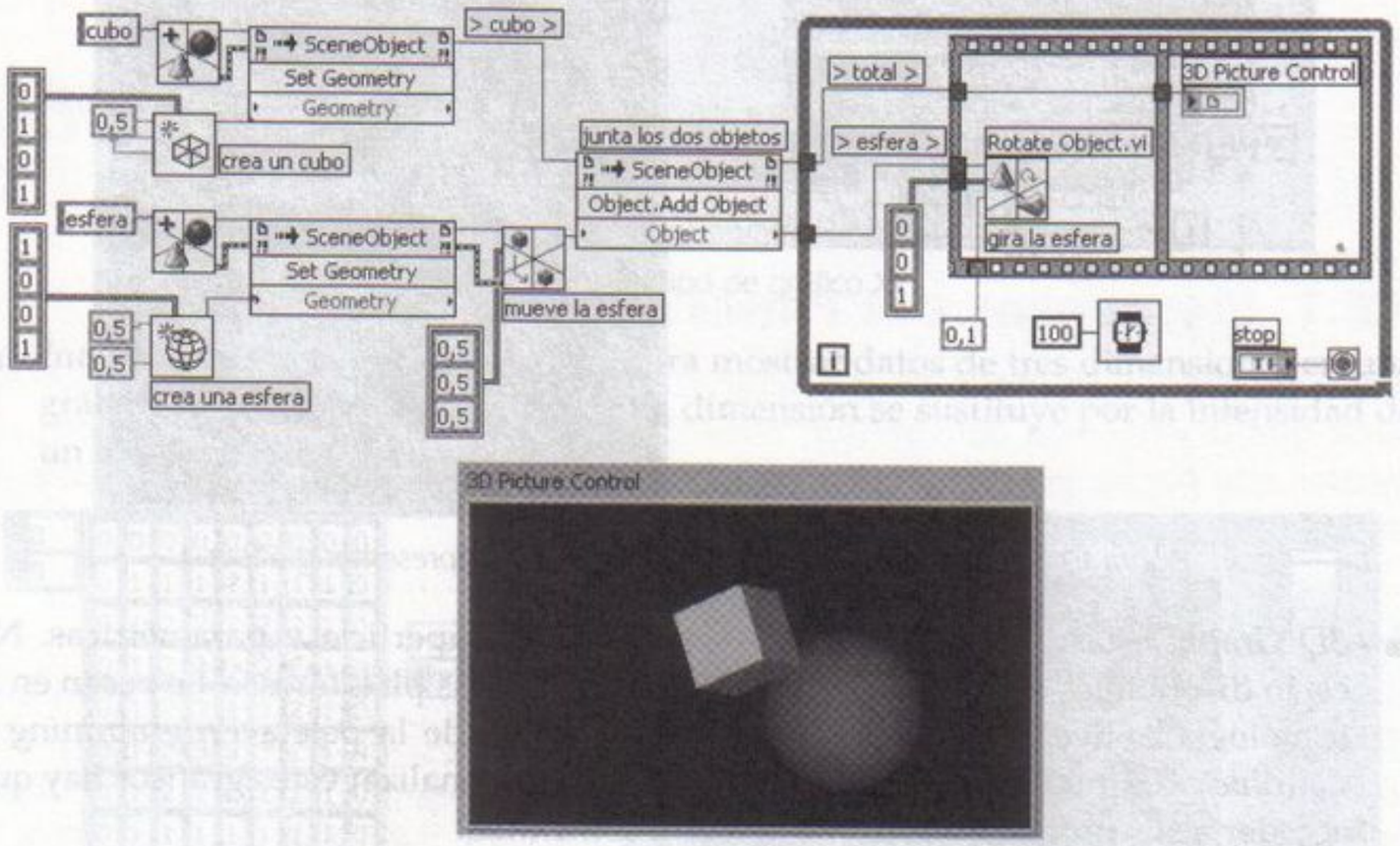


Figura 13-7. 3D Picture Control

13.2.1 Subpaneles

Los elementos subpanel son una característica introducida a partir de la versión 7.1 de LabVIEW, mediante ellos se puede insertar el Panel Frontal de un VI dentro de otro. El concepto es equivalente a los marcos o *frames* en páginas webs.

Para usar un subpanel hay que dirigirse a *Modern > Containers > SubPanel* en la paleta de controles.

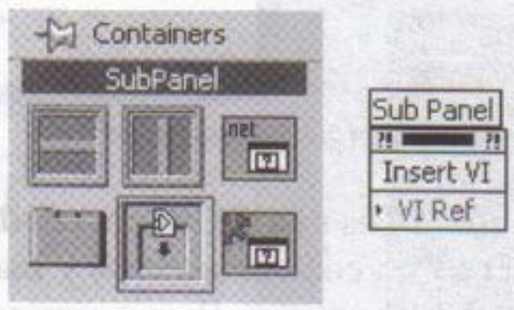


Figura 13-8. Menú Containers

Al colocar un subpanel en el Panel Frontal aparece un nodo llamado *Insert VI* en el Diagrama de Bloques, se trata de un método del control subpanel al que hay que conec-

tar una referencia del VI que se mostrará en su interior. Otro método que tiene el subpanel es el de *Remove VI,* que sirve para cerrar y quitar el VI del subpanel.

En la figura 13-9 se ve un pequeño ejemplo en el que se puede elegir entre dos VIs para mostrar en el subpanel. En el CASE de la parte superior se muestra el Panel Frontal del VI elegido y en la parte inferior se cierra y se quita del subpanel. La figura 13-9 también muestra el resultado en el Panel Frontal.

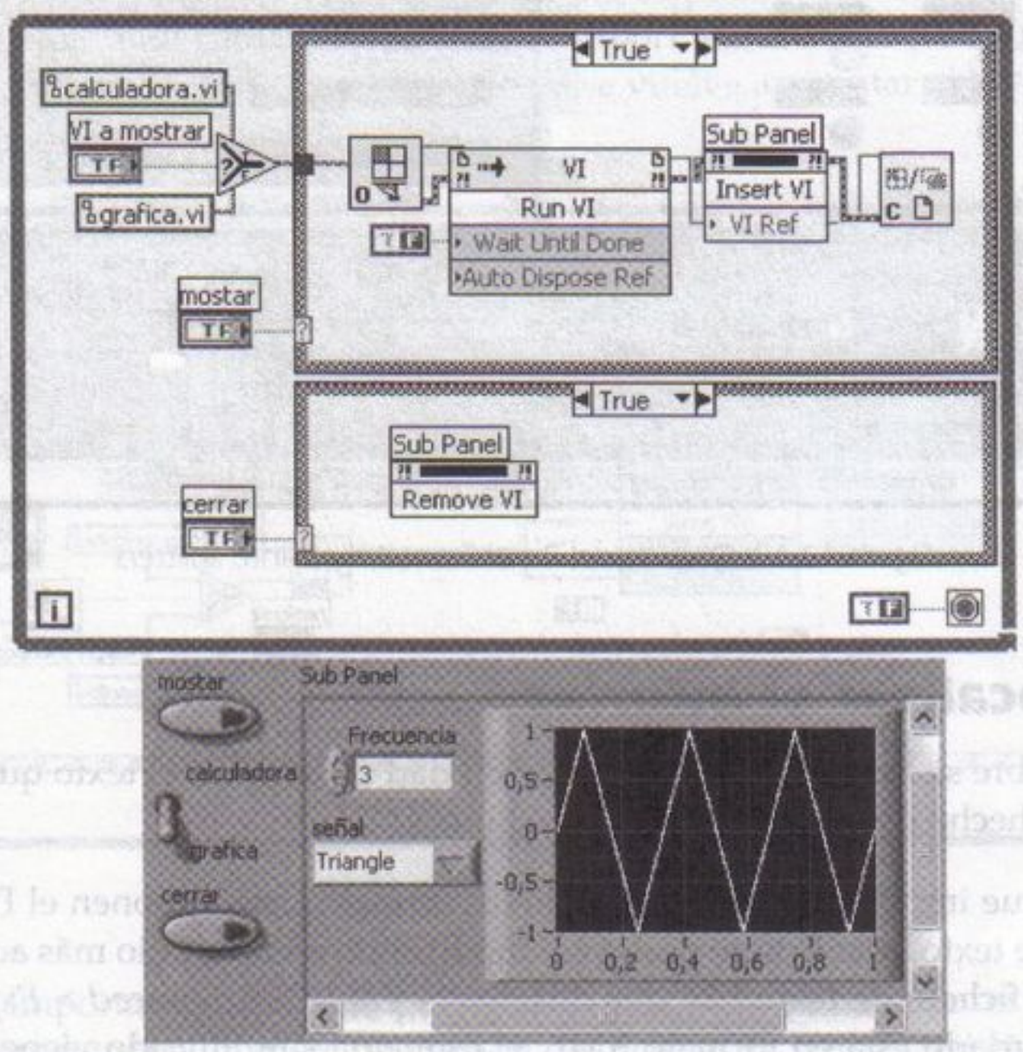


Figura 13-9. Ejemplo de utilización de subpaneles

Desde la versión 8.0 se permite abrir el Diagrama de Bloques del VI mostrado en el subpanel y usar herramientas de depuración.

13.1.3 Splitters

Los *splitters* son divisores del Panel Frontal, permiten dividirlo en varias partes o paneles; no se debe confundir con subpanel que inserta un Panel Frontal dentro de otro, en el caso de los *splitters* su función es básicamente cosmética, todos los controles e indicadores mostrados pertenecerán al mismo VI. Es útil para realizar agrupaciones cuando en una pantalla hay una gran cantidad de controles e indicadores.

Los *splitters* tienen asociados eventos a los que se puede acceder desde la estructura EVENT y usar VI Server para interactuar con ellos de forma programada.

En la figura 13-10 se puede ver un programa en el que, mediante *splitters,* se ha dividido el Panel Frontal en tres paneles.

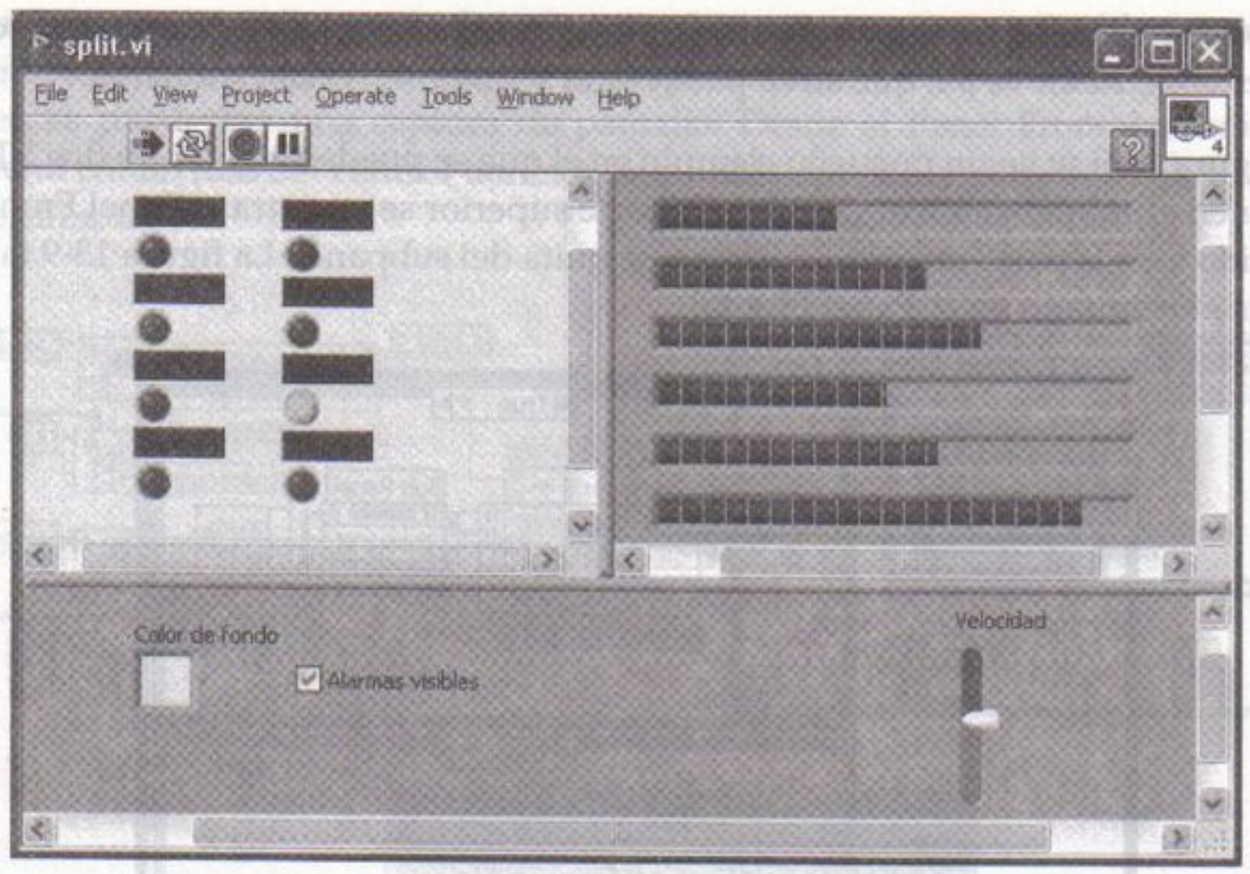


Figura 13-10. División del Panel Frontal mediante splitters

13.2 Localizar el interfaz

Bajo este nombre se hace referencia a la posibilidad de adaptar el texto que aparece en un programa hecho con LabVIEW a otros idiomas.

Esto se consigue importando y exportando los *strings* que componen el Panel Frontal a un fichero de texto, este fichero podrá ser modificado e importado más adelante. Para exportar a un fichero de texto hay que dirigirse a *Tools > Advanced > Export Strings*. El fichero guardado está en formato XML, el esquema simplificado viene dado por el siguiente texto:

```
<VI>
    <CONTENT>
        <CONTROL>
            <DESC>...</DESC>
            <TIP>...</TIP>
            <PARTS>...</PARTS>
        </CONTROL>
    </CONTENT>
</VI>
```

Cada elemento del Panel Frontal se define entre <CONTROL> y </CONTROL>. El contenido de las marcas DESC es la descripción que aparece en la ventana de la ayuda contextual. TIP es el pequeño mensaje que aparece al dejar el ratón sobre el elemento y PARTS son los otros elementos que componen al control: *caption*, texto en el interior de un botón, etc.

Una vez modificado podrá ser importado o guardado con otro nombre para luego ser cargado de forma programada. Esto último se consigue invocando un método del VI llamado *VI Strings > Import*. Un detalle importante es que este método no puede modificar un VI que se está ejecutando.

En la figura 13-11 se puede ver un VI en el que se modifica el idioma de otro VI. En primer lugar se elige el fichero en que están los datos del idioma elegido y después se abre una referencia al VI que se modificará, como no se puede importar el fichero si el VI está ejecutándose se detendrá con el método *Abort VI*, una vez parado ya se está en condición de importar el fichero y finalmente se vuelve a ejecutar el VI con el método *Run VI*.

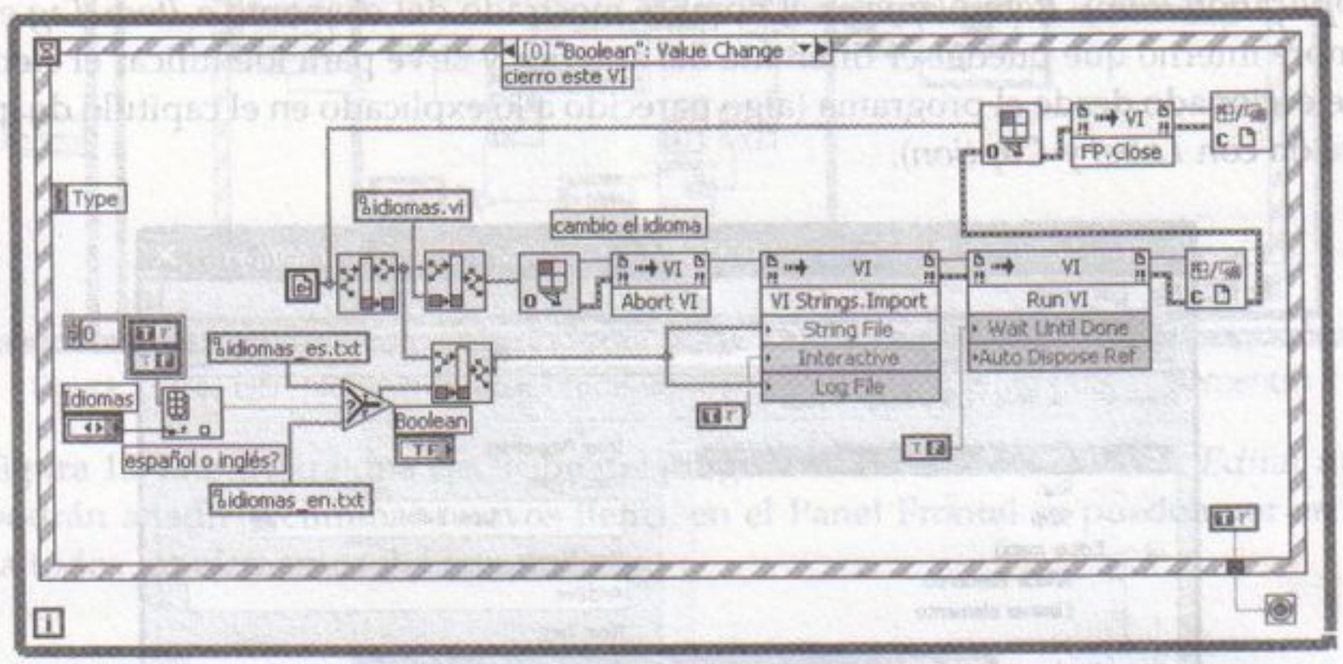


Figura 13-11. Selección del idioma de un VI

También es importante decir que por cada cambio que se haga en el Panel Frontal, LabVIEW aumentará el número de versión del fichero de *strings* con el que trabaja.

13.3 Interactuando con el usuario programadamente

A pesar de que hay una clara distinción entre Panel Frontal y Diagrama de Bloques, se pueden realizar modificaciones en el primero a través del segundo, es decir, modificar el aspecto de la ventana desde el programa. Con esto se consigue un programa que interactúa de forma mucho más dinámica con el usuario.

Las principales herramientas para conseguir esto ya se han visto en los capítulos anteriores, estos son: las propiedades de los controles, estudiadas en el capítulo 3 y VI Server, en el capítulo 9. En esta sección se tratarán otros aspectos también importantes en la comunicación usuario-máquina.

13.3.1 Menús personalizados

En los VIs de interfaz de usuario puede ser útil personalizar, o por lo menos restringir, el menú de la parte superior de la pantalla mientras el VI se está ejecutando.

Para crear o modificar un menú personalizado se debe ir a *Edit > Run-Time Menu*, una vez abierta la ventana de la figura 13-12 se pueden añadir nuevos elementos con el botón [+] y eliminar otros con [×]; los botones con forma de flecha sirven para ordenar y anidar los elementos del menú. En *Preview* se puede ver el aspecto del menú conforme se va creando. En la parte derecha, el grupo llamado *Item Properties* configura cada uno de los elementos del menú, éstos pueden ser definidos por el usuario (*User Item*), puede ser un separador o un elemento del menú habitual de LabVIEW (*Application Item*). *Item Name* es el nombre mostrado del elemento e *Item Tag* es un nombre interno que puede ser diferente del anterior y sirve para identificar el elemento seleccionado desde el programa (algo parecido a lo explicado en el capítulo de introducción con *Label* y *Caption*).

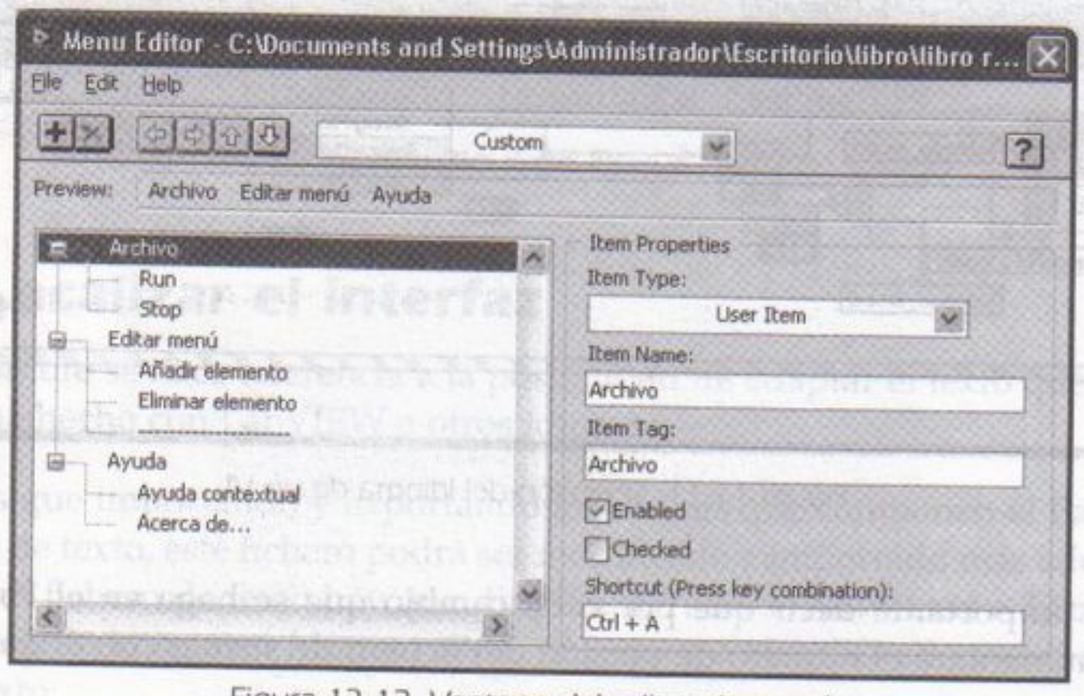


Figura 13-12. Ventana del editor de menús

El menú se puede manejar desde el programa a través de los VIs de la paleta *Programming > Dialog & User Interface > Menu*. Estos VIs permiten modificar el menú, obtener la opción seleccionada por el usuario u otra información. La estructura EVENT también tiene algunos eventos útiles para manejar el menú.

Figura 13-13. Paleta de los VIs para manejar el menú

En el ejemplo de la figura 13-14 se muestra el código de un programa que en el interior del CASE modifica un menú insertando nuevos elementos, además cada elemento nuevo tendrá asociado un atajo de teclado. Después del CASE se obtienen los nombres internos de todos los elementos que conforman el menú. Nótese que la estructura del programa empleada es una *Menu Driven State Machine* conducida por eventos y un solo hilo.

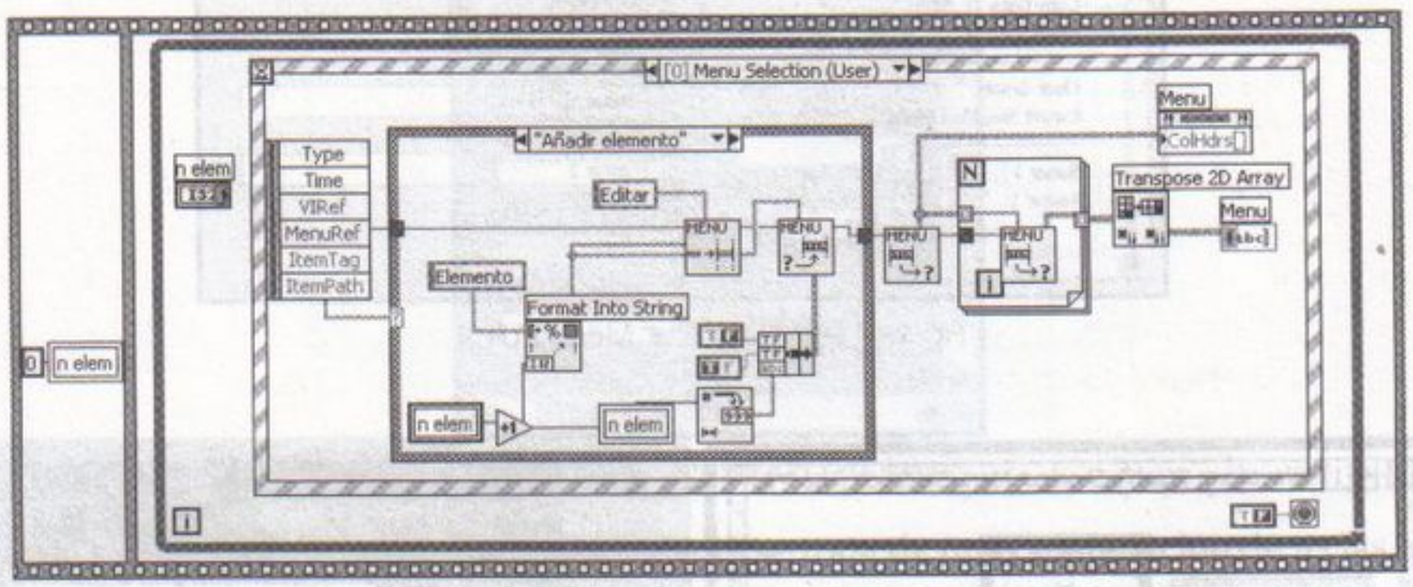


Figura 13-14. Ejemplo de modificación de un menú donde se insertan nuevos elementos

La figura 13-15 muestra una ejecución del programa anterior. En el menú *Editar menú* se podrán añadir y eliminar nuevos ítems, en el Panel Frontal se pueden ver en una tabla todos los elementos del menú.

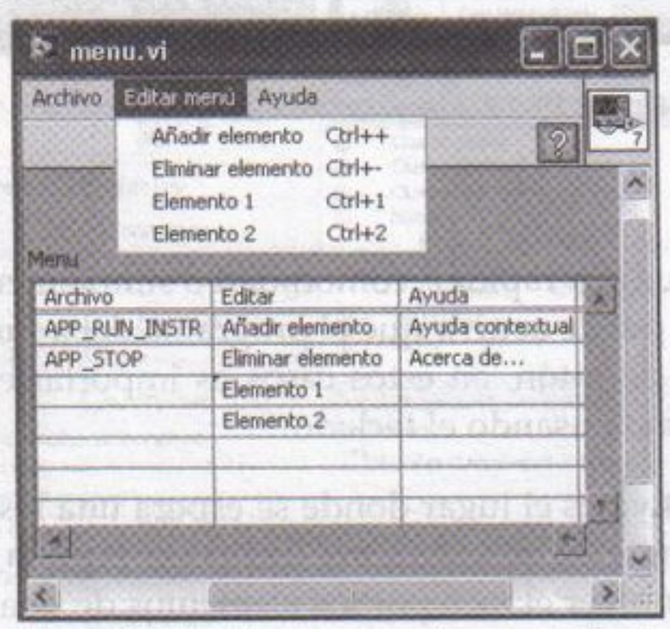


Figura 13-15. VI con un menú personalizado

Por otra parte, también se puede crear un menú contextual para cada elemento del Panel Frontal, de esta manera cuando se presione el botón secundario del ratón sobre ese elemento aparecerá el menú creado. Para esto hay que seleccionar el elemento e ir al *menú contextual > Advanced > Run-Time Shortcut Menu > Edit...*

La forma de crear el menú es igual a la explicada antes y las funciones para interactuar con el menú desde el código también serán las mismas; en este caso el evento será

Shortcut Menu Selection (User). En la figura 13-16 y figura 13-17 pueden verse el editor y el resultado.

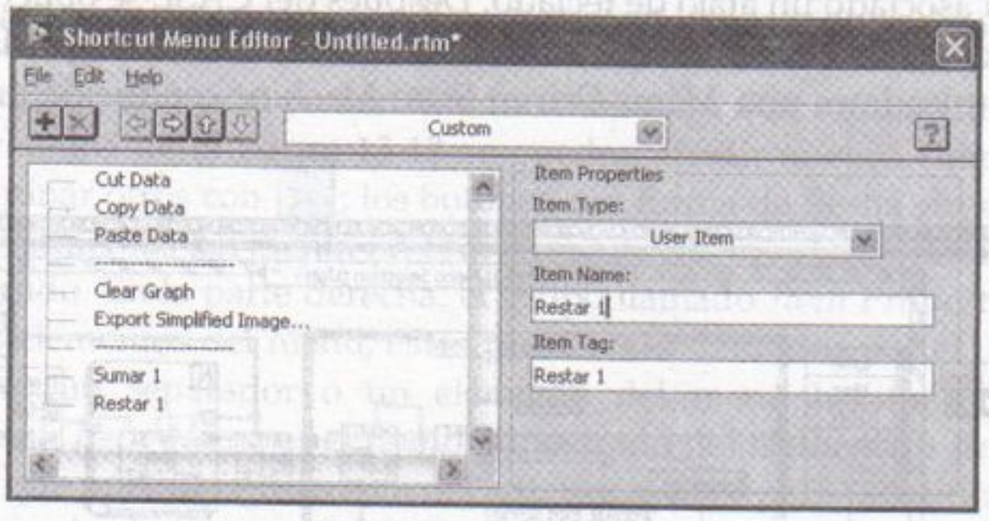


Figura 13-16. Shortcut Menu Editor

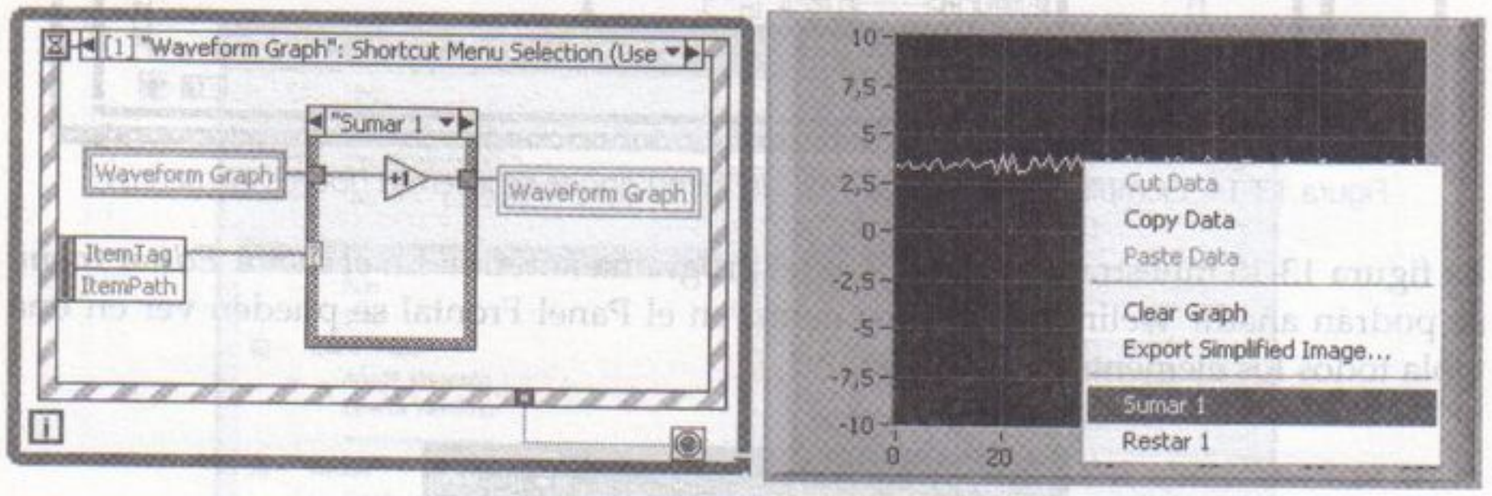


Figura 13-17. Menú contextual personalizado

13.3.2 Teclado

Hay muchos usuarios que por rapidez, comodidad o simplemente costumbre prefieren usar el teclado, también puede ocurrir que el programa deba funcionar sobre un dispositivo que no disponga de ratón, en estos casos es importarte facilitar que se pueda interactuar con el programa usando el teclado.

El foco de la ventana activa es el lugar donde se espera una respuesta del teclado, por ejemplo, la entrada de un texto en un control de tipo *string* o la activación de un botón mediante ENTER. Para mover el foco por los elementos de una ventana se usa la tecla TAB. En LabVIEW se puede definir el orden al que llamaremos orden de tabulación por medio de *Edit > Set Tabbing Order...*

En esta ventana, cuando se haga clic sobre un control, éste tendrá el orden del foco indicado en el campo llamado *Click to set to,* este número aumentará automáticamente cuando sea asignado. Cada control posee dos números en su esquina inferior derecha: el que tiene el fondo blanco era su orden de tabulación antes de modificarlo y el que tiene el fondo negro es su nuevo orden, según se puede apreciar en la figura 13-18. Una

vez que ya esté definido el orden deseado sólo hay que presionar el botón *Confirm* de la barra de herramientas para volver al Panel Frontal. También se puede modificar el orden de tabulación de los controles dentro de los *clusters* por medio de su *menú contextual > Reorder Controls In Cluster...*

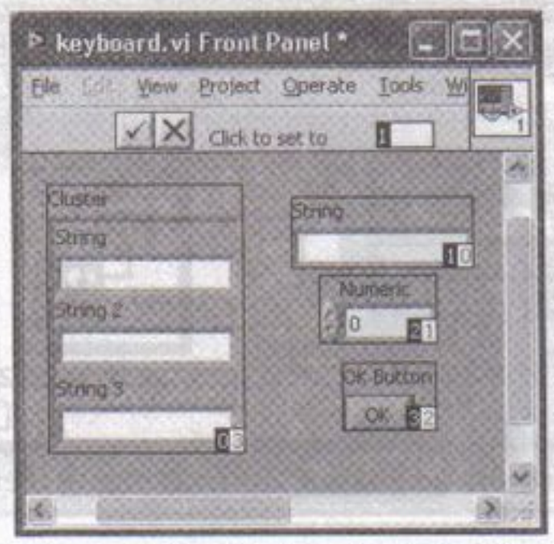


Figura 13-18. Orden de tabulación

Las *Key Navigation* son los atajos de teclado, se trata de una combinación de teclas que accionándolas provocan una acción. Se puede asociar un atajo de teclado a cada control por medio del *menú desplegable > Properties > Key Navigation* o también en *Advanced > Key Navigation*, ver figura 13-19.

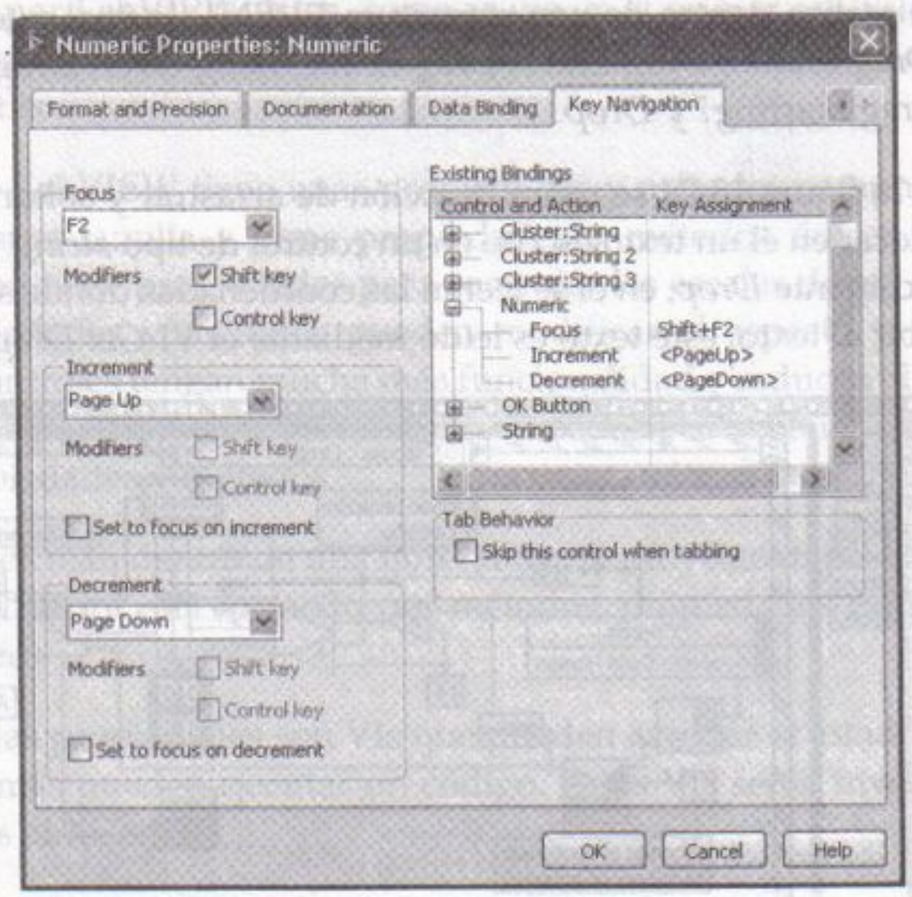


Figura 13-19. Atajos de teclado

Por último también se pueden asociar acciones a combinaciones de teclas de forma programada, sin necesidad de ningún control en el Panel Frontal. Para esto se emplearán

los VIs del menú *Connectivity > Input Device Control*. En la figura 13-20 se puede ver un ejemplo que muestra un mensaje cuando se presione la combinación de teclas CTRL+1.

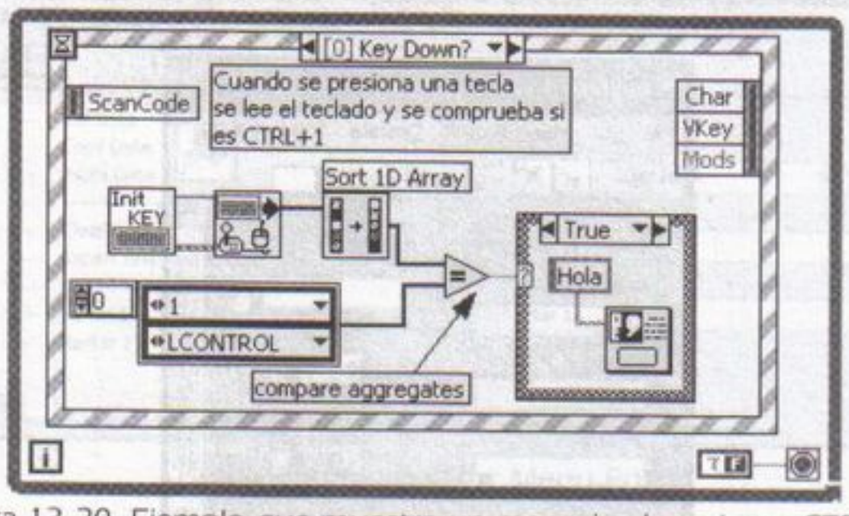


Figura 13-20. Ejemplo que muestra un mensaje al presionar CTRL+1

13.3.3 Drag & Drop

Drag & Drop es la acción de arrastrar y soltar un objeto sobre otro. Esta opción se introdujo en LabVIEW a partir de la versión 8.0.

Para usarla hay que habilitar esta característica en los controles o indicadores mediante un nodo de propiedad (si disponen de ella) o en su *menú contextual > Advanced*. La programación se realiza dentro de una estructura EVENT, la cual puede detectar distintos eventos: *Drag Ended, Drag Enter, Drag Leave, Drag Over, Drag Source Update, Drag Starting, Drag Starting?* y *Drop*.

En el ejemplo de la figura 13-21 se usará la acción de arrastrar y soltar sobre un dibujo (*Picture*) para colocar en él un texto escrito en un control de tipo *string*. El evento que se utilizará será únicamente *Drop*, en él se leerán las coordenadas donde se ha realizado la acción para escribir el texto, este texto es leído mediante el VI *Get Drag Drop Data*.

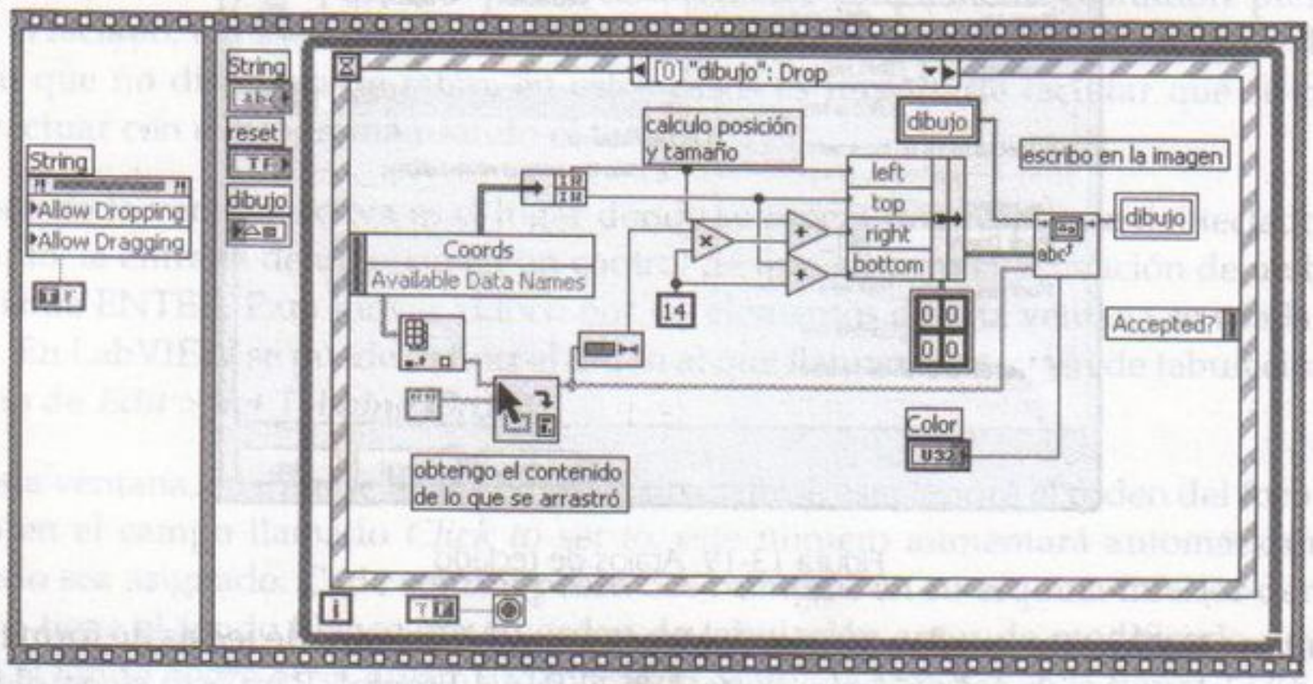


Figura 13-21. Ejemplo de Drag & Drop

Para ver el resultado se debe escribir en el *string* un texto, seleccionarlo, pinchar con el cursor sobre la selección, arrastrarlo hasta cualquier punto del dibujo y una vez allí soltar el botón del ratón.

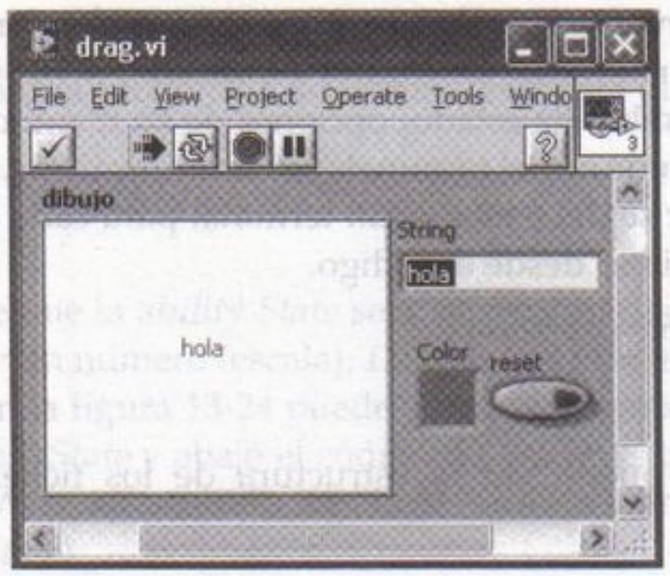


Figura 13-22. Uso de *Drag & Drop* para escribir sobre un *Picture*

13.4 XControls

La versión profesional de LabVIEW 8.0 añadía una nueva herramienta a la hora de programar: los XControls. Hasta ahora para crear el interfaz se disponía de los controles proporcionados por LabVIEW, éstos, como se vio en el primer capítulo, se pueden personalizar, pero también tiene sus límites, la personalización no deja de ser un cambio meramente estético.

Cada control de LabVIEW tiene unas propiedades y métodos comunes con otros controles de la misma familia y otras propiedades y métodos únicos, por ejemplo un *Waveform Graph* tiene propiedades para controlar las escalas de los ejes, los cursores, etc. Estas propiedades y métodos pueden modificar el aspecto y comportamiento, hacen que los controles tengan mucha más funcionalidad y reducen el tamaño del código del programa. Los XControls son controles creados por el usuario que también pueden incluir propiedades y métodos.

Los XControls se crean dentro de una librería de un proyecto; tienen datos y un estado asociado. Un XControl está formado por métodos, propiedades y otros componentes llamados *Abilities*.

Los métodos y las propiedades son VIs que pueden acceder al estado y a los datos del XControl y además pueden ejecutar un código. Estos VIs serán invocados a través de nodos asociados al XControl.

Los componentes de un XControl son:

- *State*: es el estado del XControl. Se trata de un *cluster* que contiene variables relativas a la apariencia y/o comportamiento definidas por el usuario.

- *Data*: indican el tipo de datos del XControl, por ejemplo numérico, *array*, *cluster*, etc.
- *Init*: inicialización.
- *Façade* (fachada): es el corazón del XControl. Manejan los estados y responden ante eventos. Los eventos pueden ser el cambio en el valor del dato, un cambio de dirección del terminal (pasar de control a indicador o viceversa), iniciar o finalizar la ejecución, un cambio de estado, opciones de su menú contextual, etc. Su Panel Frontal se corresponde con el terminal que verá el usuario, si el XControl puede ser control e indicador se debe añadir un terminal para cada estado, sobreponerlos y controlar su visibilidad desde el código.
- *Convert State For Save*: opcional.
- *Uninit*: opcional.

En la figura 13-23 se puede ver la estructura de los ficheros que conforman un XControl.

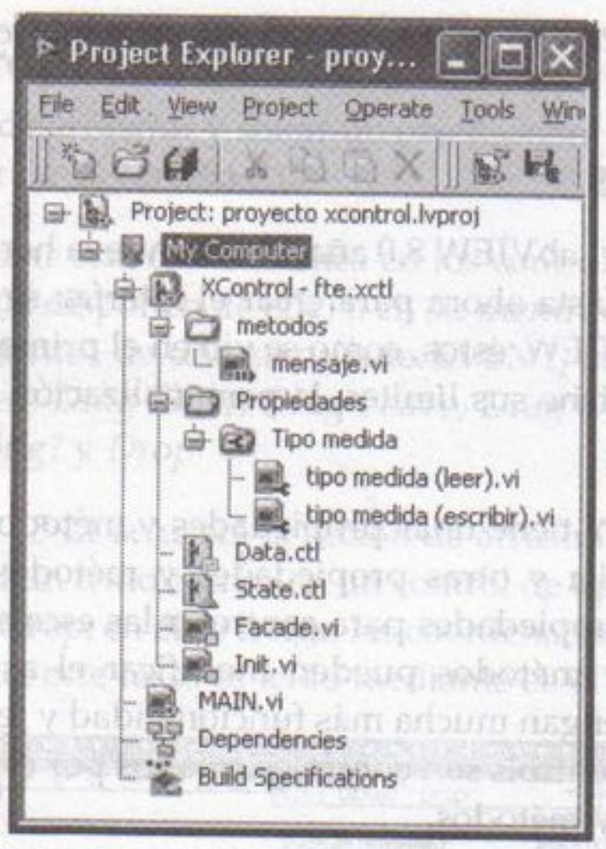


Figura 13-23. Estructura de VIs y controles que forman un XControl

Cuando se carga en memoria por primera vez un XControl se llama a la *ability Init* y cuando el usuario interactúa con el control se llama a *Façade*. Los métodos que cambian el estado del control también provocan un evento que es tratado por el *Façade VI*.

13.4.1 Ejemplo: Polímetro virtual

13.4.1.1 Explicación teórica

Para ilustrar el uso de los XControls se diseñará un control con apariencia de un sencillo polímetro. Antes de empezar a crear el código hay que aclarar algunos conceptos:

- Tipo de datos: la medida, que será un dato numérico de doble precisión.

- Estados del control: medida de corriente o tensión y la escala.
- Inicialización: escala=1 y medida=tensión.
- Métodos: tendrá un método en el que mostrará un mensaje con el valor de la medida.
- Propiedades: tendrá una propiedad de lectura/escritura que permitirá cambiar el tipo de medida.

13.4.1.2 Metodología

De lo anterior se deduce que la *ability State* será un control de tipo *cluster* de un booleano (tipo de medida) y un número (escala), *Data* (la medida) será un control numérico de doble precisión. En la figura 13-24 puede verse en la parte superior los controles correspondientes a *Data* y *State* y abajo el código de *Init* en el que, a partir de la plantilla creada por LabVIEW, se ha añadido un *cluster* con los valores iniciales del estado del XControl (*Current State*).

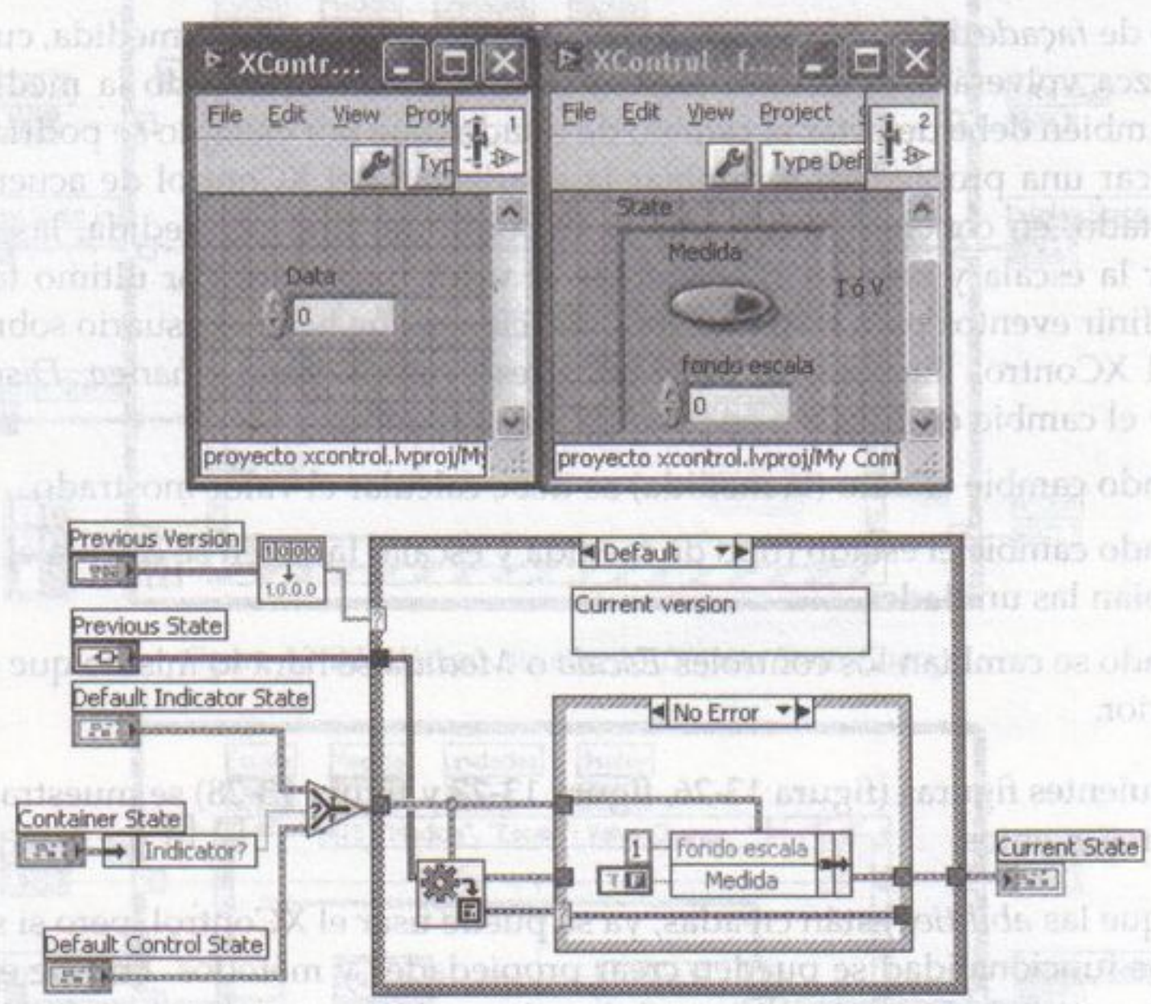


Figura 13-24. Controles *Data* y *State* y el código *Init*

La *ability façade* es la que más atención requiere. En ella, por una parte hay que crear el Panel Frontal que mostrará el XControl y por otra hay que manejar todos los eventos necesarios. El Panel Frontal será el de la figura 13-25.

Este Panel Frontal tiene un indicador numérico que muestra la medida multiplicada por la escala, un control para indicar la escala, un botón para indicar si se mide tensión o corriente y un *string* que muestra las unidades de la medida (Amperios ó Voltios), en

la figura anterior no se aprecia el *string* porque se ha hecho transparente, está situado a la derecha del indicador de la medida.

Figura 13-25. Panel Frontal de la *ability façade*

El código de *façade* debe reaccionar ante un cambio del valor de la medida, cuando esto se produzca volverá a calcular la lectura mostrada, multiplicando la medida por la escala. También debe detectar el cambio de estado (que por ejemplo se podría producir al modificar una propiedad) y cambiar la apariencia del XControl de acuerdo con el nuevo estado, en concreto debe cambiar el botón del tipo de medida, las unidades, modificar la escala y calcular nuevamente el valor mostrado. Por último también se deben definir eventos para tratar las modificaciones que haga el usuario sobre los controles del XControl. En total se utilizarán tres eventos: *Data Change, Display State Change* y el cambio en *Escala* o en *Medida*. Resumiendo:

- Cuando cambie el dato (la medida) se debe calcular el valor mostrado.
- Cuando cambie el estado (tipo de medida y escala) también se calcula el valor y se cambian las unidades.
- Cuando se cambian los controles *Escala* o *Medida* se hará lo mismo que en el caso anterior.

En las siguientes figuras (figura 13-26, figura 13-27 y figura 13-28) se muestra el código de estos tres eventos.

Una vez que las *abilities* están creadas, ya se puede usar el XControl, pero si se le quiere dar más funcionalidad se pueden crear propiedades y métodos. En este ejemplo se creará una propiedad para modificar el tipo de medida y un método para mostrar un mensaje.

Para crear una propiedad hay que elegir en primer lugar si será de lectura, de escritura o de lectura-escritura; en este último caso la propiedad consistirá en dos VIs. Para crear un método se deben enlazar los controles e indicadores del Panel Frontal en el terminal del icono y configurarlos desde la ventana del proyecto para que aparezcan en el nodo. En la parte izquierda de la figura 13-29 se ve el código de la propiedad de lectura creada para este ejemplo y en la derecha el código del método.

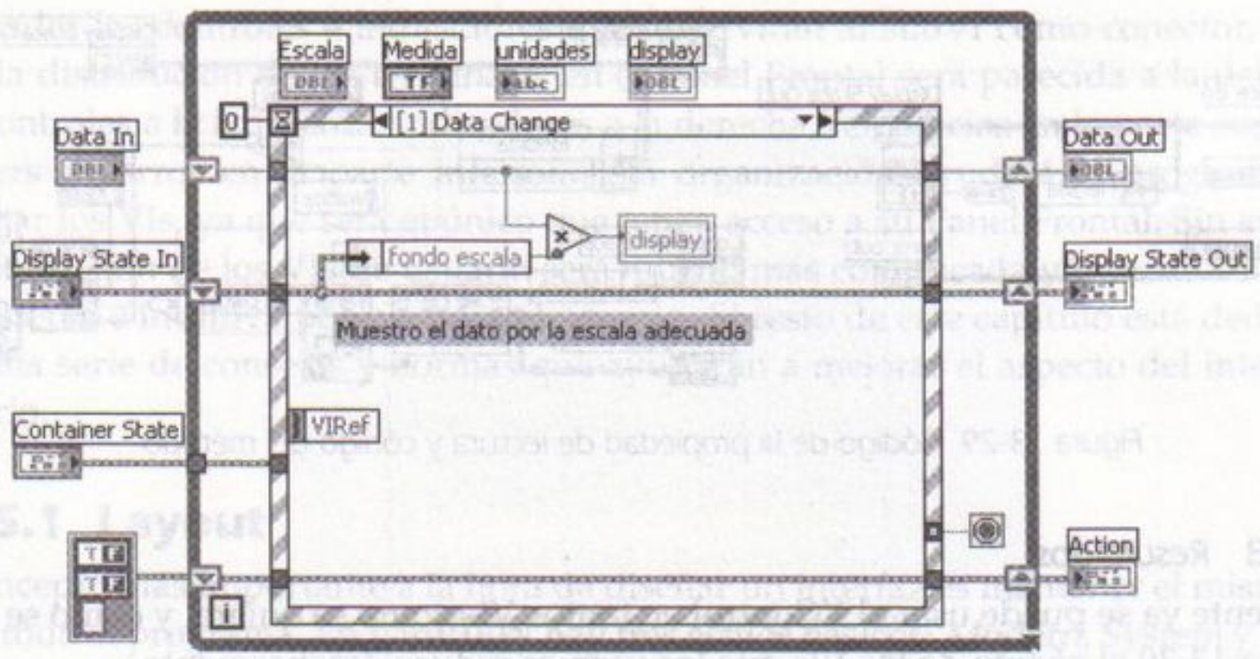


Figura 13-26. Código del evento *Data Change*

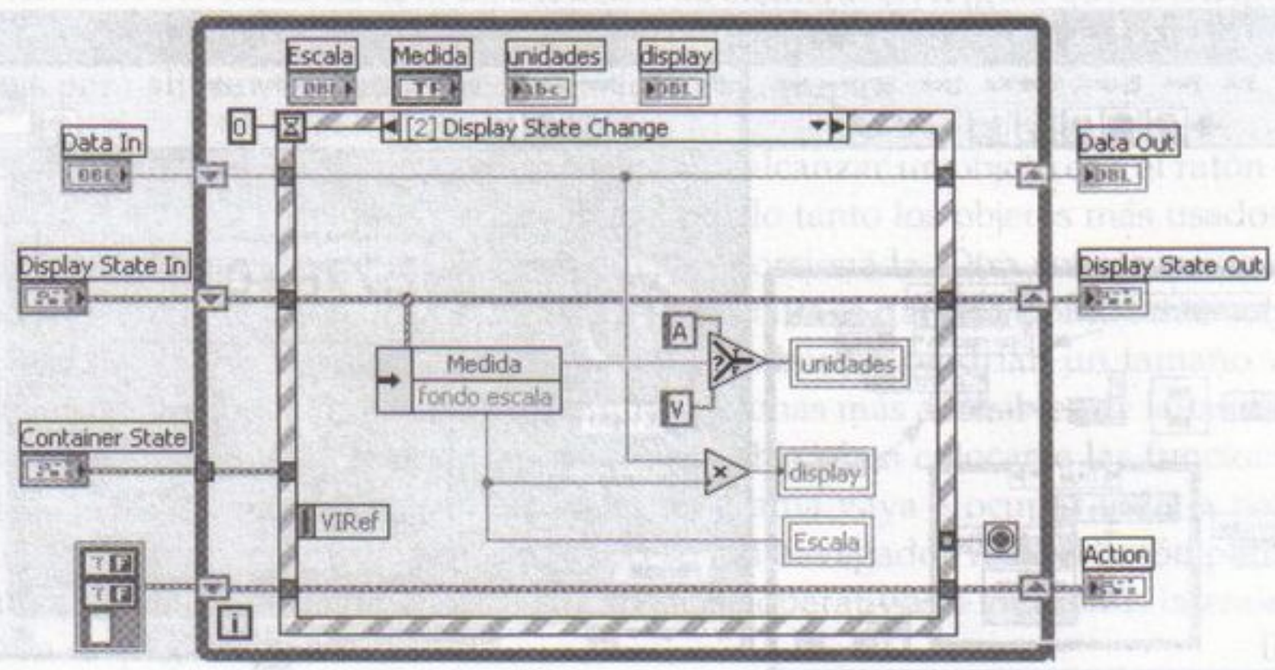


Figura 13-27. Código del evento *Display State Change*

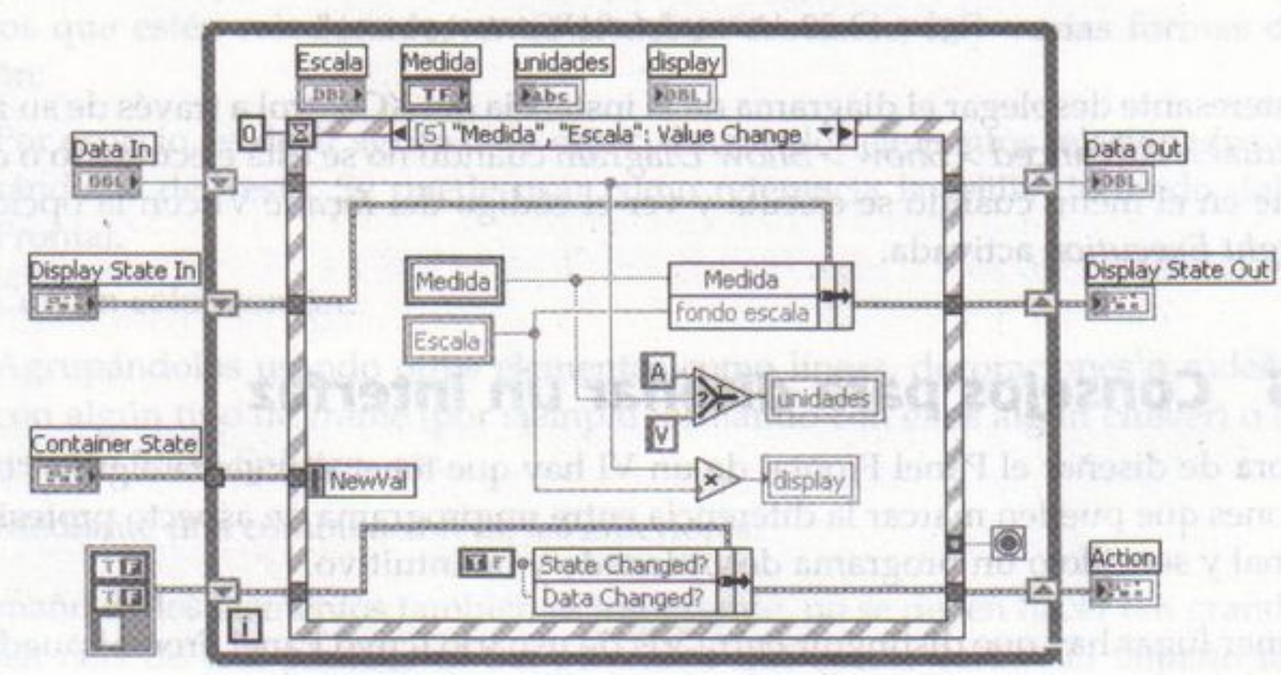


Figura 13-28. Código del evento *Medida* o *Escala*

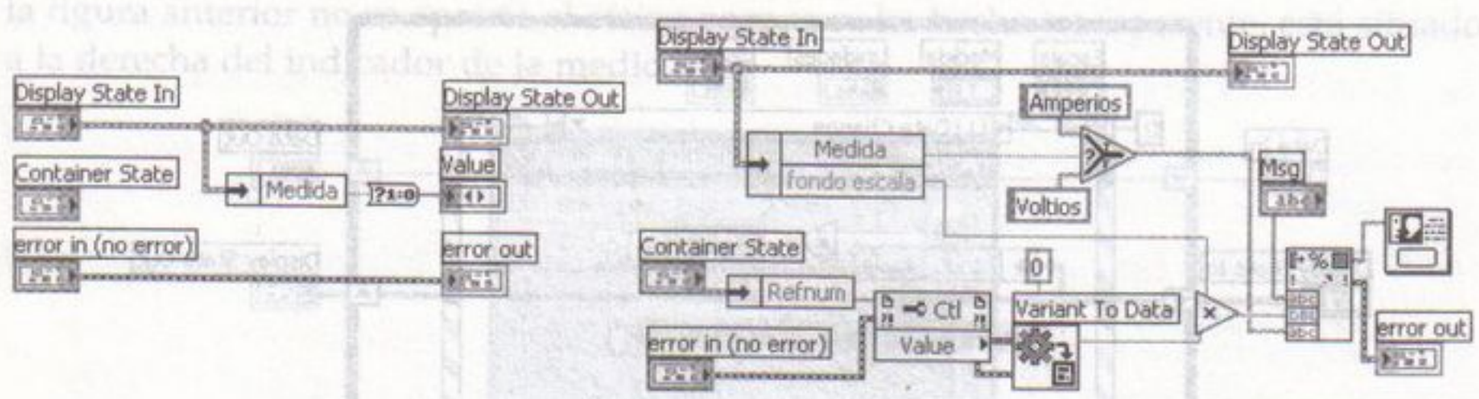


Figura 13-29. Código de la propiedad de lectura y código del método

13.4.1.3 Resultados

Finalmente ya se puede usar el XControl en tantos VIs como se quiera, y como se ve en la figura 13-30 el código de los VIs que los usan se reduce drásticamente.

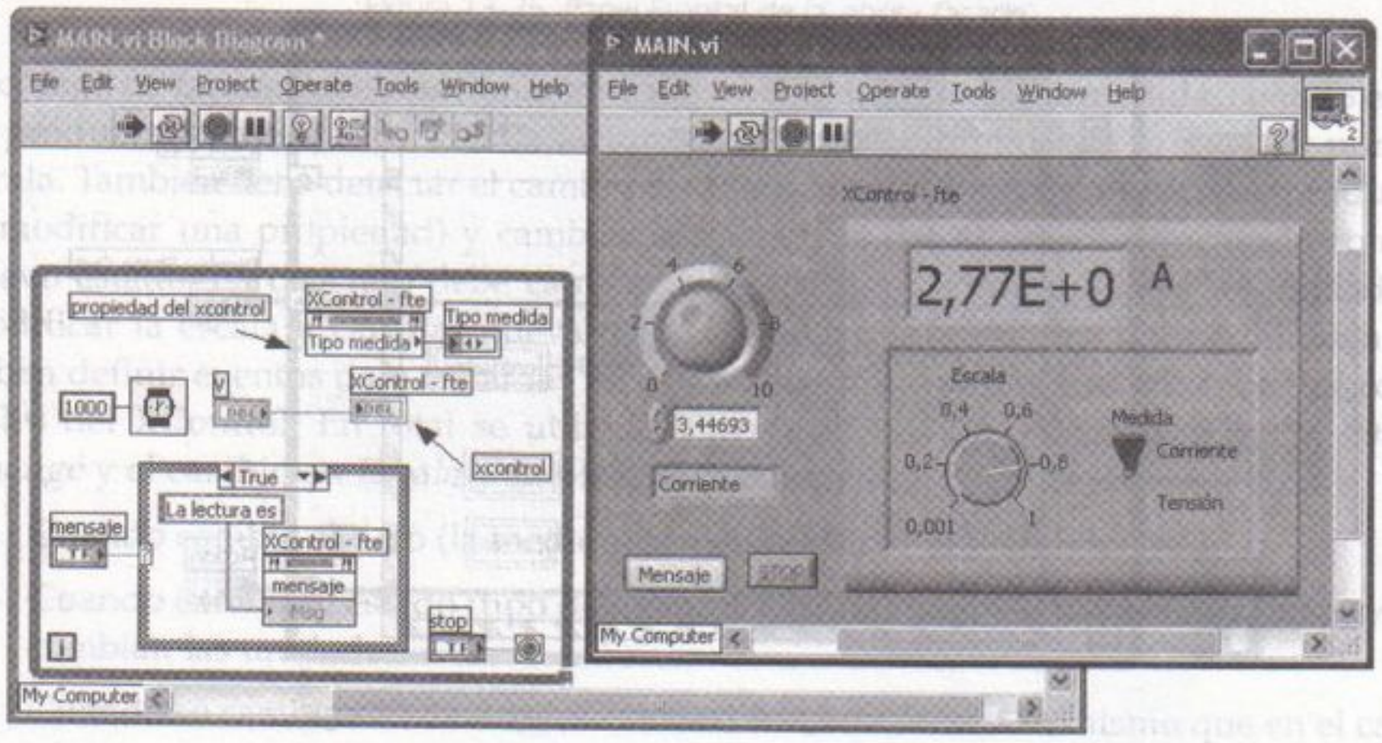


Figura 13-30. Utilización del XControl en un VI

Será interesante desplegar el diagrama de la instancia del XControl a través de su *menú contextual > Advanced > Show > Show Diagram* cuando no se está ejecutando o directamente en el menú cuando se ejecuta y ver el código del *façade* VI con la opción de *Highlight Execution* activada.

13.5 Consejos para diseñar un interfaz

A la hora de diseñar el Panel Frontal de un VI hay que tener en cuenta algunas consideraciones que pueden marcar la diferencia entre un programa de aspecto profesional, funcional y sencillo o un programa desordenado y no intuitivo.

En primer lugar hay que distinguir entre VIs de usuario (cuyo Panel Frontal puede ver el usuario) y los que no. Estos últimos serán simples VIs, por lo que su Panel Frontal

contendrá los controles e indicadores que le servirán al subVI como conector, en este caso la distribución de los terminales en el Panel Frontal será parecida a la del conector: controles a la izquierda, indicadores a la derecha, referencias en la parte superior y *clusters* de error en la parte inferior. Esta organización ayudará al programador a manejar los VIs, ya que será el único que tenga acceso a su Panel Frontal. Sin embargo la distribución de los VIs de usuario será mucho más complicada y buscará ser lo más consistente e intuitiva posible para el usuario. El resto de este capítulo está dedicado a dar una serie de consejos y normas que ayudarán a mejorar el aspecto del interfaz de usuario.

13.5.1 Layout

El concepto más importante a la hora de diseñar un interfaz es mantener el mismo estilo en todo el programa. En LabVIEW hay tres estilos básicos: *Modern, System* y *Classic*.

También debe ser consistente la distribución de elementos. Al igual que el Diagrama de Bloques, en el Panel Frontal también se deben usar las opciones de la barra de herramientas para alinear, distribuir, redimensionar y agrupar.

La ley de Fitt dice que el tiempo necesario para alcanzar un objeto con el ratón es función de la distancia y del tamaño del objeto, por lo tanto los objetos más usados deberían ser más grandes, pero no de forma desproporcionada. Otra consecuencia de esta ley es que cuando el ratón llega al borde de la pantalla se detiene, por lo tanto son zonas fáciles de llegar (los objetos situados justo en los bordes tendrían un tamaño virtualmente infinito), de este modo se deduce que las zonas más accesibles de la pantalla son las cuatro esquinas y los laterales, por lo tanto ahí deben colocarse las funciones más utilizadas (suponiendo que el interfaz del programa vaya a ocupar toda la pantalla). Algunos ejemplos serían el botón de retroceso del navegador web, el botón para cerrar la ventana de la mayoría de los GUIs de sistemas operativos o los menús laterales de la mayoría de páginas webs.

Uno de los conceptos más importantes para conseguir un interfaz sencillo es que los objetos que estén relacionados entre sí deben asociarse, hay varias formas de asociación:

- Por espacio, es decir agrupando espacialmente los elementos relacionados y separándolos del resto. Se puede usar como referencia la rejilla de fondo del Panel Frontal.
- Con un color común.
- Agrupándolos usando otros elementos como líneas, decoraciones o rodeándolos con algún tipo de *frame* (por ejemplo formando con ellos algún *cluster*) o incluso usando *Tabs, Splitters*, etc.
- Mediante una combinación de las anteriores.

El tamaño de los elementos también es importante, no se deben hacer tan grandes que ocupen más de una pantalla ni tan pequeños que cueste verlos. El tamaño también debe ser consistente en todos los elementos del programa. Para asignar el tamaño de

forma precisa pueden usarse las herramientas de *Resize Objects* de la barra de herramientas, figura 13-31.

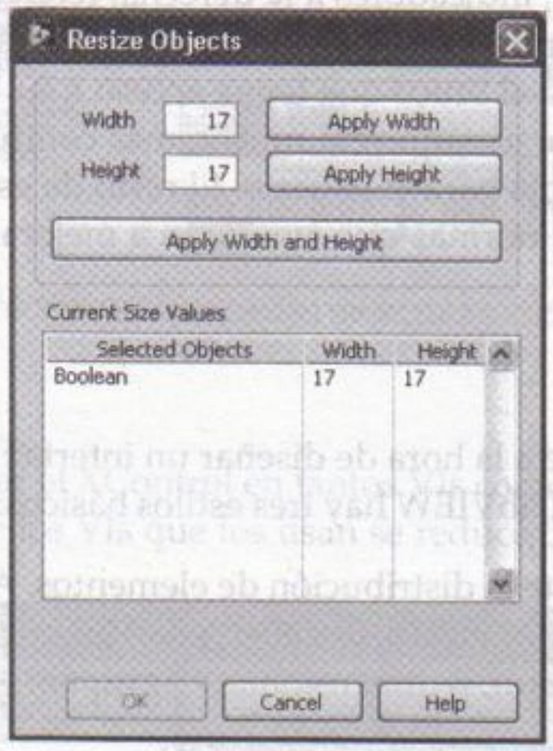


Figura 13-31. *Resize Objects* permite cambiar el tamaño de los objetos

Otro detalle relacionado con el tamaño de los objetos es la resolución de pantalla, en *File > VI Properties > Window Size* se puede hacer que al cambiar de resolución de pantalla se mantengan las proporciones o que los elementos se redimensionen al cambiar el tamaño de la ventana. También se puede conseguir que cada elemento individualmente se redimensione junto con la ventana a través de la opción *Scale Object With Pane* del menú contextual de cada control.

13.5.2 Tipografía

Las fuentes de los textos deben reducirse a *Application Font, System Font* y *Dialog Font*. Hay que tener en cuenta si el programa se ejecutará sobre sistemas operativos distintos porque los resultados pueden ser imprevistos.

Las etiquetas de los controles deben ser cortas y precisas. Para ajustar el tamaño de las etiquetas al texto que contienen se usará *Size To Text*. Si se quiere extender la información relacionada con cada elemento pueden usarse los elementos *Description and Tip* y hacer que junto al programa aparezca la ventana de ayuda contextual.

13.5.3 Color

El color puede servir para enfatizar o agrupar elementos. Su función principal es aumentar la cantidad de información para el usuario.

El color de fondo debe ser claro para que el texto y el resto de elementos que haya encima se aprecien bien. El color de estos elementos debe contrastar suficientemente con el de fondo, por ejemplo, una buena combinación es texto blanco sobre fondo azul.

En *Set Color* de la paleta de herramientas hay tres barras de colores (ver figura 13-32): la superior es una escala de grises, la segunda es para colores "neutrales" y la tercera para colores "fuertes". Para conseguir contrastes que llamen la atención se usarán colores fuertes.

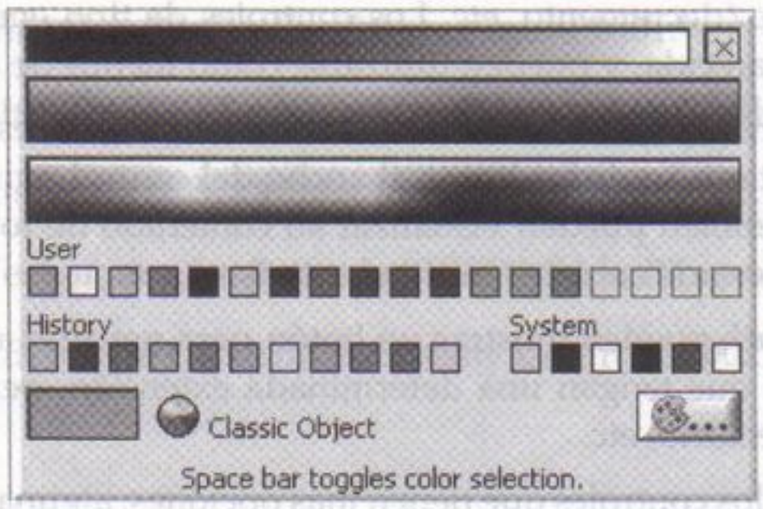


Figura 13-32. *Set Color* de la paleta de herramientas

Sobre la distribución de colores, se debe evitar el rojo y verde en la periferia, son más adecuados para el centro; sin embargo la retina es más sensible al blanco, negro, azul o amarillo en la periferia.

El número de colores que se usan no debe ser muy elevado, una cantidad recomendable estaría entre cuatro y siete.

13.5.4 Imágenes

Como ya se ha visto, los controles e indicadores pueden personalizarse. Entre otras cosas permiten añadir imágenes a los ítems de un control. Esta característica es muy útil para realizar metáforas en botones. Las metáforas son imágenes sobre controles que permiten al usuario asociar un concepto a la acción del control, por ejemplo un botón con una imagen de una flecha apuntando a la izquierda recordaría a la acción retroceder. En muchas aplicaciones se combina el uso de metáforas con texto.

13.5.5 Otros

Muchos controles tienen funcionalidades asociadas, conocerlas y usarlas puede simplificar la programación y ofrecer al programador mayor control sobre la aplicación final. Se puede acceder a ellas a través de su menú contextual o programadamente a través de propiedades y métodos.

- Los controles booleanos tienen asociada una acción mecánica *switch* o *latch*. En los dibujos que aparecen al seleccionar la acción, la letra M representa el ratón, V la salida y RD cuando el valor es leído. Los primeros cambian su valor cada vez que se presiona el ratón sobre ellos, se suelta o mientras permanece presionado; son apropiados para interruptores como el encendido o apagado de una máquina. En los segundos el valor cambia bajo las mismas condiciones que antes pero el nuevo valor sólo se mantiene hasta que es leído, volviendo entonces a su valor por defec-

to; son apropiados para asociarlos a una acción, por ejemplo la toma de una medida en un multímetro.

- Los controles numéricos pueden acotar el rango de entrada a unos determinados valores, mostrar los valores con distintas precisiones y formatos, tener un determinado incremento/decremento, etc. Los controles de tipo *enum* y *ring* tienen una funcionalidad parecida, y pueden utilizarse como menús de opciones.
- Los controles *string* pueden actualizar el valor mientras se escribe, limitarse a una línea, ajustar el tamaño de las líneas al ancho del control o mostrar barras de desplazamiento. También pueden visualizar su contenido de forma normal, mostrar el valor ASCII de cada carácter, mostrar asteriscos, etc.
- Los controles *path* pueden mostrar un botón para navegar, re*string*ir los ficheros mostrados a los que tengan una determinada extensión, permitir seleccionar un fichero o un directorio, etc.
- Las gráficas son los controles que tienen más opciones, algunas de ellas son el autoescalado de los ejes, el uso de cursores, zooms, añadir anotaciones, cambiar tipo de interpolación, cambiar el tipo de línea y su color, exportar una imagen a un formato gráfico, añadir otras escalas, etc.

Los controles deben tener como valor inicial (por defecto) el más típico. También puede considerarse crear un fichero de configuración para guardar el valor que el usuario fija en los controles.

El abuso de ventanas desplegables puede distraer al usuario de la aplicación principal. Las aplicaciones deberían tener una pantalla clave y mostrar ventanas de diálogo sólo cuando sean estrictamente necesarias; la información que pueden dar las ventanas de diálogo puede mostrarse en la pantalla principal a través de un espacio reservado para mensajes, cambiando el color de un determinado control, etc. Un ejemplo de esto es la opción de búsqueda en Internet Explorer y en Firefox, en el primero aparece una nueva ventana mientras que en el segundo aparece un pequeño recuadro en la parte inferior más cómodo y menos intrusivo.

También es importante reducir la latencia que percibe el usuario. La mejor forma de reducir retrasos es eliminándolos, para ello se utilizarán múltiples hilos de ejecución siempre que se pueda. Si se realizan tareas que impidan trabajar al usuario durante unos pocos segundos puede cambiarse la forma del cursor y cuando las tareas se alarguen más puede mostrarse una barra de progreso. Cuando la tarea acabe sería una buena idea avisar mediante un pitido o algún otro indicativo visual.

13.6 Ejercicio

1. Modificar el ejemplo sobre *Drag & Drop* para que en un control *Picture* se puedan dibujar con el ratón puntos, líneas y círculos de diferentes colores.

13.7 Bibliografía

Greg McKaskle, *LabVIEW Programming Techniques – The Good, the Bad, and the Ugly*, National Instruments, 1995.

Joel Spolsky, User *Interface Design For Programmers*, 2001.

National Instruments, *AN 161: Porting and Localizing LabVIEW VIs*, 2000.

National Instruments, *LabVIEW Development Guidelines*, 2003.

Peter Blume, *Five Techniques for Better LabVIEW Code*, Bloomy Controls, 2003.

Peter Blume, *LabVIEW Style Guidelines*, Bloomy Controls, 2002.

Capítulo 14

Optimización del código

14.1 Diseño de la aplicación

Escribir el código es sólo una más de las partes que componen el diseño de un proyecto. Cuando se aborda un proyecto de cierta envergadura que contendrá decenas o cientos de VIs, antes de empezar a programar, hay que aclarar qué es lo que se quiere y cómo se va a conseguir, además hay ciertas cualidades que serían deseables para el futuro, como hacer código reutilizable, flexible y de fácil mantenimiento.

El ciclo de vida de un proyecto son las fases por las que un proyecto debe pasar para llegar desde su inicialización hasta finalizarse en un tiempo limitado. El ciclo de vida de un producto estará constituido por las etapas que hay desde que surge la idea hasta que deja de utilizarse. En esta sección se estudiarán algunos modelos de ciclos de vida de proyectos software. Estos modelos sirven por una parte como guía al desarrollador o desarrolladores para organizar las tareas que conlleva el proyecto y por otra para ofrecer recursos para administrar y mantenerlo.

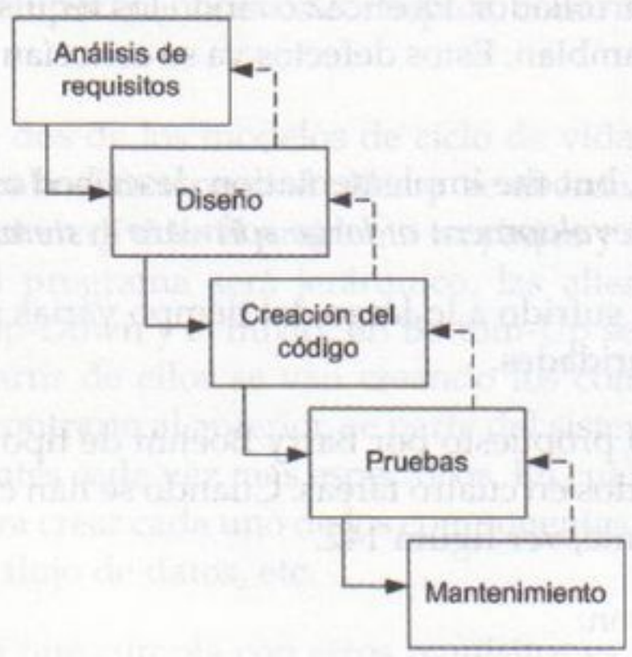


Figura 14-1. Fases de un proyecto según el modelo en cascada

El modelo en cascada (*waterfall*) fue referenciado por primera vez por Wiston Royce en los años 70. Es una de las metodologías históricamente más utilizadas. Consiste en

ordenar una tras otra las etapas del ciclo de vida del diseño, se pasa a una fase cuando se ha completado totalmente la anterior, si se detectan fallos en una fase posiblemente haya que volver a la anterior. Es muy importante que todas las fases se documenten adecuadamente.

En la figura 14-1 se aprecia un esquema de las fases propuestas por W. Royce, éstas son: análisis de los requisitos, diseño, creación del código, pruebas, implementación y mantenimiento. A continuación se detalla cada una de estas fases:

- En el análisis de los requisitos se trata de dividir el problema en los componentes hardware y software que serán necesarios, qué deben hacer y cómo se van a comunicar entre sí.
- Durante la fase de diseño hay que crear la arquitectura de la aplicación. En esta etapa también se diseñan los interfaces, que consisten en mecanismos de comunicación entre los distintos módulos del sistema y entre el sistema y el usuario.
- La creación del código es la programación propiamente dicha.
- En el testeo se corregirán fallos y se comprobará si el programa cumple las especificaciones iniciales.
- Las fases de implementación y mantenimiento no siempre están presentes en todos los proyectos, se trata de montar todo el sistema en su lugar de destino y mejorarlo o añadir nuevas funcionalidades con el tiempo.

El problema que tiene esta metodología es que la detección de un error o la introducción de una nueva característica en una de las últimas fases pueden suponer un gran esfuerzo. Por ejemplo, es muy habitual que se modifiquen las especificaciones que había en un principio, por lo que habría que volver a la primera fase. Otra desventaja es que se tarda mucho tiempo en tener listo el software, por lo que es más difícil la realimentación usuario-desarrollador. Es eficaz cuando los requisitos están muy definidos desde el principio y no cambian. Estos defectos ya se conocían cuando el método se dio a conocer:

I believe in this concept, but the implementation described above is risky and invites failure . *Managing the development of large software systems,* Winston W. Royce.

El modelo en cascada ha sufrido a lo largo del tiempo varias modificaciones cada una con sus propias particularidades.

Otro modelo es el espiral propuesto por Barry Boehm de tipo evolutivo que consta de una serie de ciclos divididos en cuatro tareas. Cuando se han cumplido los objetivos de un ciclo se pasa al siguiente, ver figura 142.

Las tareas de cada ciclo son:

- En la primera fase de cada iteración se determinan los objetivos para esa misma iteración, se proponen alternativas y se detectan las posibles restricciones.
- En el análisis de riesgos se decide si, para las alternativas elegidas, vale la pena seguir con el proyecto en esa iteración o abandonarlo definitivamente.

- El desarrollo en las primeras iteraciones consiste en un prototipado, mientras que en las últimas iteraciones se desarrollará el sistema completo con todos los detalles. Al final de esta fase se deben hacer pruebas para comprobar que el sistema cumple con los requisitos programados.
- A continuación se hace una evaluación del producto obtenido en esa fase y con los resultados obtenidos se planifica la siguiente iteración.

Figura 14-2. Fases de un proyecto según el modelo en espiral

El modelo en espiral tiene la ventaja de que el sistema se puede evaluar mucho antes que con el modelo en cascada: al final de cada iteración se tiene una versión del producto; por lo tanto es un modelo que responde mejor ante variaciones del diseño original. Además este modelo introduce nuevos conceptos, como la creación de prototipos y el análisis de riesgos.

Hasta ahora se han visto dos de los modelos de ciclo de vida de un proyecto, esta es una planificación principalmente temporal. Ahora se abordará el diseño del código, que básicamente consistirá en dividir el problema en partes y relacionarlas. En diseños de cierta complejidad el programa será jerárquico, las alternativas son: el método Bottom-Up, el método Top-Down y el mixto. En Bottom-Up se parte de los componentes más pequeños y a partir de ellos se van creando los componentes complejos. El método Top-Down es el contrario al anterior, se parte del sistema completo y desde ahí se van creando componentes cada vez más específicos. En cualquier caso también existen varias alternativas para crear cada uno de los componentes individuales del proyecto: máquinas de estados, flujo de datos, etc.

Un ejemplo de programa que cumpla con estos requisitos es el mostrado en la figura 14-3. En este programa se caracterizarán elementos electrónicos en frecuencia, para ello se usará un generador de señal y un osciloscopio. El programa consta de dos módulos: uno encargado del generador y otro del osciloscopio. También se puede ver su diseño jerárquico, desde el VI principal hasta los de más bajo nivel, que en este caso corres-

ponden a los que envían comandos a los equipos, estos VIs se podrán reutilizar en los dos módulos. A esta estructura se le podría añadir otros VIs de soporte que, por ejemplo, guardarán las medidas en un fichero, realizarán cálculos con las señales obtenidas o simularán la respuesta del osciloscopio con el fin de depurar y hacer pruebas de una forma más rápida y sencilla.

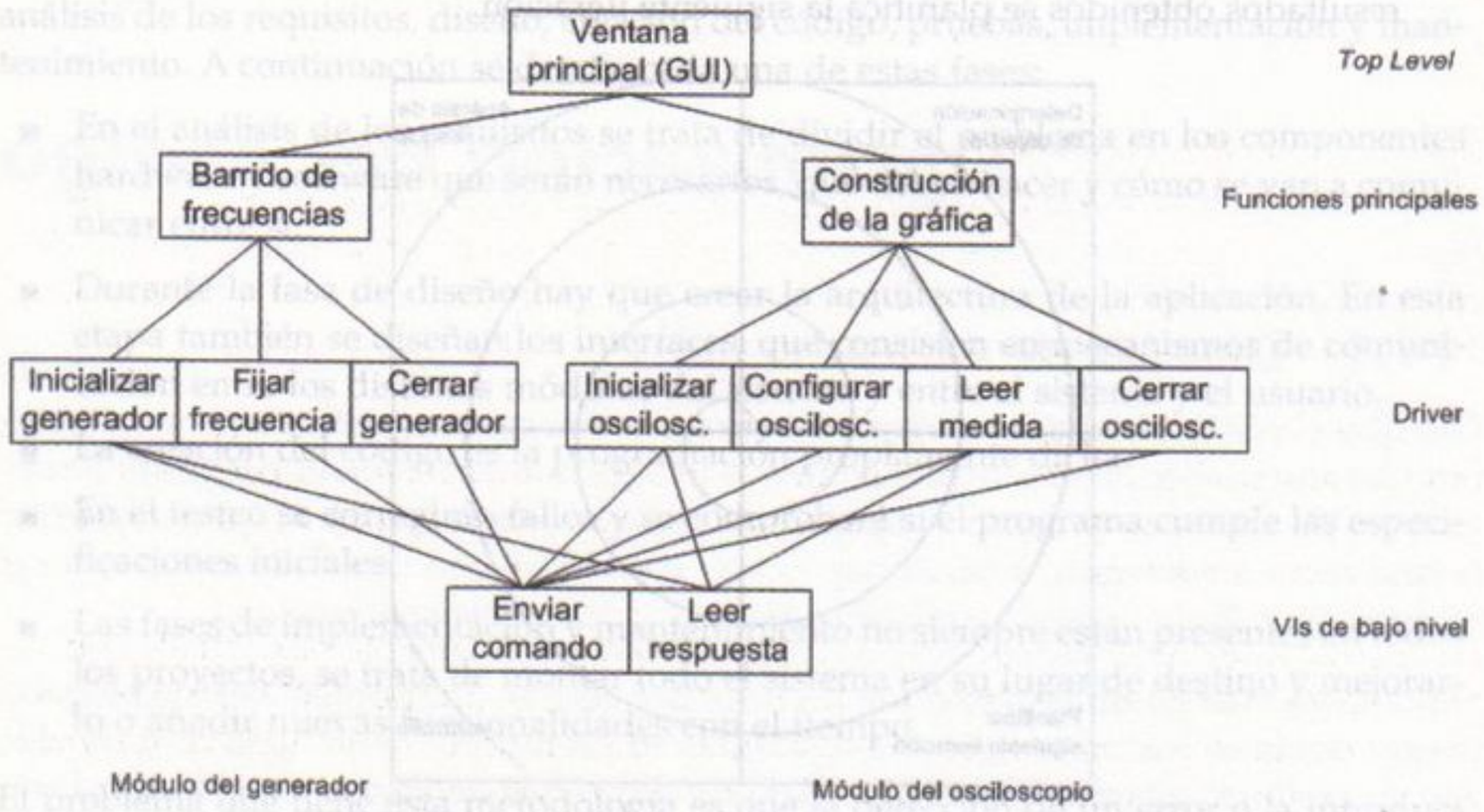


Figura 14-3. División en componentes de un proyecto tipo 'rombo'

Una vez que el código es correcto (sintaxis conforme) hay que verificar y validar el código. Verificar es comprobar que cumple con las especificaciones y validar es comprobar que el sistema cumple su función. Algunas técnicas para verificar son: tests, análisis, demostraciones, inspecciones y simulaciones.

En proyectos de cierta envergadura se puede crear un protocolo de pruebas para automatizar las pruebas, este protocolo puede consistir en una serie de programas que generen todas o un subconjunto de las entradas y comprueben que las salidas del programa para esas entradas son las esperadas. También se puede incluir en el programa partes de código cuya única finalidad es servir de ayuda a las pruebas, ese código puede eliminarse en el producto final. Este mecanismo puede implementarse fácilmente en LabVIEW usando las estructuras de deshabilitación condicional u otros VIs en *probes* personalizados.

Como se ha dicho antes, la documentación es algo común en todas las fases del proyecto: las especificaciones, el diseño, el código, los tests y el mantenimiento. Otra documentación que puede ser necesaria es la que se entrega al usuario y otro tipo de documentación comercial.

14.2 Estilo del código

A la hora de crear el código de un programa en LabVIEW es conveniente seguir unas reglas y sobre todo que sean consistentes durante todo el programa. Al hacerlo se conseguirá dotar al programa de legibilidad (el programa será más sencillo de entender y mantener), robustez (tendrá menos fallos y será más fácil depurar) y eficacia (necesitará menos recursos para ejecutarse correctamente).

14.2.1 Organización

Antes de la versión 8.0 de LabVIEW, la gestión de los distintos ficheros asociados a un proyecto sencillamente no existía. Entonces la única forma de organizar todos los VIs, controles, documentación, configuraciones y otros ficheros era tarea del programador, éste debía ordenarlos en directorios dentro de su ordenador.

Con LabVIEW 8.x se añade la ventana *Project Explorer* donde se pueden vincular y ordenar todos los ficheros que hagan falta, es una herramienta muy útil en medianos y grandes proyectos que viene a complementar (no a sustituir) la forma de organizar los ficheros por directorios.

La organización de los ficheros de un proyecto estará íntimamente relacionada con su arquitectura. Por ejemplo, para el programa de la figura 14-3 se podría tener un proyecto organizado como en la figura 14-4, donde cada directorio de la librería mostrada agrupa de forma jerárquica a los VIs que están interrelacionados.

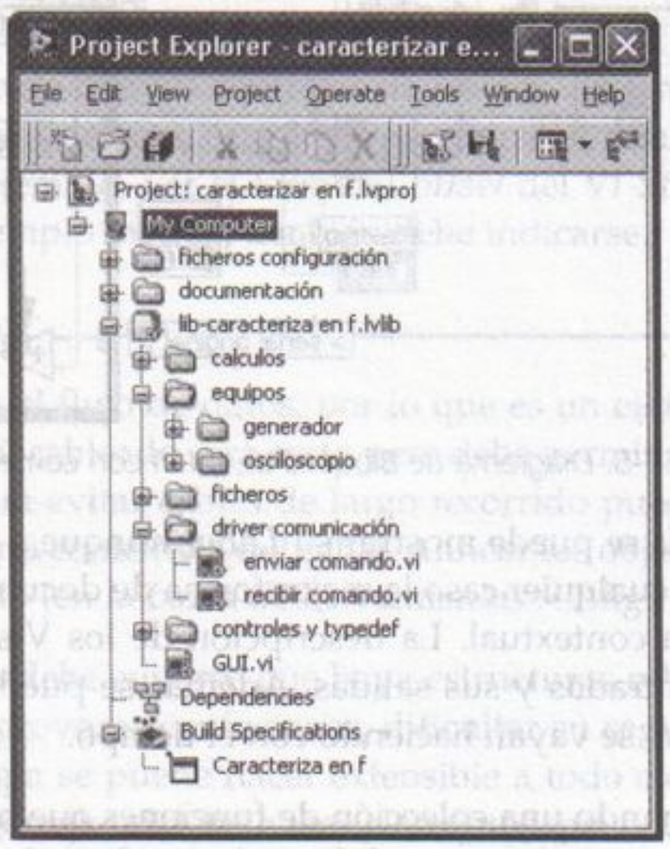


Figura 14-4. Explorador de proyectos

Dentro de un proyecto se preferirá agrupar los VIs y controles en librerías en vez de en directorios, porque las librerías aportan más posibilidades, como por ejemplo, la de hacer que los ficheros sean públicos o privados.

14.2.2 Comentar el código

Una de las cosas más importantes en todos los lenguajes de programación es comentar adecuadamente el código fuente. Se trata de embeber en el código información que explique cómo funciona el programa, el compilador ignorará estos comentarios.

En LabVIEW se dispone de *Edit Text* en la paleta de herramientas y *Free Label* en *Programming > Structures > Decorations*, con ellas se pueden añadir bloques de texto en el Diagrama de Bloques. Los elementos más sensibles a tener comentarios son las estructuras, algoritmos, funciones importantes, cables cuando su procedencia puede ser confusa (indicando mediante el símbolo > el sentido) y en general en cualquier código no evidente. Para identificar y agrupar objetos se pueden usar el resto de objetos del menú *Decorations*.

En la figura 14-5 se observa un programa que usa un bucle FOR para generar una señal aleatoria como indica el comentario que hay sobre la estructura. La señal es filtrada y se obtiene su espectro, ambos bloques bajo el comentario *Procesado de la señal* . Si el máximo de la señal filtrada es superior a un límite, la señal original (cuyo cable también está comentado antes de entrar en el CASE) se dividirá por ese máximo.

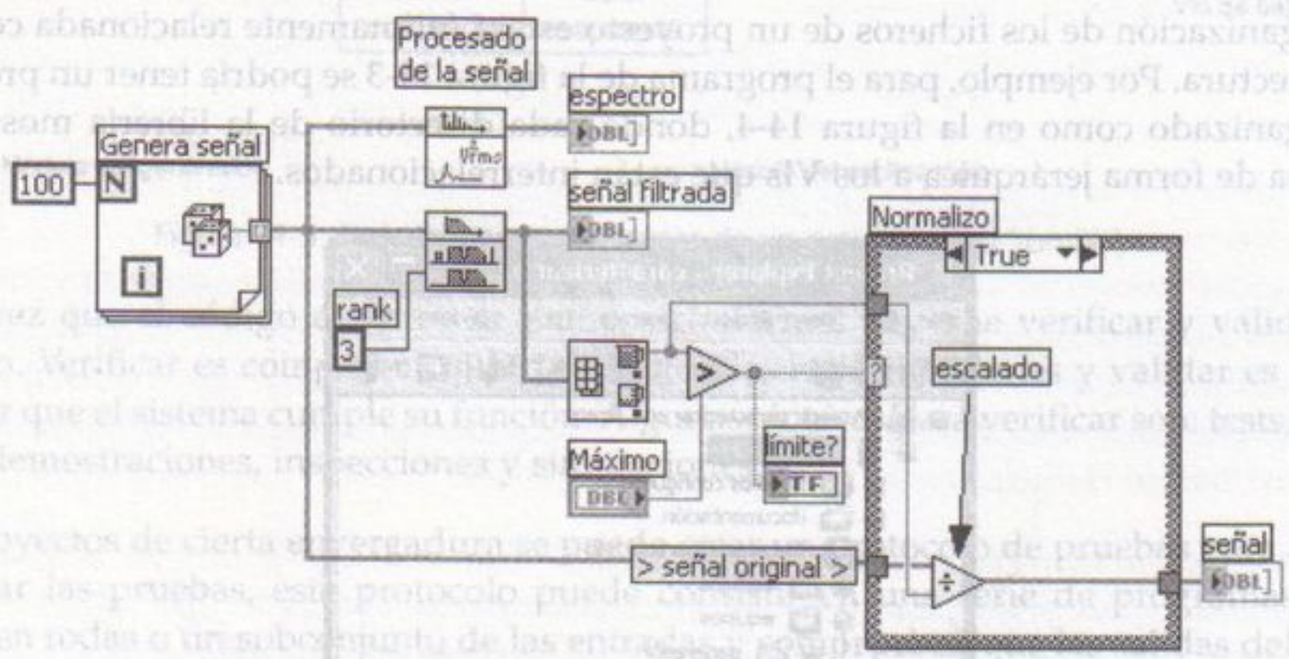


Figura 14-5. Diagrama de Bloques de un VI con comentarios

Sobre las funciones y VIs se puede mostrar su *Label*, aunque a veces su tamaño puede hacerlas incómodas. En cualquier caso la mejor forma de documentar un VI es a través de la ventana de ayuda contextual. La descripción de los VIs debe dar información sobre su función, sus entradas y sus salidas. Además se puede usar el historial para describir los cambios que se vayan haciendo con el tiempo.

Cuando se está programando una colección de funciones que más tarde se distribuirá, también se puede añadir un enlace a un fichero de ayuda en la ventana de ayuda contextual para que muestre una explicación más extensa del VI.

En muchos VIs, sobre todo los que sirven de ejemplo, también se inserta documentación en el Panel Frontal que indica cómo debe usarse, por ejemplo *introduce un valor*

en el control "Entrada" y presiona el botón "Run". También es recomendable usar la descripción de controles e indicadores para que se muestre en la ayuda contextual una explicación de su función, rangos, valor por defecto, etc.

Tampoco hay que olvidar realizar comentarios cuando se utilice código externo (librerías compartidas o CIN) o *scripts*.

14.2.3 VIs

También se debe ser consistente al diseñar los VIs de un proyecto. A la hora de dibujar los iconos suele reservarse un rectángulo en la parte superior del icono de, más o menos, un cuarto de la altura del VI para escribir el nombre de la librería a la que pertenece (para modificar la fuente de letra en los iconos hacer doble clic sobre la herramienta de texto del menú de la ventana *Icon Editor*). El resto del icono puede ser un gráfico que recuerda su función. Al dibujar el icono no debe olvidarse el *B&W*, porque aunque no se muestre en el programa si puede ser necesario si se imprime el código en una impresora en blanco y negro. Un ejemplo de VIs con este estilo serían los DAQmx.

En el conector del VI no se debe olvidar marcar los terminales como requeridos (obligatorios), recomendados (no obligatorios) u opcionales (usados con muy poca frecuencia). Cuando los VIs incluyen terminales de error, por convención, suelen situarse en las esquinas inferiores del conector. Si tienen referencias o *paths* se suelen poner en las esquinas superiores. Las entradas a la derecha y los indicadores a la izquierda. Cuando hay muchos controles o indicadores puede ser aconsejable unir los que tengan una funcionalidad parecida en *clusters*.

También es recomendable que los controles de los subVIs tengan valores por defecto. Para indicar el valor por defecto, éste suele escribirse al final del nombre del control entre paréntesis. Por ejemplo, ver el terminal *offset* del VI *String Subset*. Si el control tiene unidades, por ejemplo metros, también debe indicarse.

14.2.4 Cableado

El cableado determina el flujo de datos, por lo que es un elemento vital en la programación en LabVIEW. El cableado será corto pero debe permitir que el código tenga una claridad suficiente. Para evitar cables de largo recorrido puede usarse la herramienta *Clean Up Wire* del menú contextual del cable. Alinear los objetos puede ayudar a hacer el cableado más sencillo (en la *barra de herramientas > Align Objects*).

Al realizar el cableado debe evitarse que haya estructuras o VIs que oculten parte del cable porque pueden provocar confusiones, dificultar su seguimiento o la depuración del programa. Esta regla se puede hacer extensible a todo elemento del Diagrama de Bloques o del Panel Frontal: no debe haber elementos que oculten a otros.

El cableado debe guiar el flujo de ejecución de izquierda a derecha, por lo que en el Diagrama de Bloques los controles se situarán en la izquierda y los indicadores a la derecha. En subVIs que no sean parte del interfaz del usuario también es aconsejable seguir esta regla en el Panel Frontal.

El orden de ejecución lo determina el cableado, por lo que debe evitarse el uso de estructuras SEQUENCE porque rompen el paradigma del flujo de datos. Cuando sea necesario usar SEQUENCE es preferible FLAT SEQUENCE ya que muestra todos sus subdiagramas. La siguiente frase es una cita de Rande Johnson:

Be warned, sequence structures are a pet peeve of mine. I would not be the first person to complain should they ultimately disappear from LabVIEW . *LabVIEW Technical Resource*, Rande Johnson.

Como se ha comentado antes en este mismo capítulo, los cables que puedan dar lugar a confusiones también deben comentarse dando una descripción del dato que circula por ellos.

14.3 Control de código

El control de código o control de versiones (SCC por sus siglas en inglés) es un mecanismo para gestionar una información, desarrollo o proyecto a lo largo del tiempo y, potencialmente, por varias personas. Su principal aplicación es el control del código fuente de proyectos de software.

Hay varios programas que facilitan la tarea del control de versiones: Clear Case, CVS, Darcs, Microsoft Visual SourceSafe, MKS Source Integrity, PushOK, PVCS Version Manager, Seapine Surround SCM, Subversion, etc. Estos programas serían los servidores que almacenarían los ficheros que componen el proyecto, gestionan accesos para leer, modificar, mover o borrar los elementos, proporcionan un historial de cambios pudiendo recuperar el proyecto en una determinada fecha, etc.

Por otra parte también habría un software cliente que se conectaría al servidor para obtener información, guardar y recuperar ficheros. Casi todos los servidores también incluyen clientes, otros clientes son: lincvs, tkcvs, Tortoise, wincvs, etc.

Todos los programas de control de versiones usan una terminología parecida:

- **Repositorio**: lugar donde se almacenan todas las versiones del proyecto. Los usuarios tendrían una copia local del repositorio, cuando trabajan en el proyecto la actualizan recuperando la última versión del servidor, modifican los ficheros localmente y cuando acaban, guardan en el repositorio los ficheros modificados, construyendo así una nueva versión del proyecto.
- ***Check out***: acción que crea el directorio de trabajo local y copia los ficheros del repositorio a este directorio, el programador trabajará sobre los ficheros del directorio local.
- ***Check in*** (o *commit*): es el proceso de grabar los cambios hechos desde el directorio de trabajo local al repositario.
- ***Update*** (o *sync*): sincroniza el directorio local con la última versión del servidor. Es una operación complementaria a *check in*.

- *Merge*: cuando dos desarrolladores modifican el mismo fichero y lo guardan en el servidor podría darse el caso de una inconsistencia de los datos. El servidor tiene dos técnicas para evitarlo: una es permitir sólo a un desarrollador la escritura de ficheros y bloquear al resto (*lock-modify-unlock*); la otra es que el propio software pueda mezclar los cambios de los dos desarrolladores, lo que se conoce como *merge*. Esto es válido sólo con ficheros de texto, no funciona con ficheros binarios (como VIs).
- *Diff* (o *change*): muestra las modificaciones que se hacen de una versión respecto a otra.

14.3.1 Ejemplo en LabVIEW

En este ejemplo se supondrá que el servidor del SCC está instalado, configurado y tiene el proyecto de este ejemplo en su repositorio.

LabVIEW reconoce varios programas clientes SCC, si alguno de los instalados es soportado se detectará y automáticamente aparecerá en *Tools > Options > Source Control*. El siguiente paso será elegir el proyecto del repositorio, este paso dependerá del software elegido.

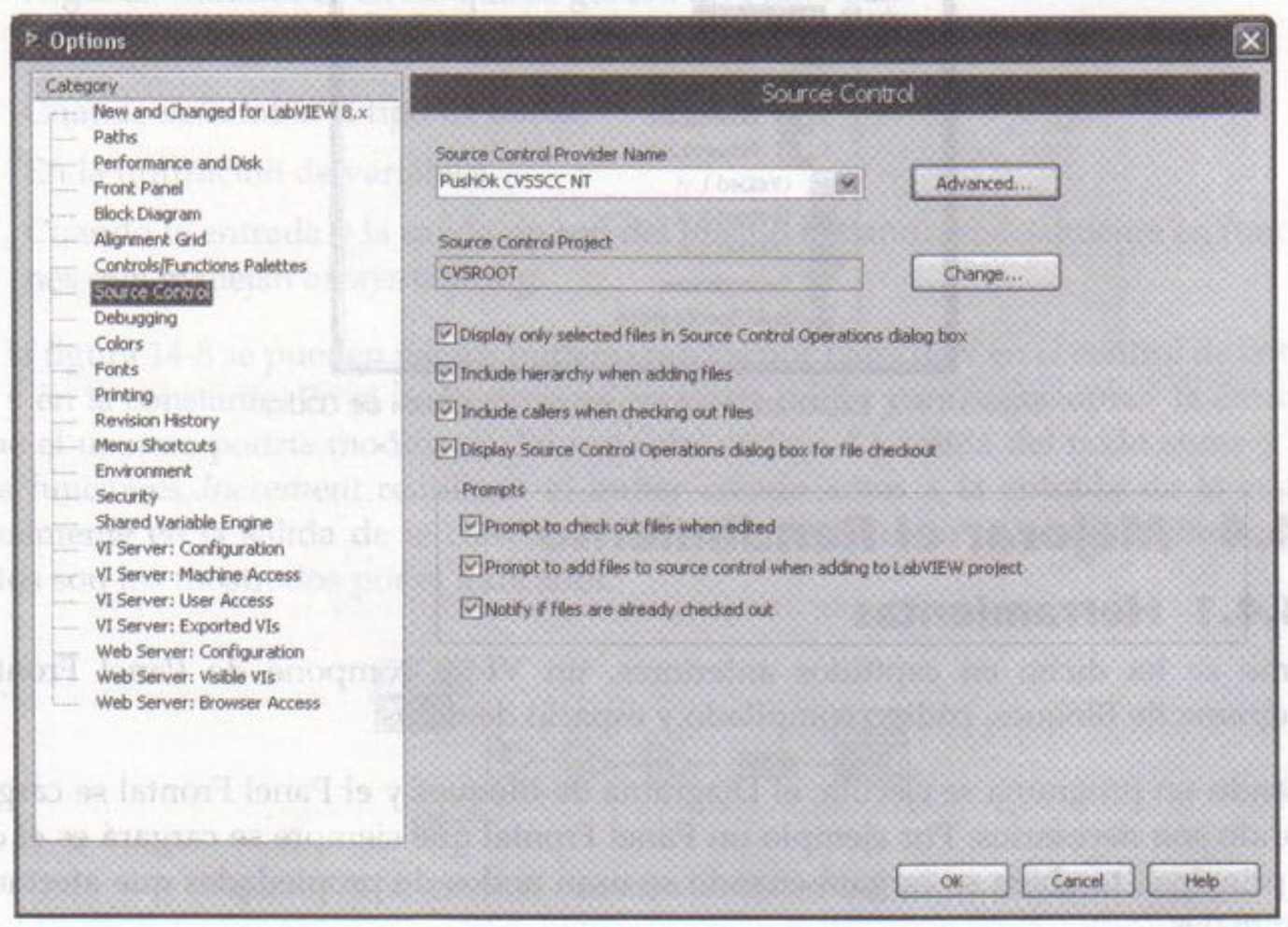


Figura 14-6. Opciones de Source Control

La gestión de ficheros puede hacerse directamente desde el software cliente o también se puede integrar el cliente en LabVIEW, en este caso se podría hacer desde el explorador de proyectos o de forma programada a través de los VIs del menú *Connectivity >*

Source Control. Ya sea con el explorador o de forma programada, LabVIEW trabajará con el programa y el proyecto que tiene configurados.

El explorador muestra una barra de botones con las opciones más útiles: *Check Out, Undo Check Out, Check In* y *Add to Source Control.* También se puede acceder a estas opciones a través del menú contextual de cada fichero o en *Tools > Source Control.*

Para añadir nuevos ficheros al repositorio habría que elegir *Add to Source Control,* después de hacerlo aparecerá una pequeña marca con forma de cuadrado en el icono del fichero.

Para trabajar con un fichero de forma local habría que chequearlo (*Check Out*), al hacerlo aparecerá una marca roja sobre la anterior y el fichero podrá ser modificado. Una vez que se termine de trabajar con este fichero se enviará el o los ficheros modificados al servidor (*Check In*), esto se puede ver en la figura 14-7.

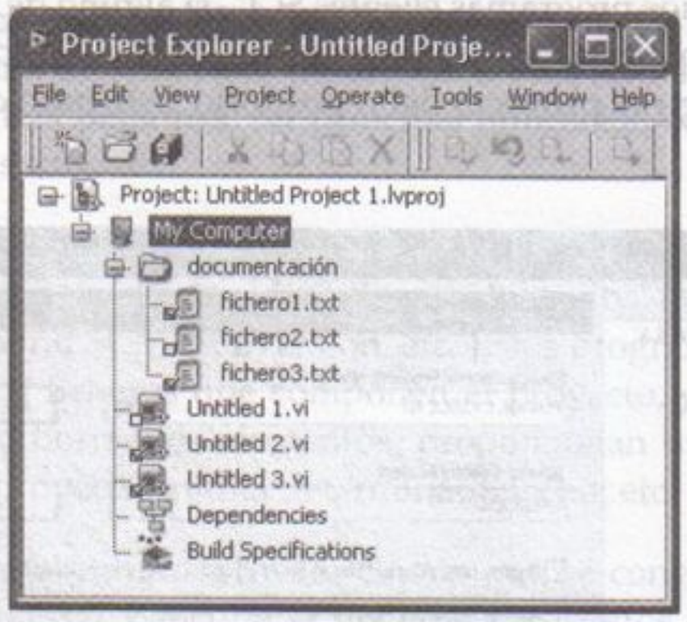


Figura 14-7. Explorador del proyecto con control de código

14.4 Mejorar el Rendimiento

14.4.1 Herramientas

Como se ha dicho en capítulos anteriores, un VI se compone de Panel Frontal, Diagrama de Bloques, código compilado y espacio de datos.

Cuando un programa se ejecuta, el Diagrama de Bloques y el Panel Frontal se cargan cuando son necesarios. Por ejemplo un Panel Frontal que siempre se cargará es el del VI principal, también se cargará cuando se usan nodos de propiedades que afectan a terminales.

Para ver el tamaño de cada una de estas partes hay que acudir a *File > VI Properties > Memory Usage.* Asimismo hay disponibles otras herramientas en el menú *Tools > Profile* para obtener más información sobre el tamaño y rendimiento de los programas de LabVIEW.

- *Performance and Memory*: muestra información sobre el tiempo que tardan en ejecutarse los VIs cargados en memoria y el tamaño de datos que usan. Para una información más detallada de cada campo puede situarse el ratón sobre cualquier elemento y abrir la ventana de ayuda contextual. Para usarlo hay que presionar el botón *Start*, ejecutar los VIs a medir y finalmente hacer clic en *Stop*.
- *Show Buffer Allocations*: muestra en el Diagrama de Bloques, mediante un punto negro en la parte derecha de los terminales o funciones, los buffers creados por LabVIEW.
- *VI Metrics*: proporciona estadísticas sobre el contenido de los VIs, como el número de subVIs, número de estructuras, controles, indicadores, variables, profundidad de anidamiento, etc.

14.4.2 Buffers

Los buffers son puntos del programa donde LabVIEW reserva un espacio de memoria para almacenar datos. LabVIEW intentará reaprovechar lo máximo posible los buffers ya creados. Como regla general una programación que respete el flujo de datos podrá aprovechar mejor el espacio de memoria.

- Algunas situaciones en las que se genera un buffer son:
- En entradas de datos (controles y constantes).
- Cuando se cambia el tipo de datos.
- En la utilización de variables.
- Cuando la entrada y la salida no son del mismo tamaño, especialmente en funciones que manejan *arrays* o *string*.

En la figura 14-8 se pueden ver los buffers creados por LabVIEW en el control de entrada y en la constante. En el nodo suma se crea otro buffer para desacoplar la entrada (que el usuario podría modificar) del resto del código. La salida del nodo suma y las dos funciones *Increment* reutilizan el buffer creado antes a la entrada de la suma. Finalmente en la salida de la conversión del tipo de datos se crea otro buffer cuyos datos son los mostrados por el indicador.

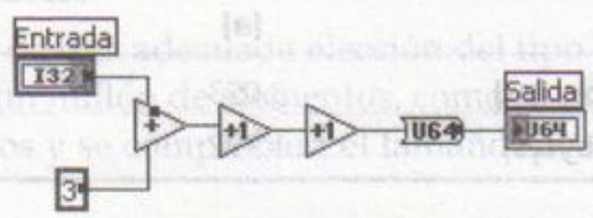


Figura 14-8. VI donde se muestran los buffers que crea LabVIEW

Las conversiones automáticas de datos numéricos (marcado con un *coercion dot*) también pueden afectar negativamente la reutilización de buffers, por lo que son algo a evitar. En general se utilizarán tipos de datos consistentes para facilitar la reutilización de memoria. La creación de nuevos buffers será especialmente delicada cuando se trate con *arrays* ya que se puede generar una gran cantidad de datos.

LabVIEW normalmente no libera el espacio de memoria que ocupan los buffers una vez usados a menos que el VI sea cerrado y descargado de la memoria. Para forzar la liberación de memoria de un subVI se puede usar la función *Request Deallocation* del menú *Programming > Application Control.*

14.4.3 Técnicas para mejorar el rendimiento

Para mejorar el rendimiento de un VI hay que seguir dos reglas principales: respetar el flujo de datos y maximizar la concurrencia. Además se debe tener siempre en mente dos aspectos importantes: el código y los datos.

En esta sección se estudiarán algunos casos más o menos habituales de programas poco eficientes, formas de optimizarlos y se darán algunas pautas generales para conseguir programas eficientes. Todos los programas se han probado en el mismo ordenador bajo las mismas condiciones y empleando LabVIEW 8.20.

14.4.3.1 Cálculos inútiles

El primer punto a revisar para optimizar el código es evitar cálculos inútiles. En la figura 14-9 se pueden ver dos programas, en el de la izquierda se realiza una suma de dos escalares antes de entrar en un bucle y en el de la derecha dentro del bucle.

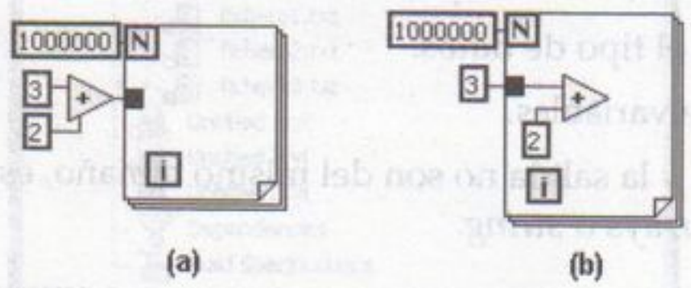


Figura 14-9. Ejemplo de cálculos no necesarios que empeoran el rendimiento

Es evidente que el código (a) tardará menos en ejecutarse. En cuando al tamaño de datos, la diferencia no es significativa.

Tabla 1 - Resultado de la ejecución del código de la figura 149

	(a)	(b)
Tiempo de ejecución (µs)	3.053	5.570
Tamaño de datos (bytes)	1.392	1.384

14.4.3.2 No reinventar la rueda

Al hilo del punto anterior, suele ser frecuente en programadores poco experimentados, el crear código para resolver problemas cuando no es necesario porque LabVIEW ya dispone de otros mecanismos alternativos más adecuados para resolver esos problemas.

Un ejemplo podría ser realizar mediante código una función para limitar la entrada de datos de un control numérico cuando esto mismo se puede realizar de una forma sencilla mediante las propiedades del control. Otro ejemplo es el mostrado en la figura 14-10 donde se quiere mostrar un mensaje durante las iteraciones 2, 3, 4 y 5 del bucle. En (a) se comprueba si el índice del bucle es mayor o igual a 2 y menor o igual a 5, si es así muestra el mensaje. En (b) se han sustituido las funciones booleanas y de comparación por *In Range And Coerce* (en el menú desplegable del VI se ha hecho que incluya los dos límites) para comprobar si el índice está dentro del rango. En (c) se cablea el índice al selector del CASE y en su menú se escribe el rango 2..5 .

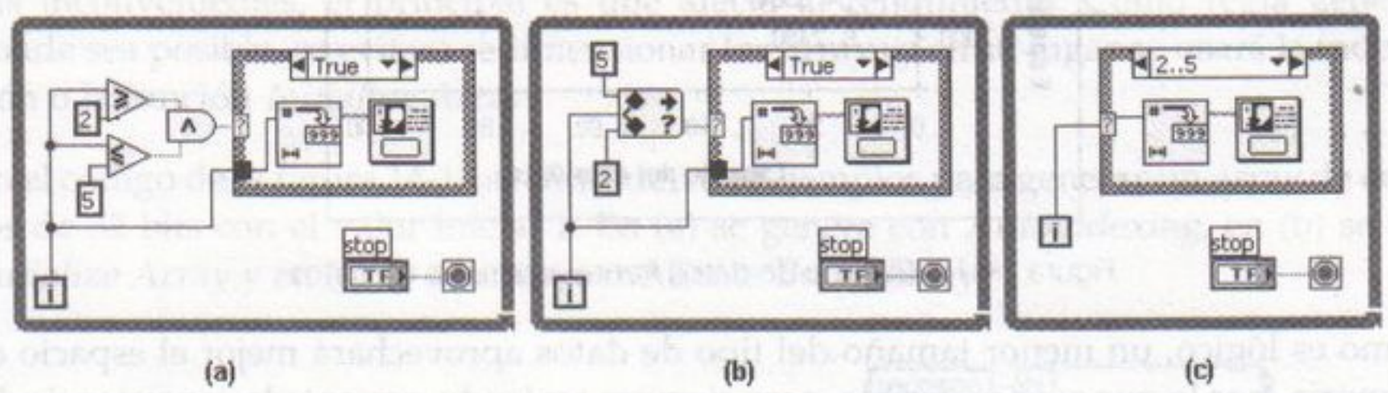


Figura 14-10. Ejemplo que muestra mensajes en diferentes iteraciones

Para comprobar el rendimiento de estas tres alternativas se ha modificado ligeramente el código: *One Button Dialog* se ha eliminado, el bucle WHILE se ha sustituido por un FOR que se repite 10 veces y como el programa se ejecutaba demasiado rápido para sacar conclusiones en la máquina en que se ha probado, se ha introducido todo el código en un segundo FOR que se repite 100 veces. El resultado muestra como la opción (c) es la más óptima.

Tabla 2 - Resultado de la ejecución de los códigos de la figura 14-10

Tiempo de ejecución (µs)	24	23	13
Tamaño de datos (bytes)	1.660	1.620	1.532

14.4.3.3 Tamaño de los datos

Para comprobar el efecto de una adecuada elección del tipo de datos se creará un *array* de dos dimensiones con un millón de elementos, como se observa en la figura 14-11. Se modificará el tipo de datos y se comprobará el tamaño que ocupan los datos en el VI.

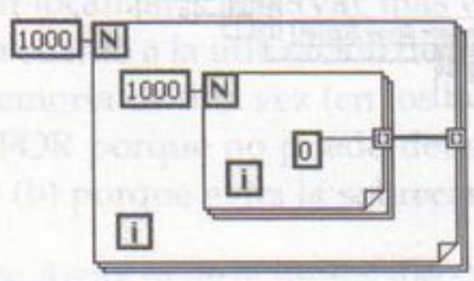


Figura 14-11. Ejemplo para ver el tamaño de los datos como varia en función del tipo de datos

Se han usado datos de tamaño 8 bits, 16, 32, 64 y 80 (números flotantes extendidos en Windows). El tamaño en kB se puede ver en la gráfica de la figura 14-12.

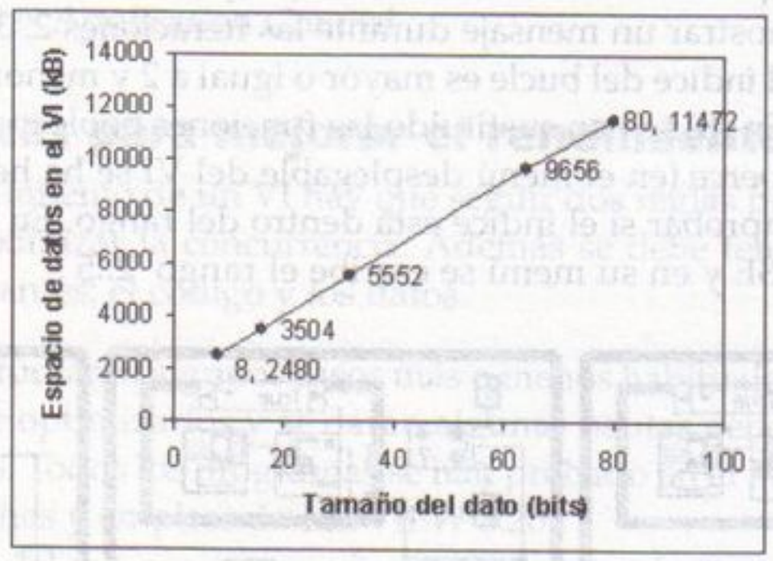


Figura 14-12. Espacio de datos frente a tamaño de datos

Como es lógico, un menor tamaño del tipo de datos aprovechará mejor el espacio de memoria, por lo que será preferible, pero siempre teniendo presente la consistencia del tipo de datos para así evitar transformaciones de tipos (ver *coercion dots*), esto es especialmente crítico si hay involucrados *arrays* de gran tamaño.

14.4.3.4 Datos simples o complejos

Muchas funciones de LabVIEW son polimórficas y permiten trabajar con varios tipos de datos, como por ejemplo con números escalares, *arrays* y *clusters* de valores numéricos. Esta característica puede simplificar el código del programa, pero si se trabaja con estructuras de datos demasiado complejas se corre el riesgo que para modificar un dato en concreto dentro de toda la estructura se creen copias innecesarias del resto de datos.

Por ejemplo, en la figura 14-13 (a) se tienen los datos en un *array* de *clusters*, cada uno compuesto por dos *arrays* numéricos y un *string*. El código modifica el segundo elemento (con valor 3 en la figura) del segundo *array* mostrado en la figura y se asigna el valor 2. Este código necesita 2.028 bytes. El código de la figura 14-13 (b) realiza la misma operación, pero en lugar de esa estructura de datos se han usado dos *arrays*

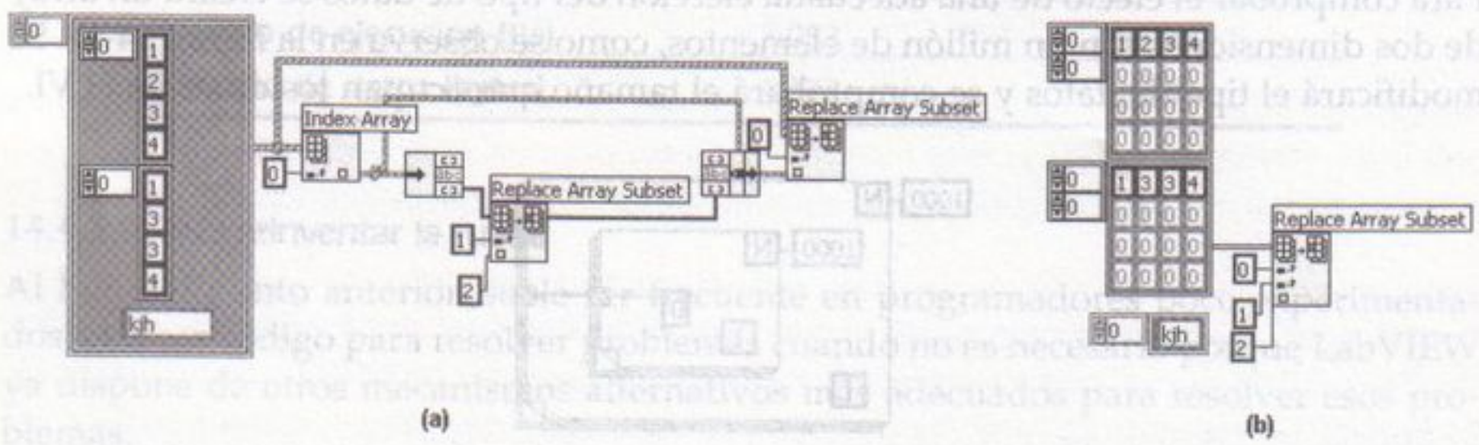


Figura 14-13. (a) Datos complejos. (b) Datos simples

numéricos de dos dimensiones y un *array* de *strings* de una dimensión. Necesita 1.615 bytes de datos. La diferencia de tamaño se debe a los buffers creados a la salida del nodo *Index Array* y de la entrada de *Replace Array Subset* del código (a).

14.4.3.5 Redimensionado de arrays

Una diferencia de LabVIEW respecto al lenguaje C es el manejo de los *arrays*. En C hay que indicar el tamaño en la declaración para que se reserve el espacio mientras que en LabVIEW se puede cambiar dinámicamente. Esto es útil porque evita tener que recurrir a las funciones `malloc` y `free` del C, evitando overflow, etc. Pero también tiene sus inconvenientes, el principal es que afecta al rendimiento. Como regla general, donde sea posible, se evitará redimensionar los *arrays* y en su lugar se usará la indexación o la función *Initialize Array*.

En el código de la figura 14-14 se muestran tres ejemplos para generar un *array* de enteros de 32 bits con el valor inicial 1. En (a) se genera con *Autoindexing*, en (b) se usa *Initialize Array* y en (c) se construye mediante *Build Array*.

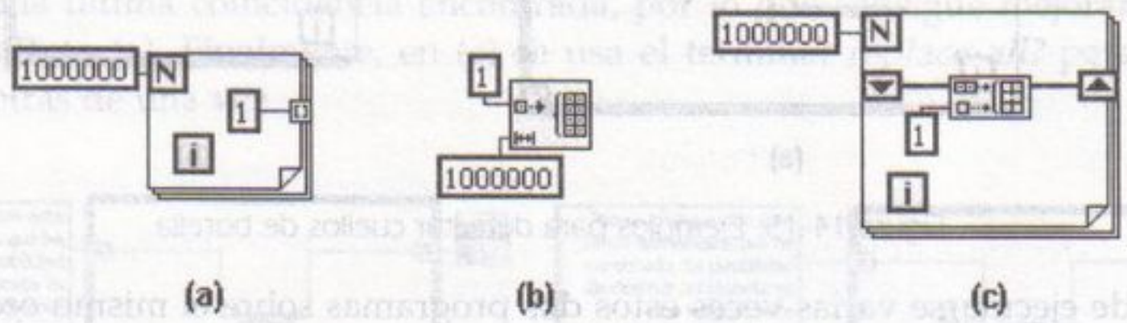


Figura 14-14. Generación de un arreglo de escalares

Los resultados de la ejecución son:

Tabla 3 - Resultados de la ejecución de la figura 14-14

	(a)	(b)	(c)
Tiempo de ejecución (µs)	3.315	1	17.031.250
Tamaño de datos (bytes)	1.364	1.320	8.001.440 (máx.)

Como era de esperar, el buffer creado por los *Shift Register* en el caso (c) genera una gran penalización en cuanto a espacio y tiempo porque ocupa una gran cantidad de memoria al tener que almacenar el *array* en cada iteración, además hay que tener en cuenta el tiempo invertido en localizar y reservar más espacio en cada iteración. Los casos (a) y (b) están parejos en cuanto a la utilización de la memoria porque los dos pueden localizar el espacio de memoria de una vez (en los bucles WHILE *autoindexing* no es tan eficiente como en los FOR porque no puede determinar a priori el tamaño del *array*); en velocidad es mejor (b) porque evita la sobrecarga que conllevan los bucles.

Se podría pensar que *Initialize Array* tiene la limitación de que todos los datos que contiene el *array* deben tener el mismo valor, es cierto, pero si después se utiliza un bucle

con una función para reemplazar elementos se tiene la misma potencia que con las otras alternativas y sigue siendo muy eficiente.

14.4.3.6 Cuellos de botella

En los tres ejemplos anteriores no se han almacenado los resultados ni mostrado en el Panel Frontal, ésta es una tarea que consume recursos, especialmente si se trata de grandes datos (como *arrays*) por lo tanto también es un candidato a optimizarse.

En el siguiente ejemplo, figura 14-15, se muestran dos códigos cuya función es generar y mostrar mil números aleatorios en una gráfica. En (a) el indicador de la gráfica se actualiza cada cinco números, mientras que en (b) se actualiza cada vez que se genera un nuevo número.

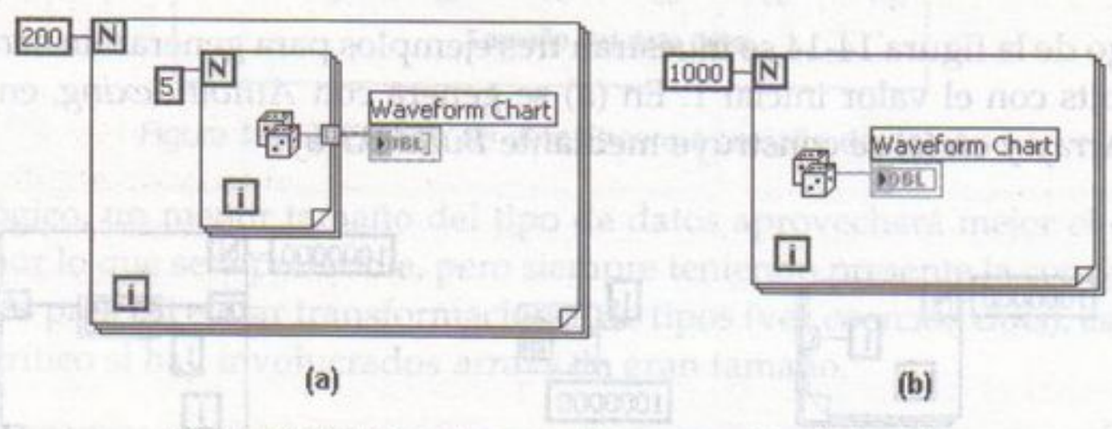


Figura 14-15. Ejemplos para detectar cuellos de botella

Después de ejecutarse varias veces estos dos programas sobre el mismo ordenador, el promedio de resultados es:

Tabla 4 - Resultado de la ejecución de los VI de la figura 14-15

	(a)	(b)
Tiempo de ejecución (µs)	384	555
Tamaño de datos (bytes)	19.553	19.133

En cuanto a memoria los dos programas están igualados, la pequeña diferencia que se aprecia a favor de (b) es debida a la salida indexada del segundo bucle FOR de (a). Sin embargo la mayor diferencia es respecto al tiempo de ejecución, donde (a) tarda mucho menos que (b) por realizar ochocientas actualizaciones del gráfico menos.

Como regla general se evitará mostrar en el Panel Frontal indicadores de tipo *array* y en el caso de gráficas se deben actualizar sólo cuando sea preciso.

Si hay varios indicadores que necesiten actualizarse en programas multihilos también puede mejorarse el rendimiento esperando a tener los todos los datos para actualizarlos a la vez en lugar de actualizarlos en cuanto lleguen los datos. Esto se consigue desplegando el menú contextual de los indicadores y eligiendo *Advanced > Syncronous Display*. Por ejemplo, dos hilos idénticos al código de figura 14-15 (b) en el mismo VI

tardarían diez veces menos en ejecutarse sincronizando las actualizaciones que sin hacerlo.

Otra opción para optimizar las gráficas es evitar el autoescalado. Cuando haya que realizar varias modificaciones sobre un indicador de una gráfica puede ser interesante bloquear el aspecto antes de empezar, usar los nodos de propiedad para efectuar los cambios y cuando ya estén todos los cambios hechos desbloquear el aspecto. El bloqueo y desbloqueo puede realizarse con una propiedad de *Generic > GObject > Panel* llamada *Defer Panel Updates*.

Además del dibujo de gráficas, también se debe prestar atención a funciones que tarden mucho tiempo en ejecutarse, por ejemplo, las que realizan tareas de I/O, especialmente si se sitúan dentro de bucles. Otras funciones contienen terminales que permiten mejorar su rendimiento, por ejemplo el terminal *offset* de muchas funciones para manejar *strings*, como muestra el ejemplo de la figura 14-16.

El código reemplaza todas las letras a por e en un *string*. En (a) se hace letra por letra en cada iteración del bucle. En (b) se usa el terminal *offset* para empezar a buscar a partir de la última coincidencia encontrada, por lo que consigue mejorar el rendimiento respecto (a). Finalmente, en (c) se usa el terminal *replace all?* para sustituir todas las letras de una vez.

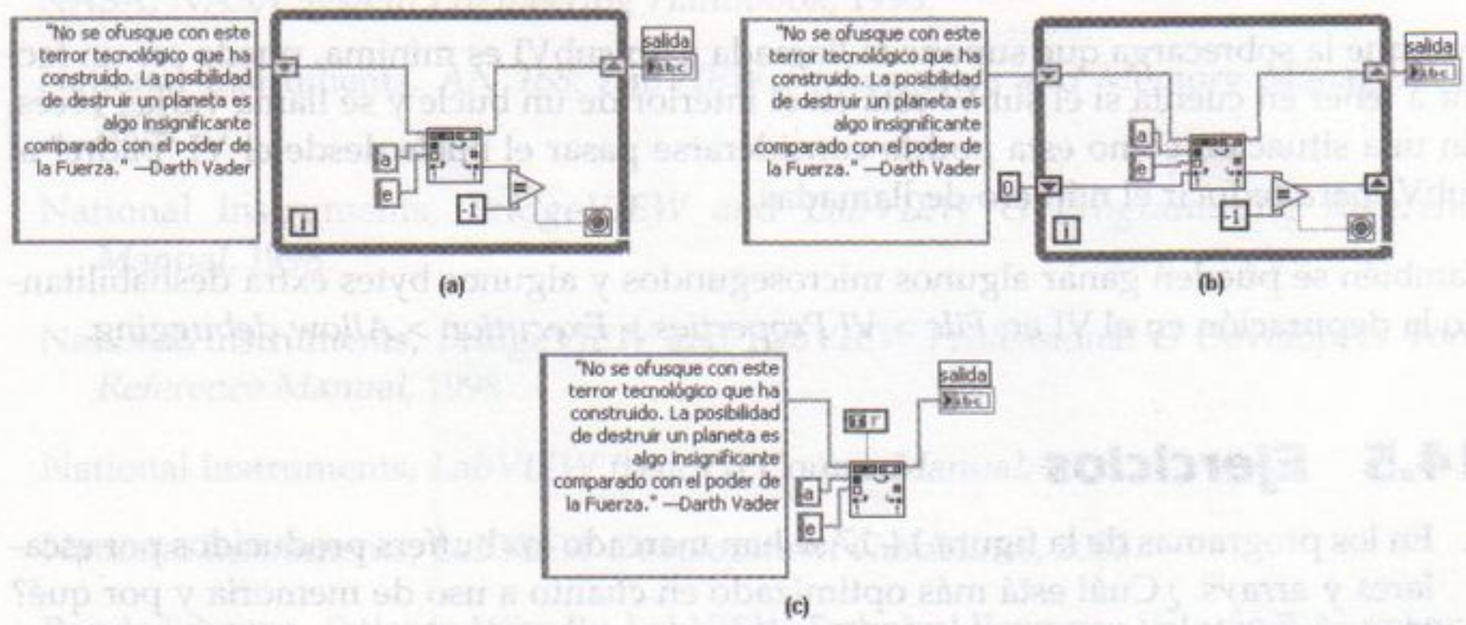


Figura 14-16. Utilización del terminal offset para mejorar el rendimiento

Tabla 5 - Resultado de la ejecución de los VI de la figura 14-16

	(a)	(b)	(c)
Tiempo de ejecución (µs)	14	7	5
Tamaño de datos (bytes)	2.427	2.475	2.496

14.4.3.7 Variables y propiedades

Para transferir datos se preferirá por este orden: cables, *shift registers* y variables. Los nodos de propiedad con el atributo *Value* raramente se deben utilizar, a lo largo de este libro sólo se han usado dos veces: en el capítulo de programación orientada a objetos y con XControls y siempre a través de referencias a controles en otros VIs.

A veces es indispensable usar variables, por ejemplo se usarán variables cuando se quiere asignar un valor de forma programada a un control; pero en general el uso de variables, tanto locales como globales, debe evitarse en la medida de lo posible, ya que rompen la estructura de flujo de datos.

Los nodos de propiedad de controles usan el Panel Frontal para trabajar, por lo que provocan que éste se cargue en memoria si no lo está ya, así que su uso se debe evitar en subVIs.

14.4.3.8 SubVIs

Las llamadas a subVIs no penalizan el rendimiento tanto como pudiera parecer, de hecho un subVI puede reutilizar buffers del VI "padre" y si no necesitan cargar el Panel Frontal pueden incluso ahorrar memoria. Por otra parte tienen otras muchas ventajas, como la reutilización del código, portabilidad, modularidad, legibilidad, etc. En conclusión, es muy aconsejable utilizarlos.

Aunque la sobrecarga que supone la llamada a un subVI es mínima, puede ser un factor a tener en cuenta si el subVI está en el interior de un bucle y se llama varias veces. En una situación como esta podría considerarse pasar el bucle desde el VI "padre" al subVI para reducir el número de llamadas.

También se pueden ganar algunos microsegundos y algunos bytes extra deshabilitando la depuración en el VI en *File > VI Properties > Execution > Allow debugging*.

14.5 Ejercicios

1. En los programas de la figura 14-17 se han marcado los buffers producidos por escalares y *arrays*. ¿Cuál está más optimizado en cuanto a uso de memoria y por qué? ¿Y en cuanto tiempo de ejecución? Téngase en cuenta que el índice del bucle FOR es un entero de 32 bits.

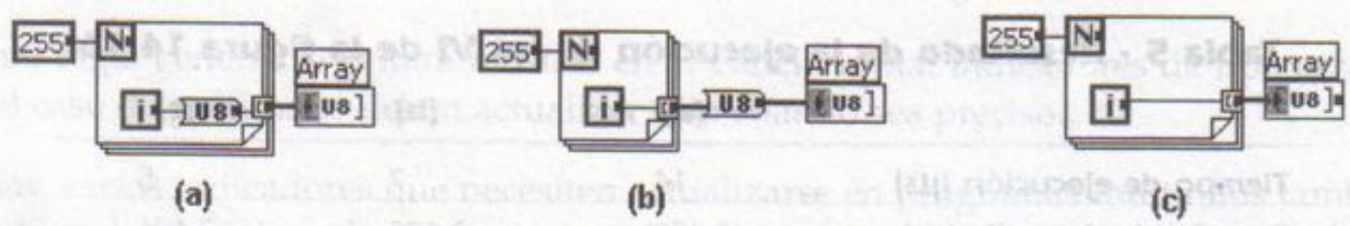


Figura 14-17. Problema propuesto 1

2. Medir la mejora en cuanto a tiempo que se produce al usar *shift register* en lugar de variables locales y nodos de propiedad.

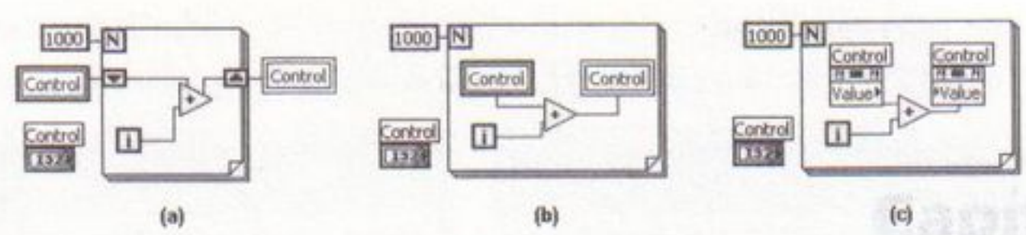


Figura 14-18. Problema propuesto 2

3. Construir una señal aleatoria de 10.000 elementos con valores entre 0 y 1 usando *Initialize Array* y *Replace Array Subset*

14.6 Bibliografía

Barry W. Boehm, *A spiral model of software development and enhancement*, ACM Sigsoft software engineering notes, 1986.

Dan Hedges, *Optimizing VI Performance*, National Instruments, 2002.

Frederick P. Brooks Jr., *The Mythical Man-Month (Anniversary Edition)*, Addison-Wesley, 1995.

Karl Fogel, *Open Source Development With CVS*, 3 edición, Paraglyph Press, 2003.

NASA, *NASA System Engineering Handbook*, 1995.

National Instruments, *AN 168: LabVIEW Performance and Memory Management*, 2004.

National Instruments, *BridgeVIEW and LabVIEW G Programming Reference Manual*, 1998.

National Instruments, *BridgeVIEW and LabVIEW Professional G Developers Tools Reference Manual*, 1998.

National Instruments, *LabVIEW Basics II Course Manual*, 2000.

National Instruments, *LabVIEW Development Guidelines*, 2003.

Rande Johnson, *Rules to Wire By*, LabVIEW Technical Resource Volume 7, Number 2, 1999.

Rick Bitter et alt., *LabVIEW Advanced Programming Techniques*, CRC Press LLC, 2001.

Steve Rogers, *Inside LabVIEW*, National Instruments, 1999.

Winston W. Royce, *Managing the development of large software systems*, IEEE WESCON, 1970.

Capítulo 15

Otras plataformas

15.1 PDA

El módulo PDA de LabVIEW permite crear aplicaciones para Windows Mobile, Windows CE y PalmOS. Antes de usarlo es necesario tener otros programas que dependerán del dispositivo final, estos programas pueden ser software de desarrollo, compiladores, herramientas de sincronización u otros, para obtener más información al respecto se puede consultar el manual del módulo. Aquí se supondrá que el software ya está instalado, tanto en el PC como en la PDA.

Para crear un programa para PDA se puede partir de un asistente al que se accede desde la ventana *Getting Started* de LabVIEW o de un simple proyecto. En el caso de empezar desde un nuevo proyecto se debe añadir un nuevo *Target* o plataforma destino, en ese caso la nueva plataforma aparecerá "al mismo nivel" que *My Computer*. Sobre esa nueva plataforma se podrán crear nuevos VIs que, en la esquina inferior izquierda, mostrarán un rectángulo blanco con la plataforma sobre la que se ejecutaría.

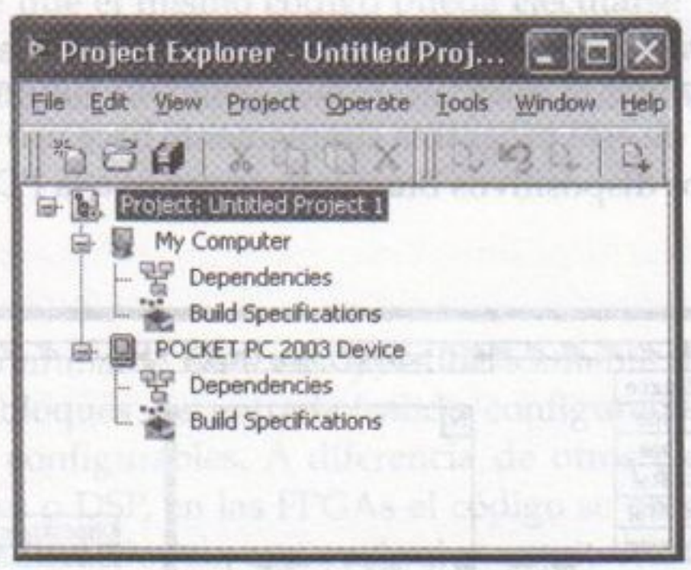


Figura 15-1. Project Explorer con un nuevo target: Pocket PC

Cuando se abre un VI destinado a una plataforma distinta al PC, las paletas de controles y de funciones cambian, mostrando nuevas funciones disponibles para estos dispositivos y ocultando otras. Por ejemplo, en la figura 15-2 puede verse el menú *Data Communication > Protocols*; en la parte izquierda de la imagen se ve el habitual de la

plataforma PC en Windows y en la derecha el de Pocket PC, este último no dispone del menú *Serial* pero en cambio tiene *Phone* y *Short Message Service*.

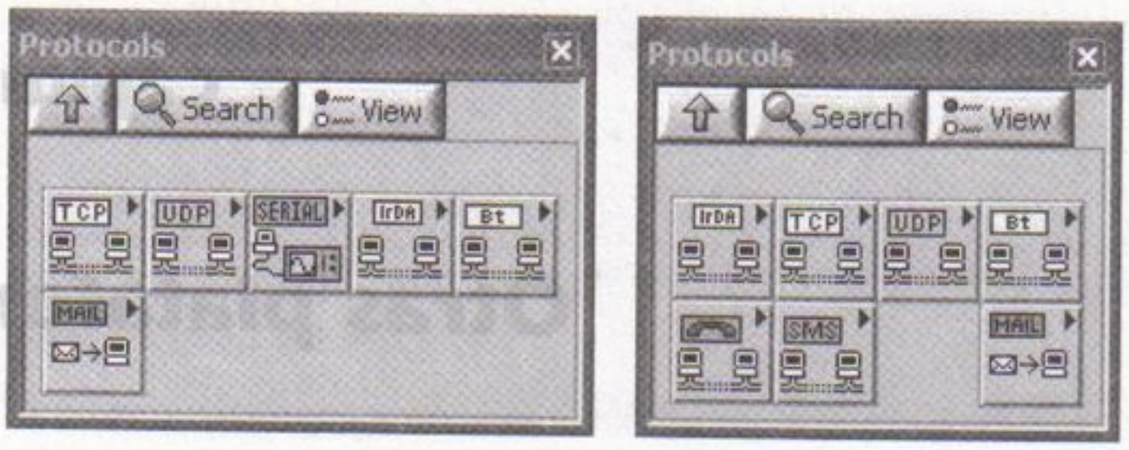


Figura 15-2. Nuevos menús al crear VIs para Pocket PC

15.1.1 Ejemplo: Escáner de dispositivos Bluetooth

15.1.1.1 Explicación teórica

Bluetooth es una especificación de un sistema de comunicación inalámbrica que permite transmitir voz y datos entre diferentes dispositivos por radiofrecuencia.

Para crear y ejecutar el programa, LabVIEW necesitará que estén instalados ActiveSync, Microsoft eMbedded Visual C++ y Microsoft eMbedded Visual Tools. Además también se requiere copiar algunas librerías en la PDA.

El programa consistirá en una aplicación que muestre el nombre y dirección de todos los dispositivos Bluetooth en el radio de alcance de la PDA.

15.1.1.2 Código

El primer paso es completar el asistente, el cual generará automáticamente los ficheros del proyecto. El código de este programa simplemente hace uso de la función *Bluetooth Discover* para buscar los dispositivos Bluetooth cercanos a la PDA y los muestra en una lista.

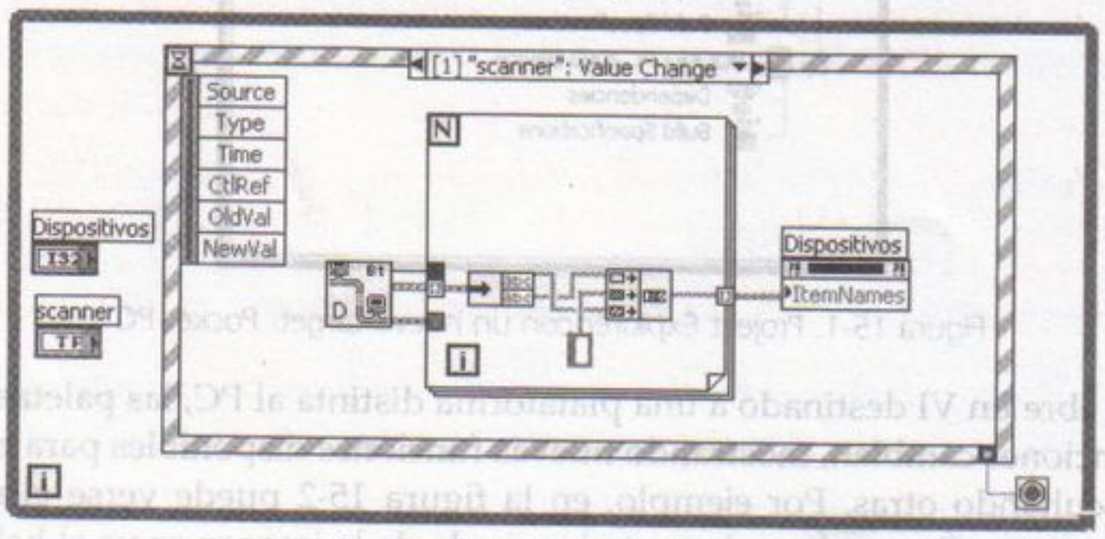


Figura 15-3. Programa para buscar dispositivos Bluetooth

15.1.1.3 Resultados

En la figura 15-4 se puede ver una captura de pantalla a la izquierda del Panel Frontal del VI en el PC y a la derecha el aspecto del programa ejecutándose sobre un dispositivo Pocket PC.

Para construir el ejecutable que funcionará sobre la PDA se debe acudir a *Build Specifications* en la ventana del proyecto.

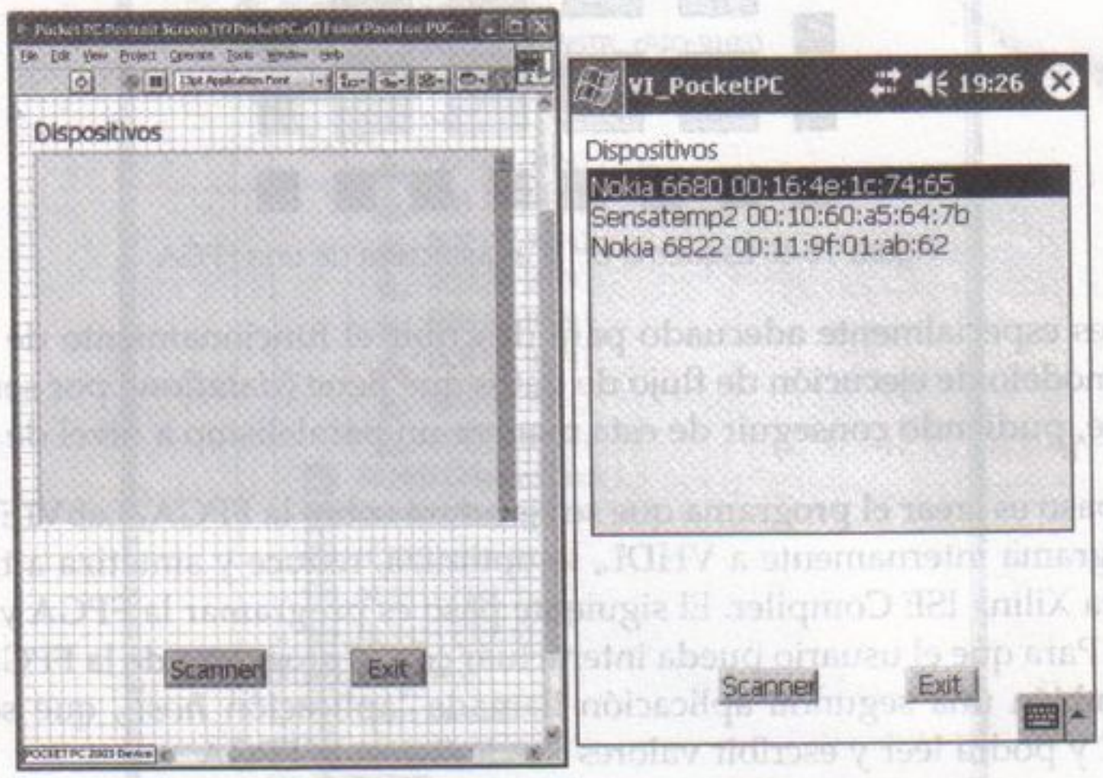


Figura 15-4. Resultado del programa

Nótese como el código de este programa no hace uso de las funciones y VIs específicos para PDA, esto permite que el mismo código pueda ejecutarse tanto en la PDA como en el PC. Si se hiciese uso de elementos dependientes de la plataforma se podría hacer uso de la estructura de deshabilitación condicional para compilar unas cosas u otras en cada dispositivo.

15.2 FPGA

Una FPGA (*Field Programmable Gate Array*) es básicamente un conjunto de bloques lógicos configurables, bloques de entrada/salida configurables y unos recursos de interconexión también configurables. A diferencia de otros elementos programables como microprocesadores o DSP, en las FPGAs el código se programa directamente en el hardware, pudiendo éste adoptar casi cualquier arquitectura. Las herramientas de síntesis determinan la forma de configurar todos los elementos que componen la FPGA para que el funcionamiento sea el descrito por el programador. En la figura 15-5 puede verse de forma esquemática la arquitectura de una FPGA donde se muestran de color gris oscuro los bloques de entrada/salida y de color más claro los bloques lógicos.

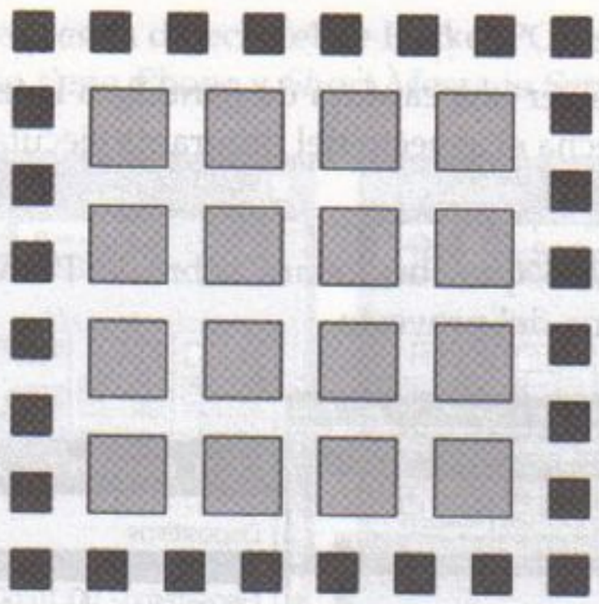


Figura 15-5. Esquema de la arquitectura de una FPGA

LabVIEW es especialmente adecuado para describir el funcionamiento de una FPGA debido al modelo de ejecución de flujo de datos que tiene (*dataflow*) por ser altamente concurrente, pudiendo conseguir de esta manera un paralelismo a nivel de hardware.

El primer paso es crear el programa que se ejecutará sobre la FPGA, LabVIEW convierte este programa internamente a VHDL, lo optimiza, reduce y sintetiza a través de la herramienta Xilinx ISE Compiler. El siguiente paso es programar la FPGA y ejecutar la aplicación. Para que el usuario pueda interactuar con el programa de la FPGA se puede diseñar también una segunda aplicación llamada "aplicación *host*", que se ejecutará sobre el PC y podrá leer y escribir valores desde/hacia la FPGA.

Al igual que al trabajar con PDAs, en este caso también debe comenzarse creando un nuevo *target* sobre el proyecto (obsérvese que los *target* están al mismo nivel que el icono *My Computer*, ambos descendiendo directamente del proyecto). Sobre el *target* podrán crearse nuevos recursos, como en la figura 15-6, donde se puede ver que el *target* es un CompactRIO (controlador 9004), el cual dispone de un chasis 9103 donde se ubica la FPGA, en el chasis hay cuatro slots en los que se insertan los módulos de entradas/salidas (9215, 9263, 9411 y 9474).

Desde las propiedades del dispositivo se podrá elegir si el programa se transferirá y ejecutará sobre la FPGA o se emulará (en *FPGA Target Properties > General > Emulator*).

15.2.1 Ejemplo: Luces del coche fantástico en CompactRIO

15.2.1.1 Explicación teórica

CompactRIO es un sistema RIO (*Reconfigurable Input Output*) que incorpora un procesador con un sistema operativo de tiempo real para operaciones determinísticas (compatible con Pentium 200 MHz), una FPGA (Xilinx Virtex 2) y varios módulos de entrada/salida.

Para este programa se usará el módulo de salidas digitales NI 9474, sobre el que se escribirá una secuencia para encender y apagar los leds que incorpora y crear un efecto de desplazamiento.

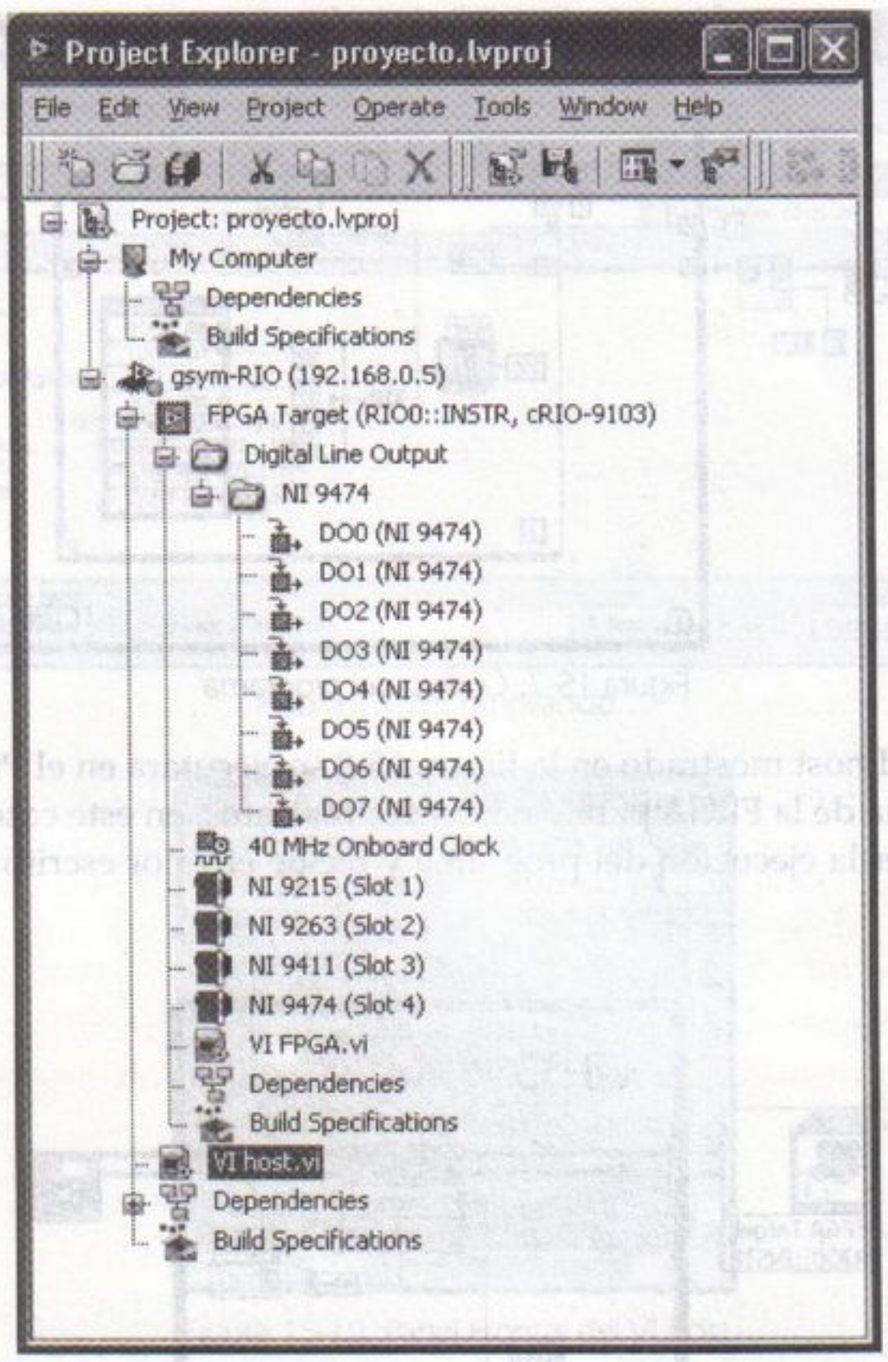


Figura 15-6. Recursos del sistema CompactRIO en el explorador de proyectos

15.2.1.2 Código

El código se divide en dos partes: el código en la FPGA y el código en PC. La comunicación entre ambos es a través de Ethernet usando el procesador del controlador CompactRIO.

El proyecto es el mostrado en la figura 15-6. El código que ejecutará la FPGA es la figura 15-7, donde se puede ver como primero inicializa un *array* de ocho booleanos con valor FALSE y luego asigna al primer elemento el valor TRUE. En cada iteración del bucle WHILE se modificará el signo de la constante numérica y su valor será el que determine el sentido del desplazamiento del *array* dentro del bucle FOR. El bucle FOR ejecutará una iteración cada 200 ms, para esto usará el reloj de la FPGA (40 MHz), en cada iteración del FOR el *array* de booleanos es rotado en una u otra dirección y se escribe en las líneas del módulo de salidas digitales.

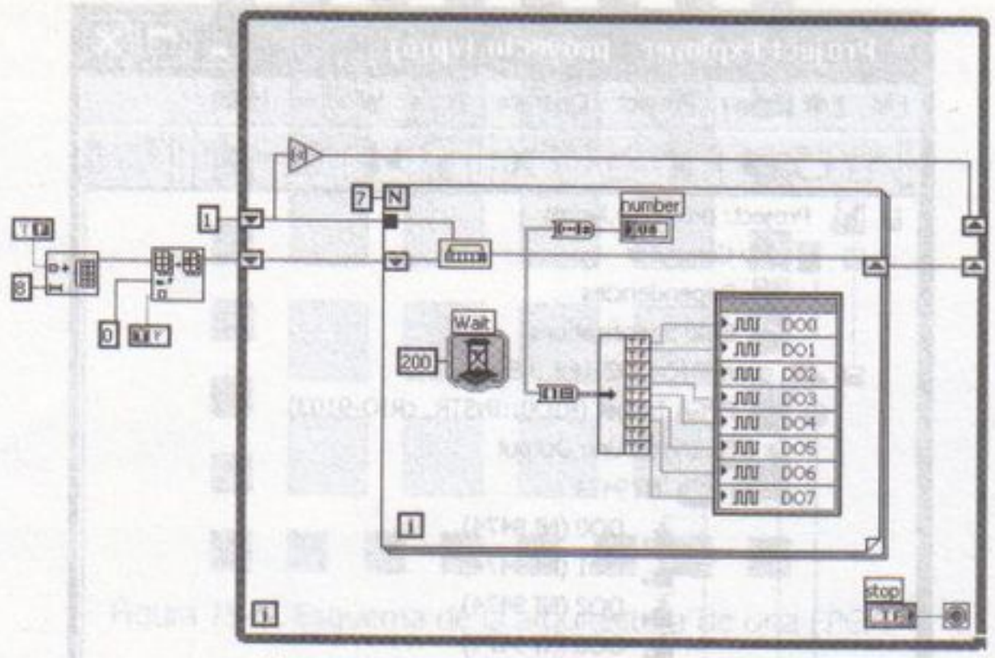


Figura 15-7. Código del programa

Por su parte, el VI host mostrado en la figura 15-8 se ejecutará en el PC y se comunicará con el programa de la FPGA para enviar y recibir datos, en este caso puede enviar la señal para detener la ejecución del programa y recibe el valor escrito en el módulo de salidas.

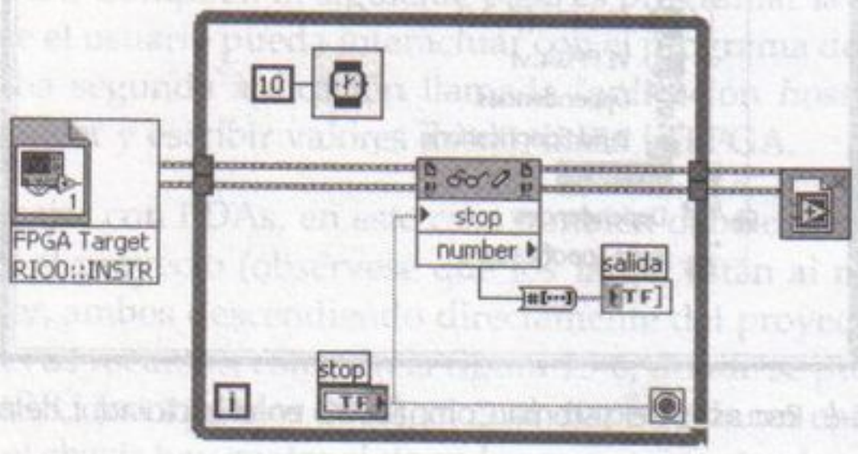


Figura 15-8. Aplicación host

15.2.1.3 Resultados

En la figura 15-9 puede verse una parte del informe de la síntesis del código anterior. Esta tarea puede realizarse desde el explorador del proyecto presionando con el botón derecho sobre el VI de la FPGA y eligiendo *Compile*.

Cuando el VI de la FPGA está compilado ya puede ejecutarse el VI host, entonces el programa es descargado a la FPGA (o emulado, si el proyecto está así configurado) y ejecutado.

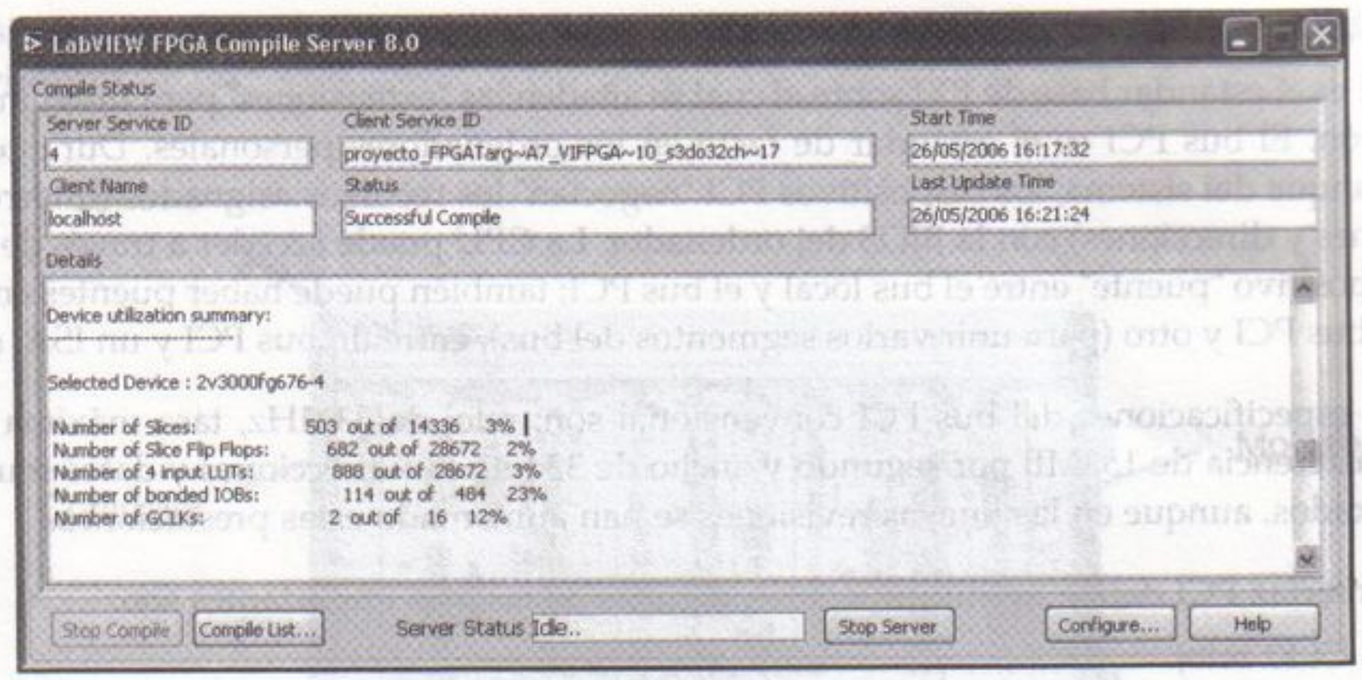


Figura 15-9. Compilando...

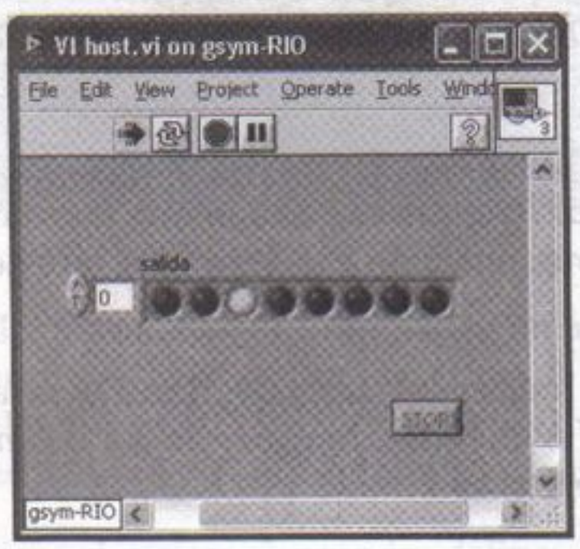


Figura 15-10. Panel Frontal del VI host

15.3 Bus PXI

PXI es el acrónimo de *PCI eXtensions for Instrumentation*, se trata de un bus de bastidor basado en CompactPCI, que a su vez es una adaptación de PCI (*Peripheral Component Interconnect*) para aplicaciones industriales. Por extensión también se llama PXI a un sistema informático basado en dicho bus.

PXI surge como respuesta a la necesidad por parte de la industria de una arquitectura de computadores orientada a la instrumentación, robusta, modular, estándar y de alto rendimiento. Las aplicaciones típicas de PXI son los bancos automáticos de pruebas (ATE) instrumentos modulares (osciloscopios, multímetros, etc.), control para prototipaje rápido y validación de sistemas (RCP y HIL), sistemas de visión artificial... Este bus se desarrolló en 1997 por un consorcio de empresas, entre las que se encuentra Nacional Instruments. En 2005 se creó PXI Express para integrar el nuevo PCI Express en el estándar PXI, aportando mayor ancho de banda, por lo que sus aplicaciones son las comunicaciones de alta velocidad, *datalogging* de de radiofrecuencia, captura de imágenes a alte velocidad, etc.

15.3.1 PCI

PCI es el estándar base de PXI sobre el cual se añaden las "extensiones" para instrumentación. El bus PCI es el estándar de facto en los ordenadores personales. Durante el arranque del sistema, los dispositivos PCI "negocian" los recursos asignados (interrupciones y direcciones) con la BIOS del ordenador. La CPU puede acceder a través de un dispositivo "puente" entre el bus local y el bus PCI; también puede haber puentes entre un bus PCI y otro (para unir varios segmentos del bus), entre un bus PCI y un ISA, etc.

Las especificaciones del bus PCI convencional son: reloj de 33MHz, tasa máxima de transferencia de 133MB por segundo y ancho de 32 bits con direcciones y datos multiplexados, aunque en las nuevas revisiones se han aumentado estas prestaciones.

El interfaz PCI define múltiples señales, algunas de las más importantes son:

- CLK: reloj.
- RST: reset.
- AD[31..0]: *Address/Data*. Son las líneas de direcciones o datos.
- FRAME: indica que se entra en la fase de direccionamiento, la fase de datos sigue a la de direcciones.
- C/BE[3..0]: *Command/Byte Enable*. Durante la fase de direccionamiento contienen un comando del bus, mientras que durante la fase de datos se usan como habilitaciones.

Las líneas de comando indican el tipo de transferencia, por ejemplo *Interrupt Acknowledge, I/O Read, I/O Write, Memory Read, Memory Write, Configuration Read, Configuration Write*, etc.

La configuración se realiza a través de una serie de registros que, entre otra información, contiene el identificador del dispositivo, del fabricante, registro de estado, registro de comando, etc.

15.3.2 PXI

Un sistema PXI está compuesto de tres elementos:

- Un chasis, el cual alberga el propio bus. Los chasis albergan varias ranuras o slots en los cuales se insertarán los módulos.
- Un controlador, que se sitúa en el primer slot de chasis. El controlador puede ser local (básicamente es un ordenador embebido en el módulo) o remoto, para que el PXI pueda ser controlado desde un PC.
- Uno o varios periféricos, por ejemplo instrumentos (osciloscopios, generadores, etc.), módulos I/O, tarjetas DAQ, módulos de conmutación, etc.

Además un sistema PXI puede conectarse con otros sistemas como VXI, VME, CAN, GPIB, etc. Para unirse a otros sistemas PXI puede usar puentes PCI-PCI, de esta forma

el sistema total tendrá más segmentos de bus en los cuales se pueden insertar más periféricos.

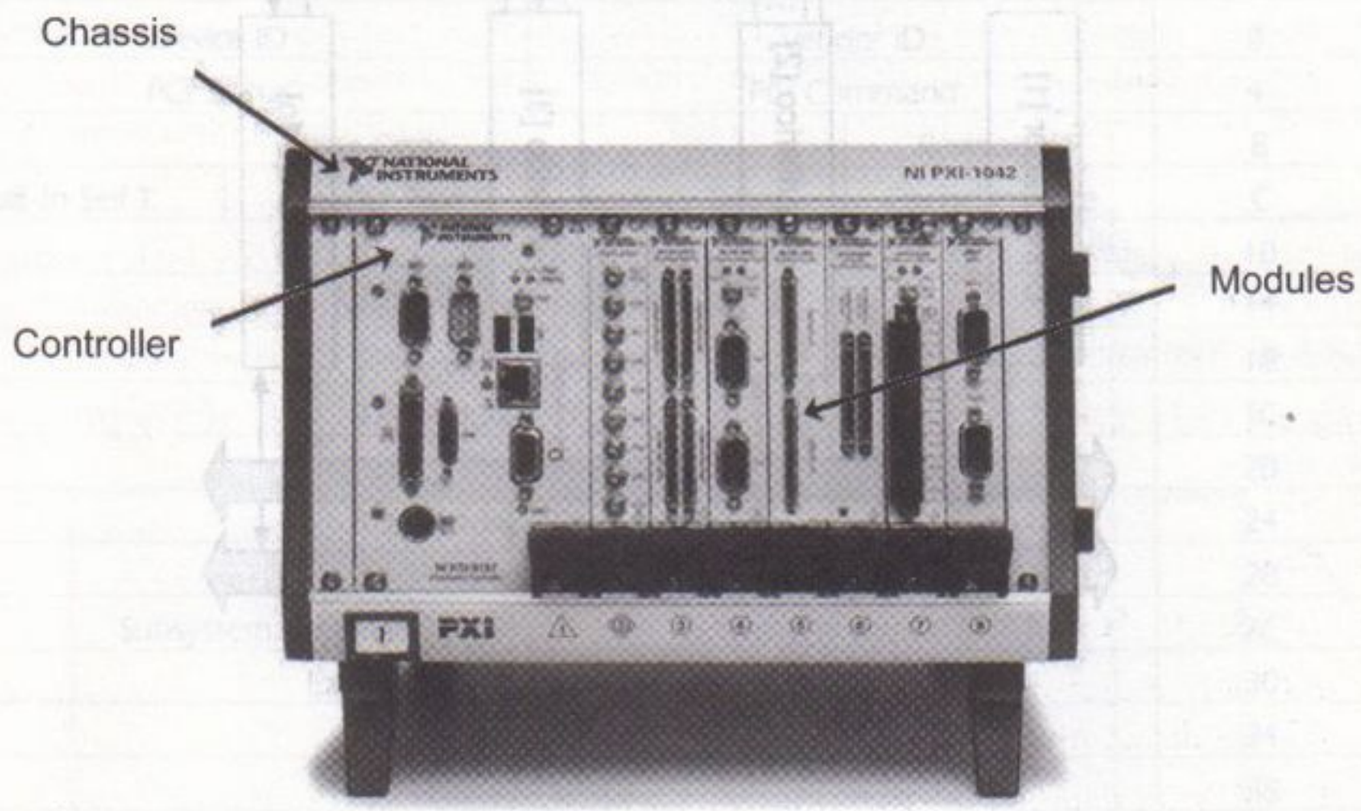


Figura 15-11. Equipo PXI

Las características que PXI añade al bus PCI se dividen en mecánicas, eléctricas y software. En cuanto a las primeras definen un conector de tipo Eurocard igual que CompactPCI de dos posibles tamaños: 3 U (1 U = 1 Rack Unit = 44,45 mm) y 6 U, también se definen otras particularidades como la forma, la refrigeración, etc.

En cuanto a las características eléctricas el estándar define:

- Reloj: 10 MHz común para todos los periféricos.
- Bus de trigger: de ocho líneas para sincronizar los diferentes módulos, el trigger puede ser síncrono o asíncrono.
- Start trigger: son unas líneas de muy alta velocidad conectadas a cada periférico desde el slot número dos; se usan cuando en este slot hay un módulo específico que pueda manejar estas señales.
- Bus local: tipo *daisy-chain* de 13 líneas para comunicaciones de alta velocidad o para pasar señales analógicas de un módulo a otro.

Un elemento característico del estándar PXI que no tienen la mayoría de buses es que también define características software. El estándar define un framework común compatible con VISA a través del cual se proporcionará información y acceso a cada uno de los elementos del sistema PXI.

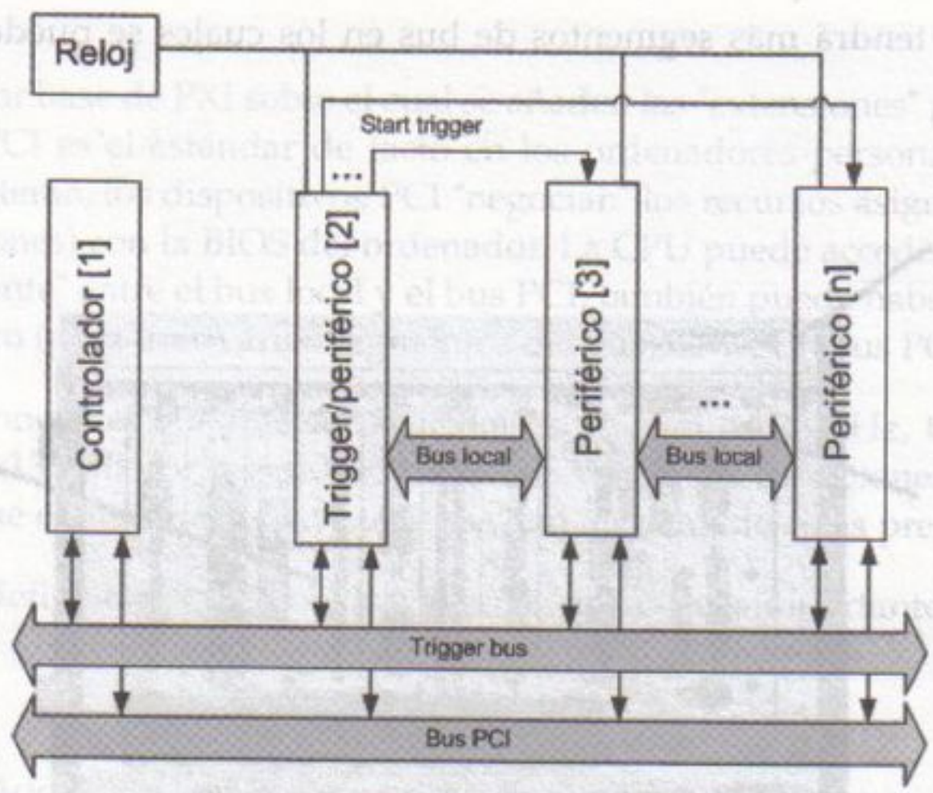


Figura 15-12. Esquema del bus PXI

15.3.3 Ejemplo I: Lectura de los registros de configuración

15.3.3.1 Explicación teórica

El equipo empleado consta de:

- Chasis NI PXI-1042 (8 slots)
- Controlador NI PXI-8186 (Pentium IV 2,20 GHz, 512 MB de RAM)
- Módulo NI-6259 (*Multifunction DAQ*)
- Módulo NI-7831R (*Intelligent DAQ*)
- Módulo NI-5122 (*Digitizer*)
- Módulo NI-4070 (*Multimeter*)
- Módulo NI-5421 (*Arbitrary Waveform Generator*)
- Módulo NI-6552 (*Digital Waveform Generator/Analyzers*)
- Módulo NI-4472 (*Dynamic Signal Acquisition*)

En este primer ejemplo se manejará el módulo PXI-6259 a bajo nivel con el objetivo de leer todos sus registros de configuración. Primero se debe configurar e instalar correctamente el driver del módulo (para ello puede emplearse *VISA Driver Development Wizard*), una vez hecho se podrá acceder a la tarjeta usando VISA.

Los registros de configuración se agrupan de la siguiente forma:

Tabla 1 - Registros de configuración PCI

Byte 3	Byte 2	Byte 1	Byte 0	Dirección
Device ID		Vendor ID		0
PCI Status		PCI Command		4
Class Code			Revision ID	8
Built-In Self T.	Header Type	Latency Timer	CacheLine Size	C
Base Address Register 0				10
Base Address Register 1				14
Base Address Register 2				18
Base Address Register 3				1C
Base Address Register 4				20
Base Address Register 5				24
Reserved Space				28
Subsystem ID		Subsystem Vendor ID		2C
Expansion ROM Base Address				30
Reserved Space				34
Reserved Space				38
Max Latency	Min. Grant	Interrupt Pin	Interrupt Line	3C

Los registros *Base Address Register X* sirven para indicar las ubicaciones del dispositivo dentro del espacio de direcciones o de entrada/salida. *Expansión ROM Base Address* define la dirección base y tamaño para la ampliación ROM del dispositivo en caso de implementarla.

15.3.3.2 Código

Para acceder a los registros del módulo se usarán los VIs del menú *Instrument I/O > VISA > VISA Advanced > Register Access.*

Las direcciones VISA para dispositivos PXI se pueden especificar como `PXI-<bus>::<dispositivo>::<función>::INSTR`. En este caso concreto la dirección del módulo es PXI2::15::INSTR (las direcciones aparecen automáticamente al crear una constante o control, también puede consultarse en MAX). El programa simplemente es un bucle que se repite tantas veces como bytes de configuración existen (64), dentro del bucle se encuentra *VISA IN 8* que lee un bloque de ocho bits indicándole un espacio de direcciones (10 para configuración, 11 para lo apuntado por BAR0, 12 para BAR1 y así sucesivamente hasta 16 para BAR5) y un *offset.*

Basándose en este programa se podrán crear otros para acceder a los registros indicados por BAR0, BAR1, etc. y leer/escribir sus valores. Para esto será indispensable un manual de programación a nivel de registros del dispositivo.

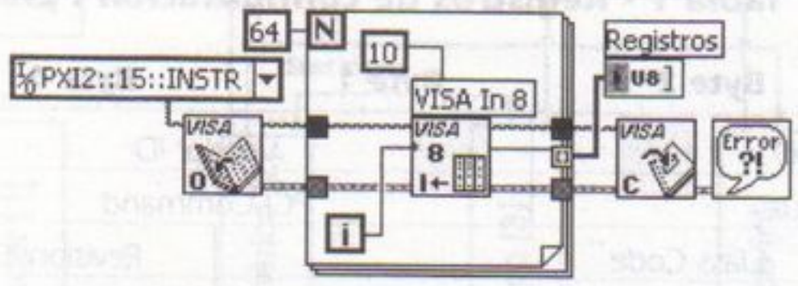


Figura 15-13. Código para visualizar los registros de configuración

15.3.3.3 Resultado

La figura 15-14 muestra el resultado de la ejecución del programa en el equipo PXI descrito antes.

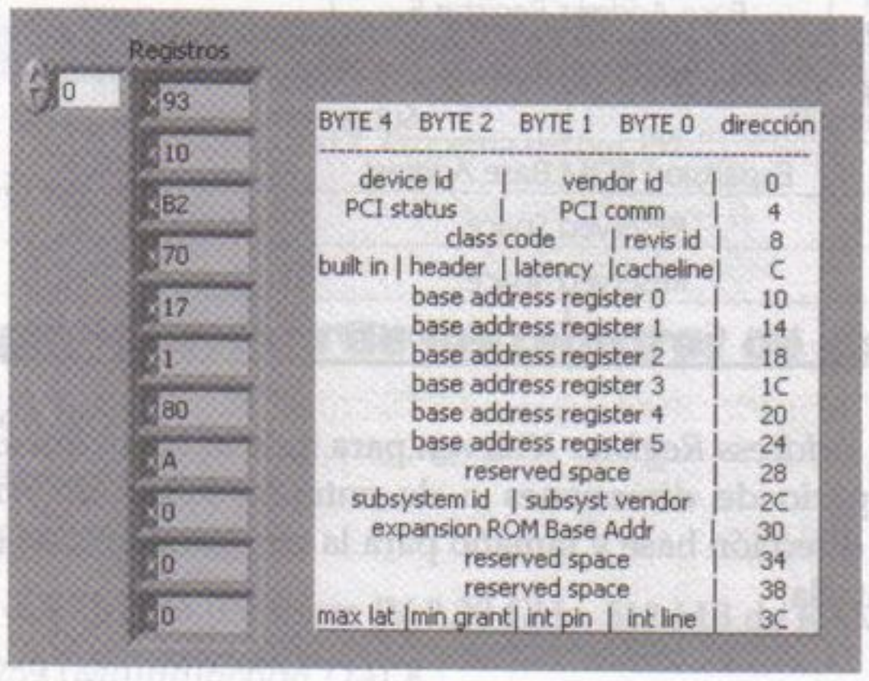


Figura 15-14. Valor de los registros de configuración

Por ejemplo, el registro *Vendor id* tiene un valor 1093h, que es el correspondiente a National Instruments.

15.3.4 Ejemplo II: Generación y adquisición de señales

15.3.4.1 Explicación teórica

Si en el caso anterior se veía un ejemplo de bajo nivel, con este se hará justo lo contrario, manejar los equipos del sistema PXI a alto nivel con el propósito de demostrar que el programador de LabVIEW puede tener total transparencia del hardware usado.

Con este propósito se utilizarán dos instrumentos: el digitalizador NI PXI-5122 y el generador de formas de onda arbitrarias NI PXI-5421, ambos conectados. El generador creará una forma de onda y el digitalizador la capturará, mostrando el resultado en el Panel Frontal.

15.3.4.2 Código

En este ejemplo no será necesario recurrir a las funciones VISA, se emplearán las funciones de NI-FGEN y NI-SCOPE, ambos compatibles con IVI y que se proporcionan junto con los equipos (también se pueden bajar de la web de National Instruments gratuitamente). Estas funciones se hallan en el menú Functions > Measurement I/O, su utilización es similar a NI DAQmx.

El programa es extremadamente sencillo, consta de dos bucles en paralelo, en el WHILE superior se usará NI-FGEN para que el usuario genere una forma de onda, los pasos son inicializar el dispositivo, configurarlo, inicializar la generación y finalmente cerrar. En el bucle TIMED LOOP inferior se empleará NI-SCOPE para leer de forma periódica la forma de onda del generador.

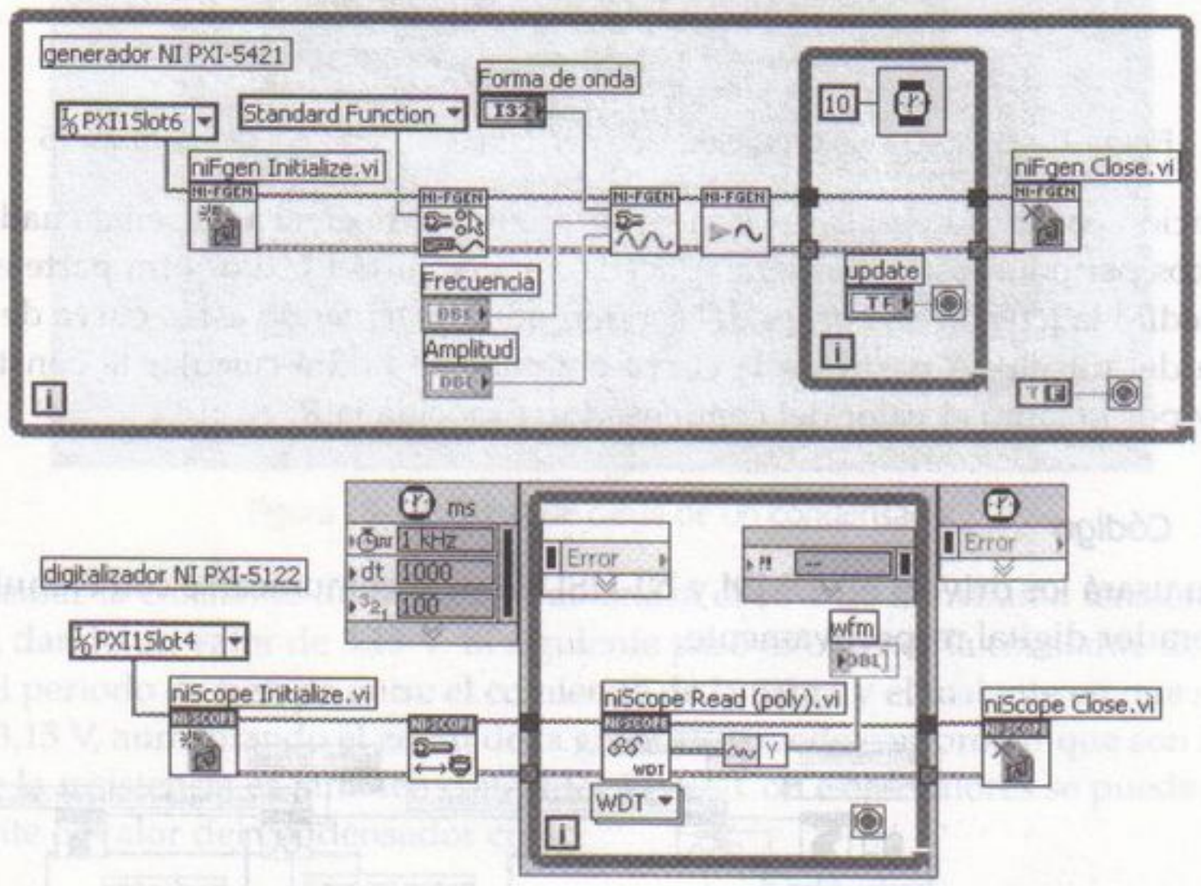


Figura 15-15. Código que maneja un generador y un digitalizador PXI

15.3.4.3 Resultados

Una vez conectados los dos instrumentos sólo queda ejecutar el programa anterior para ver los resultados, como se puede comprobar en la Figura 15-16.

15.3.5 Ejemplo III: Medida de capacidad

15.3.5.1 Explicación teórica

En este ejemplo se hará uso de otros dos instrumentos modulares del sistema PXI descrito en el ejemplo I, ahora serán el multímetro NI PXI-4070 y el generador-analizador digital NI PXI-6552.

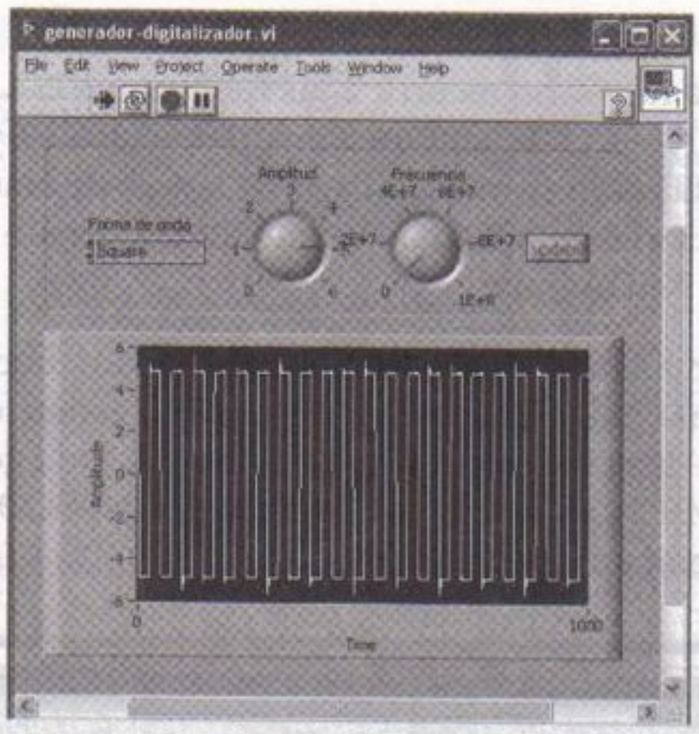


Figura 15-16. Resultado de la ejecución del programa dado por la Figura 15-15

La aplicación consistirá en usar el generador digital para crear una señal cuadrada de unos pocos periodos, esa señal será aplicada a un circuito RC. Por otra parte el multímetro medirá la tensión en bornes del condensador, obteniendo así la curva de carga y descarga del mismo. A partir de la curva obtenida se podrá calcular la constante de tiempo y por lo tanto el valor del condensador, conocida la R.

15.3.5.2 Código

El código usará los drivers NI-DMM y NI-HSDIO para comunicarse con el multímetro y el generador digital respectivamente.

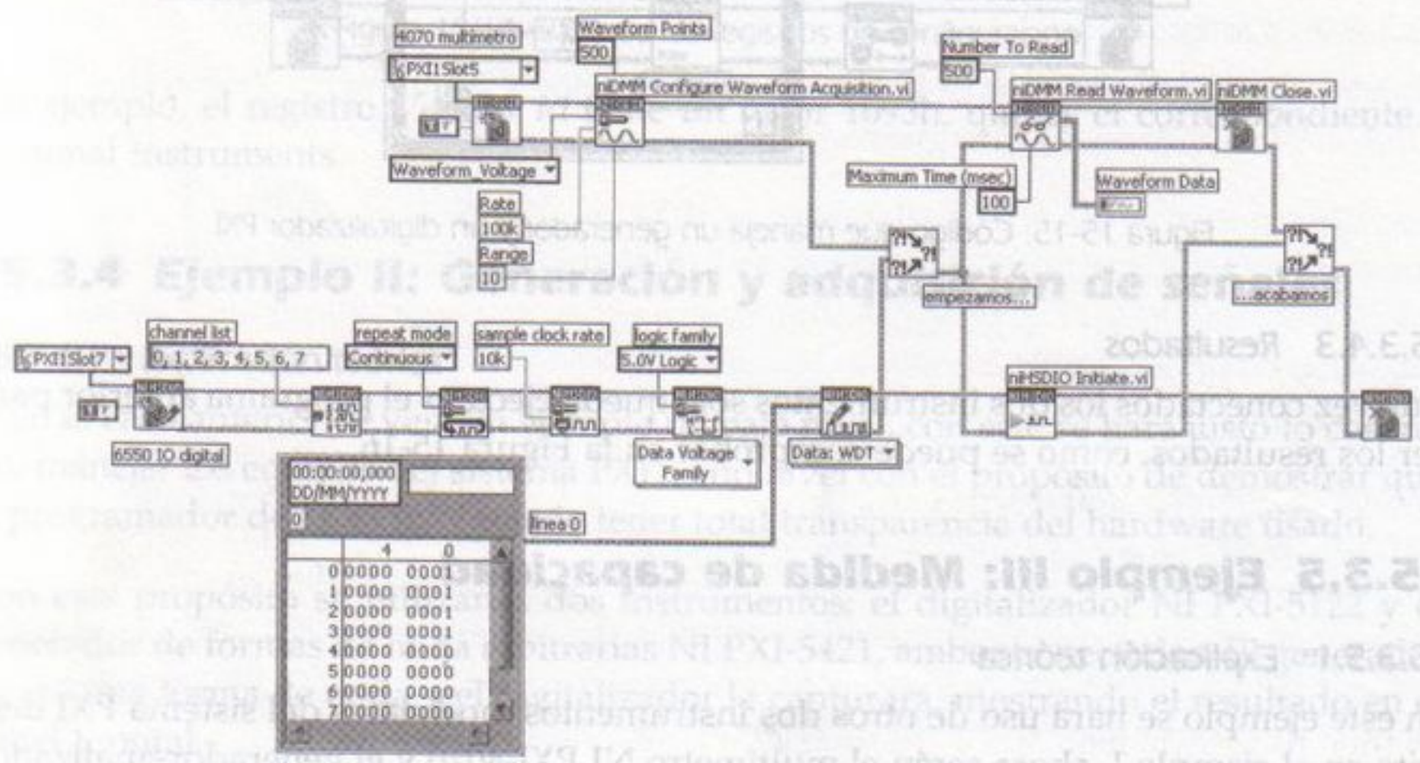


Figura 15-17. Programa para obtener la curva de carga de un condensador

El programa en una primera fase inicializa y configura el multímetro y el generador. A continuación se inicializa la generación digital y se lee una forma de onda a través del multímetro. Cuando la señal se ha capturado se cierran ambos dispositivos, primero el multímetro y después el generador.

15.3.5.3 Resultados

Al ejecutar el programa se ha obtenido la gráfica de la figura 15-18.

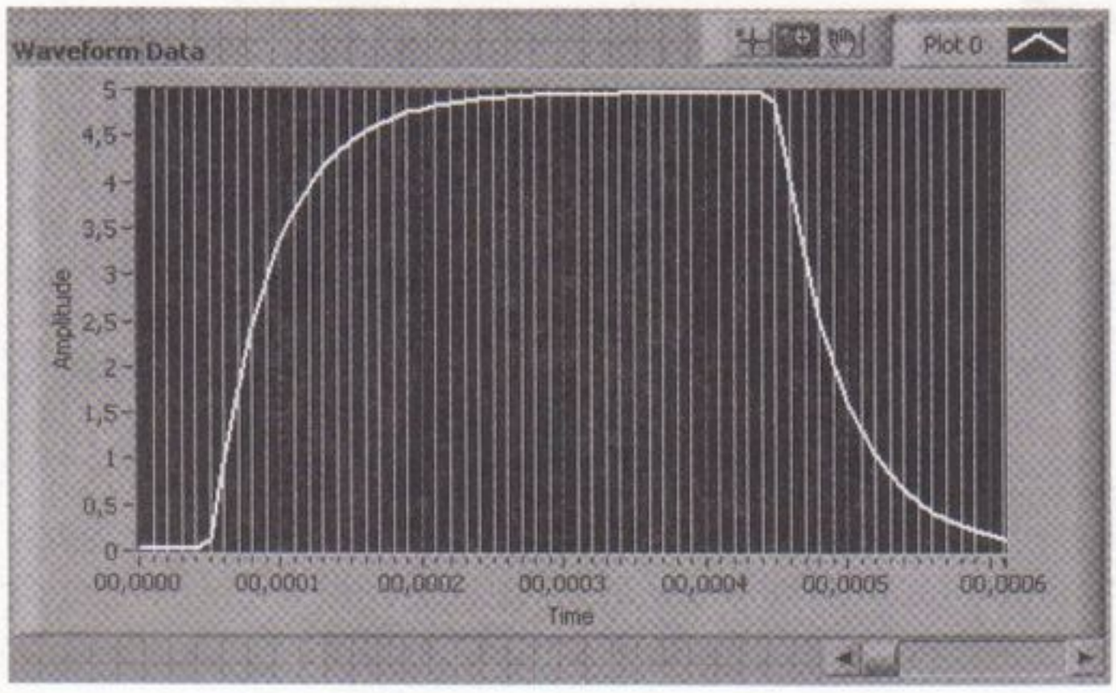


Figura 15-18. Curva de carga de un condensador

Para calcular la constante de tiempo se obtendrá el 63% de la máxima tensión, siendo ésta 5 V, dando un valor de 3,15 V. El siguiente paso es obtener la constante de tiempo, que es el periodo de tiempo entre el comienzo de la carga y el instante en que se alcanzan los 3,15 V, aumentando el zoom de la gráfica se puede comprobar que son 55 ±s. El valor de la resistencia es también conocido: 4,7 k?. Con estos valores se puede calcular fácilmente el valor del condensador como:

$$\tau = R \cdot C \rightarrow C = \frac{55\mu s}{4{,}7k\Omega} = 11{,}7nF$$

15.4 Bibliografía

Joseph DiGiovanni, Advanced LabVIEW FPGA Programming Optimizing for Speed and Size, National Instruments, 2005.

National Instruments, *Getting Started with the LabVIEW PDA Module.*

National Instruments, *NI 5421 Specifications,* NI.

National Instruments, *NI 625x Specifications,* NI.

National Instruments, *NI PXI/PCI-5122 Specifications,* NI.

National Instruments, *WP 2329: Developing Measurement and Control Applications with the LabVIEW FPGA Pioneer System*, 2003.

National Instruments, *WP 2480: NI CompactRIO - Reconfigurable Control and Acquisition System*, 2004.

PXI System Alliance, *PXI Hardware Specification*, rev. 2.2, 2004.

PXI System Alliance, WP: *The PXI Modular Instrumentation Architecture*, 2000.

LabVIEW Entorno gráfico de programación

En este libro se pretende dar al lector una visión completa de todos los aspectos de LabVIEW. El libro parte de una introducción al entorno para lectores que no tengan conocimientos previos del programa y poco a poco se va aumentando el nivel hasta llegar a abarcar prácticamente todas las posibilidades que permite LabVIEW, prácticamente infinitas, permitiendo a los lectores descubrir nuevas posibilidades.

El libro se divide en tres bloques temáticos: una introducción en la que se presenta el entorno y se explican los tipos de datos y las estructuras de control; un segundo bloque de adquisición y comunicaciones siendo ésta la aplicación más tradicional de LabVIEW; y por último los aspectos más avanzados donde el lector podrá obtener un conocimiento profundo del compilador y las técnicas de programación más adecuadas para cada situación.

En cada capítulo se parte de una introducción teórica general de la temática que se trata para, a continuación, presentar las funciones específicas de LabVIEW relacionadas. Una vez hecho esto se asentarán los conocimientos a través de varios ejemplos y ejercicios prácticos.

Los ejemplos son prácticos y amenos para que se disfrute con la lectura del libro mientras se adquieren nuevos conocimientos, esperamos que esto fomente la iniciativa de los lectores para modificarlos, mejorarlos y adaptarlos a sus requerimientos.

José Rafael Lajara Vizcaíno es Ingeniero Técnico en Telecomunicaciones e Ingeniero en Electrónica por la Universidad Politécnica de Valencia (2003) y la Universidad de Valencia (2006) respectivamente. Desde 2002 pertenece al grupo de investigación GSYM y ha trabajado en campos como sistemas embebidos, instrumentación virtual y redes de sensores. Actualmente está cursando los estudios de doctorado del departamento de Ingeniería Electrónica.

José Pelegrí Sebastiá es Doctor en Ciencias Físicas e Ingeniero en Electrónica por la Universidad de Valencia. Actualmente es Titular de Escuela Universitaria del Departamento de Ingeniería Electrónica de la Universidad Politécnica de Valencia donde imparte clases en el Área de Instrumentación desde 2000. Es el responsable del grupo de investigación GSYM cuyas líneas de interés incluye los sistemas de instrumentación, la caracterización de sensores y su aplicabilidad y las redes de sensores.

ISBN 978-970-15-1133-6

9 789701 511336

Alfaomega Grupo Editor