Revista Cubana de Ciencias Informáticas
Vol. 8, No. 2, 2014
ISSN: 1994-1536 | RNPS: 0547
Pág. 35-51
http://rcci.uci.cu

Tipo de artículo: Artículo original
Temática: Software libre
Recibido: 14/03/2014 | Aceptado: 9/04/14

Desarrollo de un driver GNU/Linux para sistemas de adquisición de datos embebidos

GNU/Linux driver development for embedded data acquisition systems

Adrian Iglesias-Benitez[1*], Abel Toledano-Hernández[1], Dainelys Toledo-Enriquez[1], Jorge Gentile Martínez-Casado[1], Dennys Julián González-Aguilera[1]

[1]Centro de Inmunoensayo, Calle 134, Ave. 25, Cubanacán, Playa, La Habana, Cuba. CP.: 11600
[*]Autor para correspondencia: inpipeta1@cie.sld.cu

Resumen

En el presente trabajo se aborda el desarrollo de un driver para el manejo de un sistema de adquisición de datos controlado por la computadora de placa única MINI2440 basada en el procesador RISC de 32 bits S3C2440 de Samsung, equipado con un núcleo ARM920T sobre el cual se ejecuta el sistema operativo embebido GNU/Linux. Se pretende ofrecer una visión global del sistema describiendo sus componentes de hardware y software. Se hace énfasis en la utilización de las estructuras, funciones y primitivas de bloqueo y sincronización que ofrece GNU/Linux para resolver problemas del tipo productor/consumidor.

Palabras clave: Drivers, GNU/Linux, sistemas embebidos, sistemas de adquisición de datos.

Abstract

The present work focuses on the development of a driver to handle a data acquisition system controlled by the MINI2440 single board computer based on the S3C2440 RISC processor from Samsung and equipped with an ARM920T 32 bits kernel, in which an embedded GNU/Linux operating system is executed. A global vision of the system is presented and the hardware and software components are described. Emphasis is made in the use of

structures, functions and blocking and synchronization primitives GNU/Linux offers to solve problems of the producer´s/consumer´s kind.

Keywords: *Data acquisition systems, drivers, embedded systems, GNU/Linux.*

Introducción

El Centro de Inmunoensayo, tiene como misión principal la investigación, desarrollo y producción de estrategias y tecnologías que permitan el pesquisaje de forma económica y científicamente sustentable de enfermedades metabólicas, transmisibles y crónicas no transmisibles, a ciclo completo, incluyendo la exportación y asistencia técnica de los laboratorios y personal que las emplean. El Centro de Inmunoensayo y la tecnología SUMA (Sistema Ultra Micro Analítico) se han desarrollado y perfeccionado en función de los requerimientos del Sistema de Salud Cubano, fortaleciendo el esquema de atención primaria al brindar la posibilidad de realizar estudios a escala masiva, cubriendo el cien por ciento de los programas de Pesquisa Neonatal, Certificación de Sangre, Vigilancia Epidemiológica y de Promoción, Prevención y Control de Enfermedades Metabólicas y Crónicas No Transmisibles. La Tecnología SUMA constituye un sistema, a escala de laboratorio, de diagnosticadores y equipamiento instrumental.

Entre los equipos médicos desarrollados para la tecnología SUMA los sistemas de adquisición de datos juegan un papel fundamental. Tradicionalmente los sistemas de adquisición de datos se han desarrollado sobre microcontroladores de 8 bits. Sin embargo, cada vez más, se hace necesario, para la inserción en el mercado internacional de los equipos producidos, tener en cuenta factores como: la conectividad a través de protocolos universales y de alta velocidad, implementar la interacción con el usuario a través de interfaces táctiles, así como aumentar las capacidades para el procesamiento digital de las señales adquiridas. Sin embargo, también es necesario mantener los precios suficientemente bajos de forma que resulten competitivos. Bajo estas imposiciones, el problema está dado porque los recursos de los microcontroladores de 8 bits, aunque son baratos, no resultan suficientes; y los precios de las computadoras tradicionales de propósito general, aunque son muy potentes, resultan excesivos. Hipotéticamente, se puede solucionar este problema migrando los sistemas de adquisición de datos a sistemas embebidos de 32 bits, que al estar destinados a realizar una tarea específica con recursos limitados, se encuentren en un punto medio entre estas dos alternativas y ofrecen las facilidades que se necesitan a precios razonables (Sloss, 2004).

Los procesadores basados en núcleos ARM (del inglés "*Advanced RISC Machine*") de 32 bits han tenido gran éxito en el ámbito de los sistemas embebidos (Sloss, 2004). Basan su funcionamiento en una arquitectura, principalmente, RISC donde instrucciones muy simples, pero poderosas, se ejecutan generalmente en un solo ciclo de máquina a altas velocidades. A menudo en los sistemas embebidos, el mismo chip que contiene el núcleo ARM realiza también otras funciones más allá del procesado convencional de instrucciones secuenciales, y se denominan SOC (del inglés "*Systemon Chip*") (Wolf, 2008). Sin embargo el aumento de las facilidades ofrecidas por el hardware conlleva al aumento de la complejidad para administrar estos recursos, resultando en la necesidad de adopción de algún sistema operativo que sirva de interfaz entre el desarrollador y la plataforma de desarrollo.

El sistema operativo GNU/Linux aplicado a sistemas embebidos ha ido ganando progresivamente el interés del sector científico e industrial (Hallinan, 2011) al ofrecer una alternativa libre y fiable, así como el apoyo de una activa comunidad de programadores. Entre las numerosas ventajas que han popularizado al sistema operativo GNU/Linux para el desarrollo de sistemas embebidos cabe citar la gran variedad de arquitecturas de hardware que soporta, la disponibilidad de un gran número de aplicaciones, bibliotecas de funciones y protocolos de comunicación, la posibilidad de contar con el código fuente, así como su escalabilidad, pudiendo adaptarse tanto a pequeños dispositivos de consumo como a grandes supercomputadoras (Jones, 2012a; Sunny, 2012). Sin embargo, a la vez que GNU/Linux administra los recursos del hardware desde un modo privilegiado de ejecución, también los protege de accesos no autorizados (Bhaira, 2013). Para que las aplicaciones de usuario puedan hacer uso de todas las facilidades que brinda el sistema, es necesario el desarrollo de drivers de dispositivo (Mauerer, 2008). Puede plantearse entonces, como objetivo central de esta investigación el desarrollo de programas en GNU/Linux para el manejo de tarjetas de adquisición datos como una vía para obtener sistemas embebidos basados en microprocesadores ARM con mayores facilidades de conexión, interfaces más amigables al usuario y mayor potencia de cómputo.

Materiales y métodos

Se entiende por sistemas embebidos una combinación de hardware y software diseñado para tener una función específica. Las principales características de un sistema embebido son el bajo costo y consumo de potencia. Generalmente, los sistemas embebidos emplean procesadores muy básicos, relativamente lentos y memorias pequeñas para minimizar los costos. Los sistemas embebidos deben reaccionar ante estímulos provenientes del ambiente, respondiendo, en muchos casos, con fuertes restricciones de tiempo. En el caso de una información que llega de

Revista Cubana de Ciencias Informáticas
Vol. 8, No. 2, 2014
ISSN: 1994-1536 | RNPS: 0547
Pág. 35-51
http://rcci.uci.cu

forma periódica, los tiempos de adquisición y tratamiento de la señal deben ser inferiores al período de actualización de dicha información (Bhaira, 2013).

Plataforma MINI2440

La plataforma MINI2440 está constituida por una computadora de placa única (SBC, del inglés "*Single Board Computer*") (Abbott, 2011), que representa un excelente compromiso entre costo y desempeño, y la cadena de herramientas necesarias para el desarrollo de aplicaciones que se ejecutan sobre ella. Está basada en el microprocesador de Samsung S3C2440 (Samsung, 2004) equipado con un núcleo ARM9 de 32 bits que opera a una frecuencia de hasta 533MHz y un conjunto de periféricos que la hacen ideal para el desarrollo de sistemas embebidos. En la Figura 1 pueden observarse los principales componentes que la integran. La plataforma MINI2440 está plenamente soportada por las versiones más recientes del "*kernel*" del sistema operativo GNU/Linux (Jones, 2012b).

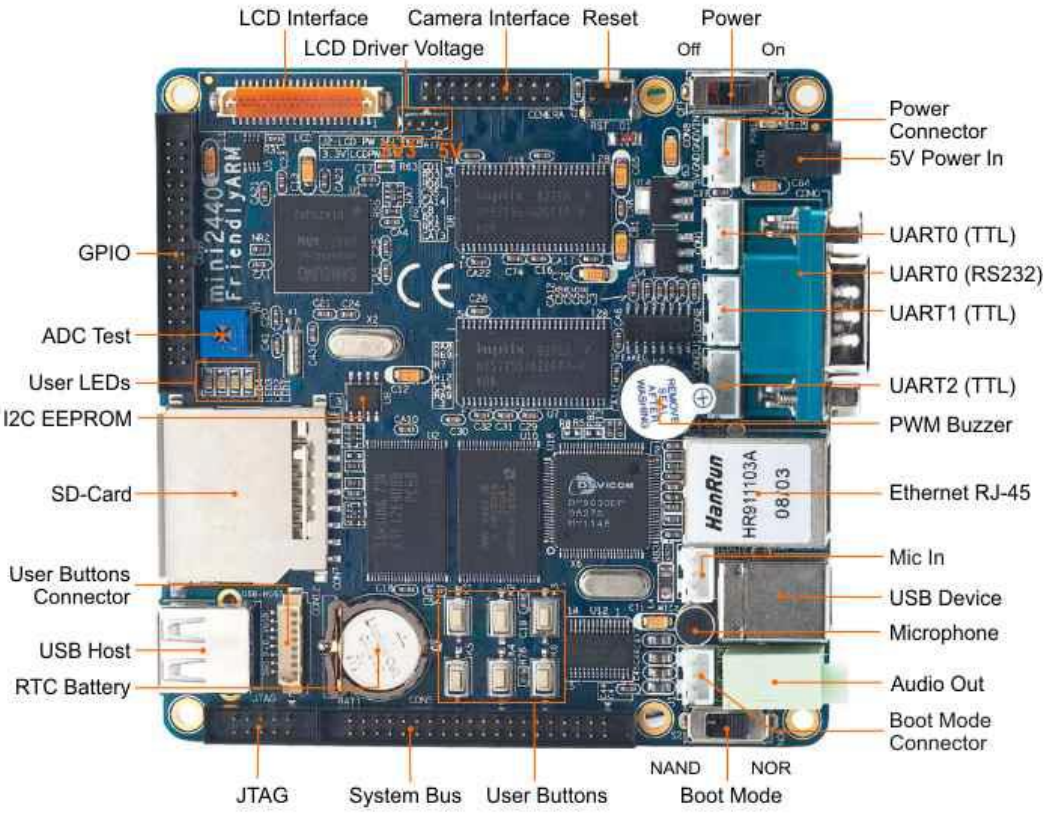

Figura 1. Localización de los componentes que integran la plataforma MINI2440.

Revista Cubana de Ciencias Informáticas
Vol. 8, No. 2, 2014
ISSN: 1994-1536 | RNPS: 0547
Pág. 35-51
http://rcci.uci.cu

En general, la SBC MINI2440 presenta interfaces de comunicación que incluyen: un puerto Ethernet RJ45 de 100MBits, 3 puertos serie RS232, 1 puerto USB maestro, 1 puerto USB esclavo, un conector JTAG, soporte para memorias MMC/SD, conexión con otros dispositivos a través los protocolos IIC y SPI, así como facilidades para la integración con un "*display*" gráfico RGB táctil con resolución de hasta 800 por 600 píxeles. El procesador S3C2440 dispone además de un controlador de interrupciones, 4 temporizadores y un bus de expansión que permite la conexión de módulos hardware diseñados a la medida de las necesidades del sistema embebido (Samsung, 2004).

La plataforma MINI2440 utiliza 64MB de SDRAM (Memoria dinámica sincrónica de acceso aleatorio) como memoria principal del sistema, distribuidos en dos bancos de memoria de 32MB; y 128MB de memoria *flash* tipo NAND que puede utilizarse como soporte de almacenamiento masivo. La memoria *flash* puede verse como un disco duro de estado sólido capaz de almacenar un gran volumen de información en un pequeño encapsulado, que al no contener partes móviles resulta muy robusto e ideal para entornos industriales. La memoria *flash* se divide de forma que en ella coexisten el cargador del programa de arranque (en inglés "*bootloader*"), el sistema operativo y el sistema de ficheros, como se muestra en la Figura 2.

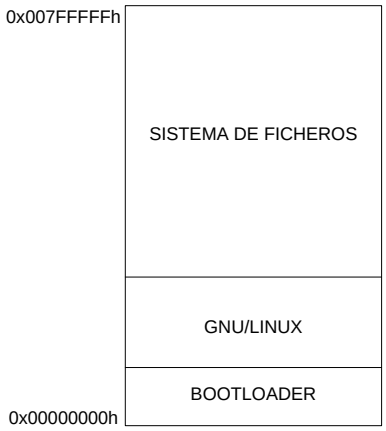

Figura 2. Particionado de la memoria flash NAND de 128MB.

En los procesadores ARM, la memoria y los puertos de entrada/salida comparten el mismo espacio de direcciones y son tratados exactamente de la misma forma por el procesador.

El sistema de adquisición de datos

Se define la adquisición de datos como el proceso mediante el cual un fenómeno físico del mundo real es transformado en una señal eléctrica que es medida y convertida a un formato digital para su posterior procesamiento, análisis y almacenamiento. Un sistema de adquisición de datos se construye alrededor de un elemento central de procesamiento, integrando diferentes componentes como por ejemplo: sensores, acondicionadores de señal, convertidores analógicos/digitales y el software de adquisición de datos (Dursun, 2013).

En la Figura 3 se muestra el diagrama en bloques de un sistema de adquisición de datos, donde se ha asumido que las señales analógicas de interés ya han sido correctamente filtradas y acondicionadas al rango de entrada del convertidor analógico/digital. En este caso, el convertidor analógico/digital utilizado fue el AD7938 (Analog-Devices, 2011), el cual trabaja por aproximaciones sucesivas y tiene una resolución de 12 *bits*. El mismo se conectó al bus de expansión del procesador por medio de líneas de direcciones, de datos y de control. Este dispositivo necesita al menos 13 pulsos de una señal de reloj externa para realizar una conversión. Al aplicar un cero lógico al terminal “CONVST” se inicia el proceso de conversión y se activa en nivel alto la señal “BUSY”, que permanece en este estado hasta que en el bus de datos del convertidor aparece un dato digital válido, regresando nuevamente al nivel lógico cero.

Revista Cubana de Ciencias Informáticas
Vol. 8, No. 2, 2014
ISSN: 1994-1536 | RNPS: 0547
Pág. 35-51
http://rcci.uci.cu

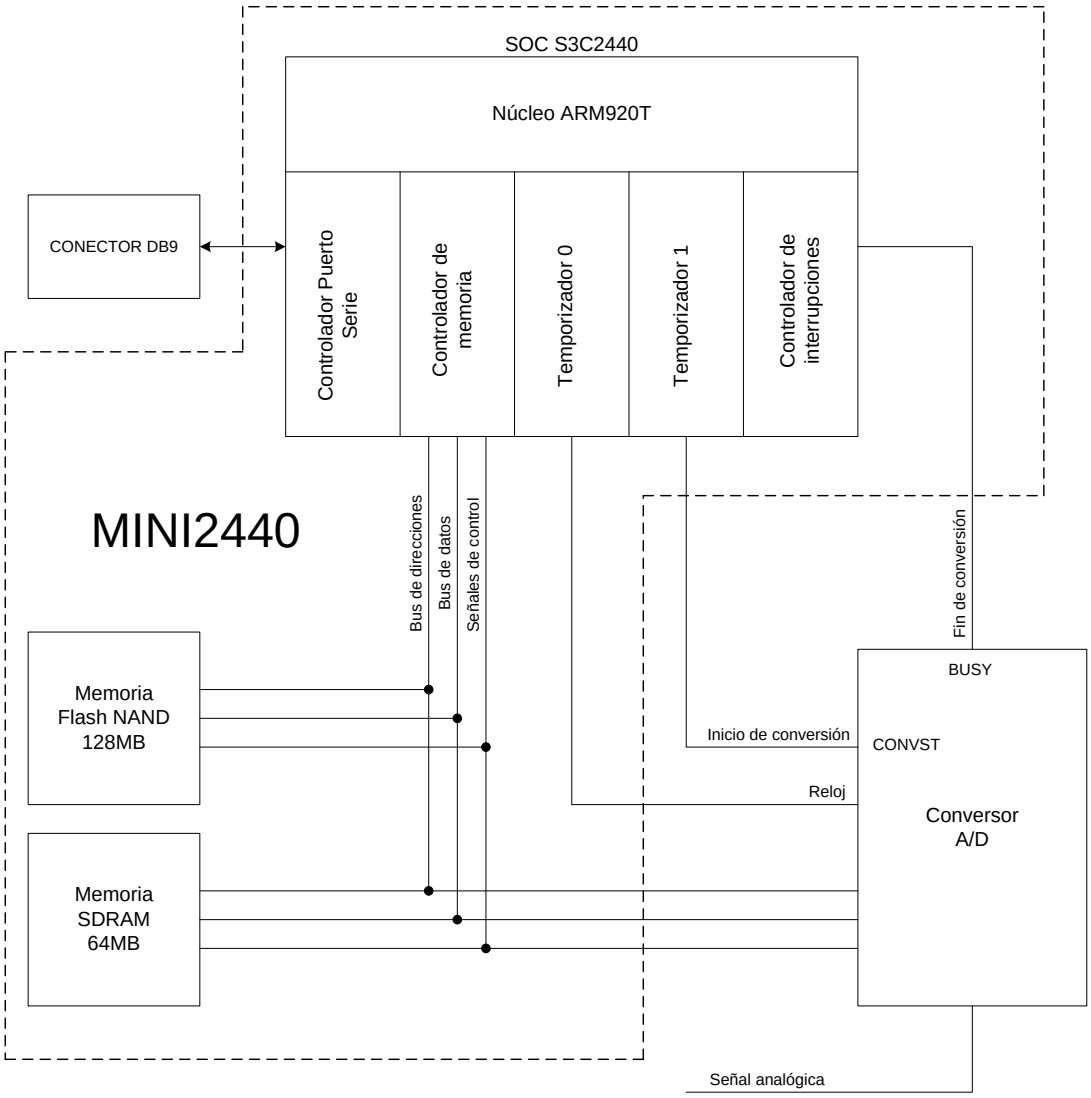

Figura 3. Diagrama en bloques del sistema de adquisición de datos.

El flanco de bajada que se produce en el terminal BUSY del convertidor analógico/digital se utiliza para solicitar la atención del procesador a través de una de las líneas de interrupción externa disponibles. Para generar las señales de reloj y los pulsos periódicos que indican el inicio de conversión se utilizan dos de los temporizadores de *hardware* que vienen integrados junto al procesador. De esta forma se garantiza que la adquisición de los datos se realice a intervalos regulares y uniformes. La estructura de los temporizadores de hardware se muestra en la Figura 4. Los temporizadores 0 y 1, que fueron los utilizados en el diseño, comparten un pre-escalador de 8 bits y los

Revista Cubana de Ciencias Informáticas
Vol. 8, No. 2, 2014
ISSN: 1994-1536 | RNPS: 0547
Pág. 35-51
http://rcci.uci.cu

temporizadores 3, 4 y 5 otro pre-escalador de 8 bits. Cada temporizador presenta un divisor de reloj que puede generar 5 señales diferentes: 1/2, 1/4, 1/8, 1/16 y TCLK. Cada bloque temporizador recibe su propia señal de reloj del divisor de reloj correspondiente, que a su vez la recibe del pre-escalador que le corresponde.

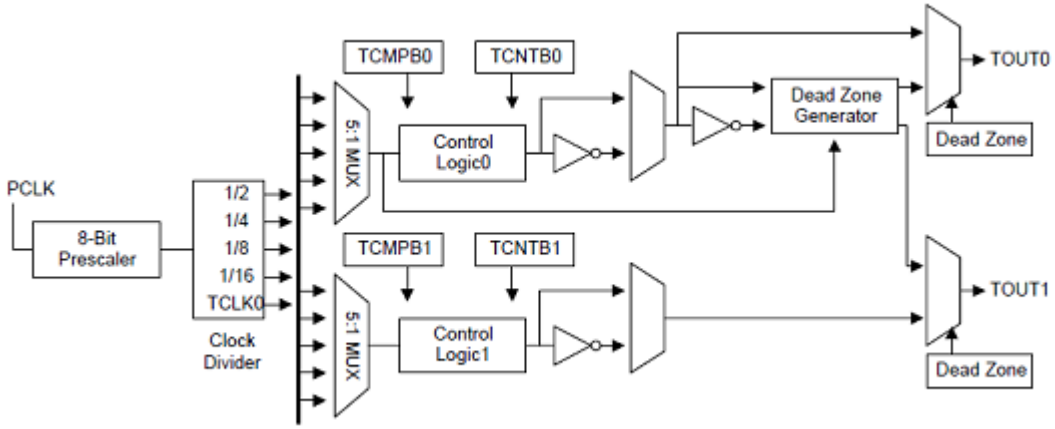

Figura 4. Estructura de los temporizadores (Samsung, 2004).

La configuración del pre-escalador y del divisor de reloj se realiza a partir de los registros de funciones especiales TCFG0 y TCFG1. El registro TCNTBn contiene el valor inicial que es cargado en el temporizador cuando es habilitado, mientras el registro TCMPBn contiene el valor inicial que es cargado en el registro de comparación (utilizado para cambiar el ciclo útil de la señal y generar señales moduladas por ancho del pulso). Cuando el contador descendente alcanza el valor de conteo cero, se lanza una solicitud de interrupción indicando que la operación ha concluido y el valor almacenado en el registro TCNTBn es automáticamente recargado.

Drivers de dispositivo

El sistema operativo GNU/Linux se ejecuta en el llamado contexto "kernel", que es un modo de operación privilegiado donde tiene total autoridad sobre todos los recursos, que incluyen la memoria física y los subsistemas de entrada/salida. Una vez que se ha inicializado el *hardware* y se ha montado el sistema de ficheros raíz, el núcleo Linux ejecuta una aplicación llamada "init". A partir de aquí las aplicaciones se ejecutan en el llamado contexto de usuario. En este estado de operación los procesos tienen acceso restringido y deben utilizar llamadas al sistema para solicitar los servicios del "kernel" y los recursos del *hardware* (Jones, 2010). Los procesos de usuario operan en un espacio de direcciones virtuales seleccionado de forma aleatoria y administrada por el "kernel". Cuando un programa de aplicación ejecuta una llamada al sistema se produce una conmutación de contexto y el "kernel" ejecuta el código

Revista Cubana de Ciencias Informáticas
Vol. 8, No. 2, 2014
ISSN: 1994-1536 | RNPS: 0547
Pág. 35-51
http://rcci.uci.cu

en representación del proceso que la invocó. Por otro lado, las subrutinas de atención a interrupción ejecutan código del "kernel" pero no representan a ningún proceso en particular por lo que, cuando se ejecuta código en este contexto se presentan algunas limitaciones: la subrutina de atención a interrupción no puede bloquearse o llamar a ninguna función del "kernel" que pueda resultar en un bloqueo.

Los drivers de dispositivos cumplen una función especial en sistemas GNU/Linux, permiten escribir código de "Kernel" para acceder directamente al *hardware* y brindar una interfaz hacia las aplicaciones de usuario que no cuentan con este privilegio (Corbet, 2005). Los *drivers* se comportan como cajas negras que hacen a una pieza de *hardware* responder a una interfaz bien definida. Los drivers esconden completamente los detalles de cómo funciona el dispositivo. Las actividades realizadas por el usuario se realizan a través de un conjunto estandarizado de llamadas al sistema, que son independientes de un *driver* en específico pero son utilizadas para realizar operaciones específicas del dispositivo. Véase la Figura 5.

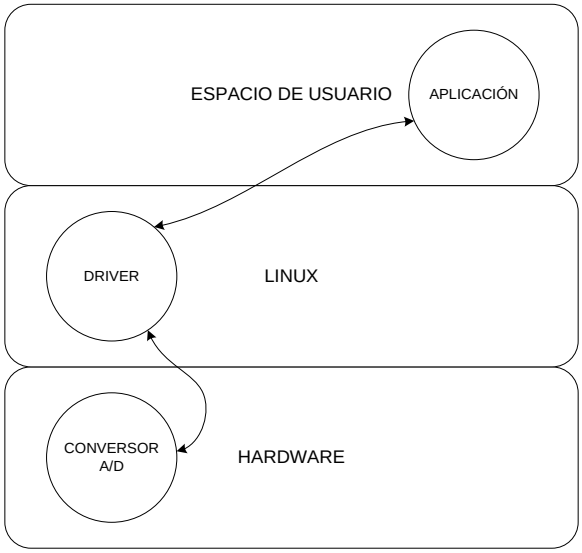

Figura 5. Acceso al hardware desde una aplicación de usuario.

Resultados y discusión

Al diseñar un *driver* para sistemas GNU/Linux debe considerarse el compromiso que se establece entre el tiempo requerido de programación y la flexibilidad del resultado. El término flexible enfatiza el hecho de que un *driver* debe proveer un mecanismo para interactuar con el dispositivo, no una política en su utilización (Corbet, 2005). Esta

distinción entre mecanismo y política es una de las mejores ideas en que se basan los *drivers* en sistemas tipo Unix. Cada problema se divide en dos partes: primero, qué facilidades debe proveer (mecanismo), y segundo, cómo utilizar esas facilidades (política). El *driver* debe lidiar con mantener el *hardware* disponible, pero no forzar la forma de utilizarlo. Esto debe dejarse a las aplicaciones de usuario. *El driver* puede verse como una capa de *software* entre la aplicación y el dispositivo.

Una de las mejores características de GNU/Linux es la posibilidad de extender el conjunto de facilidades que ofrece el "*kernel*" en tiempo de ejecución (Love, 2010). Esto significa que se pueden agregar (o eliminar) funcionalidades en el núcleo mientras el sistema está activo. Cada pieza de código que se agrega al "kernel" en tiempo de ejecución, se denomina módulo. GNU/Linux ofrece una gran variedad de clases de módulos, entre los que se encuentran (pero no se limitan a ellos) los *drivers* de dispositivo. Cada módulo está constituido por código objeto que se enlaza dinámicamente en tiempo de ejecución utilizando el comando "insmod", y puede ser desenlazado utilizando el comando "rmmod". El código modularizado se ejecuta en el espacio de direcciones del "kernel". Usualmente los *drivers* de dispositivo realizan dos tareas fundamentales, algunas funciones en el módulo se ejecutan en respuesta a llamadas al sistema y otras se encargan de manejar las solicitudes de interrupción.

El modo en que el "*kernel*" Linux clasifica los drivers de dispositivo se divide en tres categorías: *drivers* de carácter, *drivers* de bloque y *drivers* de red. Los *drivers* de carácter pueden accederse como una secuencia de *bytes* e implementan al menos cuatro llamadas al sistema: "*open*", "*read*", "*write*" y "*close*" (Mauerer, 2008). Los *drivers* tipo carácter se acceden a través de nodos en el sistema de ficheros (localizados en el directorio "/dev"). La única diferencia entre un fichero regular y un *driver* tipo carácter es que en los ficheros regulares es permitido el desplazamiento hacia atrás y hacia adelante y los *drivers* tipo carácter son solo canales de datos que pueden accederse secuencialmente.

Implementación del driver

Cuando se implementa un *driver* para sistemas GNU/Linux, generalmente, se incluyen un gran número de ficheros de cabecera que contienen las definiciones de las funciones, tipos de datos y variables. Los ficheros de cabecera incluidos dependen de la aplicación en particular, pero algunos de ellos deben aparecer en cualquier módulo, por ejemplo: <*linux/module.h*> y <*linux/init.h*>.

Revista Cubana de Ciencias Informáticas
Vol. 8, No. 2, 2014
ISSN: 1994-1536 | RNPS: 0547
Pág. 35-51
http://rcci.uci.cu

En particular, los dispositivos tipo carácter, se identifican utilizando dos números denominados números de identificación mayor y menor. La estructura "dev_t" (definida en <*Linux/types.h*>) se utiliza para almacenar estos números. Si se tienen los números mayor y menor de dispositivo y se quiere obtener la estructura "dev_t" asociada, se utiliza la macro:

```
MKDEV(int major, int minor);
```

Una de las primeras operaciones que debe realizar el driver en la función de inicialización es obtener y registrar los números que identifican al dispositivo. La función necesaria para realizar esta tarea está declarada en <linux/fs.h> y se define como:

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

Para liberar los números de dispositivo al desmontar el módulo, se utiliza la función:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Estas funciones reservan los números de dispositivo pero no le informan al "kernel" que hacer realmente con estos números. Antes de que un programa de aplicación de usuario pueda acceder uno de estos números de dispositivo, el *driver* debe conectarlos a las funciones internas que implementan las operaciones que pueden realizarse sobre él. La estructura "*file_operations*" se utiliza para realizar esta conexión. Esta estructura está definida en el fichero de cabecera <*linux/fs.h*> y está constituida por una colección de punteros a funciones. Cada fichero abierto es asociado a su propio conjunto de funciones. Las funciones referenciadas por cada puntero se encargan de implementar las llamadas al sistema "*open*", "*read*", "*ioctl*", etc. Cada campo de la estructura, apunta a la función encargada de implementar una operación específica o se pone a NULL para las operaciones no soportadas.

```
static struct file_operations SAD_fops =
{
    .owner = THIS_MODULE,
    .read = SAD_read,
    .open = SAD_open,
    .ioctl = SAD_ioctl,
    .release = SAD_release,
};
```

En este caso, solo se implementaron los métodos más importantes con el objetivo de mantener el sistema lo más simple posible.

El "kernel" utiliza además la estructura "cdev" para representar un dispositivo tipo carácter internamente. Antes de invocar las operaciones del dispositivo, deben declararse una o más de estas estructuras. Para ello debe incluirse el

fichero de cabecera <*linux/cdev.h*> donde están definidas la estructura y las funciones auxiliares necesarias. Sin embargo la mayoría de las veces se embebe la estructura "cdev" dentro de una estructura más específica que representa el dispositivo particular para el cual se está desarrollando el *driver*. Por ejemplo:

```
static struct SAD_dev
{
    wait_queue_head_t read_queue;
    struct kfifo* read_fifo;
    spinlock_t read_lock;
    atomic_t is_open;
    struct cdev cdev;
} SAD_dev;
```

Un sistema de adquisición de datos representa un caso típico de un problema del tipo productor/consumidor donde el *hardware* produce datos a una razón diferente a la que son consumidos por el *software* de aplicación. Para dar solución a este problema, en la estructura se incluyen un *buffer* circular para almacenar los datos leídos, un "*spinlock*" para garantizar la exclusión mutua a la hora de acceder a los datos compartidos por varios hilos de ejecución, una estructura del tipo "*wait_queue_head_t*" para lograr la sincronización del productor y el consumidor, y una variable atómica que permite determinar si el dispositivo ya ha sido abierto para ejecutar la inicialización de las estructuras una sola vez. La función de inicialización quedaría entonces:

```
static int __init SAD_init(void)
{
    int result;
    dev_t dev;
    // Inicialización de la estructura dev_t
    dev = MKDEV(major_number, minor_number);
    result = register_chrdev_region(dev, 1, "SAD");
    // Inicialización de la estructura "cdev" de SAD_dev
    cdev_init(&SAD_dev.cdev, &SAD_fops);
    SAD_dev.cdev.owner = THIS_MODULE;
    SAD_dev.cdev.ops = &SAD_fops;
    result = cdev_add (&SAD_dev.cdev, dev, 1);
    // Inicialización de otros campos de SAD_dev
    init_waitqueue_head(&SAD_dev.read_queue);
    // Inicialización de la FIFO
    SAD_dev.read_fifo = kfifo_alloc(read_buffersize * sizeof(SAMPLE_SIZE), GFP_KERNEL, &SAD_dev.read_lock);
    atomic_set(&SAD_dev.is_open,-1);
    // Todo se ejecutó correctamente
    #ifdef DEBUG
    printk(KERN_ALERT "El modulo SAD ha sido cargado\n");
    #endif
   return 0;
}
```

De forma similar, cualquier módulo no trivial necesita una función encargada de liberar los recursos utilizados y devolverlos al sistema:

Revista Cubana de Ciencias Informáticas
Vol. 8, No. 2, 2014
ISSN: 1994-1536 | RNPS: 0547
Pág. 35-51
http://rcci.uci.cu

```
static void __exit SAD_exit(void)
{
    dev_t dev;
    dev = MKDEV(major_number, minor_number);
    unregister_chrdev_region(dev, 1);
    kfifo_free(SAD_dev.read_fifo);
    #ifdef DEBUG
    printk(KERN_ALERT "El modulo SAD ha sido desmontado\n");
    #endif
}
```

Entre las operaciones que pueden realizarse sobre el dispositivo, el método "*open*" se utiliza para inicializar y preparar el dispositivo antes de proceder al resto de las operaciones. En la mayoría de los *drivers* esta función debe llevar a cabo las siguientes tareas:

1. Chequear el dispositivo y detectar posibles errores.
2. Inicializar el dispositivo si es abierto por vez primera.
3. Actualizar los punteros a las operaciones de ser necesario.
4. Registrar e inicializar las estructuras de datos.

En este caso, el método "*open*" se implementó como:

```
static int SAD_open(struct inode *inode, struct file *filp)
{
    int result;
    printk (KERN_INFO "opened\n");
    if (atomic_inc_and_test (&SAD _dev.is_open)) // Si el dispositivo está siendo utilizado, denegar el acceso.
    {
        printk (KERN_INFO "Inicializado\n");
        // Configuración del pin GPF0 como entrada de interrupción externa
        s3c2410_gpio_cfgpin (S3C2410_GPF(0), S3C2410_GPF0_EINT0);
        // Inicialización de la interrupción externa EXT0
        result = request_irq (IRQ_EINT0, ext0_isr, IRQ_FLAGS,"SAD", NULL);
        // Obtener acceso a la dirección de memoria del convertidor analógico/digital
        if (!request_mem_region (AD_address, 1, "SAD"))
        {
            printk (KERN_INFO "No pudo obtenerse la dirección solicitada \n");
            return -ENODEV;
        }
        // Obtener la dirección de memoria virtual del convertidor analógico/digital
        vAD_address = (unsignedlong) ioremap (AD_address, 1);
        // Selección de la configuración del convertidor analógico/digital
        iowrite16(0x0001, vAD_address);
        set_timer0_frequency(TMR0FREQ); // Configuración del timer0
        printk (KERN_INFO "El Timer0 ha sido inicializado\n");
    }
    return 0;
}
```

El método "*read*" copia datos desde el sistema de adquisición de datos hasta el código de aplicación. El método "*read*", puede implementarse como:

Revista Cubana de Ciencias Informáticas
Vol. 8, No. 2, 2014
ISSN: 1994-1536 | RNPS: 0547
Pág. 35-51
http://rcci.uci.cu

```
static size_t SAD_read(struct file *filp, char __user * buf, size_t count, loff_t *f_pos)
{
    unsignedint _count;
    _count = kfifo_get(SAD_dev.read_fifo, buf, count *
    sizeof(SAMPLE_SIZE));
    while(_count==0)    // Debe bloquearse hasta que hallan datos disponibles para leer
    {
        if (wait_event_interruptible (SAD_dev.read_queue,(_count = kfifo_get(SAD_dev.read_fifo,
                        buf, count * sizeof(SAMPLE_SIZE)))))
        return -ERESTARTSYS;
    }
    return _count;
}
```

La variable "*filp*", es el puntero al fichero y "*count*" es el número de *bytes* que se solicitan transferir. El argumento "*buf*" apunta al *buffer* del usuario de donde los datos deben leerse o escribirse según sea el caso. Finalmente "f_pos" es el puntero que indica la posición a la que el usuario está accediendo. El método "*read*" devuelve un valor negativo si ocurrió algún error. Un valor mayor o igual a cero, indica cuantos *bytes* fueron transferidos correctamente a la aplicación de usuario. El código de la rutina de atención a la interrupción externa puede implementarse como:

```
static irqreturn_t ext0_isr(int irq_in, void *dev_id)
{
    u16 ad_sample;
    ad_sample = ioread16 (vAD_address) & 0x0FFF;
    if(kfifo_put(SAD_dev.read_fifo,(char*)&ad_sample, 1*sizeof(SAMPLE_SIZE)))
    {
        wake_up_interruptible(&SAD_dev.read_queue);
    }
    return IRQ_HANDLED;
}
```

Además se puede utilizar el método "*ioctl*" para enviar comandos a la aplicación, como por ejemplo la detención o puesta en marcha del temporizador 1 con el objetivo de detener o reiniciar el proceso de conversión.

```
static int SAD_ioctl(structinode *inode, struct file *filp, unsignedint cmd, unsignedlon garg)
{
    switch (cmd)
    {
        case POT_IOCTL_START:
            printk (KERN_INFO "Se ha iniciado el Timer1\n");
            set_timer1_frequency(TMR1FREQ);     //Iniciar timer 1
        break;
        case POT_IOCTL_STOP:
            printk (KERN_INFO "Se ha detenido el Timer1\n");
            timer1_stop();                      //Detener timer1
       break;
   }
   return 0;
}
```

Revista Cubana de Ciencias Informáticas
Vol. 8, No. 2, 2014
ISSN: 1994-1536 | RNPS: 0547
Pág. 35-51
http://rcci.uci.cu

Por último es necesario implementar la llamada al método "*release*" que debe deshacer los efectos producidos por el método "*open*", dejando listo el sistema para cuando vuelva a ser necesitado por otro proceso de aplicación:

```
static int SAD_release(struct inode *inode, struct file *filp)
{
    printk (KERN_INFO "El dispositivos se ha cerrado\n");
    atomic_inc(&SAD_dev.is_open);
    if (atomic_dec_and_test (&SAD_dev.is_open))
    {
        printk (KERN_INFO "Los recursos se han liberado\n");
        // Detiene el TIMER0 y el TIMER1
        timer0_stop();
        timer1_stop();
        // Configuración del pin de interrupción como salida digital
        s3c2410_gpio_cfgpin(S3C2410_GPF(0), S3C2410_GPIO_OUTPUT);
        //Libera la interrupción.
        free_irq (IRQ_EINT0, NULL);
        // Libera las direcciones virtuales del convertidor analógico/digital
        iounmap ((void __iomem *) vAD_address);
        release_mem_region (AD_address, 1);
    }
    atomic_dec(&SAD_dev.is_open);
    return 0;
}
```

El principal aporte del *driver* desarrollado y que lo diferencia de forma única del resto, radica en su capacidad para aprovechar las características específicas del *hardware* de la SBC MINI2440 (temporizadores, controlador de interrupciones, puertos de entrada/salida, buses de expansión, etc.) e integrarlo junto a un convertidor analógico/digital para formar un sistema de adquisición de datos completamente funcional que da solución a un problema real. Además, el sistema tiene a su disposición la gran capacidad de procesamiento que brinda el microprocesador S3C2440 de Samsung y al estar soportado sobre el sistema operativo GNU/Linux permite al desarrollador de la aplicación de usuario contar con todas las ventajas que este ofrece: una gran variedad de librerías de funciones, diversidad en estándares de comunicación y soporte para plataformas orientadas al desarrollo de aplicaciones gráficas enriquecidas en entornos embebidos tales como GTK+ y Qt.

Conclusiones

En el presente trabajo se ha desarrollado un *driver* para el sistema operativo GNU/Linux capaz de controlar un sistema de adquisición de datos conectado al bus de expansión de la computadora de tarjeta única MINI2440 como una alternativa a los tradicionales sistemas de adquisición de datos basados en microcontroladores de 8 *bits* cuyos recursos resultan muchas veces insuficientes. Mediante la utilización de la arquitectura propuesta se amplían las facilidades ofrecidas por el sistema, permitiendo contar con diversas tecnologías de comunicación, una interfaz gráfica enriquecida y más amigable al usuario, así como mayor capacidad de procesamiento. A partir del *driver*

desarrollado se pueden generar por *hardware* las señales de reloj y de inicio de conversión utilizando los temporizadores 1 y 2 de la SBC MINI2440, y atender el fin de conversión por interrupción, garantizando la adquisición regular y uniforme de las muestras. La propuesta presentada aunque enfocada a un caso particular puede extenderse a otros problemas del tipo productor/consumidor.

Referencias

- ABBOTT, D. The embedded Linux learning kit version 2 user's guide. Silver City: Intellimetrix; 2011. p. 103.
- ANALOG-DEVICES. DATASHEET AD7938-6. In: Devices, editor: Analog Devices; 2011. p. 32.
- BHAIRA, A. Development and implementation of a Linux-Xenomai based hard real-time device driver for PCI data acquisition system (DAS) card. International Journal of Computer Applications. 2013;81(12):24-28.
- CORBET, J. Linux Device Drivers. Sebastopol: O'Reilly; 2005. p. 636.
- DURSUN, M. PC-based data acquisition system for PLC-controlled linear switched reluctance motor. Turkish Journal of Electrical Engineering & Computer Sciences. 2013;21:71-80.
- HALLINAN, C. Embedded Linux primer, a practical real world approach. Boston: Prentice Hall; 2011. p. 652.
- JONES, T. Look at Linux, the operating system and universal platform. Colorado: Developers Works IBM; 2012a.
- JONES, T. Introducing the 3.3 and 3.4 Linux kernels. Colorado: Developers Works IBM; 2012b.
- JONES, T. Kernel command using Linux system calls. Colorado: Developers Works IBM; 2010.
- LOVE, R. Linux kernel development. Boston: Addison-Wisley; 2010. p. 647.
- MAUERER, W. Professional Linux kernel architecture. Indianapolis: Wiley Publishing, Inc; 2008. p. 1370.
- SAMSUNG. S3C2440A 32-bit RISC microprocessor user's manual. Samsung Electronics; 2004. p. 560.

– SLOSS, A. N. ARM system developers guide, designing and optimizing system software. San Francisco: Elsevier; 2004. p. 703.
– SUNNY, S. Data acquisition and control system using embedded web server. International Journal of Engineering Trends and Technology. 2012;3(3):411-414.
– WOLF, W. Computers as components, principles of embedded computing system design. Burlington: Elsevier; 2008. p. 533.