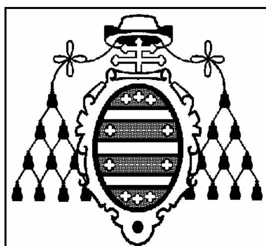
UNIVERSIDAD DE OVIEDO



ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA INFORMÁTICA Y TELEMÁTICA DE GIJÓN

DOCUMENTO Nº 2

MANUALES DEL PROGRAMADOR

CONJUNTO DE DRIVERS PARA TARJETAS DE ADQUISICIÓN DE SEÑALES BAJO LINUX

Adolfo Antonio Fernández Trabanco

JUNIO 2005

Índice

1 .- Introducción 1
1.1 .- Identificación del proyecto 1
1.2 .- Visión general del documento 1
1.3 .- El puerto serie en Linux 2
1.3.1 .- Conceptos de entrada para dispositivos serie 5
1.3.1.1 .- Proceso de entrada canónico.............................. 5
1.3.1.2 .- Proceso de entrada no canónico 5
1.3.1.3 .- Entrada asíncrona 5
1.3.1.4 .- Espera de entradas de origen múltiple.................. 6
1.3.2 .- Configuración del puerto serie en Linux 6
1.3.2.1 .- Opciones de control 10
1.3.2.2 .- Opciones locales 12
1.3.2.3 .- Opciones de entrada..13
1.3.2.4 .- Opciones de salida 14
1.3.2.5 .- Caracteres de control.......................................15
1.3.3 .- Implementación realizada 16
1.3.3.1 .- Abrir_Puerto_Serie 18
1.3.3.2 .- Cerrar_Puerto_Serie 19
1.3.3.3 .- Baudios 19
1.3.3.4 .- Enviar_Datos..20
1.3.3.5 .- Leer_Datos 20
1.3.3.6 .- Leer_Datos_Plazo..21
1.3.3.7 .- Get_fd 21
1.3.3.8 .- Get_Velocidad 22
1.3.3.9 .- Get_Puerto...22

2 .- FieldPoint, manual del programador 23
2.1 .- Protocolo de comunicación 23
2.1.1 .- Formato de las tramas FieldPoint.................................23
2.1.2 .- Respuestas de los módulos FieldPoint 25
2.1.2.1 .- Respuestas de éxito 25
2.1.2.2 .- Respuestas de error 25

2.2 .- Implementación del driver 26

2.2.1 .- Conceptos previos: el bus local 27

2.2.2 .- Código C..28

2.2.2.1 .- Estructuras y variables utilizadas........................28

2.2.2.2 .- Funciones locales 33

2.2.2.3 .- Funciones globales 35

2.2.3 .- Código Ada 37

2.2.4 .- Descripción de las funciones 38

NI_Inicializar 38

NI_Liberar...39

NI_EscribeLinea_DO 39

NI_EscribeModulo_DO...40

NI_LeeLinea_DI 40

NI_LeeModulo_DI..41

ObtenerError 42

ObtenerErrorString 42

Num_Lineas 43

Num_Modulos...43

Get_Tipo..44

Get_Nombre..44

Get_Id...45

Get_Direccion 45

2.3 .- Anexo A: FieldPoint Commands 46

Comandos utilizados..49

2.4 .- Anexo B: FieldPoint Responses 56

3 .- NuDAM, manual del programador 59

3.1 .- Protocolo de comunicación 59

3.1.1 .- Formato de las tramas...59

3.1.2 .- Respuestas de los módulos NuDAM...............................60

3.2 .- Implementación del driver 61

3.2.1 .- Conceptos previos: el bus local 62

3.2.2 .- Código C..63

3.2.2.1 .- Estructuras y variables utilizadas........................63

3.2.2.2 .- Funciones locales 67

3.2.2.3 .- Funciones globales 69
3.2.3 .- Código Ada 71
3.2.4 .- Descripción de las funciones 72
ND_Inicializar 72
ND_Liberar....................................73
ND_EscribeLinea_DO 73
ND_EscribePuerto_DO....................................74
ND_EscribeModulo_DO....................................75
ND_LeeLinea_DI 76
ND_LeeModulo_DI....................................77
ObtenerError 78
ObtenerErrorString 78
Num_Lineas 79
Num_Modulos....................................79
Get_Tipo....................................80
Get_Nombre....................................81
Get_Id....................................81
Get_Direccion 82
3.3 .- Anexo A: Command Set (digital I/O modules)....................................83
Comandos utilizados....................................85

4 .- LabJack U12, manual del programador 92
4.1 .- Implementación del driver 92
4.1.1 .- Conceptos previos....................................92
4.1.2 .- Código C....................................92
4.1.2.1 .- Estructuras y variables utilizadas....................................92
4.1.2.2 .- Funciones locales 94
4.1.2.3 .- Funciones globales 94
4.1.3 .- Código Ada 96
4.1.4 .- Descripción de las funciones 97
LJ_Inicializar 97
LJ_EscribeLinea_DO....................................98
LJ_EscribeModulo_DO 98
LJ_LeeLinea_DI....................................99
LJ_LeeModulo_DI 100

LJ_EscribeLinea_AO 101

ObtenerError 101

ObtenerErrorString 102

4.2 .- Anexo A: Descripción de las funciones propias del fabricante 103

4.3 .- Anexo B: Description of errorcodes 106

Listado de Tablas

Tabla 1.1 Asociación de puertos serie a ficheros de dispositivo 2

Tabla 1.2 Miembros de la estructura *termios* 6

Tabla 1.3 Constantes para el miembro *c_cflag* 11

Tabla 1.4 Constantes para el miembro *c_lflag* 12

Tabla 1.5 Constantes para el miembro *c_iflag* 13

Tabla 1.6 Constantes para el miembro *c_oflag* 14

Tabla 1.7 Caracteres de control para el miembro *c_cc* 15

Tabla 2.1 Ejemplo de asignación de direcciones en el bus local 27

Tabla 2.2 Variables de error 29

Tabla 2.3 Standard FieldPoint Commands and Syntax 46

Tabla 2.4 Extended FieldPoint Commands and Syntax 47

Tabla 2.5 Standard Error Responses 57

Tabla 2.6 Extended Error Responses 58

Tabla 3.1 Ejemplo de asignación de direcciones en el bus local 62

Tabla 3.2 Variables de error 64

Tabla 3.3 Command set of digital I/O modules 84

Listado de ejemplos

Ejemplo 1.1 Utilización de la llamada al sistema *open* 3
Ejemplo 1.2 Utilización de la llamada al sistema *write* 4
Ejemplo 1.3 Utilización de la llamada al sistema *read* 4
Ejemplo 1.4 Utilización de las funciones de configuración del puerto 9
Ejemplo 2.1 Interfaz C-Ada 37
Ejemplo 3.1 Interfaz C-Ada 71
Ejemplo 4.1 Interfaz C-Ada 96

1.- Introducción

1.1.- Identificación del proyecto

Título: Conjunto de drivers para tarjetas de adquisición de señales bajo Linux

Directores: Víctor Manuel González Suárez
José Antonio Cancelas Caso

Autor: Adolfo Antonio Fernández Trabanco

Fecha: Junio 2005

1.2.- Visión general del documento

En el presente documento se incluyen los manuales del programador de las librerías desarrolladas.

Previamente al desarrollo de cada uno de los manuales, se muestra en la Introducción los conceptos necesarios sobre programación del puerto serie en Linux. Dado que la implementación realizada para acceder al puerto serie es prácticamente igual para los módulos FieldPoint y NuDAM, también se incluye en la Introducción la explicación de ésta, así como las diferencias que existen entre ambos sistemas cuando se desea realizar la configuración del puerto.

Se incluyen en el documento tres manuales:

- **FieldPoint, manual del programador** (epígrafe 2)
  Manual del programador de la librería desarrollada para el sistema FieldPoint de la empresa National Instruments.

- **NuDAM, manual del programador** (epígrafe 3)
  Manual del programador de la librería desarrollada para el sistema NuDAM de la empresa ADLink Technology.

- **LabJack U12, manual de usuario** (epígrafe 4)
  Manual del programador de la librería desarrollada para la Tarjeta de Adquisición de Datos LabJack U12 de la empresa LabJack.

1.3.- El puerto serie en Linux

En Linux, los periféricos se encuentran representados como simples ficheros del sistema de archivos. De esta forma, si se quiere enviar datos a través del puerto serie, basta con abrir el fichero asociado al puerto y escribir con las funciones de siempre. Cada uno de los puertos serie tiene asociado uno o más ficheros de dispositivo. En la Tabla 1.1 se muestra la asociación más habitual entre los puertos serie y los ficheros del sistema de archivos.

Puerto	Fichero
COM1	/dev/ttyS0
COM2	/dev/ttyS1
COM3	/dev/ttyS2
COM4	/dev/ttyS3

Tabla 1.1 Asociación de puertos serie a ficheros de dispositivo

Para acceder al puerto únicamente es necesario abrir el fichero asociado mediante la llamada al sistema *open*. Se debe tener en cuenta que, en Linux, los ficheros asociados a los dispositivos no suelen tener permiso de acceso para los usuarios.

Para evitar este problema existen varias soluciones:

- Cambiar los permisos del fichero/s asociado al dispositivo y dar permisos de lectura y escritura a los usuarios.

- Ejecutar el programa realizado en modo super-usuario (root).

La especificación de la función *open* es la siguiente:

```
int open(const char *camino, int flags);
```

Se debe indicar el fichero que se desea abrir y las opciones de apertura necesarias. Estas opciones se indican mediante el uso de constantes definidas por el operativo. Para obtener una descripción detallada de todas las opciones existentes ejecute *man 2 open* en su consola. La función retorna un número negativo si no se pudo abrir el fichero indicado. Si el fichero se abrió correctamente, se retorna un descriptor de fichero (un entero no negativo que se utiliza en las operaciones de E/S posteriores, como en *read*, *write*, etc.).

En el Ejemplo 1.1 se muestra el uso de la llamada al sistema *open*. La opción *O_RDWR* indica que el fichero */dev/ttyS1* (fichero asociado al puerto COM2) se debe abrir en modo lectura y escritura.
La opción *O_NOCTTY* indica que el fichero no se abrirá como controlador de terminal tty. Si no se especifica esta opción una señal de finalización (abort) enviada por la línea finalizaría la ejecución.

```
strcpy(_puerto,"/dev/ttyS1");
_fd = open (_puerto, O_RDWR | O_NOCTTY);
  if (_fd <  0) { // No se pudo abrir el puerto
    return _fd;
  }
```

Ejemplo 1.1 Utilización de la llamada al sistema *open*

Para mandar datos al puerto se emplea la llamada al sistema *write*:

```
ssize_t write(int fd, const void *buf, size_t num);
```

Escribe hasta *num* bytes en el fichero referenciado por el descriptor de fichero *fd* desde el búfer que comienza en *buf*. En el Ejemplo 1.2 se puede ver una forma de uso. Para obtener más información sobre esta llamada al sistema, acuda a la ayuda en línea de su operativo: *man 2 write*.

```
int Enviar_Datos(const char * comando){
  ssize_t escribo;
  //comprobamos que existe un descriptor de fichero valido
  if (_fd < 0) return (-2);

  escribo = write(_fd, comando, strlen(comando));
  return escribo;
}
```

Ejemplo 1.2 Utilización de la llamada al sistema *write*

Para leer datos del puerto se emplea la llamada al sistema read:

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

Intenta leer hasta *nbytes* bytes del fichero cuyo descriptor de fichero es *fd* y guardarlos en la zona de memoria que empieza en *buf*. En el Ejemplo 1.2 se puede ver una forma de uso. La llamada al sistema *read,* no finaliza el buffer donde deposita los datos con el ASCII nulo. Es por ello, que esta tarea se hace explícitamente: `buffer[leo]='\0';` . Para obtener más información sobre esta llamada al sistema, acuda a la ayuda en línea de su operativo: *man 2 read*.

```
int Leer_Datos(char * buffer, int length){
  ssize_t leo;

  //comprobamos que existe un descriptor de fichero valido
  if (_fd < 0) return (-2);

  leo = read (_fd, buffer, length);
  buffer[leo]='\0'; //ponemos el caracter nulo de final
  return leo;
}
```

Ejemplo 1.3 Utilización de la llamada al sistema *read*

1.3.1.- Conceptos de entrada para dispositivos serie

Hay tres conceptos diferentes de entrada para dispositivos serie. Según la aplicación que se quiera realizar se debe escoger el concepto apropiado. Siempre que sea posible, se recomienda no obtener una cadena completa mediante un bucle que vaya leyendo carácter a carácter.

1.3.1.1.- Proceso de entrada canónico

Es el modo de proceso normal para terminales, pero puede ser útil también para comunicaciones con otros dispositivos. Toda la entrada es procesada en unidades de líneas, lo que significa que un *read* sólo devolverá una línea completa de entrada. Una línea, por defecto, está finalizada con un NL y fin de fichero o con un carácter fin de línea. Un CR (el fin de línea por defecto en DOS/Windows) no terminará una línea con la configuración por defecto.

El proceso de entrada canónica también puede manejar los caracteres borrado, borrado de palabra, reimprimir carácter, traducir CR a NL, etc.

1.3.1.2.- Proceso de entrada no canónico

Maneja un número fijo de caracteres por lectura y permite un carácter temporizador. Este modo se debería usar si la aplicación siempre lee un número fijo de caracteres o si el dispositivo conectado envía ráfagas de caracteres.

1.3.1.3.- Entrada asíncrona

Los dos procesos de entrada descritos anteriormente se pueden usar en dos modos:

- **Modo síncrono**: la sentencia *read* se bloqueará hasta que la lectura esté completa (este es el modo por defecto).

- **Modo asíncrono**: la sentencia *read* devolverá inmediatamente y enviará una señal al programa llamador cuando esté completa. Esta señal puede ser recibida por un manejador de señales que la tratará adecuadamente.

1.3.1.4.- Espera de entradas de origen múltiple

No es un modo diferente de entrada, pero puede ser útil si se están manejando dispositivos múltiples. En aplicaciones donde se esperen entradas de distintos orígenes, se procesarán aquellas entradas que estén disponibles y, una vez procesadas, se esperará la llegada de nuevas entradas.

En el epígrafe 3.4 del documento *"Cómo programar el puerto serie en Linux"* se puede obtener más información sobre este modo de entrada.

1.3.2.- Configuración del puerto serie en Linux

En este epígrafe se mostrará la configuración del puerto serie en lenguaje C utilizando la interfaz que proporciona el estándar POSIX (acrónimo de Portable operating system interface). La información aquí mostrada se ha obtenido, entre otras fuentes, del documento "*Serial Programming Guide for POSIX Operating Systems*".

En el archivo de cabecera `<termios.h>` se definen las estructuras y funciones necesarias para llevar a cabo la configuración del puerto serie. La estructura más importante es la estructura `termios` formada por los miembros mostrados en la Tabla 1.2.

Miembro	Descripción
c_cflag	Opciones de control
c_lflag	Opciones locales
c_iflag	Opciones de entrada
c_oflag	Opciones de salida
c_cc	Caracteres de control
c_ispeed	Velocidad (baudios) de entrada
c_ospeed	Velocidad (baudios) de salida

Tabla 1.2 Miembros de la estructura *termios*

Las funciones que se utilizarán para llevar a cabo la configuración del puerto serie son:

- ***tcgetattr***: Se utiliza para obtener los parámetros actuales del puerto.

```
int tcgetattr ( int fd, struct termios *termios_p );
```

  Obtiene los parámetros asociados con el objeto referido por *fd* y los guarda en la estructura termios referenciada por *termios_p*.

- ***tcsetattr***: Se utiliza para actualizar los parámetros del puerto.

```
int tcsetattr ( int fd, int optional_actions,
                struct termios *termios_p);
```

  Establece los parámetros de la terminal con descriptor de fichero *fd*, desde la estructura termios referenciada por *termios_p*. El argumento *optional_actions* especifica cuándo tienen efecto los cambios:

  - TCSANOW: El cambio ocurre inmediatamente.

  - TCSADRAIN: El cambio ocurre después de que toda la salida escrita a fd haya sido transmitida. Esta función debería emplearse cuando se cambien parámetros que afecten a la salida.

  - TCSAFLUSH: El cambio ocurre después de que toda la salida escrita al objeto referenciado por fd haya sido transmitida, y toda la entrada que se haya recibido pero no leído será descartada antes de que se haga el cambio.

- ***tcflush***: Se utiliza para vaciar los buffers de entrada/salida.

```
int tcflush ( int fd, int queue_selector );
```

  Descarta datos escritos al objeto referido por *fd* pero no transmitidos, o datos recibidos pero no leídos, dependiendo del valor de *queue_selector*:

  - TCIFLUSH: Vuelca datos recibidos pero no leídos.

  - TCOFLUSH: Vuelca datos escritos pero no transmitidos.

  - TCIOFLUSH: Vuelca tanto los datos recibidos pero no leídos, como los escritos pero no transmitidos.

- ***cfsetospeed***: Se utiliza para establecer la velocidad de salida.

  ```
  int cfsetospeed ( struct termios *termios_p, speed_t speed );
  ```

  Establece la velocidad de salida, guardada en la estructura termios apuntada por *termios_p*, a *speed*, que debe ser una de estas constantes: B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, B38400, B57600, B115200, B230400.
  Los nuevos valores no hacen efecto hasta que se llame con éxito a *tcsetattr()*.

- ***cfsetispeed***: Se utiliza para establecer la velocidad de entrada.

  ```
  int cfsetispeed ( struct termios *termios_p, speed_t speed );
  ```

  Establece la velocidad de entrada, guardada en la estructura termios apuntada por *termios_p*, a *speed*, que debe ser una de estas constantes: B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, B38400, B57600, B115200, B230400.
  Si la velocidad de entrada se pone a cero, entonces será igual a la de salida. Los nuevos valores no hacen efecto hasta que se llame con éxito a *tcsetattr()*.

En el Ejemplo 1.4 se puede ver un fragmento de código que muestra la utilización de alguna de las funciones comentadas.

```
//VARIABLES GLOBALES
char _puerto [15];
int _npuerto = -2;
int _velocidad = -2;
int _fd = -2;
struct termios _oldtio, _newtio;
…
…
int Baudios (int vel)
{
  speed_t baud;

  //comprobamos que existe un descriptor de fichero valido
  if (_fd < 0) return (-2);

  // Leer atributos del puerto
  if (tcgetattr(_fd,&_newtio)==-1) {
    return -1;
  }

  switch(vel) {
  case 300    : baud=B300;    _velocidad=vel; break;
  case 1200   : baud=B1200;   _velocidad=vel; break;
  case 2400   : baud=B2400;   _velocidad=vel; break;
  case 9600   : baud=B9600;   _velocidad=vel; break;
  case 19200  : baud=B19200;  _velocidad=vel; break;
  case 38400  : baud=B38400;  _velocidad=vel; break;
  case 57600  : baud=B57600;  _velocidad=vel; break;
  case 115200 : baud=B115200; _velocidad=115200; break;
  default     : return -1;
  }

  cfsetospeed(&_newtio,baud);
  cfsetispeed(&_newtio,baud);

  if (tcsetattr(_fd,TCSANOW,&_newtio)==-1)
    return -1;
  return _velocidad;
}
```

Ejemplo 1.4 Utilización de las funciones de configuración del puerto

1.3.2.1.- Opciones de control

El miembro *c_cflag* de la estructura *termios* permite controlar la velocidad del puerto, el número de bits de datos y de parada, la paridad y el control del flujo de datos por hardware. Entre todas las opciones de configuración (bTabla 1.3), existen dos que siempre se deben activar, CLOCAL y CREAD.

La velocidad de comunicación admite varias posibilidades de configuración, que dependen del sistema operativo. En implementaciones antiguas se emplea el miembro *c_cflag* mediante las constantes definidas en la Tabla 1.3. En las nuevas implementaciones, se han definido dos miembros nuevos, *c_ispeed* y *c_ospeed*. Como ya se ha visto, existen dos llamadas al sistema operativo que permiten llevar a cabo la configuración de la velocidad de comunicación, *cfsetospeed* y *cfsetispeed* (Ejemplo 1.4).

Constante	Descripción
CBAUD	Bit mask for baud rate
B0	0 baud (drop DTR)
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud
B57600	57,600 baud
B76800	76,800 baud
B115200	115,200 baud
EXTA	External rate clock
EXTB	External rate clock
CSIZE	Bit mask for data bits
CS5	5 data bits
CS6	6 data bits
CS7	7 data bits
CS8	8 data bits
CSTOPB	2 stop bits (1 otherwise)
CREAD	Enable receiver
PARENB	Enable parity bit
PARODD	Use odd parity instead of even
HUPCL	Hangup (drop DTR) on last close
CLOCAL	Local line – do not change "owner" of port
LOBLK	Block job control output
CNEW_RTSCTS CRTSCTS	Enable hardware flow control (not supported on all platforms)

Tabla 1.3 Constantes para el miembro *c_cflag*

1.3.2.2.- Opciones locales

El miembro para las opciones locales *c_lflag*, permite controlar la forma en que serán tratados los caracteres de entrada. En general, se configurará para entrada canónica o para no canónica. En la Tabla 1.4 se muestra las constantes para este miembro de la estructura *termios*.

Constante	Descripción
ISIG	Enable SIGINTR, SIGSUSP, SIGDSUSP, and SIGQUIT signals
ICANON	Enable canonical input (else raw)
XCASE	Map uppercase \lowercase (obsolete)
ECHO	Enable echoing of input characters
ECHOE	Echo erase character as BS−SP−BS
ECHOK	Echo NL after kill character
ECHONL	Echo NL
NOFLSH	Disable flushing of input buffers after interrupt or quit characters
IEXTEN	Enable extended functions
ECHOCTL	Echo control characters as ^char and delete as ~?
ECHOPRT	Echo erased character as character erased
ECHOKE	BS−SP−BS entire line on line kill
FLUSHO	Output being flushed
PENDIN	Retype pending input at next read or input char
TOSTOP	Send SIGTTOU for background output

Tabla 1.4 Constantes para el miembro *c_lflag*

1.3.2.3.- Opciones de entrada

El miembro *c_iflag* permite controlar el procesamiento de los caracteres de entrada, que son los que se reciben por el puerto. En la Tabla 1.5 se muestra las constantes para este miembro de la estructura *termios.*

Constante	Descripción
INPCK	Enable parity check
IGNPAR	Ignore parity errors
PARMRK	Mark parity errors
ISTRIP	Strip parity bits
IXON	Enable software flow control (outgoing)
IXOFF	Enable software flow control (incoming)
IXANY	Allow any character to start flow again
IGNBRK	Ignore break condition
BRKINT	Send a SIGINT when a break condition is detected
INLCR	Map NL to CR
IGNCR	Ignore CR
ICRNL	Map CR to NL
IUCLC	Map uppercase to lowercase
IMAXBEL	Echo BEL on input line too long

Tabla 1.5 Constantes para el miembro *c_iflag*

1.3.2.4.- Opciones de salida

El miembro c_oflag contiene las opciones para el procesamiento de la salida. En la Tabla 1.6 se muestra las constantes para este miembro de la estructura *termios.*

Constante	Descripción
OPOST	Postprocess output (not set = raw output)
OLCUC	Map lowercase to uppercase
ONLCR	Map NL to CR−NL
OCRNL	Map CR to NL
NOCR	No CR output at column 0
ONLRET	NL performs CR function
OFILL	Use fill characters for delay
OFDEL	Fill character is DEL
NLDLY	Mask for delay time needed between lines
NL0	No delay for NLs
NL1	Delay further output after newline for 100 milliseconds
CRDLY	Mask for delay time needed to return carriage to left column
CR0	No delay for CRs
CR1	Delay after CRs depending on current column position
CR2	Delay 100 milliseconds after sending CRs
CR3	Delay 150 milliseconds after sending CRs
TABDLY	Mask for delay time needed after TABs
TAB0	No delay for TABs
TAB1	Delay after TABs depending on current column position
TAB2	Delay 100 milliseconds after sending TABs
TAB3	Expand TAB characters to spaces
BSDLY	Mask for delay time needed after BSs
BS0	No delay for BSs
BS1	Delay 50 milliseconds after sending BSs
VTDLY	Mask for delay time needed after VTs
VT0	No delay for VTs
VT1	Delay 2 seconds after sending VTs
FFDLY	Mask for delay time needed after FFs
FF0	No delay for FFs
FF1	Delay 2 seconds after sending FFs

Tabla 1.6 Constantes para el miembro *c_oflag*

1.3.2.5.- Caracteres de control

El array de caracteres *c_cc* contiene los caracteres de control y los parámetros de configuración para los timeout (Tabla 1.7).

Los elementos VSTART y VSTOP contienen los caracteres utilizados para el control de flujo de datos por software. Normalmente se emplean DC1 (021 octal) y DC3 (023 octal) que representan los caracteres XON y XOFF en el estándar ASCII.

Para establecer los timeouts se utilizan los elementos VMIN y VTIME. Mediante VMIN se especifica el número mínimo de caracteres a leer. En función del valor de VMIN, VTIME tiene uno u otro significado:

- Si VMIN es cero, el valor de VTIME especifica el tiempo que se ha de esperar para todas las lecturas.
- Si VMIN es distinto de cero, el valor de VTIME especifica el tiempo que se ha de esperar hasta la llegada del primer carácter. Si un carácter llega antes de cumplirse el tiempo, la lectura se bloqueará hasta que lleguen el número de caracteres indicados en VMIN. Si no llega ningún carácter en el tiempo especifico, la llamada a la función de lectura retornará cero. Esta forma de utilizar los elementos VMIN y VTIME permite indicarle al driver que deseamos hacer lecturas de N bytes, de manera que un llamada a la función *read* retornará cero o los N bytes indicados en VMIN.

Constante	Descripción	Clave
VINTR	Interrupt	CTRL−C
VQUIT	Quit	CTRL−Z
VERASE	Erase	Backspace (BS)
VKILL	Kill−line	CTRL−U
VEOF	End−of−file	CTRL−D
VEOL	End−of−line	Carriage return (CR)
VEOL2	Second end−of−line	Line feed (LF)
VMIN	Minimum number of characters to read	−
VSTART	Start flow	CTRL−Q (XON)
VSTOP	Stop flow	CTRL−S (XOFF)
VTIME	Time to wait for data (tenths of seconds)	−

Tabla 1.7 Caracteres de control para el miembro *c_cc*

1.3.3.- Implementación realizada

Para acceder al puerto serie se han implementado un conjunto de funciones. En el archivo *ComPort.h* se puede encontrar la especificación de estas funciones y en el archivo *ComPort.c* la implementación.

Este código se ha desarrollado para poder comunicarse con los dispositivos FieldPoint y NuDAM. La implementación realizada es prácticamente común para ambos. Únicamente se diferencia en los parámetros de configuración del puerto serie:

Los parámetros de configuración más importantes a tener en cuenta para los módulos FieldPoint son:

- El formato de las tramas: 8 bits de datos, 1 bit de parada, sin paridad.
- El control del flujo de datos: control por hardware.

Los parámetros de configuración más importantes a tener en cuenta para los módulos NuDAM son:

- El formato de las tramas: 8 bits de datos, 1 bit de parada, sin paridad.
- El control del flujo de datos: **NO** existe control por hardware.

En el archivo ComPort.h se definen un conjunto de variables que se utilizan en la implementación:

- `char _puerto [15];`

  Nombre del fichero asociado al puerto, por ejemplo, "/dev/ttyS1".

- `int _npuerto = -2;`

  Número del puerto identificado por las macros COM1 y COM2. Estas macros están definidas en el archivo *ComPort.*h y deben ser utilizadas para llamar a la función *Abrir_Puerto_Serie ()* y para comprobar el valor devuelto por *Get_Puerto ()*.

- `int _velocidad = -2;`

  Velocidad de comunicación del puerto serie.

- `int _fd = -2;`

  Descriptor del fichero asociado al puerto una vez que se ha abierto. Para ejecutar gran parte de las funciones se comprueba previamente que exista un descriptor de fichero válido. Es por ello que se inicializa con un valor negativo.

- `struct termios _oldtio, _newtio;`

  Estructuras para almacenar la vieja y la nueva configuración del puerto respectivamente.

A continuación se ofrece una descripción detallada de cada una de las funciones, incluyendo la estructura interna de cada una de ellas.

1.3.3.1.- Abrir_Puerto_Serie

◇ **Descripción**

Esta función abre el puerto serie con la configuración adecuada.

La estructura de la función es la siguiente:

- Se abre el fichero asociado al puerto.
- Se obtiene la configuración actual del puerto. Este valor se utilizará para restaurar el puerto a su estado original cuando se cierre.
- Se configura el puerto mediante los distintos miembros de la estructura *termios*. La configuración ha de ser acorde con las características de los módulos. En este punto es donde se diferencian las dos implementaciones de acceso al puerto serie para los sistemas FieldPoint y NuDAM.
- Se activa la configuración del puerto realizada.
- Se establece la velocidad de comunicación.
- Si falla alguno de los pasos anteriores, se retornará un estado de error.

◇ **Sintaxis**

```
int Abrir_Puerto_Serie (int puerto, int velocidad);
```

◇ **Argumentos**

Puerto: Puerto serie del PC que se desea abrir. Se deben utilizar las macros COM1 y COM2 definidas.

Velocidad: Velocidad del puerto serie a la que se va a llevar a cabo la comunicación. Los valores permitidos son: 300, 1200, 2400, 9600, 19200, 28400, 57600, 115200. Si se indica un valor distinto a estos se producirá un error.

◇ **Valor retornado**

-1 en caso de error.

El valor del descriptor de fichero retornado por la función *open* si todo fue correcto.

1.3.3.2.- Cerrar_Puerto_Serie

◇ **Descripción**

Esta función cierra el puerto serie.

La estructura de la función es la siguiente:

- Se restaura la configuración original del puerto.
- Se cierra el fichero asociado al puerto.
- Si falla alguno de los pasos anteriores se retornará un estado de error.

◇ **Sintaxis**

```
int Cerrar_Puerto_Serie (void);
```

◇ **Valor retornado**

0 si todo fue correcto.

-1 en caso de error.

-2 si no existe un descriptor de fichero válido.

1.3.3.3.- Baudios

◇ **Descripción**

Esta función configura la velocidad de comunicación del puerto.

◇ **Sintaxis**

```
int Baudios (int vel);
```

◇ **Argumentos**

Vel: Velocidad del puerto serie a la que se va a llevar a cabo la comunicación. Los valores permitidos son: 300, 1200, 2400, 9600, 19200, 28400, 57600, 115200. Si se indica un valor distinto a estos se producirá un error.

◇ **Valor retornado**

La velocidad a la que se ha configurado el puerto si todo fue correcto.

-1 en caso de error.

-2 si no existe un descriptor de fichero válido.

1.3.3.4.- Enviar_Datos

◇ **Descripción**

Envía el comando que se le pasa como argumento al puerto serie.

◇ **Sintaxis**

```
int Enviar_Datos (const char * comando);
```

◇ **Argumentos**

Comando: Cadena que contiene la información a enviar a través del puerto.

◇ **Valor retornado**

El número de bytes enviados si todo fue correcto.

-1 en caso de error.

-2 si no existe un descriptor de fichero válido.

1.3.3.5.- Leer_Datos

◇ **Descripción**

Esta función lee *length* caracteres del puerto serie y los retorna en la cadena apuntada por *buffer*. Si no existen datos para leer en el puerto, esta función se quedará bloqueada.

◇ **Sintaxis**

```
int Leer_Datos(char * buffer, int length);
```

◇ **Argumentos**

Buffer: Puntero al buffer en donde se almacenarán la información leída.

Length: Numero de caracteres que se desean leer.

◇ **Valor retornado**

El número de bytes leídos si todo fue correcto.

-1 en caso de error.

-2 si no existe un descriptor de fichero válido.

1.3.3.6.- Leer_Datos_Plazo

◇ **Descripción**

Esta función lee *length* caracteres del puerto serie y los retorna en la cadena apuntada por *buffer*. Si en el momento de la llamada no existen caracteres a leer en el puerto, la función se quedará bloqueada durante *plazo* microsegundos. Transcurrido este tiempo, si no se ha llevado a cabo la lectura retornará un código de error.

◇ **Sintaxis**

```
int Leer_Datos_Plazo (char * buffer, int length, int plazo);
```

◇ **Argumentos**

Buffer: Puntero al buffer en donde se almacenará la información leída.

Length: Número de caracteres que se desean leer.

Plazo: El tiempo de plazo para leer la información (en microsegundos).

◇ **Valor retornado**

El número de bytes leídos si se leyó con éxito en el plazo indicado.

-3 si no se ha podido leer ningún byte en el plazo indicado.

-2 si no existe un descriptor de fichero válido.

1.3.3.7.- Get_fd

◇ **Descripción**

Esta función retorna el valor del descriptor de fichero asociado al puerto.

◇ **Sintaxis**

```
int Get_fd (void);
```

◇ **Valor retornado**

El descriptor de fichero asociado. Un valor negativo indica que el descriptor de fichero no es válido.

1.3.3.8.- Get_Velocidad

◇ **Descripción**

Esta función retorna la velocidad de comunicación del puerto.

◇ **Sintaxis**

```
int Get_Velocidad (void);
```

◇ **Valor retornado**

La velocidad de comunicación a la que está configurado el puerto. Un valor negativo indica una configuración errónea del puerto.

1.3.3.9.- Get_Puerto

◇ **Descripción**

Esta función retorna el número que identifica el puerto utilizado.

◇ **Sintaxis**

```
int Get_Puerto (void);
```

◇ **Valor retornado**

El puerto utilizado identificado por las macros COM1 y COM2 definidas. Un valor negativo indica que no se ha inicializado el puerto.

2.- FieldPoint, manual del programador

2.1.- Protocolo de comunicación

La comunicación a través del puerto serie, entre el PC y un dispositivo, se basa en el envío y recepción de una serie de tramas. Para poder llevar a cabo esta comunicación, es necesario conocer el protocolo que utiliza el fabricante. En este caso, National Instruments, mediante el documento *"FP-1000/FP-1001 Programmer Reference Manual"*, explica en detalle el protocolo que siguen los módulos FieldPoint.

En el epígrafe 2.3, se pueden consultar las tablas resumen de todos los comandos que soportan los módulos FieldPoint y los detalles de los comandos utilizados para implementar el driver.

2.1.1.- Formato de las tramas FieldPoint

Todas las tramas FieldPoint tienen la sintaxis mostrada en este epígrafe y contienen los cinco campos mostrados a continuación:

`[start] [address] [command] [checksum] [end]`

`[start]` Todas las tramas comienzan con el carácter ">" (ASCII 62).

`[address]` Dos caracteres ASCII en hexadecimal. Tras el campo `[start]`, todas las tramas deben especificar la dirección del módulo a quien va dirigido. Este campo ha de tener un valor entre 00 y F9 para redes construidas con los módulos de red FP-1000 y FP-1001.

`[command]` Este campo es el cuerpo de la trama. Se subdivide en cuatro campos, algunos de los cuales no se utilizan siempre.

- `[cmdchars]` Uno o dos caracteres ASCII en hexadecimal. Todas las tramas contienen este campo. Indica el código de la operación a llevar a cabo.
  Para los comandos estándar está formado por un único carácter (dentro del rango "A"-"Z" o "a"-"i"). Para los comandos

extendidos está formado por dos caracteres, siendo el primero el carácter "!" (ASCII 33) y el segundo el carácter que identifica la operación (dentro del rango "A"-"Z" o "a"-"z").

- `[position]` De cero a cuatro caracteres ASCII en hexadecimal. Algunas tramas no contienen este campo.
  Este campo especifica los canales del módulo que se verán afectados por el comando. Cada bit representa un canal, con un máximo de 16 canales por módulo FieldPoint. El bit 15 representa el canal 15 y el bit 0 representa el canal 0.

- `[modifier]` Cero o más caracteres ASCII en hexadecimal. Algunas tramas no contienen este campo. Este campo es utilizado por comandos que realizan acciones distintas en función de un valor.

- `[data]` Cero o más caracteres ASCII en hexadecimal. Algunas tramas no contienen este campo. Este campo contiene caracteres específicos del comando.

`[checksum]` Dos caracteres ASCII en hexadecimal. Este campo activa el control de paridad para la trama. Este valor se calcula de la siguiente manera: se suman los valores numéricos de los caracteres de los campos `[address]` y `[command]` y se haya el resto de dividir este valor por 256 (módulo-256 de la suma). Este resto se convierte a dos caracteres ASCII en hexadecimal para formar el campo.

Si no se desea controlar el error de paridad, este campo ha de contener dos interrogaciones, "??".

`[end]` Un carácter ASCII en hexadecimal. Este campo indica el final de la trama. Se puede utilizar un retorno de carro (ASCII 13) o un "." (ASCII 46).

2.1.2.- Respuestas de los módulos FieldPoint

2.1.2.1.- Respuestas de éxito

Si un comando se ejecuta correctamente, el módulo retornará una trama indicando esta situación. Los formatos más comunes de estas tramas son los siguientes:

- `A[cr]` Este tipo de respuesta se obtiene de los comandos que no solicitan información al módulo.

- `A[response data][checksum][cr]` Este tipo de respuesta, se obtiene de los comandos que han solicitado información al módulo. El campo `[checksum]` se obtiene sumando los valores numéricos de todos los caracteres del campo `[response data]` y calculando el módulo-256 de la suma. El módulo de la división se convierte a dos caracteres ASCII en hexadecimal.

2.1.2.2.- Respuestas de error

Un módulo retornará una respuesta de error, cuando detecte condiciones erróneas durante la recepción o ejecución del comando. La respuesta de error a un comando tiene la siguiente sintaxis:

`N[error number][cr]` donde `[error number]` son dos caracteres ASCII en hexadecimal.

En el epígrafe 2.3 se ofrece la descripción de los errores que pueden devolver los dispositivos FieldPoint.

2.2.- Implementación del driver

Los objetivos del presente proyecto planteaban la realización de un driver que permitiera utilizar los dispositivos bajo Linux y la realización de una interfaz en Ada para elaborar los programas de control en este lenguaje.

La implementación realizada se divide en dos partes:

- El driver, que permite utilizar los dispositivos en sistemas operativos Linux, junto con la interfaz en lenguaje C, para que el usuario pueda realizar sus programas de control en este lenguaje. Esta parte está programada en lenguaje C y consta de los siguientes ficheros:
    - El fichero *"DriverC_NI.h"*: Contiene la especificación de los tipos y funciones desarrolladas para manejar los módulos.

    - El fichero *"DriverC_NI.c"*: Contiene la implementación de las funciones desarrolladas para manejar los módulos.

- Una interfaz en lenguaje Ada, para que el usuario pueda realizar los programas de control en este lenguaje. Esta parte está programada en lenguaje Ada y se basa en la facilidad que proporciona el propio lenguaje para realizar llamadas a funciones programadas en otros lenguajes de programación (como por ejemplo C). Los ficheros involucrados son:
    - El fichero *"driverada_ni.ads"*: Contiene la especificación de los tipos y funciones desarrolladas para manejar los módulos.

    - El fichero *"driverada_ni.adb"*: Contiene la implementación de las funciones desarrolladas para manejar los módulos.

2.2.1.- Conceptos previos: el bus local

Los distintos módulos conectados para formar el sistema, se identifican en el bus mediante una dirección física. La dirección base se configura en el módulo de red mediante unos pequeños interruptores alojados en el frontal del módulo. El resto de módulos de E/S, tendrán como dirección física, la dirección base incrementada según su posición en el bus. El módulo de E/S contiguo al módulo de red tendrá la dirección física base+1, el siguiente base+2 y así sucesivamente.

La implementación del driver, se ha realizado para que el usuario no necesite conocer en ningún momento las direcciones físicas de los módulos conectados al bus. El dato que ha de manejar es la dirección del módulo en el ***bus local***. Este bus local, está formado únicamente por los módulos de E/S y se asignan las direcciones comenzando en cero a partir del primer módulo de E/S, el contiguo al módulo de red. De esta forma, el módulo contiguo al módulo de red tendrá la dirección cero, el siguiente la uno y así sucesivamente. En la Tabla 2.1 podemos ver un ejemplo.

	Módulo FP-1000	Módulo FP-DI-301	Módulo FP-RLY-420	Módulo FP-RLY-420
Dirección física	base	base + 1	base + 2	base + 3
Posición en el bus		0	1	2

Tabla 2.1 Ejemplo de asignación de direcciones en el bus local

La detección de los módulos que componen el bus local y la asignación de direcciones se hacen de forma dinámica. No es necesario que los módulos estén dispuestos físicamente siempre en el mismo orden. Podemos obtener tantas configuraciones hardware como posibilidades de conexión. Algunos ejemplos se muestran a continuación:

(FP-1000) – (FP-DI-301) – (FP-RLY-420) – (FP-RLY-420)

(FP-1000) – (FP-RLY-420) – (FP-DI-301)

2.2.2.- Código C

2.2.2.1.- Estructuras y variables utilizadas

En el código fuente, se definen una serie de macros, estructuras y variables que se utilizan para realizar la implementación del driver.

Variables globales

- **COM1** y **COM2**: Indican el puerto serie a utilizar para la comunicación.

- **_Digital_IN** y **_Digital_OUT**: Indican el tipo de módulo: entradas digitales y salidas digitales respectivamente.

- **Variables de error**: Proporcionan información sobre el estado en que finalizó la última operación. En la Tabla 2.2 se muestran las variables de error utilizadas, el valor que toman cada una de ellas y la descripción.

Macros internas

```
#define _FPDI301   0x0105
#define _FPRLY420  0x0108
```

Contienen los códigos del fabricante para identificar los distintos módulos. Este identificador es único para cada tipo de módulo.

Error	Valor	Descripción
CORRECTO	0	En la última operación no se produjo ningún error
ERROR_COM	-101	Error en la comunicación con el puerto serie
FD_INVALIDO	-102	Descriptor del puerto inválido
TIMEOUT	-103	Timeout al leer del puerto. El dispositivo no ha respondido en el tiempo esperado
RESPUESTA_ERRONEA	-200	El dispositivo ha devuelto una respuesta considerada errónea
ERROR_MEM	-300	No hay memoria dinámica reservada. Se puede producir por dos causas: • Si no se ha llamado a la función de inicialización antes de llamar a cualquier otra función • Si ha fallado la reserva dinámica de memoria durante la inicialización
NO_MODULO	-400	El número de módulo al que se quiere acceder no existe en el bus local
NO_DINPUT	-401	El módulo no es un módulo de entradas digitales
NO_DOUTPUT	-402	El módulo no es un módulo de salidas digitales
NO_LINEA	-500	La línea a la que se quiere acceder no existe

Tabla 2.2 Variables de error

Estructuras

- `typedef struct _Tdevice { … } DEVICE;`

  Esta estructura contiene los campos que identifican a cada uno de los módulos:

  - `int _direccion;`

    Contiene la dirección física del módulo, necesaria para enviar las tramas al bus.

  - `int _deviceType;`

    Identifica el tipo de módulo. Para ello se utilizan las macros `_Digital_IN` y `_Digital_OUT` definidas en el fichero de cabecera. Estas dos macros identifican los módulos de entradas y de salidas digitales respectivamente.

  - `int _deviceNumber;`

    Contiene la identificación del dispositivo. Este identificador es único y se corresponde con las macros `_FPDI301` y `_FPRLY420`.

  - `char _deviceName [15];`

    Contiene el nombre del dispositivo.

  - `float * _lineas;`

    Contiene el valor de cada uno de los canales que tiene el módulo. Esta variable es un array dinámico de flotantes. Una vez que se conoce el número de canales que forman el módulo se reserva la memoria necesaria.

  - `int _numLineas;`

    Contiene el número de canales de que dispone el módulo.

- `typedef struct _Tbus { … } BUS;`

  Esta estructura contiene los campos con las propiedades del bus:

  - `int _dirBase;`

    Contiene la dirección base del módulo de red.

  - `int _puerto;`

    Identifica el puerto serie utilizado. Se deben utilizar las variables COM1 y COM2 definidas en el fichero de cabecera.

  - `int _velocidad;`

    Contiene la velocidad de comunicación del puerto serie.

  - `int _numModulos;`

    Contiene el número de módulos que forman el bus.

  - `DEVICE * _bus;`

    Contiene la información de cada uno de los módulos que forman el bus. Esta variable es un array dinámico de estructuras DEVICE que, como vimos anteriormente, contienen la información de cada módulo. Se reserva dinámicamente la memoria tras conocer el número de módulos que forman el bus.

Variables internas

- `BUS DeviceBus = {0};`

  Esta variable contiene toda la información del bus.

- `int _nerror;`

  Contiene el número que identifica el estado en que finalizó la última función.

- `char _serror[255]`

  Cadena que contiene la descripción del estado en que finalizó la última función. Si la operación finalizó correctamente la cadena estará vacía. En caso contrario, contendrá una descripción del error producido. Esta cadena incluye una traza de las funciones en que ha sucedido el error.

- `int estado = 0;`

  Indica si en algún momento se ha producido un error al asignar memoria dinámica. Se inicializa a cero y en el momento de asignar memoria dinámica se actualiza a uno si se hizo correctamente la asignación. Si no se asignó memoria dinámica correctamente, no se permitirá realizar ninguna operación. Por lo tanto, en toda función se debe comprobar, antes de nada, el valor de esta variable y, si fuera cero, retornar error.

2.2.2.2.- Funciones locales

Existen un conjunto de funciones que se utilizan internamente para el correcto funcionamiento del driver.

expon

◇ **Sintaxis**

```
int expon (int base, int exp);
```

◇ **Descripción**

Realiza la operación *base* elevado a *exp* y retorna el resultado. Aunque existe la función *pow* en C que realiza la misma operación, se decidió implementar aquí esta función para evitar problemas de librerías en la implementación en Ada.

Reset_bus

◇ **Sintaxis**

```
int Reset_bus (void);
```

◇ **Descripción**

Esta función inicializa el módulo de red. Para ello se siguen los siguientes pasos:

- Se envía el comando *Reset Module (!Z)* al módulo. Tras enviar este comando los módulos pasan a estar un tiempo inaccesibles. Es por ello, que se ha de esperar un tiempo prudencial para continuar.
- Se envía el comando *Power Up Clear (A)* al módulo.
- Se desactiva el watchdog. Para ello se utiliza el comando *Set Watchdog Delay (!Q)*.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`) y se retornará -1.

Reset_modulo

◇ Sintaxis

```
int Reset_modulo (int modulo);
```

◇ Descripción

Esta función inicializa el módulo que se le indica como argumento. Para ello se envía el comando *Power Up Clear (A)* al módulo indicado. Si se produce algún fallo durante la ejecución se actualizarán convenientemente las variables de error (`_nerror` y `_serror`) y se retornará -1.

BuscaModulos

◇ Sintaxis

```
int BuscaModulos (void);
```

◇ Descripción

Esta función lee todos los dispositivos que están conectados al bus e inicializa correctamente todas las estructuras de datos y variables utilizadas. Para ello sigue los siguientes pasos:

- Se inicializa el bus mediante la función *Reset_bus*.
- Se hace una lectura de los módulos que forman el bus.
- Si la lectura anterior tuvo éxito, se reserva memoria para el vector que contendrá los módulos que forman el bus (`DeviceBus._bus`).
- Se continúa la inicialización de cada una de las variables que identifican los módulos. Actualmente sólo se soportan los módulos FP-DI-301 y FP-RLY-420.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`) y se retornará -1.

2.2.2.3.- Funciones globales

En este epígrafe únicamente se describe la estructura interna de aquellas funciones que por su importancia o complejidad así lo requieran. La descripción detallada de todas las funciones que forman la librería se proporciona en el epígrafe 2.2.4.

NI_Inicializar

- Se inicializan los componentes `_dirBase`, `_puerto` y `_velocidad` de la variable `DeviceBus` con lo parámetros de la función.
- Se abre el puerto serie.
- Se buscan los módulos en el bus mediante la función *BuscaModulos*.
- Se resetean cada uno de los módulos de E/S que forman el bus local.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

NI_Liberar

- Se cierra el puerto serie.
- Se libera la memoria asignada dinámicamente durante la inicialización.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

NI_LeeModulo_DI

- Se hacen las comprobaciones previas a la ejecución de la función.
- Se envía al módulo el comando *Read Discrete with Status (!K)* para leer el estado de todos los canales.
- Se actualizan los valores locales de los canales del módulo.
- Se retorna codificado en un entero el estado de todas las líneas.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

NI_EscribeModulo_DO

- Se hacen las comprobaciones previas a la ejecución de la función.
- Se envía al módulo el comando *Write Discrete with Status (!M)* para escribir el estado de todos los canales.
- Se actualizan los valores locales de los canales del módulo.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

NI_LeeLinea_DI

- Se hacen las comprobaciones previas a la ejecución de la función.
- Se llama a la función *NI_LeeModulo_DI*.
- Se retorna el valor del canal solicitado.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

NI_EscribeLinea_DO

- Se hacen las comprobaciones previas a la ejecución de la función.
- Se actualiza el valor del canal en las variables locales.
- Se codifica el valor de los canales del módulo para enviar el comando *Write Discrete with Status (!M)*.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

2.2.3.- Código Ada

El lenguaje Ada proporciona la facilidad de reutilizar código escrito en otro lenguaje. El paquete *Interfaces.C* contiene los tipos básicos, constantes y subprogramas que permiten a un programa en Ada pasar parámetros y cadenas a las funciones escritas en C.

Para llevar a cabo la utilización de las funciones escritas en C se hace uso del *pragma import*:

- Primero se declara la función como si se fuera a implementar en lenguaje Ada.
- Luego se utiliza *pragma import* para utilizar la función escrita en C en nuestra implementación en Ada. Este pragma necesita tres parámetros:
    - El lenguaje en que está implementada la función a importar.
    - El nombre que tendrá la función en nuestra implementación en Ada.
    - El nombre de la función en lenguaje C.

En el Ejemplo 2.1 se muestra la utilización de este pragma. La llamada a la función *NI_EscribeLinea_DO* en la interfaz en Ada, hace uso de la función definida en lenguaje C con el mismo nombre.

```
function NI_EscribeLinea_DO (Modulo: Integer; Linea: Num_Linea_Salida;
                             Valor: U1) return Integer is
   function NI_Ada_EscribeLinea_DO (Modulo: Integer; Linea: Integer; Valor:
                                    Integer) return Integer;
   pragma Import (C, NI_Ada_EscribeLinea_DO, "NI_EscribeLinea_DO");
begin
   return NI_Ada_EscribeLinea_DO (Modulo, Linea, Valor);
end NI_EscribeLinea_DO;
```

Ejemplo 2.1 Interfaz C-Ada

La definición de las variables de acceso al puerto serie, variables de error y de identificación del tipo de módulo, se definen como valores enumerados. También se definen nuevos tipos, derivados del tipo *Integer* y con el rango de valores limitado. En las funciones que utilicen estos tipos, si se realiza una llamada con un valor fuera del rango, se producirá una excepción en tiempo de ejecución.

2.2.4.- Descripción de las funciones

El argumento *modulo*, utilizado en varias funciones, hace referencia a la posición en el bus local del módulo al que se desea acceder.

Si la ejecución de una función da lugar a un error, se devuelve un valor negativo y se actualizan las variables internas de error. Se recomienda utilizar las funciones *ObtenerError* y *ObtenerErrorString* para obtener el código de error y una descripción de éste respectivamente.

NI_Inicializar

◇ **Descripción**

Esta función sirve para inicializar el bus de comunicaciones de los módulos FieldPoint. Se debe invocar esta función antes de llamar a cualquier otra.

◇ **Sintaxis**

C/C++

```
int NI_Inicializar (int puerto,
                    int velocidad,
                    int dir_modulo);
```

Ada

```
function NI_Inicializar (Puerto: Puerto_Serie;
                         Velocidad: Integer;
                         Dir_Modulo: Integer)
return Integer;
```

◇ **Argumentos**

Puerto: Puerto serie del PC que se va ha emplear para la comunicación. Se deben utilizar las variables COM1 y COM2 definidas.

Velocidad: Velocidad del puerto serie a la que se va a llevar a cabo la comunicación. Este valor ha de coincidir con el valor fijado en el módulo de red. Los valores permitidos son: 300, 1200, 2400, 9600, 19200, 28400, 57600, 115200. Si se indica un valor distinto a estos se producirá un error.

Dir_Modulo: Dirección física que se ha asignado al módulo de red.

◇ **Valor retornado**

0 si todo fue correcto.

-1 en caso de error.

NI_Liberar

◇ **Descripción**

Esta función libera el bus de comunicaciones y los recursos comprometidos durante la inicialización.

◇ **Sintaxis**

C/C++

```
int NI_Liberar (void);
```

Ada

```
function NI_Liberar return Integer;
```

◇ **Valor retornado**

0 si todo fue correcto.

-1 en caso de error.

NI_EscribeLinea_DO

◇ **Descripción**

Esta función actualiza el valor de un canal de salida de un módulo de salidas digitales FP-RLY-420.

◇ **Sintaxis**

C/C++

```
int NI_EscribeLinea_DO (const int modulo,
                        const int linea,
                        const int valor);
```

Ada

```
function NI_EscribeLinea_DO (Modulo: Integer;
                             Linea: Num_Linea_Salida;
                             Valor: U1) return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

Linea: Número del canal de salida que se desea actualizar (entre 0 y 7).

Valor: Valor de la línea a actualizar (1: activar; 0: desactivar).

◇ **Valor retornado**

0 si todo fue correcto.

-1 en caso de error.

NI_EscribeModulo_DO

◇ **Descripción**

Esta función actualiza los ocho canales del módulo de salidas digitales FP-RLY-420.

◇ **Sintaxis**

C/C++

```
int NI_EscribeModulo_DO (const int modulo,
                         const U8 valor);
```

Ada

```
function NI_EscribeModulo_DO (Modulo: Integer;
                              Valor: U8)
return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

Valor: Entero donde estén codificados los valores de los ocho canales del módulo.

◇ **Valor retornado**

El número de canales del módulo si todo fue correcto.

-1 en caso de error.

NI_LeeLinea_DI

◇ **Descripción**

Esta función lee el estado de un canal de entrada de un módulo de entradas digitales FP-DI-301.

◇ **Sintaxis**

C/C++

```
int NI_LeeLinea_DI (const int modulo,
                    const int linea);
```

Ada

```
function NI_LeeLinea_DI (Modulo: Integer;
                         Linea: Num_Linea_Entrada)
return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

Linea: Número del canal de entrada que se desea leer (entre 0 y 15).

◇ **Valor retornado**

El estado del canal de entrada (1: activo; 0: inactivo) si todo fue correcto.

Un número negativo si hubo algún error.

NI_LeeModulo_DI

◇ **Descripción**

Esta función lee el estado de los dieciséis canales de entrada del módulo de entradas digitales FP-DI-301. El estado de las entradas se devuelve codificado en un entero que se ha de pasar a la función como argumento.

◇ **Sintaxis**

C/C++

```
int NI_LeeModulo_DI (int modulo,
                     U16 * valor);
```

Ada

```
procedure NI_LeeModulo_DI (Modulo: in Integer;
                           Valor: in out U16;
                           Retorno: out Integer);
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

Valor: Entero donde se codificará el estado de los dieciséis canales de entrada del módulo.

◇ **Valor retornado**

El número de líneas del módulo si todo fue correcto.

-1 en caso de error.

La implementación de esta función en Ada se hace a través de un procedimiento. El valor de retorno se devuelve a través de la variable *Retorno.*

ObtenerError

◇ **Descripción**

Esta función sirve para obtener el estado en que finalizó la última operación que se ha realizado.

◇ **Sintaxis**

C/C++

```
int ObtenerError (void);
```

Ada

```
function ObtenerError return Var_Error;
```

◇ **Valor retornado**

C/C++

Un entero que identifica la situación en que finalizó la última operación. Los posibles valores están definidos en las variables de error.

Ada

Una variable de tipo *Var_Error* (valor enumerado donde se definen los errores).

ObtenerErrorString

◇ **Descripción**

Esta función sirve para obtener una descripción del estado en que finalizó la última operación realizada. Si en la última llamada a una función se ha producido un error, mediante una llamada a esta función se obtendrá una descripción de éste.

◇ **Sintaxis**

C/C++

```
char * ObtenerErrorString (void);
```

Ada

```
function ObtenerErrorString return string;
```

◇ **Valor retornado**

Una cadena con la descripción del estado en que finalizó la última operación realizada. Si en la última operación no se produjo ningún error se retornará una cadena vacía.

Num_Lineas

◇ **Descripción**

Esta función proporciona información sobre el número de canales que forman un módulo.

◇ **Sintaxis**

C/C++

```
int Num_Lineas (int modulo);
```

Ada

```
function Num_Lineas (Modulo: Integer) return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

◇ **Valor retornado**

El número de canales del módulo si todo fue correcto.

-1 en caso de error.

Num_Modulos

◇ **Descripción**

Esta función proporciona información sobre el número de módulos que se han detectado en el bus local.

◇ **Sintaxis**

C/C++

```
int Num_Modulos (void);
```

Ada

```
function Num_Modulos return Integer;
```

◇ **Valor retornado**

El número de módulos que forman el bus local si todo fue correcto.

-1 en caso de error.

Get_Tipo

◇ **Descripción**

Esta función retorna información sobre el tipo del módulo (entradas digitales, salidas digitales,…).

◇ **Sintaxis**

C/C++

```
int Get_Tipo (int modulo);
```

Ada

```
function Get_Tipo (Modulo: Integer) return Device_Type;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

◇ **Valor retornado**

C/C++

El número que identifica el tipo de módulo (variables *_Digital_IN* y *_Digital_OUT*).

-1 en caso de error.

Ada

Una variable de tipo *Device_Type* (valor enumerado donde están definidos los tipos de dispositivos). En caso de error el valor del enumerado retornado será *ERROR.*

Get_Nombre

◇ **Descripción**

Esta función permite obtener el nombre de un módulo.

◇ **Sintaxis**

C/C++

```
char * Get_Nombre (int modulo);
```

Ada

```
function Get_Nombre (Modulo: Integer) return String;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

◇ **Valor retornado**

Una cadena con el nombre del módulo si todo fue correcto.

Una cadena con la descripción del error sucedido en caso de error.

Get_Id

◇ **Descripción**

Esta función permite obtener el identificador de un módulo. Este identificador es único para cada tipo de módulo y se corresponde con el valor asignado por el fabricante.

◇ **Sintaxis**

C/C++

```
int Get_Id (int modulo);
```

Ada

```
function Get_Id (Modulo: Integer) return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

◇ **Valor retornado**

El número del fabricante que identifica el módulo si todo fue correcto.

-1 en caso de error.

Get_Direccion

◇ **Descripción**

Esta función permite obtener la dirección física que tiene asignada un módulo.

◇ **Sintaxis**

C/C++

```
int Get_Direccion (int modulo);
```

Ada

```
function Get_Direccion (Modulo: Integer) return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

◇ **Valor retornado**

La dirección física que tiene asignada el módulo si todo fue correcto.

-1 en caso de error.

2.3.- Anexo A: FieldPoint Commands

En este anexo se proporciona información sobre los comandos que utilizan los módulos FieldPoint. En las tablas se muestran todos los comandos que soportan los módulos. Únicamente se incluye la descripción de aquellos comandos que se han utilizado para implementar el driver. Esta información se ha obtenido directamente de los capítulos 3, 4 y 5 del manual de usuario "*FP-1000/1001 Programmer Reference Manual*".

Command Type and Name	Command Syntax	Success Response
Utility Commands:		
Power Up Clear	A	A
Reset	B	A
Configuration Commands:		
Set Turn-around Delay	C	A
Identify Optomux Type	F	A[type]
Configure Positions	G[positions]	A
Configure As Inputs	H[positions]	A
Configure As Outputs	I[positions]	A
Read Module Configuration	j	A[config]
Watchdog Commands:		
Set Analog Watchdog Delay	D[positions][wdgTmo]	A
Set Analog Watchdog Data	m[positions][data]	A
Enhanced Discrete Watchdog	m[positions][wdgTmo]	A
Discrete Commands:		
Write Outputs	J[positions]	A
Activate Outputs	K[positions]	A
Deactivate Outputs	L[positions]	A
Read ON/OFF Status	M	A[data]
Analog Commands:		
Write Analog Outputs	J[positions][data]	A
Read Analog Outputs	K[positions]	A[data]
Read Analog Inputs	L[positions]	A[data]
Update Analog Outputs	S[positions][data]	A

Tabla 2.3 Standard FieldPoint Commands and Syntax

Command Type and Name	Command Syntax	Success Response
Utility Commands:		
Read Module ID	!A	A[modID]
Read All Module IDs	!B	A[number][modarray]
HotSwap Reporting Mode	!b[flag]	A
Resend Last Response	!c	A
Read Firmware Revision	!e	
Read Module Status	!N	A[mod_status]
Read Channel Status	!O[positions]	A[chnl_status]
Read Bank Status	!P	A[bank_status]
Reset Module	!Z	A
Execute Channel Command	!n[positions]{[command type][command] pairs}	A
Configuration Commands:		
Set Attributes	!D[positions]{[attrMask] [rangeMask][settings] triplets}	A
Get Attributes	!E[positions]{[attrMask] [rangeMask] pairs}	A{[settings] array}
Discrete Commands:		
Read Discrete	!J	A[data]
Read Discrete with Status	!K	A[status][data]
Write Discrete	!L[positions][data]	A
Write Discrete with Status	!M[positions][data]	A[status]
Analog Commands:		
Read 16-bit Data	!F[positions]	A[data]
Read 16-bit Data with Status	!G[positions]	A[status][data]
Write 16-bit Data	!H[positions][data]	A[positions][data]
Write 16-bit Data with Status	!I[positions][data]	A[status]

Tabla 2.4 Extended FieldPoint Commands and Syntax

Command Type and Name	Command Syntax	Success Response
Watchdog Commands:		
Set Watchdog Delay	!Q[wdgTmo]	A[wdgTmo]
Set Discrete Watchdog Data	!R[positions][wdgData]	A
Set 16-bit Watchdog Data	!S[positions][wdgData]	A
Set Watchdog Data Status	!T[positions][mask]	A
Get Watchdog Info	!U	A[moduleInfo] [chnlEnable] [chnlWdgData]
SnapShot/Programmable Power-Up Commands:		
Store Attributes	!f[positions]{[attrMask] [rangeMask][settings] triplets}	A
Store 16-bit Data	!g[positions][data]	A
Store Discrete	!h[positions][data]	A
Store Discrete Watchdog Data	!i[positions][wdgData]	A
Store 16-bit Watchdog Data	!j[positions][wdgData]	A
Store Watchdog Data Status	!k[positions][mask]	A
Store Watchdog Enable	!l[wdgTmo]	A
Store Watchdog Delay	!V[wdgTmo]	A
Store SnapShot	!W	A
Use SnapShot	!X[flag]	A[flag]
Read SnapShot Status	!Y	A[status]

Tabla 2.4 Extended FieldPoint Commands and Syntax (Continued)

Comandos utilizados

Power Up Clear [cmdchars] = A

Description

Power Up Clear should be the first command issued to a FieldPoint module. This command prevents the FieldPoint module from returning a power-up clear expected error (E_PONCLR_EXP) message in response to the first command following application of power.

This command functions only if it is the first command sent after power-up. If this command is sent to a module after the first command has been sent, the module responds with a success response. A power-up clear expected error is returned if any other command is sent first. After a power-up clear expected error is returned this command does not need to be sent; the next command executes normally.

This command has no effect on the FieldPoint module operation or setup; the power-up clear expected error provides an indication to the host that there has been a power failure and that the FieldPoint module has been reset to power-up configuration.

Syntax

```
A
```

Success Response

```
A
```

Example

```
>33A??[cr]
```

This example sends Power Up Clear to the FieldPoint module at address 51 (0x33). “??” is the [checksum] field.

Read All Module IDs [cmdchars] = !B

Description

The Read All Module IDs command reads the number of modules in the bank, and the module IDs for the network module and all the modules in the bank. This command should be sent to a network module only. Use the *Read Module ID command* (!A) to read the module ID for a single module.

Syntax

`!B`

Success Response

`A[number][mod array]`

`[number]` Two ASCII-hex characters representing the number of modules in the bank, including the network module.

`[mod array]` Four ASCII-hex characters per module in the bank representing the module ID. The first entry (four characters) is the ID of the network module that was addressed. Each subsequent entry corresponds to the module at the next higher address in the bank.

Each module ID is four ASCII-hex characters. It is a 16-bit number representing the module ID as shown in the following table.

[modID] Value	Module Name
0001	FP-1000
0002	FP-1001
0101	FP-AI-110
0102	FP-AO-200
0103	FP-DI-330
0104	FP-DO-400
0105	FP-DI-301
0106	FP-DO-401
0107	FP-TC-120
0108	FP-RLY-420
0109	FP-DI-300
010A	FP-AI-100
010B	FP-RTD-122
010C	FP-AI-111
010D	FP-CTR-500

[modID] Value	Module Name
010E	FP-PWM-520
010F	FP-AO-210
0110	FP-DO-410
0111	FP-DO-403
FFFF	Empty Base

Example

```
>00!B??[cr]
```

This command requests the FieldPoint network module at address 0x00 to return the IDs of all the modules in its bank.

```
A03000101020103[cs][cr]
```

This response from the network module indicates that there are three modules in the bank with module IDs 0x0001(FP-1000 module), 0x0102 (FP-AO-200 module), and 0x0103 (FP-DI-330 module).

Read Discrete with Status [cmdchars] = !K

Description

The Read Discrete with Status command reads discrete data from all discrete input and output channels of the addressed module. In addition, the status of the targeted channels is reported, to enable additional error checking.

Use the *Read Module Status* or *Read Channel Status* commands for details on the error condition of a bad channel.

Syntax

```
!K
```

Success Response

```
A[status][data]
```

`[status]` Four ASCII-hex characters, specifying the status of the channels targeted by this command. The most significant bit represents channel 15, the least significant bit represents channel 0. A "1" in any bit means that the corresponding channel's status is bad. A zero in any bit means that the corresponding channel's status is good.

Channels that do not exist return a zero in the corresponding status bit.

Channels that are not discrete return a zero in the corresponding status bit.

`[data]` Four ASCII-hex characters, specifying the channel levels. A "1" in any bit means that the corresponding channel is ON. A zero in any bit means that the corresponding channel is OFF.

Channels that do not exist return a zero in the corresponding data bit.

Channels that are not discrete return a zero in the corresponding data bit.

Note: Even if a channel has a bad status, a value is returned for that channel in `[data]`.

Example

```
>33!K??[cr]
```

This command tells the FieldPoint module at address 51 (0x33) to return the status of all its channels.

```
A000000FF[cs][cr]
```

This response from a 16-channel discrete FieldPoint module indicates that all channels have a good status, channels 15 through 8 are OFF, and channels 7 through 0 are ON.

Write Discrete with Status [cmdchars] = !M

Description

The Write Discrete with Status command drives outputs ON or OFF on targeted channels of the addressed FieldPoint module. In addition, the status of the targeted channels is reported, to enable additional error checking. Use the !O command for more information on a channel's status.
Use the *Read Module Status* or *Read Channel Status* commands for details on the error condition of a bad channel.

Syntax

`!M[positions][data]`

`[positions]` Four ASCII-hex characters, specifying the channels targeted by this command. A "1" in any bit means that the corresponding channel's data is to be written. A zero in any bit means that the corresponding channel is not targeted by this command.

Targeting channels that are inputs returns an error.

Targeting channels that do not exist returns an error.

`[data]` Four ASCII-hex characters, specifying the channel levels. A "1" in any bit means that the corresponding channel is to be driven ON. A zero in any bit means that the corresponding channel is to be driven OFF.

Success Response

`A[status]`

`[status]` Four ASCII-hex characters, specifying the status of the channels targeted by this command. The most significant bit represents channel 15, the least significant bit represents channel 0. A "1" in any bit means that the corresponding channel's status is bad. A zero in any bit means that the corresponding channel's status is good.

Channels that are not targeted return a zero in the corresponding status bit.

Example

`>33!M00010000??[cr]`

This command tells the FieldPoint module at address 51 (0x33) to turn channel 0 to OFF. This command also tells the module to return the channel status.

`A0000[cs][cr]`

This response from the FieldPoint module indicates that the channel status is good.

Set Watchdog Delay [cmdchars] = !Q

Description

If a network module is addressed, the Set Watchdog Delay command sets up the timeout value for the FieldPoint bank's watchdog timer. If a module other than a network module is addressed, the addressed module is enabled or disabled for watchdog timer expiration, but the bank's watchdog timeout value is not affected. You should set the watchdog data before you issue this command.

Syntax

`!Q[wdgTmo]`

`[wdgTmo]` Four ASCII-hex characters, specifying the watchdog timeout value. The result of this command depends on the type of module that is addressed.

Success Response

`A`

Command Sent to Network Module

The watchdog timeout value is set equal to 10 times the number of milliseconds specified in this field. In addition, the watchdog timer is started. Timeout values of less than 200 ms (0 < `[wdgTmo]` < 20) result in an error. A `[wdgTmo]` of 0 disables the watchdog timer function for the bank.

Command Sent to I/O Module

The watchdog timeout value for the bank is not affected, and the running/stopped state of the watchdog timer is not altered. Non-zero delays of less than 200 ms (`[wdgTmo]` < 20) result in an error.

A `[wdgTmo]`≥ 20 enables the I/O module for watchdog timer expiration. A `[wdgTmo]` of 0 disables the addressed I/O module from being affected by a watchdog timer expiration.

Note: The channel watchdog setups are not altered. Therefore, if you desire to re-enable the I/O module for watchdog timer expiration, you need to re-send this command to the I/O module with a valid `[wdgTmo]` value. All the channel setup you performed for setting watchdog data and enabling channels to source that pre-specified is re-enabled automatically.

Example

`>00!Q0015??[cr]`

This command tells the FieldPoint module at address 0x00 to set its watchdog timeout value to 210 ms, and to start the watchdog timer.

`>33!Q0021??[cr]`

This command tells the FieldPoint module at address 51 (0x33) to enable itself to react to watchdog timer expiration.

Reset Module [cmdchars] = !Z

Description

If a network module is addressed, the Reset Module command resets all the I/O modules in the bank and resets the turn-around delay for all modules to zero. If a module other than a network module is addressed, the addressed module is reconfigured and the turn-around delay for the addressed module is set to zero. In both cases, the FieldPoint modules are configured to factory default settings (if the SnapShot feature is disabled) or to stored SnapShot information (if the SnapShot feature is enabled).

Syntax

```
!Z
```

Success Response

```
A
```

Example

```
>33!Z??[cr]
```

This command tells the FieldPoint module at address 51 (0x33) to return to its power-up state.

2.4.- Anexo B: FieldPoint Responses

En este anexo se proporciona la descripción de los errores que pueden devolver los dispositivos. Esta información se ha obtenido directamente del capítulo 2 "*FieldPoint Responses*" del manual de usuario "*FP-1000/1001 Programmer Reference Manual*".

A FieldPoint module returns an error response when an erroneous condition is detected during the reception or execution of a command. FieldPoint modules return only the Standard errors (N00 through N07) in response to all standard commands, which enables FieldPoint modules to work with host software that is written for the Optomux protocol. In response to the extended commands, FieldPoint modules return either standard or extended errors, depending on which is most appropriate. The error response to a FieldPoint command (standard or extended) has the following form:
`N[error number][cr]` where [error number] is two ASCII-hex characters.

Error Number (Hex)	Error Tag	Description
00	E_PUCLR_EXP	**Power Up Clear** expected. The command was ignored. A command other than **Power Up Clear** was attempted after power-up or power failure. Once the error is received by the host, it is unnecessary to send the **Power Up Clear** command, because the next command is executed normally. If this error message is received, the FieldPoint network module has gone through its power-up sequence and has reset all characteristics to defaults. If SnapShot is enabled, the module is configured in accordance with the stored SnapShot information.
01	E_INVALID_CMD	Undefined command. The command character was not a legal command character, or the addressed module does not support this command. The command was ignored.
02	E_BAD_CHECKSUM	Checksum error. The checksum received in the command did not match the calculated checksum of the characters in the command. The command was ignored.
03	E_INBUF_OVRFLO	Input buffer overrun. The received command was too long. The command was ignored.
04	E_ILLEGAL_CHAR	Non-printable ASCII character received. Only characters from ASCII value 33 to 127 are permitted within commands. The command was ignored.
05	E_INSUFF_CHARS	Data field error. An insufficient or incorrect number of characters were received for the specified command.
06	E_WATCHDOG_TMO	Communications link network watchdog timed out. The command was ignored.
07	E_INV_LIMS_GOT	Specified limits invalid for the command. This incluyes notification that an invalid digit (hex or decimal) was received.

Tabla 2.5 Standard Error Responses

Error Number (Hex)	Error Tag	Description
80	E_ILLEGAL_DIGIT	One or more characters sent in the command could not be correctly converted to a digit (hex or decimal).
81	E_BAD_ADDRESS	The command is valid, but the addressed module does not support the command received.
82	E_INBUF_FRMERR	The FieldPoint network module detected a serial framing error in the command. The command was ignored.
83	E_NO_MODULE	The addressed module does not exist.
84	E_INV_CHNL	One or more channels specified in the command either do not exist or do not support the operation specified. The command was ignored.
85	E_INV_RANGE	One or more ranges specified in the command either do not exist or do not support the setting specified. The command was ignored.
86	E_INV_ATTR	One or more attributes specified in the command either do not exist or do not support the setting specified. The command was ignored.
88	E_HOTSWAP	The module has been hot-swapped since it was last sent a command. This response is sent only if the network module is enabled to report hot-swaps, and if the hot-swap occurred after hot-swap reporting mode was enabled. The command was ignored. This error number can be sent in response to a standard command if you have enabled hot-swap reporting for the bank.
89	E_ADDR_NOT_SAME	The module addressed by the Resend Last Response is not the same as the module addressed by the previous command.
8A	E_NO_RESEND_BUF	The response to the last command is unavailable
8B	E_HW_FAILURE	An irrecoverable fault has occurred.
8C	E_UNKNOWN	An unidentifiable error condition has occurred.

Tabla 2.6 Extended Error Responses

3.- NuDAM, manual del programador

3.1.- Protocolo de comunicación

La comunicación a través del puerto serie, entre el PC y un dispositivo, se basa en el envío y recepción de una serie de tramas. Para poder llevar a cabo esta comunicación, es necesario conocer el protocolo que utiliza el fabricante. El protocolo utilizado por lo módulos de E/S digitales de la empresa NuDAM se puede obtener de los manuales de usuario de estos módulos (capítulo 3, "*Command Set*").

En el epígrafe 3.3, se puede consultar la tabla resumen de todos los comandos que soportan los módulos de E/S digitales NuDAM y los detalles de los comandos utilizados para implementar el driver.

3.1.1.- Formato de las tramas

Todas las tramas NuDAM tienen la sintaxis mostrada en este epígrafe y contienen los cinco campos mostrados a continuación:

`(Leading Code)(Addr)(Command)[Data][Checksum]<CR>`

`(Leading Code)` Es el primer carácter de todas las tramas (%,$,#,@,…).

`(Addr)` Dos caracteres ASCII en hexadecimal. Dirección del módulo a quien va dirigida la trama. El rango de valores posibles está comprendido entre 00 y FF.

`(Command)` Longitud variable. Indica el comando a ejecutar.

`[Data]` Longitud variable. Es un campo opcional. Utilizado por algunos comandos para indicar valores de datos.

`[Checksum]` Dos caracteres ASCII en hexadecimal. Este campo es opcional. Este campo activa el control de paridad para la trama. Este valor se calcula según la siguiente fórmula:

`[Checksum] = ((LeadingCode)+(Addr)+(Command)+[Data]) MOD 0x100`

<CR> Un carácter ASCII en hexadecimal. Indica el final de la trama. Para finalizar las tramas se ha de utilizar el carácter de retorno de carro (CR: ASCII 13).

3.1.2.- Respuestas de los módulos NuDAM

Los mensajes de respuesta dependen del comando enviado. La respuesta puede incluir varios campos de información. Las posibles respuestas a un comando se diferencian por el primer carácter (campo *Leading Code*):

- Si se responde al comando con "!" o ">" el comando enviado es correcto.
- Si se responde al comando con "?" el comando enviado es inválido.

Existen condiciones en las que un comando no producirá un mensaje de respuesta:

- La dirección especificada no existe.
- La sintaxis del comando es incorrecta.
- Se produjo un error en la comunicación.
- Algunos comandos especiales no producen mensajes de respuesta.

3.2.- Implementación del driver

Los objetivos del presente proyecto planteaban la realización de un driver que permitiera utilizar los dispositivos bajo Linux y la realización de una interfaz en Ada para elaborar los programas de control en este lenguaje.

La implementación realizada se divide en dos partes:

- El driver, que permite utilizar los dispositivos en sistemas operativos Linux, junto con la interfaz en lenguaje C, para que el usuario pueda realizar sus programas de control en este lenguaje. Esta parte está programada en lenguaje C y consta de los siguientes ficheros:
  - El fichero *"DriverC_ND.h"*: Contiene la especificación de los tipos y funciones desarrolladas para manejar los módulos.

  - El fichero *"DriverC_ND.c"*: Contiene la implementación de las funciones desarrolladas para manejar los módulos.

- Una interfaz en lenguaje Ada, para que el usuario pueda realizar los programas de control en este lenguaje. Esta parte está programada en lenguaje Ada y se basa en la facilidad que proporciona el propio lenguaje para realizar llamadas a funciones programadas en otros lenguajes de programación (como por ejemplo C). Los ficheros involucrados son:
  - El fichero *"driverada_nd.ads"*: Contiene la especificación de los tipos y funciones desarrolladas para manejar los módulos.

  - El fichero *"driverada_nd.adb"*: Contiene la implementación de las funciones desarrolladas para manejar los módulos.

3.2.1.- Conceptos previos: el bus local

Los distintos módulos conectados para formar el sistema, se identifican en el bus mediante una dirección física. Estas direcciones ni siquiera tienen porque ser correlativas, de tal forma, que podríamos tener dos módulos, uno con dirección cinco y otro con dirección treinta.

La implementación del driver, se ha realizado para que el usuario no necesite conocer en ningún momento las direcciones físicas de los módulos conectados al bus. El dato que ha de manejar es la dirección del módulo en el ***bus local***. Este bus local, está formado únicamente por los módulos de E/S.

La detección de los módulos que componen el bus local y la asignación de direcciones se hacen de forma dinámica durante la fase de inicialización. En esta fase se buscan módulos en el bus entre las direcciones cero y *limite*, siendo *limite* un valor indicado por el usuario. Se comprueba que exista el módulo con *dirección x* en el bus y, en caso afirmativo, se le asigna una posición en un *bus local*. Estas posiciones son correlativas y se asignan, empezando por cero, según se van encontrando módulos. De esta forma podríamos tener conectados al bus tres módulos con direcciones físicas cinco, ciento cinco y noventa y durante la fase de inicialización se les asignaría las direcciones cero, dos y uno en el bus local respectivamente (Tabla 3.1).

	Módulo ND-6053	Módulo ND-6058	Módulo ND-6053
Dirección física	5	105	90
Posición en el bus	0	2	1

Tabla 3.1 Ejemplo de asignación de direcciones en el bus local

3.2.2.- Código C

3.2.2.1.- Estructuras y variables utilizadas

En el código fuente, se definen una serie de macros, estructuras y variables que se utilizan para realizar la implementación del driver.

Variables globales

- **COM1** y **COM2**: Indican el puerto serie a utilizar para la comunicación.

- **PUERTO_A**, **PUERTO_B** y **PUERTO_C**: indican el puerto de la tarjeta DIN-24R sobre el que actuar.

- **_Digital_IN** y **_Digital_OUT**: Indican el tipo de módulo: entradas digitales y salidas digitales respectivamente.

- **Variables de error**: Proporcionan información sobre el estado en que finalizó la última operación. En la Tabla 3.2 se muestran las variables de error utilizadas, el valor que toman cada una de ellas y la descripción.

Error	Valor	Descripción
CORRECTO	0	En la última operación no se produjo ningún error
ERROR_COM	-101	Error en la comunicación con el puerto serie
FD_INVALIDO	-102	Descriptor del puerto inválido
TIMEOUT	-103	Timeout al leer del puerto. El dispositivo no ha respondido en el tiempo esperado
RESPUESTA_ERRONEA	-200	El dispositivo ha devuelto una respuesta considerada errónea
ERROR_MEM	-300	No hay memoria dinámica reservada. Se puede producir por dos causas: • Si no se ha llamado a la función de inicialización antes de llamar a cualquier otra función • Si ha fallado la reserva dinámica de memoria durante la inicialización
NO_MODULO	-400	El número de módulo al que se quiere acceder no existe en el bus local
NO_DINPUT	-401	El módulo no es un módulo de entradas digitales
NO_DOUTPUT	-402	El módulo no es un módulo de salidas digitales
NO_LINEA	-500	La línea a la que se quiere acceder no existe
NO_PORT	-600	El puerto especificado no es un puerto válido para la tarjeta DIN-24R
BUS_VACIO	-700	No se ha encontrado ningún módulo en el bus

Tabla 3.2 Variables de error

Macros internas

```
#define _ND6053   6053
#define _ND6058   6058
```

Contienen los códigos del fabricante para identificar los distintos módulos. Este identificador es único para cada tipo de módulo.

Estructuras

- `typedef struct _Tdevice { … } DEVICE;`

  Esta estructura contiene los campos que identifican a cada uno de los módulos:

  - `int _direccion;`

    Contiene la dirección física del módulo, necesaria para enviar las tramas al bus.

  - `int _deviceType;`

    Identifica el tipo de módulo. Para ello se utilizan las macros `_Digital_IN` y `_Digital_OUT` definidas en el fichero de cabecera.

  - `int _deviceNumber;`

    Contiene la identificación del dispositivo. Este identificador es único y se corresponde con las macros `_ND6053` y `_ND6058`.

  - `char _deviceName [15];`

    Contiene el nombre del dispositivo.

  - `float * _lineas;`

    Contiene el valor de cada uno de los canales que tiene el módulo. Esta variable es un array dinámico de flotantes. Una vez que se conoce el número de canales que forman el módulo se reserva la memoria necesaria.

  - `int _numLineas;`

    Contiene el número de canales de que dispone el módulo.

- `typedef struct _Tbus { … } BUS;`

  Esta estructura contiene los campos con las propiedades del bus:

  - `int _puerto;`

    Identifica el puerto serie utilizado. Se deben utilizar las macros `COM1` y `COM2` definidas en el fichero de cabecera.

  - `int _velocidad;`

    Contiene la velocidad de comunicación del puerto serie.

  - `int _numModulos;`

    Contiene el número de módulos que forman el bus.

  - `DEVICE * _bus;`

    Contiene la información de cada uno de los módulos que forman el bus. Esta variable es un array dinámico de estructuras DEVICE que, como vimos anteriormente, contienen la información de cada módulo. Se reserva dinámicamente la memoria tras conocer el número de módulos que forman el bus.

Variables internas

- `BUS DeviceBus = {0};`

  Esta variable contiene toda la información del bus.

- `int _nerror;`

  Contiene el número que identifica el estado en que finalizó la última función.

- `char _serror[255]`

  Cadena que contiene la descripción del estado en que finalizó la última función. Si la operación finalizó correctamente la cadena estará vacía. En caso contrario, contendrá una descripción del error producido. Esta cadena incluye una traza de las funciones en que ha sucedido el error.

- `int estado = 0;`

  Indica si en algún momento se ha producido un error al asignar memoria dinámica. Se inicializa a cero y en el momento de asignar memoria dinámica se actualiza a uno si se hizo correctamente la asignación. Si no se asignó memoria dinámica correctamente, no se permitirá realizar ninguna operación. Por lo tanto, en toda función se debe comprobar, antes de nada, el valor de esta variable y, si fuera cero, retornar error.

3.2.2.2.- Funciones locales

Existen un conjunto de funciones que se utilizan internamente para el correcto funcionamiento del driver.

expon

◇ **Sintaxis**

```
int expon (int base, int exp);
```

◇ **Descripción**

Realiza la operación *base* elevado a *exp* y retorna el resultado. Aunque existe la función *pow* en C que realiza la misma operación, se decidió implementar aquí esta función para evitar problemas de librerías en la implementación en Ada.

Init_bus

◇ **Sintaxis**

```
int Init_bus (void);
```

◇ **Descripción**

Esta función inicializa todos los módulos del bus.

Init_modulo

◇ Sintaxis

```
int Init_modulo (int modulo);
```

◇ Descripción

Esta función inicializa el módulo que se le indica como argumento. Para ello se siguen los siguientes pasos:

- Se desactiva el watchdog en el módulo mediante el comando "*Set Host Watchdog/Safety Value*".
- Si el módulo a inicializar es el ND-6058, se configuran las 24 líneas de E/S programables como salidas, mediante el comando "*Set Programmable I/O Mode*", y se inicializan a cero.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`) y se retornará -1.

BuscaModulos

◇ Sintaxis

```
int BuscaModulos (int limite);
```

◇ Descripción

Esta función busca los módulos que forman el bus, entre las direcciones físicas cero y *limite*, e inicializa correctamente todas las estructuras de datos y variables utilizadas. Para ello sigue los siguientes pasos:

- Comprueba que existan en el bus los módulos con dirección física entre cero y *limite*. Para ello se envía el comando "*Read Configuration*" a todos los módulos.
- Tras conocer el número de módulos que forman el bus, se reserva memoria para el vector `DeviceBus._bus`, que contendrá la información de todos ellos.
- Se continúa la inicialización de cada una de las variables que identifican los módulos. Para inicializar las variables correctamente es necesario conocer la identidad de los módulos. Para ello, se envía el comando "*Read Module Name*" a cada uno de ellos. Actualmente sólo se soportan los módulos ND-6053 y ND-6058.

- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`) y se retornará -1.

3.2.2.3.- Funciones globales

En este epígrafe únicamente se describe la estructura interna de aquellas funciones que por su importancia o complejidad así lo requieran. La descripción detallada de todas las funciones que forman la librería se proporciona en el epígrafe 3.2.4.

ND_Inicializar

- Se inicializan los componentes `_puerto` y `_velocidad` de la variable `DeviceBus` con lo parámetros de la función.
- Se abre el puerto serie.
- Se buscan los módulos que forman el bus local mediante la función *BuscaModulos*.
- Se inicializa el bus.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

ND_Liberar

- Se cierra el puerto serie.
- Se libera la memoria asignada dinámicamente durante la inicialización.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

ND_LeeModulo_DI

- Se hacen las comprobaciones previas a la ejecución de la función.
- Se envía al módulo el comando *Digital Input* para leer el estado de todos los canales.
- Se actualizan los valores locales de los canales del módulo.
- Se retorna codificado en un entero el estado de todas las líneas.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

ND_EscribePuerto_DO

- Se hacen las comprobaciones previas a la ejecución de la función.
- Se envía al módulo el comando *Digital Output* para escribir el estado de todos los canales del puerto indicado.
- Se actualizan los valores locales de los canales del módulo.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

ND_EscribeModulo_DO

- Se hacen las comprobaciones previas a la ejecución de la función.
- Se llama a la función *ND_EscribePuerto_DO* para establecer los valores de cada uno de los puertos.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

ND_LeeLinea_DI

- Se hacen las comprobaciones previas a la ejecución de la función.
- Se llama a la función *ND_LeeModulo_DI*.
- Se retorna el valor del canal solicitado.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

ND_EscribeLinea_DO

- Se hacen las comprobaciones previas a la ejecución de la función.
- Se actualiza el valor del canal en las variables locales.
- Se codifica el valor de los canales del módulo para enviar el comando *Digital Output* para escribir el estado de todos los canales del puerto al que pertenece la línea.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

3.2.3.- Código Ada

El lenguaje Ada proporciona la facilidad de reutilizar código escrito en otro lenguaje. El paquete *Interfaces.C* contiene los tipos básicos, constantes y subprogramas que permiten a un programa en Ada pasar parámetros y cadenas a las funciones escritas en C.

Para llevar a cabo la utilización de las funciones escritas en C se hace uso del *pragma import*:

- Primero se declara la función como si se fuera a implementar en lenguaje Ada.
- Luego se utiliza *pragma import* para utilizar la función escrita en C en nuestra implementación en Ada. Este pragma necesita tres parámetros:
  - El lenguaje en que está implementada la función a importar.
  - El nombre que tendrá la función en nuestra implementación en Ada.
  - El nombre de la función en lenguaje C.

En el Ejemplo 3.1 se muestra la utilización de este pragma. La llamada a la función *ND_LeeLinea_DI* en la interfaz en Ada, hace uso de la función definida en lenguaje C con el mismo nombre.

```
function ND_LeeLinea_DI (Modulo: Num_Modulo; Linea: Num_Linea_Entrada)
return Integer is
   function ND_Ada_LeeLinea_DI (Modulo: Integer; Linea: Integer)
   return Integer;
   pragma Import (C, ND_Ada_LeeLinea_DI, "ND_LeeLinea_DI");
begin
   return ND_Ada_LeeLinea_DI (Modulo, Linea);
end ND_LeeLinea_DI;
```

Ejemplo 3.1 Interfaz C-Ada

La definición de las variables de acceso al puerto serie, variables de error, de identificación del tipo de módulo y del puerto de la tarjeta DIN-24R, se definen como valores enumerados. También se definen nuevos tipos, derivados del tipo *Integer* y con el rango de valores limitado. En las funciones que utilicen estos tipos, si se realiza una llamada con un valor fuera del rango, se producirá una excepción en tiempo de ejecución.

3.2.4.- Descripción de las funciones

El argumento *modulo*, utilizado en varias funciones, hace referencia a la posición en el bus local del módulo al que se desea acceder.

Si la ejecución de una función da lugar a un error, se devuelve un valor negativo y se actualizan las variables internas de error. Se recomienda utilizar las funciones *ObtenerError* y *ObtenerErrorString* para obtener el código de error y una descripción de éste respectivamente.

ND_Inicializar

◇ **Descripción**

Esta función sirve para inicializar el bus de comunicaciones de los módulos NuDAM. Se debe invocar esta función antes de llamar a cualquier otra.

◇ **Sintaxis**

C/C++

```
int ND_Inicializar (int puerto,
                    int velocidad,
                    int limite);
```

Ada

```
function ND_Inicializar (Puerto: Puerto_Serie;
                         Velocidad: Integer;
                         Limite: Num_Modulo)
return Integer;
```

◇ **Argumentos**

Puerto: Puerto serie del PC que se va ha emplear para la comunicación. Se deben utilizar las variables COM1 y COM2 definidas.

Velocidad: Velocidad del puerto serie a la que se va a llevar a cabo la comunicación. Este valor ha de coincidir con el valor fijado en los módulos. Los valores permitidos son: 300, 1200, 2400, 9600, 19200, 28400, 57600, 115200. Si se indica un valor distinto a estos se producirá un error.

Limite: Dirección física hasta la que se buscarán módulos en el bus.

◇ **Valor retornado**

0 si todo fue correcto.

-1 en caso de error.

ND_Liberar

◇ Descripción

Esta función libera el bus de comunicaciones y los recursos comprometidos durante la inicialización.

◇ Sintaxis

C/C++

```
int ND_Liberar (void);
```

Ada

```
function ND_Liberar return Integer;
```

◇ Valor retornado

0 si todo fue correcto.

-1 en caso de error.

ND_EscribeLinea_DO

◇ Descripción

Esta función actualiza el valor de una línea de salida de uno de los puertos de la tarjeta DIN-24R, conectada al módulo ND-6058.

◇ Sintaxis

C/C++

```
int ND_EscribeLinea_DO (const int modulo,
                        const int puerto,
                        const int linea,
                        const int valor);
```

Ada

```
function ND_EscribeLinea_DO (Modulo: Num_Modulo;
                             Puerto: Puerto_Tarjeta;
                             Linea: Num_Linea_Salida;
                             Valor: U1)
return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

Puerto: Puerto de la tarjeta donde se encuentra la línea a actualizar (utilizar las variables *PUERTO_A*, *PUERTO_B* o *PUERTO_C*).

Linea: Número de línea del puerto que se desea actualizar (entre 0 y 7).

Valor: Valor de la línea a actualizar (1: activar; 0: desactivar).

◇ **Valor retornado**

0 si todo fue correcto.

-1 en caso de error.

ND_EscribePuerto_DO

◇ **Descripción**

Esta función actualiza los ocho canales de uno de los puertos de la tarjeta DIN-24R, conectada al módulo ND-6058.

◇ **Sintaxis**

C/C++

```
int ND_EscribePuerto_DO (const int modulo,
                         const int puerto,
                         const U8 valor);
```

Ada

```
function ND_EscribePuerto_DO (Modulo: Num_Modulo;
                              Puerto: Puerto_Tarjeta;
                              Valor: U8)
return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

Puerto: Puerto de la tarjeta que se desea actualizar (utilizar las variables *PUERTO_A*, *PUERTO_B* o *PUERTO_C*).

Valor: Entero donde estén codificados los valores de los ocho canales del puerto.

◇ **Valor retornado**

El número de canales del puerto si todo fue correcto.

-1 en caso de error.

ND_EscribeModulo_DO

◇ **Descripción**

Esta función actualiza los 3 puertos de la tarjeta DIN-24R, conectada al módulo ND-6058.

◇ **Sintaxis**

C/C++

```
int ND_EscribeModulo_DO (const int modulo,
                         const U8 valorA,
                         const U8 valorB,
                         const U8 valorC);
```

Ada

```
function ND_EscribeModulo_DO (Modulo: Num_Modulo;
                              ValorA: U8;
                              ValorB: U8;
                              ValorC: U8)
return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

ValorA: Entero donde estén codificados los valores de los ocho canales del puerto A de la tarjeta DIN-24R.

ValorB: Entero donde estén codificados los valores de las ocho canales del puerto B de la tarjeta DIN-24R.

ValorC: Entero donde estén codificados los valores de las ocho canales del puerto C de la tarjeta DIN-24R.

◇ **Valor retornado**

El número de líneas del módulo si todo fue correcto.

-1 en caso de error.

ND_LeeLinea_DI

◇ **Descripción**

Esta función lee el estado de un canal de entrada de un módulo de entradas digitales ND-6053.

◇ **Sintaxis**

C/C++

```
int ND_LeeLinea_DI (const int modulo,
                    const int linea);
```

Ada

```
function ND_LeeLinea_DI (Modulo: Num_Modulo;
                         Linea: Num_Linea_Entrada)
return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

Linea: Número del canal de entrada que se desea leer (entre 0 y 15).

◇ **Valor retornado**

El estado del canal de entrada (1: activo; 0: inactivo) si todo fue correcto.

Un número negativo si hubo algún error.

ND_LeeModulo_DI

◇ **Descripción**

Esta función lee el estado de los dieciséis canales de entrada del módulo de entradas digitales ND-6053. El estado de las entradas se devuelve codificado en un entero que se ha de pasar a la función como argumento.

◇ **Sintaxis**

C/C++

```
int ND_LeeModulo_DI (const int modulo,
                     U16 * valor);
```

Ada

```
procedure ND_LeeModulo_DI (Modulo: in Num_Modulo;
                           Valor: in out U16;
                           Retorno: out Integer);
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

Valor: Entero donde se codificará el estado de los dieciséis canales de entrada del módulo.

◇ **Valor retornado**

El número de líneas del módulo si todo fue correcto.

-1 en caso de error.

La implementación de esta función en Ada se hace a través de un procedimiento. El valor de retorno se devuelve a través de la variable *Retorno.*

ObtenerError

◇ Descripción

Esta función sirve para obtener el estado en que finalizó la última operación que se ha realizado.

◇ Sintaxis

C/C++

```
int ObtenerError (void);
```

Ada

```
function ObtenerError return Var_Error;
```

◇ Valor retornado

C/C++

Un entero que identifica la situación en que finalizó la última operación. Los posibles valores están definidos en las variables de error.

Ada

Una variable de tipo *Var_Error* (valor enumerado donde se definen los errores).

ObtenerErrorString

◇ Descripción

Esta función sirve para obtener una descripción del estado en que finalizó la última operación realizada. Si en la última llamada a una función se ha producido un error, mediante una llamada a esta función se obtendrá una descripción de éste.

◇ Sintaxis

C/C++

```
char * ObtenerErrorString (void);
```

Ada

```
function ObtenerErrorString return string;
```

◇ Valor retornado

Una cadena con la descripción del estado en que finalizó la última operación realizada. Si en la última operación no se produjo ningún error se retornará una cadena vacía.

Num_Lineas

◇ **Descripción**

Esta función proporciona el número de canales que forman el módulo.

◇ **Sintaxis**

C/C++

```
int Num_Lineas (int modulo);
```

Ada

```
function Num_Lineas (Modulo: Integer) return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

◇ **Valor retornado**

El número de canales del módulo si todo fue correcto.

-1 en caso de error.

Num_Modulos

◇ **Descripción**

Esta función proporciona el número de módulos que se han detectado en el bus local.

◇ **Sintaxis**

C/C++

```
int Num_Modulos (void);
```

Ada

```
function Num_Modulos return Integer;
```

◇ **Valor retornado**

El número de módulos que forman el bus local si todo fue correcto.

-1 en caso de error.

Get_Tipo

◇ **Descripción**

Esta función retorna información sobre el tipo del módulo (entradas digitales, salidas digitales,…).

◇ **Sintaxis**

C/C++

```
int Get_Tipo (int modulo);
```

Ada

```
function Get_Tipo (Modulo: Integer) return Device_Type;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

◇ **Valor retornado**

C/C++

El número que identifica el tipo de módulo (variables *_Digital_IN* y *_Digital_OUT*).

-1 en caso de error.

Ada

Una variable de tipo *Device_Type* (valor enumerado donde están definidos los tipos de dispositivos). En caso de error el valor del enumerado retornado será *ERROR.*

Get_Nombre

◇ **Descripción**

Esta función permite obtener el nombre de un módulo.

◇ **Sintaxis**

C/C++

```
char * Get_Nombre (int modulo);
```

Ada

```
function Get_Nombre (Modulo: Integer) return String;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

◇ **Valor retornado**

Una cadena con el nombre del módulo si todo fue correcto.

Una cadena con la descripción del error sucedido en caso de error.

Get_Id

◇ **Descripción**

Esta función permite obtener el identificador de un módulo. Este identificador es único para cada tipo de módulo y se corresponde con el valor asignado por el fabricante.

◇ **Sintaxis**

C/C++

```
int Get_Id (int modulo);
```

Ada

```
function Get_Id (Modulo: Integer) return Integer;
```

◇ **Argumentos**

Modulo: Posición del módulo en el bus local.

◇ **Valor retornado**

El número del fabricante que identifica el módulo si todo fue correcto.

-1 en caso de error.

Get_Direccion

◇ Descripción

Esta función permite obtener la dirección física que tiene asignada un módulo.

◇ Sintaxis

C/C++

```
int Get_Direccion (int modulo);
```

Ada

```
function Get_Direccion (Modulo: Integer) return Integer;
```

◇ Argumentos

Modulo: Posición del módulo en el bus local.

◇ Valor retornado

La dirección física que tiene asignada el módulo si todo fue correcto.

-1 en caso de error.

3.3.- Anexo A: Command Set (digital I/O modules)

En este anexo se proporciona información sobre los comandos que utilizan los módulos NuDAM de E/S digitales. En la Tabla 3.3 se muestran todos los comandos que soportan los módulos. Únicamente se incluye la descripción de aquellos comandos que se han utilizado para implementar el driver. Esta información se ha obtenido directamente del capítulo tres, "*Command Set*", del manual de usuario de los módulos de E/S digitales del sistema NuDAM.

Command Set of Digital I/O Modules		
Command	**Syntax**	**Module**
General Commands		
Set Configuration	%(OldAddr)(NewAddr)(TypeCode)(BaudRate)(CheckSumFlag)	ALL
Read Configuration	$(Addr)2	ALL
Read Module Name	$(Addr)M	ALL
Read Firmware Version	$(Addr)F	ALL
Reset Status	$(Addr)5	ALL
Functional Commands		
Synchronized Sampling	#**	6050, 6052, 6053, 6054, 6058, 6060
Read Synchronized Data	$(Addr)4	6050, 6052, 6053, 6054, 6058, 6060
Digital Output	#(Addr)(ChannelNo)(OutData)	6050, 6060, 6063
	#(Addr)(Port)(Odata)	6056, 6058
	#(Addr)(Port)(ChannelNo)(BitData)	6056,6058
	#(Addr)T(OdataA)(OdataB)(OdataC)	6058
Digital Input	$(Addr)6	ALL
Set Programmable I/O Mode	$(Addr)S(IOSts)	6058
Special Commands		
Read Command Leading Code Setting	~(Addr)0	ALL
Change Command Leading Code Setting	~(Addr)10(C1)(C2)(C3)(C4)(C5)(C6)	ALL
Set Host Watchdog / Safety Value	~(Addr)2(Flag)(TimeOut)(SafeValue)	ALL
Read Host WatchDog / Safe Value	~(Addr)3	ALL
Change Polarity	~(Addr)CP(Status)	ALL
Read Polarity	~(Addr)CR	ALL
Host is OK	~**	ALL

Tabla 3.3 Command set of digital I/O modules

Comandos utilizados

Read Configuration

Description

Read the configuration of module on a specified address ID.

Syntax

`$(Addr)2<CR>`

- `$` Command leading code
- `(Addr)` Address ID.
- `2` Command code for reading configuration

Response

`!(Addr)(TypeCode)(BaudRate)(CheckSumFalg)<CR>` or `?(Addr)<CR>`

- `!` Command is valid.
- `?` Command is invalid.
- `(Addr)` Address ID.
- `(TypeCode)` It always be 40 (Hex) for digital I/O modules.
- `(BaudRate)` Current setting of communication baud rate, refer to Table 0-1 for details.
- `(CheckSumFlag)` Current setting of check-sum flag, refer to Table 0-3. for details.

Example

User command: `$302<CR>`

Response: `!30400600<CR>`

- ! Command is valid.
- 30 Address ID.
- 40 Digital I/O module.
- 06 Baud rate is 9600 bps.
- 00 checksum is disable.

Code	Baudrate
03	1200 bps
04	2400 bps
05	4800 bps
06	9600 bps
07	19200 bps
08	38400 bps
09	115200 bps

Table 0-1 Baud rate setting code

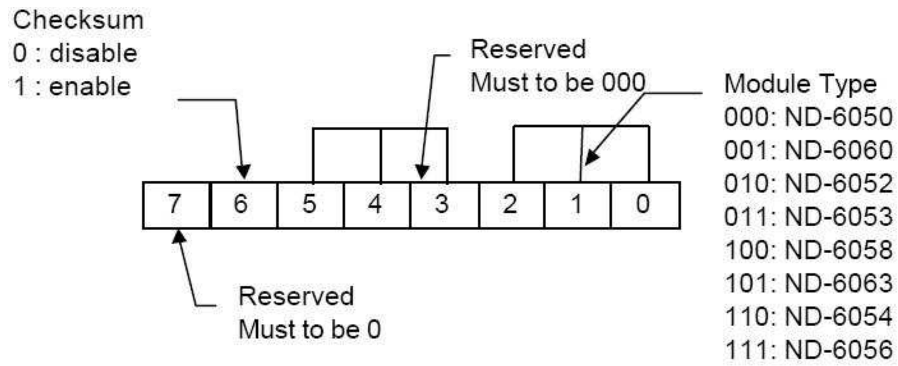


Table 0-3 Response of check sum flag

Read Module Name

Description

Read NuDAM module's name.

Syntax

`$(Addr)M<CR>`

`$`	Command leading code.
`(Addr)`	Address ID.
`M`	Read module name.

Response

`!(Addr)(ModuleName) <CR>` or `?(Addr)<CR>`

`!`	Command is valid.
`?`	Command is invalid.
`(Addr)`	Address ID.
`(ModuleName)`	NuDAM module's name.

Example

User command: `$30M<CR>`

Response: `!306050<CR>`

!	Command is valid
30	Address.
6050	ND-6050 (Digital I/O module).

Digital Output

Description

Set digital output port channel value at specified address. This command is only available to modules involving the multiport digital output function.

Syntax

`#(Addr)(Port)(OutData)<CR>` (6056, 6058 only)

`#`	Command leading code. (1-character)
`(Addr)`	Address ID (2-character)
`(Port)`	Set value to indivisual port
	0H: for 6056 channel 14 to 8
	0L: for 6056 channel 7 to 0
	0A: for 6058 port A
	0B: for 6058 port B
	0C: for 6058 port C (2-character)
`(OutData)`	Each bit is mapping to each channel number (2-character)

*** if the port of ND-6058 is in input mode, output data to this port will beignore.**

Response

<CR> or ?(Addr)<CR>

`>`	Command is valid.
`?`	Command is invalid.
`(Addr)`	Address ID.

Example

User command: `#2F0A10<CR>`

Response: `<CR>`

2F Address ID

0A Set output to port A

10 Set channel 4 of port A ON

Digital Input

Description

Read the digital input channel value and readback the digital output channel value.

Syntax

`$(Addr)6<CR>`

- `$` Command leading code.
- `(Addr)` Address ID
- `6` Digital data input command.

Response

ND-6053 module response : !(DataInH)(DataInL)00<CR>

or

`?(Addr)<CR>`

- ! Command is valid.
- ? Command is invalid.
- (DataInH) Value of digital input channel 15-8. (2-character)
- (DataInL) Value of digital input channel 7-0.(2-character)

Host is OK

Description

When host watchdog timer is enable, host computer must send this command to every module before timeout otherwise “host watchdog timer enable” module‘s output value will go to safety state output value.

Timeout value and safety state output value is defined “Set Host Watchdog Timer & Safety Value”.

Syntax

`~**<CR>`

- ~ Command leading code.
- `**` Host is OK.

Response

Note: Host is OK command has NO response.

Programmable I/O Mode Setting

Description

Set the programmable input or output mode for ND-6058.

Syntax

`$(Addr)S(IOFlag)<CR>` (6058 only)

`$`	Command leading code.
`(Addr)`	Address ID
`S`	Set programmable I/O mode
`(IOFlag)`	Status of programmable I/O

0x00: A(O/P) B(O/P) CH(O/P) CL(O/P)

0x01: A(O/P) B(O/P) CH(O/P) CL(I/P)

0x02: A(O/P) B(O/P) CH(I/P) CL(O/P)

0x03: A(O/P) B(O/P) CH(I/P) CL(I/P)

0x04: A(O/P) B(I/P) CH(O/P) CL(O/P)

0x05: A(O/P) B(I/P) CH(O/P) CL(I/P)

0x06: A(O/P) B(I/P) CH(I/P) CL(O/P)

0x07: A(O/P) B(I/P) CH(I/P) CL(I/P)

0x08: A(I/P) B(O/P) CH(O/P) CL(O/P)

0x09: A(I/P) B(O/P) CH(O/P) CL(I/P)

0x0A: A(I/P) B(O/P) CH(I/P) CL(O/P)

0x0B: A(I/P) B(O/P) CH(I/P) CL(I/P)

0x0C: A(I/P) B(I/P) CH(O/P) CL(O/P)

0x0D: A(I/P) B(I/P) CH(O/P) CL(I/P)

0x0E: A(I/P) B(I/P) CH(I/P) CL(O/P)

0x0F: A(I/P) B(I/P) CH(I/P) CL(I/P)

*I/P input mode, O/P output mode.

Response

`!(Addr)<CR>` or `?(Addr)<CR>`

! Command is valid.

? Command is invalid.

(Addr) Address ID

Example

User command: `$06S0C<CR>`

Response: `!06<CR>`

! Command is valid.

0C Port A and B are input mode, high and low half byte of port C are output mode.

Set Host Watchdog Timer & Safety Value

Description

Set host watchdog timer, module will change to safety state when host is failure. Define the output value in this command.

Syntax

```
~(Addr)2(Flag)(TimeOut)(SafeValue)<CR>

~(Addr)2(Flag)(TimeOut)(SafeH)(SafeL)<CR> (6056 only)

~(Addr)2(Flag)(TimeOut)(SafeA)(SafeB)(SafeC)<CR> (6058 only)
```

~	Command leading code.
(Addr)	Address ID, range (00 - FF).
2	Set host watchdog timer and safe state value.
(Flag)	0: Disable host watchdog timer 1: Enable host watchdog timer (1-character)
(TimeOut)	Host timeout value, between this time period host must send (Host is OK) command to module, otherwise module will change to safety state. Range 01 - FF. (2-character) One unit is 100 ms 01 = 1 * 100 = 100 ms FF = 255 * 100 = 25.5 sec
(SafeValue)	8 channels safety value of digital output channels when host is failure. (2-character)
(SafeH)	Safety value of digital output channels 14 ~ 8 when host is failure. (2-character)
(SafeL)	Safety value of digital output channels 7 ~ 0 when host is failure. (2-character)
(SafeA)	Safety value of port A channels 7 ~ 0 when host is failure while A in output mode. (2-character)

(SafeB) Safety value of port B channels 7 ~ 0 when host is failure while B in output mode. (2-character)

(SafeC) Safety value of port C channels 7 ~ 0 when host is failure while C in output mode. (2-character)

Response

!(Addr)<CR> or ?(Addr)<CR>

! Command is valid.

? Command is invalid.

(Addr) Address ID

Example

Example for NuDAM-6050 :

User command: ~0621121C<CR>

Response: !06<CR>

06 Address ID

2 Set host watchdog timer and safe state value.

1 Enable host watchdog timer.

12 Timeout value. 0x12 = 18
18 * 100 = 1800 ms

1C 1C (00011100) Digital output channel DO2, DO3 and DO4 are high, the others are low.

Example for NuDAM-6058 :

User command: ~0621121C1C1C<CR>

Response: !06<CR>

06 Address ID

2 Set host watchdog timer and safe state value.

1 Enable host watchdog timer.

12 Timeout value. 0x12 = 18
18 * 100 = 1800 ms

1C1C1C 1C (00011100) port A, B and C channel 2, 3 and 4 are high, the other are low.

4.- LabJack U12, manual del programador

4.1.- Implementación del driver

4.1.1.- Conceptos previos

La empresa LabJack, fabricante de la Tarjeta de Adquisición de Datos LabJack U12, proporcionan un driver para manejar esta tarjeta bajo sistemas operativos Linux. Por lo tanto, la implementación de la interfaz en lenguaje C y Ada utilizará los drivers proporcionados por el fabricante. En el epígrafe 4.2, se encuentra una descripción detallada de las funciones de la librería del fabricante que se han utilizado para desarrollar la interfaz.

4.1.2.- Código C

4.1.2.1.- Estructuras y variables utilizadas

En el código fuente, se definen una serie de variables y funciones que se utilizan para realizar la implementación del driver.

Variables globales

- **INPUT** y **OUTPUT**: Indican la configuración de la tarjeta RB16, como entradas y salidas respectivamente.

Variables internas

- `Int _numLineas = 16;`

  Esta variable contiene el número de líneas de la tarjeta RB16.

- `long _nerror;`

  Contiene el número que identifica el estado en que finalizó la última función.

- `char _serror[255]`

  Cadena que contiene la descripción del estado en que finalizó la última función. Si la operación finalizó correctamente la cadena estará vacía. En caso contrario, contendrá una descripción del error producido.

Estas dos últimas variables, son las variables de error que identifican el estado en que finalizó una operación. Si la ejecución de una función da lugar a un error, se devuelve un valor negativo y se actualizan estas variables. El driver proporcionado por el fabricante define sus propios códigos de error. En el epígrafe 4.3 se encuentra un listado de todos los posibles errores.

4.1.2.2.- Funciones locales

expon

◇ **Sintaxis**

```
int expon (int base, int exp);
```

◇ **Descripción**

Realiza la operación *base* elevado a *exp* y retorna el resultado. Aunque existe la función *pow* en C que realiza la misma operación, se decidió implementar aquí esta función para evitar problemas de librerías en la implementación en Ada.

4.1.2.3.- Funciones globales

En este epígrafe únicamente se describe la estructura interna de aquellas funciones que por su importancia o complejidad así lo requieran. La descripción detallada de todas las funciones que forman la interfaz de programación, se proporciona en el epígrafe 4.1.4.

Las funciones definidas, utilizan internamente la librería proporcionada por el fabricante. En el epígrafe *4.2.- Anexo A*, se puede encontrar la descripción detallada de aquellas funciones de la librería del fabricante que se han utilizado para realizar nuestra propia interfaz.

Las funciones de la librería del fabricante se representan con el siguiente formanto: `InitLabjack()`.

LJ_Inicializar

- Se invoca a la función `InitLabjack()` (de la librería del fabricante) para inicializar el módulo.
- Se configura la tarjeta RB16, en función del valor del parámetro *tipo*, utilizando la función `DigitalIO()`.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

LJ_EscribeLinea_DO

- Se tratan adecuadamente los valores, pues la tarjeta trabaja con lógica invertida.
- Se llama a la función `EDigitalOut()` para actualizar el valor de la línea.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

LJ_LeeLinea_DI

- Se llama a la función `EDigitalIn()` para obtener el valor de la línea.
- Se retorna el valor adecuado, teniendo en cuenta que la tarjeta trabaja con lógica invertida.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

LJ_EscribeModulo_DO

- Se tratan adecuadamente los valores, pues la tarjeta trabaja con lógica invertida.
- Se llama a la función `DigitalIO()` para actualizar los valores de las dieciséis líneas de la tarjeta RB16.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

LJ_LeeModulo_DI

- Se llama a la función `DigitalIO()` para obtener los valores de las dieciséis líneas de la tarjeta RB16.
- Se tratan adecuadamente los valores, pues la tarjeta trabaja con lógica invertida, para retornar en un entero la codificación de las dieciséis líneas.
- Si falla alguno de los pasos anteriores se actualizarán convenientemente las variables de error (`_nerror` y `_serror`).

4.1.3.- Código Ada

El lenguaje Ada proporciona la facilidad de reutilizar código escrito en otro lenguaje. El paquete *Interfaces.C* contiene los tipos básicos, constantes y subprogramas que permiten a un programa en Ada pasar parámetros y cadenas a las funciones escritas en C.

Para llevar a cabo la utilización de las funciones escritas en C se hace uso del *pragma import*:

- Primero se declara la función como si se fuera a implementar en lenguaje Ada.
- Luego se utiliza *pragma import* para utilizar la función escrita en C en nuestra implementación en Ada. Este pragma necesita tres parámetros:
    - El lenguaje en que está implementada la función a importar.
    - El nombre que tendrá la función en nuestra implementación en Ada.
    - El nombre de la función en lenguaje C.

En el Ejemplo 4.1 se muestra la utilización de este pragma. La llamada a la función *LJ_LeeLinea_DO* en la interfaz en Ada, hace uso de la función definida en lenguaje C con el mismo nombre.

```
function LJ_LeeLinea_DI (Modulo: Integer; Linea: Num_Linea) return Integer
is
   function LJ_Ada_LeeLinea_DI (Modulo: Integer; Linea: Integer)
   return Integer;
   pragma Import (C, LJ_Ada_LeeLinea_DI, "LJ_LeeLinea_DI");
begin
   return LJ_Ada_LeeLinea_DI (Modulo, Linea);
end LJ_LeeLinea_DI;
```

Ejemplo 4.1 Interfaz C-Ada

Las variables de identificación del tipo de módulo, se definen como valores enumerados. También se definen nuevos tipos, derivados del tipo *Integer* y del tipo *Float*, con el rango de valores limitado. En las funciones que utilicen estos tipos, si se realiza una llamada con un valor fuera del rango, se producirá una excepción en tiempo de ejecución.

4.1.4.- Descripción de las funciones

Se puede ver que varias funciones necesitan el argumento *modulo*. Este argumento hace referencia al número de serie del módulo o al Local ID.

Si la ejecución de una función da lugar a un error, se devuelve un valor negativo. Se recomienda utilizar las funciones *ObtenerError* y *ObtenerErrorString* para obtener el código de error y una descripción de este respectivamente.

LJ_Inicializar

◇ **Descripción**

Esta función sirve para inicializar el módulo y configurarlo para manejar las entradas o salidas del proceso. Se debe invocar esta función antes de llamar a cualquier otra.

◇ **Sintaxis**

C/C++

```
int LJ_Inicializar (int modulo,
                    int tipo);
```

Ada

```
function LJ_Inicializar (Modulo: Integer;
                         Tipo: Device_Type)
return Integer;
```

◇ **Argumentos**

<u>*Modulo*</u>: Número de serie o Local ID.

<u>*Tipo*</u>: Configuración que se desea otorgar a la tarjeta RB16 conectada al módulo, como entradas o salidas. Utilizar las variables *INPUT* y *OUTPUT.*

◇ **Valor retornado**

0 si todo fue correcto.

-1 en caso de error durante la inicialización.

<u>Sólo para la versión en C/C++</u>: -2 si la variable *tipo* no se corresponde con ninguna de las macros definidas.

LJ_EscribeLinea_DO

◇ **Descripción**

Esta función actualiza el valor de un canal de salida de la tarjeta RB16 conectada al módulo LabJack U12 correspondiente.

◇ **Sintaxis**

C/C++

```
int LJ_EscribeLinea_DO (int modulo,
                        int linea,
                        int valor);
```

Ada

```
function LJ_EscribeLinea_DO (Modulo: Integer;
                             Linea: Num_Linea;
                             Valor: U1)
return Integer;
```

◇ **Argumentos**

Modulo: Número de serie del módulo o Local ID.

Linea: Número del canal de salida que se desea actualizar (entre 0 y 15).

Valor: Valor de la línea a actualizar (1: activar; 0: desactivar).

◇ **Valor retornado**

0 si todo fue correcto.

-1 en caso de error.

LJ_EscribeModulo_DO

◇ **Descripción**

Esta función actualiza los dieciséis canales de salida de la tarjeta RB16 conectada al módulo LabJack U12 correspondiente.

◇ **Sintaxis**

C/C++

```
int LJ_EscribeModulo_DO (int modulo,
                         U16 valor);
```

Ada

```
function LJ_EscribeModulo_DO (Modulo: Integer;
                              Valor: U16)
return Integer;
```

◇ **Argumentos**

Modulo: Número de serie o Local ID del módulo.

Valor: Entero donde estén codificados los valores de los dieciséis canales de la tarjeta RB16.

◇ **Valor retornado**

El número de canales de la tarjeta RB16 si todo fue correcto.

-1 en caso de error.

LJ_LeeLinea_DI

◇ **Descripción**

Esta función lee el estado de un canal de entrada de la tarjeta RB16 conectada al módulo LabJack U12 correspondiente.

◇ **Sintaxis**

C/C++

```
int LJ_LeeLinea_DI (int modulo,
                    int linea);
```

Ada

```
function LJ_LeeLinea_DI (Modulo: Integer;
                         Linea: Num_Linea)
return Integer;
```

◇ **Argumentos**

Modulo: Número de serie o Local ID del módulo.

Linea: Número del canal de entrada que se desea leer (entre 0 y 15).

◇ **Valor retornado**

El estado del canal de entrada (1: activo; 0: inactivo) si todo fue correcto.

Un número negativo si hubo algún error.

LJ_LeeModulo_DI

◇ **Descripción**

Esta función lee el estado de los dieciséis canales de entrada de la tarjeta RB16 conectada al módulo LabJack U12 correspondiente. El estado de las entradas se devuelve codificado en un entero que se ha de pasar a la función como argumento.

◇ **Sintaxis**

C/C++

```
int LJ_LeeModulo_DI (int modulo,
                     U16 * valor);
```

Ada

```
procedure LJ_LeeModulo_DI (Modulo: in Integer;
                           Valor: in out U16;
                           Retorno: out Integer);
```

◇ **Argumentos**

Modulo: Número de serie del módulo o Local ID.

Valor: Entero donde se codificará el estado de los dieciséis canales de la tarjeta RB16.

◇ **Valor retornado**

0 si todo fue correcto.

-1 en caso de error.

La implementación de esta función en Ada se hace a través de un procedimiento. El valor de retorno se devuelve a través de la variable *Retorno.*

LJ_EscribeLinea_AO

◇ **Descripción**

Esta función actualiza el valor de uno de los dos canales de salida analógicos de los que dispone el módulo LabJack U12 (AO0-AO1).

◇ **Sintaxis**

C/C++

```
int LJ_LeeModulo_DI (int modulo,
                     U16 * valor);
```

Ada

```
function LJ_EscribeLinea_AO (Modulo: Integer;
                             Linea: Num_Linea_Salida_Analog;
                             Valor: Valor_Salida_Analog)
return Integer;
```

◇ **Argumentos**

Modulo: Número de serie o Local ID del módulo.

Linea: Número del canal de salida analógico que se desea actualizar (0 ó 1).

Valor: Valor de la línea a actualizar (entre 0.0 y 5.0).

◇ **Valor retornado**

0 si todo fue correcto.

-1 en caso de error.

ObtenerError

◇ **Descripción**

Esta función sirve para obtener el estado en que finalizó la última operación que se ha realizado. En el epígrafe *4.3.- Anexo C* se pueden consultar todos los códigos de error que puede devolver la función *ObtenerError*.

◇ **Sintaxis**

C/C++

```
int ObtenerError (void);
```

Ada

```
function ObtenerError return Integer;
```

◇ **Valor retornado**

Un entero que identifica la situación en que finalizó la última operación.

ObtenerErrorString

◇ Descripción

Esta función sirve para obtener una descripción del estado en que finalizó la última operación realizada. Si en la última llamada a una función se ha producido un error, mediante una llamada a esta función se obtendrá una descripción de éste.

◇ Sintaxis

C/C++

```
char * ObtenerErrorString (void);
```

Ada

```
function ObtenerErrorString return string;
```

◇ Valor retornado

Una cadena con la descripción del estado en que finalizó la última operación realizada. Si en la última operación no se produjo ningún error, se retornará una cadena vacía.

4.2.- Anexo A: Descripción de las funciones propias del fabricante

InitLabjack

◇ **Descripción**

Esta función inicializa las variables necesarias del driver y se debe invocar antes de cualquier otra función.

◇ **Sintaxis**

```
int LJ_Inicializar (int modulo,
                    int tipo);
```

DigitalIO

◇ **Descripción**

Lee y escribe los veinte canales de entrada/salida digitales.

◇ **Sintaxis**

```
long DigitalIO (long *idnum,
                long demo,
                long *trisD,
                long trisIO,
                long *stateD,
                long *stateIO,
                long updateDigital,
                long *outputD);
```

◇ **Argumentos**

**idnum*: Número de serie, Local ID ó -1 para el primer módulo encontrado. Identifica el módulo sobre el que hay que ejecutar la operación.

demo: Permite emplear la función sin necesidad de tener conectados los módulos LabJack. Se debe indicar cero para operar normalmente y un valor mayor que cero para operar en modo demo.

**trisD:* Configuración de las direcciones D0-D15. 0=Input, 1=Output.

trisIO: Configuración de las direcciones IO0-IO3. 0=Input, 1=Output.

**stateD:* Estado de las líneas D0-D15.

**stateIO:* Estado de las líneas IO0-IO3.

updateDigital: Si es mayor que cero se escribirán los valores de las variables *tris* y *state*. En caso contrario, se realizará una lectura de los canales y se retornará en estas variables.

◇ **Valor retornado**

0 si todo fue correcto.

Un código de error (*Labjack errorcodes*) si se produjo algún error.

EDigitalOut

◇ **Descripción**

Esta función es una versión simplificada de la función DigitalIO, que actúa sobre un solo canal de entrada/salida.

◇ **Sintaxis**

```
long EDigitalOut ( long *idnum,
                   long demo,
                   long channel,
                   long writeD,
                   long state );
```

◇ **Argumentos**

**idnum*: Número de serie, Local ID ó -1 para el primer módulo encontrado. Identifica el módulo sobre el que hay que ejecutar la operación.

demo: Permite emplear la función sin necesidad de tener conectados los módulos LabJack. Se debe indicar cero para operar normalmente y un valor mayor que cero para operar en modo demo.

channel: Canal a actualizar. 0-3 para los canales IO, 0-15 para los canales D.

writeD: Si es mayor que cero se actualiza un canal D, sino un canal IO.

state: Si es cero el canal se activa, sino se desactiva.

◇ **Valor retornado**

0 si todo fue correcto.

Un código de error (*Labjack errorcodes*) si se produjo algún error.

EDigitalIn

◇ Descripción

Esta función es una versión simplificada de la función DigitalIO, que lee un canal de entrada/salida.

◇ Sintaxis

```
long EDigitalIn ( long *idnum,
                  long demo,
                  long channel,
                  long readD,
                  long *state );
```

◇ Argumentos

idnum: Número de serie, Local ID ó -1 para el primer módulo encontrado. Identifica el módulo sobre el que hay que ejecutar la operación.

demo: Permite emplear la función sin necesidad de tener conectados los módulos LabJack. Se debe indicar cero para operar normalmente y un valor mayor que cero para operar en modo demo.

channel: Canal a leer. 0-3 para los canales IO, 0-15 para los canales D.

readD: Si es mayor que cero se lee un canal D, sino un canal IO.

state: Variable en la que se retorna el estado del canal leído.

◇ Valor retornado

0 si todo fue correcto.

Un código de error (*Labjack errorcodes*) si se produjo algún error.

4.3.- Anexo B: Description of errorcodes

En este anexo se proporciona la descripción de los errores que puede devolver la tarjeta LabJack U12. Esta información se ha obtenido directamente del epígrafe *4.40 "Description of errorcodes"* del manual de usuario "*LabJack U12 Users Guide*".

0 – No error.
1 – Unknown error.
2 – No LabJacks found.
3 – LabJack n not found.
4 – Set USB buffer error.
5 – Open handle error.
6 – Close handle error.
7 – Invalid ID.
8 – Invalid array size or value.
9 – Invalid power index.
10 – FCDD size too big.
11 – HVC size too big.
12 – Read error.
13 – Read timeout error.
14 – Write error.
15 – Turbo error.
16 – Illegal channel index.
17 – Illegal gain index.
18 – Illegal AI command.
19 – Illegal AO command.
20 – Bits out of range.
21 – Illegal number of channels.
22 – Illegal scan rate.
23 – Illegal number of samples.
24 – AI response error.
25 – LabJack RAM checksum error.
26 – AI sequence error.
27 – Maximum number of streams.
28 – AI stream start error.
29 – PC buffer overflow.
30 – LabJack buffer overflow.
31 – Stream read timeout.
32 – Illegal number of scans.
33 – No stream was found.
40 – Illegal input.
41 – Echo error.
42 – Data echo error.
43 – Response error.
44 – Asynch read timeout error.
45 – Asynch read start bit error.
46 – Asynch read framing error.
47 – Asynch DIO config error.
48 – Caps error.
49 – Caps error.
50 – Caps error.
51 – HID number caps error.
52 – HID get attributes warning.
57 – Wrong firmware version error.
58 – DIO config error.
64 – Could not claim all LabJacks.
65 – Error releasing all LabJacks.
66 – Could not claim LabJack.
67 – Error releasing LabJack.
68 – Claimed abandoned LabJack.
69 – Local ID –1 thread stopped.
70 – Stop thread timeout.

71 – Thread termination failed.

72 – Feature handle creation error.

73 – Create mutex error.

80 – Synchronous CS state or direction error.

81 – Synchronous SCK direction error.

82 – Synchronous MISO direction error.

83 – Synchronous MOSI direction error.

89 – SHT1X CRC error.

90 – SHT1X measurement ready error.

91 – SHT1X ack error.

92 – SHT1X serial reset error.