TALLER DE LINUX DE TIEMPO REAL

CONSTRUCCIÓN DE DRIVERS

INFORME:
CONSTRUCCION DE DRIVER
DE UN MOTOR PASO A PASO (STEPPER ENGINE)

Profesor: Nelson Acosta
Alumno: Néstor Daniel Altamirano

UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES

ÍNDICE

ÍNDICE ..... **- 2 -**

INTRODUCCIÓN ..... **- 3 -**

MOTORES PASO A PASO (STEPPER ENGINES) ..... **- 4 -**

QUÉ ESPERABAMOS LOGRAR ..... **- 11 -**

CONSTRUCCIÓN DEL DRIVER ..... **- 12 -**

INSTALACIÓN Y USO DEL DRIVER ..... **- 18 -**

QUÉ LOGRAMOS ..... **- 19 -**

CONCLUSIÓN ..... **- 20 -**

REFERENCIAS ..... **- 21 -**

INTRODUCCIÓN

A medida que la popularidad del sistema operativo Linux crece, el interés por escribir controladores para dicha plataforma crece. Ella misma está diseñada hoy en día para que la mayor parte de lo que se realiza sea independiente del hardware que se utiliza y con una estructura modularizada que permite cargar cualquier controlador que se necesite sin necesidad de bootear de nuevo el nucleo, simplemente con un par de comandos. Sin embargo, aunque el usuario final puede tener la facilidad de cargar en su sistema solo las partes de software que vaya a utilizar, para cada pieza de la computadora, alguien, en algún lado, debió escribir algunas lineas de código que manejaran ese dispositivo. Esta tarea no es algo trivial, pero afortunadamente se encuentra lo suficientemente estandarizada para que cualquiera con ciertas habilidades de programación en lenguaje C y conocimiento acerca de qué es lo que hace que el núcleo del sistema funcione, logre escribir un driver que controle el dispositivo que ha construido (o que otro construyó).

Se utilizó para construir el presente trabajo práctico la distribución provista por la cátedra: ROCK & Real Time Linux. Se decidió trabajar sobre esta distribución a fin de concentrar los esfuerzos en la construcción misma del driver ya que nos proveía de un entorno ya probado y sobre el que podíamos fácilmente desarrollar y probar nuestro controlador y aplicación, la cual también fue escrita en el lenguaje C.

También se construyó la interfaz puerto paralelo – motor. Esto se logró utilizando el chip ULN 2003, el cual consiste en un arreglo de 7 transistores que regulan las señales enviadas por el controlador a través del puerto paralelo, permitiendo el paso o no de corriente desde una fuente hacia el motor. También se incorporó dicha fuente y los componentes de conexión se ensamblaron sobre una plaqueta con las correspondientes sendas de cobre que señalan las interconexiones que hacen posible su funcionamiento. Todo esto permitió el correcto funcionamiento del motor, controlando su velocidad, sentido de rotación y tipo de movimiento directamente con una combinación de teclas.

MOTORES PASO A PASO
(STEPPER ENGINES)

Un motor paso a paso es un dispositivo muy común, muy simple y ampliamente usado en miles de dispositivos electrónicos hoy en día. Desde las disqueteras hasta las impresoras, pasando por cámaras fotográficas, máquinas de fax, fotocopiadoras, discos rígidos y aplicaciones de la robótica aprovechan las capacidades de estos componentes que nos permiten regular muy precisamente el movimiento, a través de series de señales que se les envían.

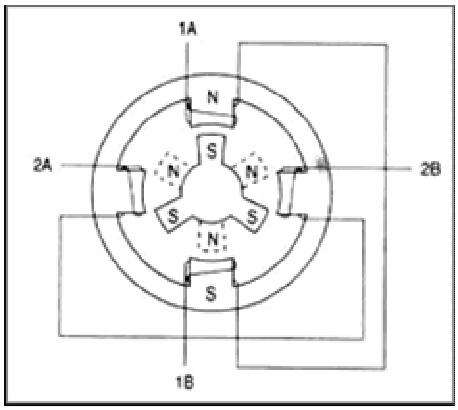
Características especiales:

- Torque de posición: Estos motores se mueven bien a bajas velocidades, y sostienen un muy buen torque, dada una posición cualquiera aplicada mediante impulsos sostenidos. Aún más, pueden mantenerse fijos aún sin energía aplicada a sus bobinas (aunque en menor grado) gracias al llamado “torque detenido”.
- Posicionamiento de ciclo abierto: No necesitamos ninguna linea de entrada que nos indique si el motor se movió o no. Si garantizamos que el torque que el motor puede generar es el suficiente para mover la carga que se le aplique, podemos estar seguros de que al enviarle la correcta secuencia de pulsos, este se moverá los pasos que le indiquemos en cualquier dirección, y mediante la frecuencia que esté dentro de los parámetros posibles. Esta es, quizá, una de las más valoradas características de este tipo de motores. Aquellos que sí poseen un retorno que nos informa la posición actual del motor se denominan “servos”, y son más complicados de controlar que los que aquí tratamos.
- Independencia de carga: La velocidad de rotación del motor es independiente de la carga, siempre que el torque sea el suficiente para prevenir que el motor “patine”, es decir que intente moverse y no lo logre. A mayores velocidades más torque es necesario para moverse, por lo que

eventualmente todos los motores paso a paso comienzan a patinar bajo la carga, por lo que los controladores que se utilizan poseen un límite superior de frecuencia para evitar que esto suceda.

Funcionamiento:

El motor paso a paso utiliza la teoría de operación de imanes para su funcionamiento, para lograr hacer que el motor se mueva una distancia precisa cuando se le aplica un impulso eléctrico a sus bobinas. En la siguiente figura podemos ver un motor con 2 bobinas estacionarias y 6 polos en el rotor. El rotor necesitará 12 pulsos de electricidad a traves de las bobinas para moverse los 12 pasos necesarios para completar una vuelta completa (30 grados por paso). La mínima cantidad de grados que se moverá el motor al aplicarle un impulso eléctrico puede ser calculada dividiendo la cantidad de grados en una revolución (360º) en la cantidad de bobinas que posee, y luego en la cantidad de polos del rotor (norte y sur, considerando todas las posiciones). En este caso se divide por 12, lo que nos da una resolución de 30º por paso.



En el caso de que no se brinde energía al motor, el magnetismo residual hará que el rotor alinee uno de sus polos con uno de los polos de las bobinas, lo que producirá una fuerza lo suficientemente fuerte para que se mantenga en su lugar. Esto es lo que produce que, al mover el eje con la mano, el motor gire haciendo “clicks”, pasando de una posición a otra.

Cuando se hace pasar electricidad, solo estará dirigida a una de las dos bobinas estacionarias, convirtiendo sus extremos en polos sur y norte. Al ocurrir esto, el polo norte de la bobina atraerá al polo opuesto del eje, más cercano a su posición, lo que provocará un pequeño movimiento (si es que el eje ya no se encontraba en la posición en la que caería) y subsecuente fijación del eje, con más resistencia

que antes, ya que ahora no se trata solo de magnetismo residual, sino que estamos generando un campo magnetico a través del enrollado de cobre.
Al cambiar el flujo de corriente a la siguiente bobina, la alineación de campo magnético cambiará 90º. Sin embargo, el motor solo se moverá 30º, ya que solo eso necesita para mover el polo opuesto más cercano. Al continuar cambiando los polos de las bobinas en la dirección horaria o antihoraria se irá moviendo el eje y sus polos siguiendo la alineación de los campos magnéticos.

Tipos de motores:

Existen dos tipos de motores de iman permanente:

- Bipolares
- Unipolares

Bipolares

Este tipo de motor requiere de un cambio de dirección en la corriente para ser controlado, por lo que suele ser más complicado de controlar, aunque más eficiente que el unipolar que mencionaremos luego. Esto último viene dado por las secuencias que se usan para su funcionamiento, relacionado con la naturaleza misma del motor y sus bobinas. Generalmente poseen 4 cables de salida.

Unipolares

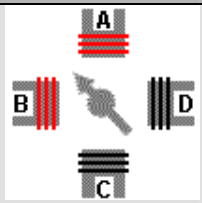
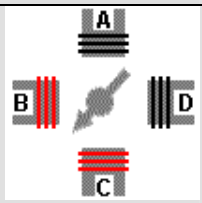
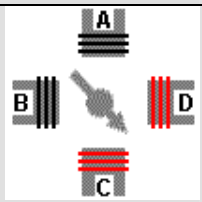
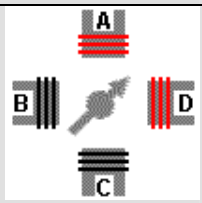
Estos motores suelen tener 6 o 5 cables de salida dependiendo de su conexionado interno. Son más fáciles de controlar, ya que solo dependen de proveer las combinaciones de descargas en secuencia apropiadas para que la corriente circule y polarice el motor, lo cual puede ser realizado por un simple arreglo de transistores.

Cada uno de estos tipos de motores posee 3 secuencias de funcionamiento que describiremos a continuación. Recordemos que mientras el motor unipolar posee una salida para cada bobina y la entrada de la corriente se realiza por un cable

aparte, el motor bipolar funciona invirtiendo la dirección de salida de la corriente por cada par de bobinas. Así, en el bipolar, si la corriente entra por A, debe salir por C, o viceversa, pero nunca entrar por ambas a la vez. Esto tampoco tiene sentido hacerlo en el unipolar, pero nada nos impide intentar poner a tierra ambas conexiones, el motor simplemente no se moverá ya que se cancelaran mutuamente las fuerzas. Sí se permite que ambas se encuentren desconectadas.

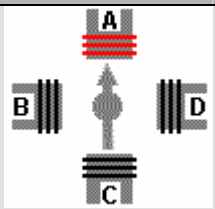
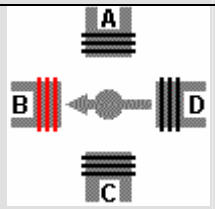
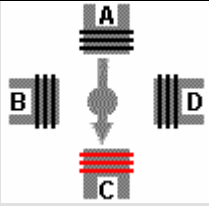
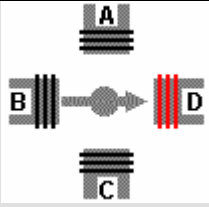
Secuencia normal

En esta secuencia se activan dos bobinas por cada paso, con lo que se obtiene un alto poder de torque y de retención. Es la que recomiendan los fabricantes y la más utilizada.

PASO	Bobina A	Bobina B	Bobina C	Bobina D	
1	NORTE	NORTE	SUR	SUR	
2	SUR	NORTE	NORTE	SUR	
3	SUR	SUR	NORTE	NORTE	
4	NORTE	SUR	SUR	NORTE	

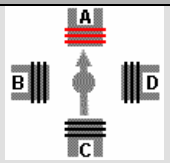
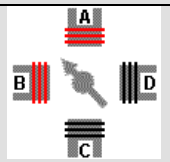
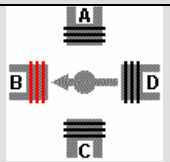
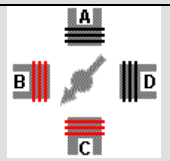
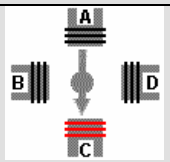
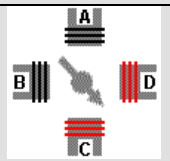
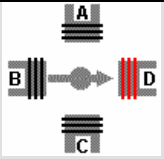
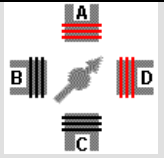
Secuencia Wave-Drive

En esta secuencia solo se energiza una bobina a la vez, lo que resulta en un menor torque y menor retencion. La ventaja es que el movimiento es más suave en algunos motores que la anterior secuencia, pero no tanto como en la siguiente que veremos.

PASO	Bobina A	Bobina B	Bobina C	Bobina D	
1	NORTE	SUR	SUR	SUR	
2	SUR	NORTE	SUR	SUR	
3	SUR	SUR	NORTE	SUR	
4	SUR	SUR	SUR	NORTE	

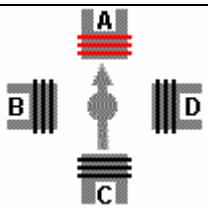
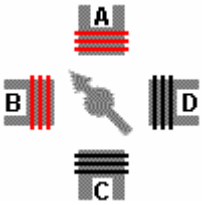
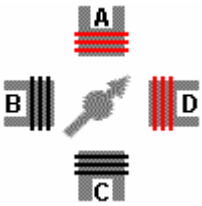
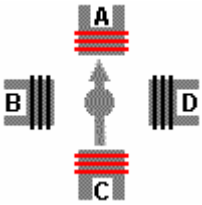
Secuencia Half-Step

En esta secuencia se activan las bobinas para brindar un movimiento de medio paso, polarizando primero una, luego dos, de nuevo solo una, y así sucesivamente. Esto nos brinda mayor precisión en el movimiento del motor.

PASO	Bobina A	Bobina B	Bobina C	Bobina D	
1	NORTE	SUR	SUR	SUR	
2	NORTE	NORTE	SUR	SUR	
3	SUR	NORTE	SUR	SUR	
4	SUR	NORTE	NORTE	SUR	
5	SUR	SUR	NORTE	SUR	
6	SUR	SUR	NORTE	NORTE	
7	SUR	SUR	SUR	NORTE	
8	NORTE	SUR	SUR	NORTE	

Como comentario final de esta sección, podemos brindar una técnica que nos permitirá detectar la secuencia correcta de cables para un motor que no conocemos o no tenemos su hoja de datos (esto fue útil en el presente práctico, ya que no se conocía ningún dato previo del dispositivo).

Para lograr identificar los cables debemos previamente aplicar un voltaje al cable común (en nuestro caso, 12 volts). Luego, seguiremos los siguientes pasos:

Seleccionar uno de los cables y conectarlo a masa. Denominaremos a este cable **A**	
Tomando cada uno de los restantes 3 cables, probar hasta identificar cual produce un movimiento en sentido antihorario. Este cable se llamará **B**	
Desconectando el cable anterior y poniendo a masa cualquiera de los otros dos, identificar el que produce un movimiento horario. Este será el cable **D**	
Por descarte, el único de los cables que queda es el **C**. Este cable no debe producir ningún tipo de movimiento ya que es la bobina opuesta a la bobina **A**.	

Realizados cada uno de estos pasos habremos identificado cada uno de los cables de control y podremos luego proceder a su conexión de forma tal que las señales que lleguen al motor sean las apropiadas para el movimiento requerido.

QUÉ ESPERABAMOS LOGRAR

Las ambiciones de este proyecto estaban puestas muy alto. Se quería construir un controlador que fuera posible instalarlo fácilmente en cualquier distribución, así como su aplicación que lo controlaba. Esta iba a estar escrita en TCL/TK, un lenguaje de scripting con un entorno visual, e iba a permitir manejar cada aspecto que fuera posible del motor.

Asimismo, se construiría la plaqueta con todos los componentes que fueran necesarios para que, al conectar el motor paso a paso y enchufar el cable al puerto paralelo de la computadora, las señales del driver lograran llegar correctamente al dispositivo, así como la necesaria energía desde la fuente para su movimiento.

Estos objetivos fueron con los que se inició este proyecto. Sin embargo, luego de demasiados intentos infructuosos para lograrlos, se debió modificar algunos de ellos para así poder finalizar el proyecto. Esto, aunque no nos permitió presentar un proyecto como nos hubiera gustado, no impidió que se lograrán un mejor entendimiento del funcionamiento tanto de los componentes electrónicos como del mismo sistema operativo y de los diferentes inconvenientes que surgen del diseño y construcción de drivers bajo un cierto entorno.

CONSTRUCCIÓN DEL DRIVER

Los drivers de linux se componen de ciertas funciones básicas, las cuales deben estar presentes para que mínimamente el comportamiento sea el correspondiente a un controlador. También el tipo de dispositivo debe caer dentro de una de las dos categorías siguientes (también existe la categoría de interfaz de red, pero no la consideraremos aquí):

- **Carácter:** Se denomina así a un dispositivo que puede ser accedido como si fuera un flujo de bytes, uno por vez. El controlador que maneja este tipo de dispositivo usualmente implementa las funciones *open, close, read, write* y alguna más que veremos luego. Poseen similitud con los archivos, pero a diferencia de ellos, en este tipo de dispositivos no podemos volver hacia atrás en el flujo de bytes, ya que funcionan más como un canal de información.
- **Bloque:** Este tipo de dispositivos puede tener asignado un disco si se necesitara. Generalmente solo se pueden acceder leyendo de él múltiplos de un *bloque*.

Para el usuario final estos detalles son transparentes: no hay diferencia entre un dispositivo de carácter o de bloque. Sin embargo, las diferencias persisten y es en ese momento que cobra aún más importancia el código que se encuentra entre el programa que utiliza el usuario y el pedazo de silicio y metal que intentamos que se mueva. En nuestro caso, el driver será uno de carácter, ya que así se considera a las señales que enviaremos a través del puerto paralelo.
Bien, luego de esta breve aclaración, las funciones que se implementaron para el manejo del motor paso a paso fueron:

- **void** *ctrl_motor_tim()*: Esta función se utilizó como la que haría funcionar el motor a intervalos regulares sin ninguna intervención externa (salvo su interrupción). Su funcionamiento es extremadamente simple, ya que

depende de los estados de variables globales y semáforos que guíen a una serie de condicionales, los cuales a su vez poseen como bloque de código a las señales que se envían al motor, también a su vez controladas por tipo de secuencia y sentido de giro. Luego, cada una de las iteraciones de esta función envía una señal por el puerto paralelo, la cual permite que el motor se mueva a una nueva posición, y según las variables, se prepara para el siguiente movimiento en las iteraciones. Es el tiempo entre diferentes llamados lo que hace que se mueva más lento o rápido, lo que se logró mediante timers que, al finalizar cada ejecución de la función, se seteaba un reloj que al cabo de una equis cantidad de pulsos disparara a la misma función, una y otra vez. Esta función de librería la consideraremos luego de explicar las otras funciones que se implementaron (que no son demasiadas ni mucho más complicadas).

- **Dos funciones** que un driver debe implementar son *init_module()* y *cleanup_module()*. Como sus nombres lo indican, sus funciones son setear los valores iniciales de las variables al ser cargadas, y remover cualquier instalación que se haya realizado en el sistema, respectivamente. La función *init_module()* comprueba que el puerto al que se desea acceder se encuentre disponible, y de ser así lo registra para uso propio. Registra el número *Major*, que será el identificador del tipo de dispositivo dentro del sistema, y por último (en nuestro caso particular) inicializa la variable del semáforo, la cual se utilizará para sincronizar los llamados al driver, de modo que dos funciones no actúen sobre la misma variable al mismo tiempo, dejándola en un estado inconsistente. Luego, la función *cleanup_module()* detiene cualquier ejecución de la función *ctrl_motor_tim()* que pudiera haber estado corriendo en ese momento, envía al puerto paralelo los valores nulos, para evitar que el motor continúe con alguna de sus bobinas activa, devuelve el número *Major* desregistrandolo, así como el puerto que estabamos utilizando.
- **Las funciones** *motor_open, motor_release y motor_write* se encuentran principalmente por convención. No son necesarias en nuestro caso, aunque

se podría haber brindado la funcionalidad de poder ingresar los valores a pasar al puerto paralelo mediante la función write. Ni siquiera se puso por convención la función read, ya que no tiene sentido. Las funciones open y release podrían utilizarse si hubiera muchas aplicaciones utilizando el driver al mismo tiempo, de manera de mantener un contador que nos indique si podemos bajar el driver del nucleo o no, ya que al estar alguna aplicación aún utilizando el controlador, esta quedaría en un estado que podría dar lugar a cuelgues de la aplicación o incluso de todo el sistema.

- **int motor_ioctl()** es nuestra función más utilizada. Mediante ella podemos manejar cada aspecto de nuestro dispositivo a través de dos parámetros que son el número del comando que queremos activar y, opcionalmente, un parámetro extra que indica un valor a setear en una cierta variable del controlador. Los diferentes comandos (los valores elegidos para representarlos son completamente arbitrarios) que tenemos en nuestra función son los siguientes:
  - **cmd = 10:** Setea la variable entera *clockwise* en cero, lo que indica que el tipo de movimiento será a favor de las agujas del reloj, horario, lo que significa que nos moveremos hacia la derecha dentro del arreglo de posiciones que tenemos almacenado, sea cual sea el tipo de movimiento.
  - **cmd = 20:** Setea la variable entera *clockwise* en uno, lo que indica que el tipo de movimiento será contrario al de las agujas del reloj, antihorario, lo que significa que nos moveremos hacia la izquierda dentro del arreglo de posiciones que tenemos almacenado, sea cual sea el tipo de movimiento.
  - **cmd = 30:** Selección de la secuencia NORMAL. Este tipo de secuencia se encuentra almacenada en el arreglo *norm_secuence*, el cual nos indica a cada momento cual es el siguiente valor que debemos enviar a través del puerto, mediante la variable *position*. Identificamos que debemos utilizar esta variable en conjunción con ese arreglo a través de verificar que el valor de la tercera variable,

norm_sec, se encuentre en un valor distinto de cero. En caso de que se encuentre seleccionado uno de los otros dos tipos de movimiento, se realizará una búsqueda de la posición que más se parezca a la siguiente que debería enviarse en el movimiento que ya se encuentra realizándose, y considerando un movimiento horario o antihorario (variable *clockwise*). Para esto analizamos el arreglo *hstep_secuence*, el cual corresponde al movimiento Half-Step o de Medio Paso, y que contiene los valores intercalados de las otras dos secuencias. Al principio y al final de todos estos pasos, debemos deshabilitar y habilitar el semáforo *stop_mutex*, para así evitar el conflicto con la función *ctrl_motor_tim* si llegara a estar ejecutándose en ese momento.

- **cmd = 40:** Se selecciona la secuencia WAVE-DRIVE. Identico al anterior comando, solo que se selecciona esta secuencia al finalizar.
- **cmd = 50:** Se elige la secuencia HALF-STEP, o medio paso. Es mucho más fácil cambiar a esta secuencia desde una de las otras, ya que sólo se debe buscar la siguiente posición en el arreglo del movimiento que ya se encuentra realizándose, y luego ubicar ese mismo valor en nuestro arreglo del nuevo movimiento. Si vemos el código podremos observar que son muchas menos líneas y menos consideraciones las que debemos tener. Igualmente, se debe manejar la variable *stop_mutex* como siempre para evitar conflictos entre funciones.
- **cmd = 60:** Mueve nuestro dispositivo a la siguiente posición, dependiendo del tipo de movimiento que se haya seleccionado. En caso de que no haya sido elegido uno, o que el motor se encuentre moviéndose en una secuencia mediante la función *ctrl_motor_tim*, no hará nada. Luego de enviar la señal correspondiente, posicionará la variable *position* en el siguiente valor que se debería enviar. De no haber ninguna secuencia definida, devolverá el código de error de valor 1.

- **cmd = 70:** Si se ha seleccionado una posición previamente, este comando iniciará el movimiento mediante la función *ctrl_motor_tim*. Para ello se igualará a cero la variable *stopped* a fin de indicar que la función se encuentra en ejecución, se inicializará el timer *ctrl_tim*, se le pasará el nombre de la función y también el tiempo en el cual el temporizador debe activarse, se agregará este timer a una cola de ejecución y por último se habilitará el semáforo *stop_mutex*. El tiempo en el cual la función se ejecutará dependerá de la cantidad de pulsos de reloj con que se haya seteado la variable entera *delay*. Al imprimir la variable HZ, que está en la librería delay.h, nos informó que hay 100 pulsos por cada segundo, por lo que la mayor resolución que se logrará será de un paso por cada centésima de segundo. Si no hay ninguna secuencia definida, devuelve el código de error de valor 1.
- **cmd = 80:** Detiene cualquier secuencia que se encuentre en ejecución mediante la función *ctrl_motor_tim* al eliminar el próximo evento del timer *ctrl_tim* de la cola de ejecución. De no encontrarse ninguna secuencia en ejecución, devuelve el código de error de valor 3.
- **cmd = 90:** Permite darle un valor a la variable *delay*, la cual define la cantidad de pulsos de reloj que pasarán antes de que la función *ctrl_motor_tim* vuelva a ejecutarse. De pasar un valor negativo como argumento, por defecto la variable se asigna con valor 1.
- **Cualquier otro valor de cmd:** Por defecto se devuelve un valor 4, indicando una opción no válida.

Funciones de Timers

La librería *delay.h* nos brinda funciones que permiten encolar una tarea en el scheduler del sistema, para que, con un tiempo arbitrario de vencimiento, dicha tarea se ejecute automáticamente. Luego de incluir dicha librería, todo lo que se debe hacer es utilizarla, definiendo para ello una variable de tipo *timer_list*, y

mediante llamadas a la función *init_timer* que recibe a esta misma variable como parámetro, podemos luego asignar la función y el tiempo que debe pasar antes de que se ejecute. Aquí podemos ver una parte de nuestro código como ejemplo:

```
init_timer(&ctrl_tim);
ctrl_tim.function = ctrl_motor_tim;
ctrl_tim.expires = jiffies + delay;
add_timer(&ctrl_tim);
```

Al llamar a la función *add_timer()* estamos agregando nuestro timer a la cola de ejecución del scheduler. Como se puede ver, a fin de definir el tiempo en el cual se debe ejecutar nuestra tarea, debemos sumar la variable *jiffies* (indica la cantidad de pulsos del mother desde que se encendió la computadora) más la variable *delay* (que indica la cantidad de pulsos que pasarán antes de que nuestra función se ejecute), lo que nos da como total un número que indica al sistema la cantidad de pulsos que pasarán desde que se encendió la máquina, y señalando el tiempo futuro de la ejecución de nuestra función.

INSTALACIÓN Y USO DEL DRIVER

Para su utilización se construyó el archivo *makefile*, que acompaña los fuentes provistos. Mediante dos comandos, *make* y *make install*, se puede compilar e instalar en nuestro sistema el controlador y la aplicación para probarlo correspondiente. El comando *make* llama a otros comandos que hemos ubicado en el script de instalación, los cuales compilan el driver, la aplicación, crean los nodos (archivos mediante los cuales se accede a los dispositivos) necesarios para su utilización y suben al nucleo el módulo, exportando así su funcionamiento y dejándolo preparado para que cualquier aplicación que desee utilizarlo así lo haga. Por defecto, se utilizaron ciertos nombres para la aplicación y el driver respectivamente, y números de major arbitrarios que se sabía estaban disponibles en la distribución utilizada. Estos pueden ser fácilmente modificados editando el archivo *makefile*. Los valores elegidos se presentan a continuación:

Aplicación: *app* = stmtrctrl
Archivo nodo: *node_name* = stmtrdrv
Nombre del módulo: *module_name* = stmtrdrv
Major: *major_number* = 253

Códigos de error:

Si bien se mencionaron anteriormente, es conveniente ubicarlos en una sección de este informe para su mayor claridad. A continuación presentamos los códigos de error que se podrían recibir desde las diversas funciones del módulo, en caso de que se decidiera no utilizar la aplicación provista y construir la propia:

0: *Procedimiento completado exitosamente.*
-10: *La secuencia no ha sido definida apropiadamente.*
-20: *El motor se encuentra en movimiento mediante una secuencia.*
-30: *El motor no se encuentra en movimiento mediante una secuencia.*
-40: *Opción ioctl() inválida.*

QUÉ LOGRAMOS

Aunque el proyecto no debio ser demasiado complicado ya que más allá de tener que fabricar la plaqueta, todas las herramientas estaban disponibles, se extendió en el tiempo porque se deseaba que el driver fuera lo más portable posible. Con este fin, se instaló la distribución Debian “Sarge” en una PC y se procedió a escribir el driver sobre esta plataforma, solo para descubrir que a la hora de probarlo el driver no compilaba por falta de librerías apropiadas, vínculos simbólicos, etc. Se buscó infructuosamente bibliografía e información sobre la forma de compilar un driver y también de configurar apropiadamente el sistema, y no se halló más que la misma información que la cátedra ya había proporcionado, incluso en libros como “Linux Device Drivers” de O’Reilly.

También se intentó escribir la aplicación en TCL. Sin embargo, por más que se recorrió la web y Google buscando alguna referencia a cómo utilizar una llamada a la función IOCTL desde un simple script perteneciente a este lenguaje, no se logró hallar ninguna información que permitiera construirla. Una alternativa que se barajó fue la de utilizar un programa C en el medio que permitiera comunicar el script TCL/TK con el driver, pero se descartó por cuestiones ya de tiempo y de que al utilizar ROCK & Real Time Linux, la programación en TK quedó descartada al no poder hacer funcionar el entorno gráfico en la computadora que se utilizó para desarrollo, ni tampoco instalar el software que permitiera construir la aplicación gráfica.

CONCLUSIÓN

Si bien, como dijimos al comienzo del presente informe, la construcción de módulos de controlador para este sistema operativo llamado Linux se encuentra definida y estandarizada mayormente, las dificultades propias del sistema en si mismo fueron el mayor obstáculo en nuestro desarrollo. La poca información acerca de la ubicación, versión, etc, de las librerías necesarias para la compilación del driver fueron la razón por la que las metas que inicialmente se habían puesto se vieran modificadas en el camino, con el único afan de llegar a buen puerto.

Como se dijo, se decidió compilar y utilizar nuestro driver desde la distribución ya probada de ROCK & Real Time Linux, la que nos permitió muy fácilmente probar que la programación del mismo había sido exitosa, aunque no nos permitió utilizar el entorno gráfico TK para el desarrollo de la aplicación que lo utilizara. Sin embargo, dado que la estructura del driver es la correcta, no se descarta que mediante la apropiada configuración de otro sistema, el mismo driver funcione sin la más mínima modificación a su código fuente. Con este anhelo y la mira puesta en continuar la investigación del sistema operativo Linux por propia cuenta, a fin de lograr el correcto funcionamiento del controlador y de futuros proyectos en esta línea, se dio por finalizada la construcción del presente driver.

REFERENCIAS

"***Understanding the Linux Kernel, 3rd Edition***",
Daniel P. Bovet, Marco Cesati

"***Write a linux hardware device driver***",
Andrew O'Shaughnessy. UnixWorld. 2001.

"The Linux kernel module programming guide",
Peter Jay Salzman

"The Linux Kernel"
David A. Rusling

"***Writing linux device drivers***",
Michael K. Johnson. Spring of DECUS'95, Washington DC, 1995.

"***Linux Device Drivers***",
Alessandro Rubini. Ed. O'Reilly. Feb. 1998.

"***Device drivers***",
Michael K. Johnson. 1996.