Programación de Sistemas Embebidos 2020
Dispositivos de E/S (periféricos)

Índice general

Chapter 5

Dispositivos de E/S (periféricos)

Rafael Ignacio Zurita
Universidad Nacional del Comahue

Además del procesador y memoria, la mayoría de los sistemas embebidos contienen varios dispositivos de hardware extra. Algunos de estos dispositivos son específicos a la aplicación, mientras que otros -como los relojes y puertos seriales- son útiles de una manera general a una gran variedad de sistemas distintos. Los que residen junto con el procesador dentro del mismo chip se los denomina periféricos internos, u on-chip. Los que se encuentran fuera del chip donde está el procesador se los denomina de manera opuesta, es decir, periféricos externos. En este capítulo se detalla la mayoría de los problemas de software comunes que surgen cuando se programan periféricos de ambos tipos.

Keywords: sistema embebido, dispositivos de E/S, periféricos, registro de E/S, controladores de E/S, drivers de dispositivos

> The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well-installed microcode bug will be almost impossible to detect.
> —Ken Thompson aug-1984, Reflections on Trusting Trust.

1 Registros de estado y de control

La interfaz entre un procesador embebido y un dispositivo periférico es un conjunto de registros de estado y control. Estos registros son parte del hardware del periférico, y sus ubicaciones, sus tamaños (en bits) y el significado individual de cada bit en cada registro son característicos de cada dispositivo. Por ejemplo, el significado de los bits de los registros de un dispositivo serial son muy diferentes a los de los contadores o relojes. En esta sección se encuentra explicado la manera de manipular el contenido de los registros de estado y control directamente desde programas en C/C++.

Dependiendo del diseño del procesador y la placa, los dispositivos periféricos se encuentran localizados en el espacio de memoria del procesador o dentro

del espacio de E/S. De hecho, es común en sistemas embebidos que se incluyan periféricos de ambos tipos. Se los denomina *periféricos mapeados en memoria* y *periféricos mapeados en E/S* respectivamente (estos últimos tambien son denominados como *aislados*). De los dos tipos, los periféricos mapeados en memoria son los más fáciles de utilizar y por lo tanto su popularidad aumenta.

Los registros de estado y control de periféricos mapeados en memoria se pueden parecer a variables comunes en un programa. Por ejemplo, es posible declarar un puntero en C a un registro o bloque de registros, y establecer la dirección explícitamente (valor del puntero). Supongamos el registro `puerto_b` utilizado en el capítulo 2, el cual es mapeado en memoria en la dirección física 0x25. La función `led_toggle` estudiada en ese capitulo puede ser escrita completamente en C, como se muestra a continuación.

```
volatile unsigned char * puerto_b = (unsigned char *) 0x25;

void led_toggle(void)
{
    *puerto_b ^= LED_ROJO;    /* Read, xor, and modify. */
}    /* led_toggle() */
```

Aquí se declara un puntero a un registro de 8 bits y explícitamente se lo inicializa con la dirección 0x25. A partir de este momento el puntero al registro trabaja como cualquier puntero a una variable de tipo char de 8 bits.

1.0.1 Uso de volatile en C o C++

Note que existe una diferencia muy importante entre registros de dispositivos y variables. El contenido de un registro de dispositivo puede cambiar sin que el programa intervenga o se notifique del cambio. Esto sucede porque el contenido del registro puede ser modificado por el hardware del dispositivo. De manera contraria, el contenido de una variable no cambiará a menos que el programa modifique la misma explícitamente. Por esa razón, se dice que el contenido de un registro de dispositivo es volátil, o sujeto a cambiar sin notificación previa.

En C y C++ se debe utilizar la palabra clave volatile cuando se declaran punteros a registros de dispositivos. La misma le indica al compilador que no puede realizar suposiciones acerca del dato almacenado en esa dirección. Por ejemplo, si el compilador observa una escritura a una ubicación volátil y luego inmediatamente le sigue otra instrucción de escritura a la misma ubicación, el compilador no asume que la primer escritura es innecesaria. En otras palabras, la palabra

clave volatile instruye a la fase de optimización del compilador a tratar a esa variable como volátil, y que su conducta no puede ser predecible en tiempo de compilación.

Prosiguiendo con el ejemplo anterior, se muestra aquí debajo el uso de volatile para advertir al compilador acerca del registro `puerto_b`:

```
volatile unsigned char * puerto_b = (unsigned char *) 0x25;
```

Es importante notar que es erróneo interpretar, en esta declaración, que el puntero en si mismo es volátil. De hecho, el valor de la variable `puerto_b` tendrá el valor 0x25 durante toda la ejecución del programa (a menos que el valor sea modificado por alguna instrucción del programa, por supuesto). Mas bien es el dato apuntado el que está sujeto a cambiar sin que el programa intervenga. Este detalle es muy sutil y es fácilmente confundible al pensar mucho en el tema. Solamente recuerde que la ubicación de un registro es fija aunque su contenido podría no serlo. Por lo tanto, si se utiliza la palabra clave volatile, el compilador asumirá lo mismo.

Los otros tipos de registros en dispositivos de E/S (registros en el espacio de E/S -aislados-) no pueden accederse de la manera anterior, y no hay una manera estándar de acceder a ellos desde C o C++. A estos registros se los accede mediante el uso de instrucciones específicas de la arquitectura (en lenguaje máquina), las cuales no son soportadas por los estándares del lenguaje C o C++. Por lo tanto, es necesario utilizar rutinas de bibliotecas especiales, o código en lenguaje ensamblador, usualmente incrustado entre sentencias del programa embebido en C, para leer y escribir los mismos.

2 Filosofía del controlador de dispositivo

Objetivo de diseño
Al diseñar un controlador de dispositivo (device driver) se debe tener presente la siguiente meta: ocultar el hardware completamente.

Un controlador de dispositivo o manejador de dispositivo (en inglés: device driver, o simplemente driver) es el software que controla la operación de un dispositivo de E/S. Le permite a la aplicación embebida, o al núcleo del sistema operativo, acceder a las funciones del dispositivo, sin que estos tengan que conocer los detalles precisos del hardware siendo utilizado, lo que resulta en una abstracción del hardware mediante una interfaz de programación.

Cuando se completó el desarrollo de un sistema embebido, los módulos controladores de dispositivos (device drivers) deben ser las únicas piezas del software en el sistema entero que leen o escriben directamente a los registros de estado, control y datos de un periférico. En la figura 1 se puede observar el diagrama de bloques de alto nivel de este concepto. Existirá un único controlador por cada dispositivo de E/S[1]. Además, si un dispositivo genera interrupciones, la rutina de servicio de interrupciones que atiende las mismas debe ser parte integral del código fuente del controlador de dispositivo de E/S. En esta sección se explica el por qué se recomienda esta filosofía y cómo puede ser llevada a cabo.

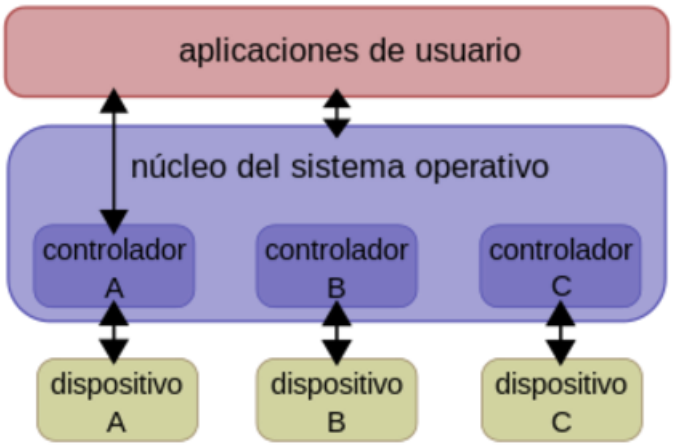

Figura 1: Diagrama de bloques de un sistema con controladores de dispositivos.

Por supuesto, intentar ocultar el hardware completamente es difícil. Algunas caracterísiticas generales de los dispositivos pueden llegar a ser visibles siempre, sea cual sea la interfaz de programación que se diseñe o seleccione. Esa característica es esperada e inevitable. La meta tiene que ser que la interfaz de programación desarrollada no debería necesitar cambios si el periférico final es reemplazado con otro de su misma clase general. Por ejemplo, todos los dispositivos de memoria Flash comparten el concepto de sectores (aunque el tamaño del sector puede diferir entre chips). Una operación de borrado puede ser realizada únicamente en un sector entero, y es probable que la aplicación deba indicar qué sector borrar, o el sistema de archivos. Esa característica del dispositivo Flash (sectores) estará visible en la interfaz de programación, sin poder ser ocultada

[1] A nivel de código fuente, un único controlador puede implicar varios archivos fuentes

completamente. De todas maneras, debido a que todas las memorias de tipo Flash se acceden de a sectores, la interfaz desarrollada para un controlador de memoria Flash debería ser útil con cualquier dispositivo de memoria similar.

Los controladores de dispositivos para sistemas embebidos son bastante diferentes a los controladores de dispositivos para estaciones de trabajo (por ej. PC). En una computadora general, los controladores de dispositivos están desarrollados con el objetivo de satisfacer los requerimientos del sistema operativo, debido a que imponen requerimientos estrictos en la interfaz de software entre el sistema operativo y controlador de dispositivo. Por ejemplo, el controlador de dispositivo para una placa de red particular debe cumplir con la interfaz de software del sistema operativo, sin importar las características y capacidades del hardware subyacente. Los programas de aplicación que necesitan utilizar la placa de red están forzados a utilizar la API de red provista por el sistema operativo, y no tienen acceso directo a la placa en si misma. En este caso la meta de ocultar el hardware completamente es alcanzada más fácilmente.

En contraste, el software de aplicación en un sistema embebido puede acceder al hardware directamente. De hecho, debido a que todo el software es vinculado en conjunto en una imagen binaria individual, raramente existe una distinción entre la aplicación, sistema operativo, y controladores de dispositivos. Esta distinción, como así también la aplicación de restricciones de acceso al hardware, son meramente responsabilidades de los desarrolladores de software embebido (a nivel de código fuente). Ambas son decisiones de diseño que los desarrolladores deben realizar concienzudamente. En otras palabras, los programadores de software embebido pueden tener más flexibilidad en el diseño del software que sus pares no embebidos (lo cual no significa que esto sea una ventaja). De hecho, muchas veces no lo es, y los sistemas embebidos suelen presentar código fuente pobremente modularizado, y muchas veces tambien con poco encapsulamiento de los datos de sus módulos.

Los beneficios de un buen diseño del controlador de dispositivo son tres. Primero, debido a la modularización, la estructura de todo el software se comprende más fácilmente. Segundo, como hay únicamente un modulo que interactúa directamente con los registros de un periférico, el estado del hardware puede ser verificado mas precisamente. Finalmente, el ultimo beneficio pero no por eso menos importante, los cambios en el software que resultan de los cambios en el hardware están localizados en el controlador de dispositivo. Todos estos beneficios ayudan a reducir el numero total de errores en el software embebido. Pero, se debe estar dispuesto a realizar el esfuerzo extra que conlleva lograr estos objetivos, sobre todo en la etapa de diseño de la arquitectura del sistema.

Si acuerda con esta filosofía de ocultar todas las especificaciones del hardware e interacciones dentro del controlador entonces usualmente debe implementar y utilizar los cinco componentes listados a continuación:

1. Una estructura de datos que se superpone (overlay) a los registros de estado y control del dispositivo mapeados en memoria
2. Un conjunto de variables para mantener el estado actual del hardware y del controlador de dispositivo
3. Una rutina para inicializar el hardware a un estado conocido
4. Un conjunto de rutinas que, en su conjunto, presentan una API
5. Una o varias rutinas de atención de interrupciones

Para realizar una implementación de un controlador de dispositivo lo más simple e incremental como sea posible, se debería desarrollar los cinco elementos anteriores en el orden presentado.

Una estructura de datos que se superpone (overlay) a los registros de estado y control

El primer paso en el proceso de desarrollo del driver es crear una estructura (struct), al estilo del lenguaje C, que represente de manera exacta los registros mapeados en memoria del dispositivo. Generalmente, se necesita estudiar la hoja de datos del periférico y crear una tabla para los registros de estado y de control junto con sus desplazamientos (en memoria). Entonces, comenzando con el registro con la dirección más baja, se va completando la tabla de la estructura. Si existen ubicaciones no utilizadas o reservadas entre los registros se puede ubicar variables vacías para ocupar espacio adicional acorde.

Un ejemplo de una estructura de datos del estilo se presenta a continuación. La misma describe los registros del hardware serial USART presente dentro del atmega328p. El dispositivo tiene seis registros, dispuestos como se muestra a continuación en la estructura de datos serial_st. Cada registro es de 8 bits y deberían ser manipulados como un unsigned char.

```
struct serial_st
{
    uint8_t data_es; /* udr0 i/o data */
    uint8_t baud_rate_h;     /* ubrr0h baud rate high */
```

```
    uint8_t baud_rate_l;     /* ubrr0l baud rate low */;
    uint8_t _reserved;     /* espacio sin utilizar */
    uint8_t status_control_c;   /* ucsr0c USART Control and Status C */
    uint8_t status_control_b;   /* ucsr0b USART Control and Status B */
    uint8_t status_control_a;   /* ucsr0a USART Control and Status A */
}
```

Para leer y escribir bits individuales mas fácilmente es una practica común definir mascaras de bits para usos comunes:

```
#define RECEIVER_ENABLE 0x10        /* RXEN0 Habilitar la recepcion */
#define TRANSMITTER_ENABLE 0x08     /* TXEN0 Habilitar la transmicion */
#define CHARACTER_SIZE_0 0x20       /* UCSZ00 Numero de bits del dato de e/s */
#define CHARACTER_SIZE_1 0x40       /* UCSZ01 Numero de bits del dato de e/s */
#define READY_TO_READ 0x80          /* RXC0 Dato listo para leer */
#define READY_TO_WRITE 0x20         /* UDRE0 Buffer vacio listo para escribir */
```

Un conjunto de variables para mantener el estado actual del hardware y del controlador de dispositivo

El segundo paso en el proceso de desarrollo del controlador es definir qué variables se necesitan para mantener el estado del hardware y del controlador de dispositivo. Por ejemplo, en el caso del controlador serial probablemente se necesita conocer si el hardware ha sido inicializado.

Algunos controladores de dispositivos crean tambien *dispositivo de software*. Esto es un dispositivo puramente lógico, que es implementado en el código fuente por encima del controlador del hardware periférico. Por ejemplo, es fácil imaginar que más de un reloj por software puede ser implementado desde un reloj de hardware (timer/counter). El dispositivo de E/S del reloj se configura para generar un tick de reloj periódico, y el controlador de dispositivo gestionaría un conjunto de relojes virtuales por software de varias duraciones, manteniendo información del estado de cada *reloj software*.

Una rutina para inicializar el hardware a un estado conocido

Una vez que se conoce como mantener el estado de los dispositivos lógicos y físicos es momento de comenzar a escribir las funciones que interactúan y controlan el dispositivo. Es conveniente comenzar con la rutina de inicialización del hardware. Además, necesitará esa rutina primero, y esa es una buena manera de entender como interactuar con el dispositivo.

Un conjunto de rutinas que serán la API para los usuarios del controlador de dispositivo

Luego de que se haya inicializado el dispositivo correctamente es momento de agregar otras funcionalidades al controlador. Es en este paso en donde se deben establecer los nombres y propósitos de varias rutinas, como así también sus respectivos parámetros y valores devueltos. Una vez definidas es el momento de implementar cada función junto con un caso de test mínimo. Se presentan ejemplos de tales rutinas en la siguiente sección.

Una o varias rutinas de atención de interrupciones

Es conveniente diseñar, implementar y verificar la mayoría de las rutinas del controlador de dispositivo antes de habilitar las interrupciones. Encontrar el origen de los problemas relacionados con las interrupciones es muy difícil. Además, si existen errores (bugs) en otros módulos controladores, la búsqueda del problema relacionado con las interrupciones puede no tener fin. Por lo que es mejor utilizar E/S programada (polling) primero, para obtener el controlador funcionando. De esta manera se comprenderá cómo trabaja el dispositivo (y que realmente esté funcionando) cuando se comienza a buscar la fuente de los problemas relacionados con las interrupciones. Finalmente, cabe decir que, desafortunadamente, suele haber casi siempre problemas relacionados con las interrupciones cuando se implementa el control por software de las mismas.

3 Controlador de dispositivo serial

El controlador de dispositivo que se presenta como ejemplo permite operar un puerto serial. El hardware para este controlador de dispositivo utiliza un periférico llamado *UART* (pronunciado 'you-art' y es el acrónimo de *Universal Asynchronous Receiver Transmitter*) que se encuentra en el atmega328p. Un UART es un componente de hardware para comunicaciones, que recibe y envía información de manera serial y asincrónica. Asincrónico significa que el dato que arriba puede llegar en cualquier momento (no está controlado por un reloj), similar a la entrada provista por un teclado. Un UART acepta un byte de forma paralela desde el procesador, y el dispositivo lo serializa, transmitiendo cada bit en el momento apropiado. La recepción trabaja de manera inversa. Los dispositivos UART están disponibles en la mayoría de las computadoras (embebidas o no), por lo que usualmente se utiliza para comunicación entre sistemas. Tambien es la interfaz

utilizada por un sin fin de periféricos externos, por lo tanto, implementar un controlador para dispositivo UART será siempre útil.

Es importante aclarar que el atmega328p tiene en su chip un USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter) que puede trabajar en modo sincrónico, y también, como periférico SPI maestro. En nuestro controlador utilizamos únicamente el modo asincrónico no SPI.

Antes de diseñar y desarrollar un controlador de dispositivo es importante entender el diagrama de bloques del hardware. Este diagrama presenta cómo las señales van y vienen desde el periférico al exterior del mismo.

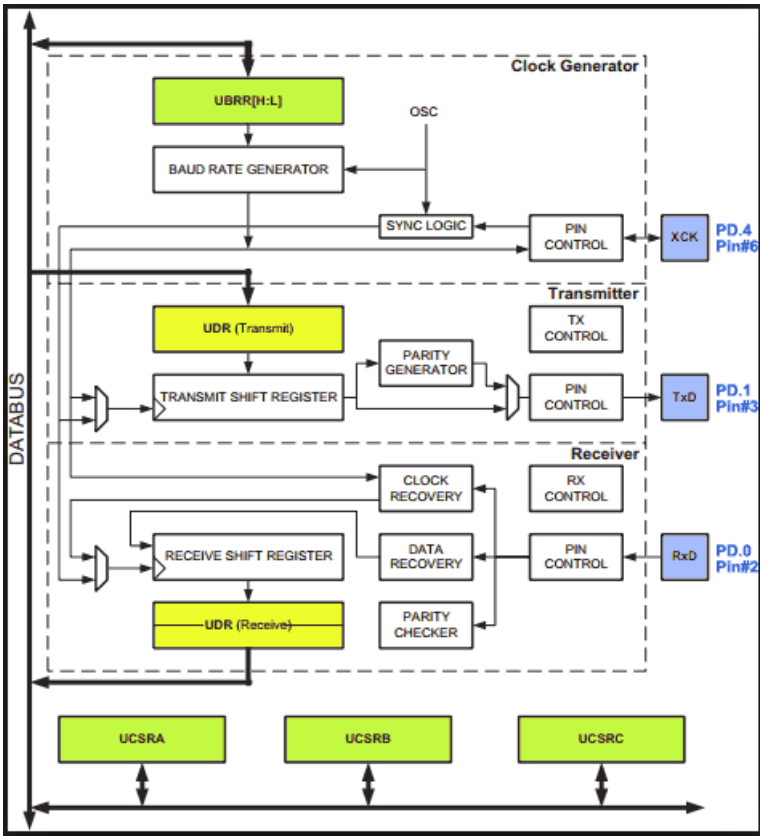

Figura 2: Diagrama de bloques del USART en el atmega328p.

Típicamente, esta tarea se logra observando las porciones relevantes de los esquemáticos, y leyendo las hojas de datos de los diferentes integrados. Un diagrama de bloques para el puerto serial se muestra en la Figura 2. Observe que

se han rellenado con colores amarillo y verde los registros conectados al Databus. Estos son los registros de estado, control y datos que pueden ser accedidos desde la CPU (conectados al bus). Si existen otros registros no conectados al bus con la CPU entonces son registros para uso interno del hardware del periférico. Los bloques en color celeste son los componentes que controla el hardware del dispositivo, y se encargan de enviar señales al exterior (la E/S real).

¿Qué se necesita conocer para controlar el periférico?

Para este ejemplo de controlador de dispositivo de E/S nos enfocamos únicamente en el dispositivo USART. La información de sus registros está localizada en la hoja de datos (en este caso del microcontrolador atmega328p). Mientras se lee esta información se debe tener en cuenta que la meta es entender varios conceptos, algunos de ellos son :

- La estructura de los registros para controlar el periférico, que incluye el cómo configurar las comunicaciones y como obtener y enviar datos desde el periférico.
- Las direcciones de memoria de los registros de estado y de control.
- El método que se debe utilizar para la operación del periférico (E/S programada o con interrupciones).
- Si se utilizan interrupciones entonces se debe entender bajo qué condiciones se producen las interrupciones, cómo el controlador de software se informa de las mismas, y cómo debe atenderlas.

3.1 Registros interfaz

El primer paso en el disenio de un controlador de dispositivo serial es definir la interfaz de los registros. En nuestro ejemplo utilizamos una estructura que se superponga a los registros del USART, los cuales están mapeados a memoria. La estructura uart_t se muestra a continuación :

```
typedef struct
{
    uint8_t data_es; /* udr0 i/o data */
    uint8_t baud_rate_h;    /* ubrr0h baud rate high */
    uint8_t baud_rate_l;    /* ubrr0l baud rate low */;
    uint8_t _reserved;    /* espacio sin utilizar */
```

```
    uint8_t status_control_c;    /* ucsr0c USART Control and Status C */
    uint8_t status_control_b;    /* ucsr0b USART Control and Status B */
    uint8_t status_control_a;    /* ucsr0a USART Control and Status A */
} volatile uart_t
```

La variable puerto_serial se utiliza para acceder a los registros en la dirección 0xc6 (que es la dirección mas baja del registro data_es), y es definida así :

```
uart_t *puerto_serial = (uart_t *) (0xc6);
```

3.2 Variables de estado

El siguiente paso es definir variables para mantener el estado actual del hardware. Se declara entonces, en el ejemplo, una estructura que contiene los parámetros del controlador serial, llamado `serialparams_t`. También, se crea una variable global que contendrá los valores de configuración actual definidos en la estructura anterior.

Una variable más, llamada `initialized` se define, para mantener el estado de si el controlador de dispositivo está inicializado o no.

```
typedef struct
{
    uint8_t dataBits;
    uint8_t stopBits;
    uint8_t baudRate;
    parity_t parity;
} serialparams_t;

serialparams_t serial_params;
```

3.3 Rutina de inicialización

La rutina de inicialización `serial_init` configura los parámetros de comunicación predeterminados. Los registros del USART son programados utilizando `serial_params`, la cual debe ser inicializada primero. La variable `initialized` es utilizada para asegurar que el puerto serial es configurado únicamente una vez.

```
void serial_init()
```

2

[2] La omisión tiene un propósito, se dejan como tarea para el lector

3.4 API del controlador de dispositivo

Ahora ya es posible incorporar funcionalidad adicional definiendo otras funciones en el controlador de dispositivo serial. La API para el controlador de dispositivo serial debería tener, al menos, funciones para enviar y recibir caracteres. En nuestro ejemplo se implementan las funciones `serial_put_char` y `serial_get_char`, para enviar y recibir caracteres individuales respectivamente.

La función `serial_put_char` debe esperar hasta el que hardware transmisor del periférico esté listo, y luego envía un caracter individual a través del puerto serial (E/S programada). La transmisión es realizada al escribir el dato al registro de datos del USART. El siguiente código muestra la función `serial_put_char`.

```
void serial_put_char(char c)
```

La función `serial_get_char` debe esperar hasta que un caracter es recibido, y luego es posible leer el mismo desde el registro de datos del puerto serial. Para determinar si un caracter ha sido recibido se puede verificar el bit data ready, del registro de estado del USART. El caracter recibido es devuelto a la función llamadora. El siguiente código implementa esta función.

```
char serial_get_char()
```

Otras funciones extras útiles para anexar a la API del controlador de dispositivo podrían ser las funciones `serial_get_str()` y `serial_put_str()` [3]. Debido a que este controlador de dispositivo serial no utiliza interrupciones el paso final en la filosofía del controlador de dispositivo (implementar la rutinas que atienden las interrupciones del controlador de dispositivo) no se implementa.

Verificando (testing) el controlador de dispositivo serial

Ahora que el controlador de dispositivo está implementado se necesita verificar que funciona correctamente. Es importante verificar las funciones de la nueva API de manera individual, antes de integrar el controlador junto con el resto del sistema.

Para verificar el controlador debe conectar el puerto serial de la placa arduino pro mini a la PC. Debido a que los pines del puerto serial de la placa funcionan con lógica ttl, se debe utilizar un adaptador. Un adaptador común en estos días

[3] API posible:
void serial_put_str(char * s);
char * serial_get_str();

es el que adapta estos niveles ttl a USB serial, util en caualquier PC. Luego de conectar el hardware se necesita un programa 'terminal' en la PC, tal como minicom, picocom, o screen. Ejecute uno de estos programas y configure el puerto serial del sistema operativo en la PC y la velocidad de la comunicación, que debe ser la misma que la del driver implementado.

La función main demuestra como ejecutar las funcionalidades implementadas en el controlador de dispositivo.

```
/*********************************************************************
 *
 * Function:    main
 *
 * Description: Exercise the serial device driver.
 * Notes:
 * Returns:     This routine contains an infinite loop, which can
 *              be exited by entering q.
 *
 ********************************************************************/
int main(void)
{
    char rcv_char = 0;

    /* Configure the UART for the serial driver. */
    serial_init();

    serial_put_char('s');
    serial_put_char('t');
    serial_put_char('a');
    serial_put_char('r');
    serial_put_char('t');
    serial_put_char('\r');
    serial_put_char('\n');

    while (rcv_char != 'q')
    {
        /* Wait for an incoming character. */
        rcv_char = serial_get_char();

        /* Echo the character back along with a carriage return and line feed. */
        serial_put_char(rcv_char);
        serial_put_char('\r');
        serial_put_char('\n');
    }
```

```
    /* The embedded program never ends */
    for(;;);

    return 0;
}
```

Primero, el controlador de dispositivo serial es inicializado llamando a `serial_init`. Luego, se envían varios caracteres hacia la PC, para verificar la función `serial_put_char`. Si el controlador de dispositivo serial opera correctamente se obtiene el mensaje 'start' en la pantalla terminal de la PC.

Finalmente un bucle while es ejecutado, chequeando si un caracter ha sido recibido (llamando a `serial_get_char`). Si se recibe un caracter en el puerto serial este es enviado nuevamente a la PC a través de un 'eco'. Si el usuario presiona 'q' en el programa terminal de la PC el programa embebido finaliza. De otra manera, el bucle continua y comprueba si otro caracter de entrada arriba volviendo a realizar el 'eco'.

3.5 Extendiendo la funcionalidad del controlador

Aunque el controlador es muy básico, tiene funcionalidad como para escribir una aplicación útil mínima. Además, el controlador de dispositivo presentado es suficiente para aprender acerca de la operación de los UARTs. La lista a continuación son posibles extensiones que puede realizar si se desea agregar funcionalidad extra.

NOTA: Mantenga esta lista en mente para otros controladores que desarrolle.

Configuración seleccionable

Se puede cambiar `serial_init` para que reciba parámetros de entrada, que permita a la aplicación que utiliza el controlador especificar los parámetros de comunicación inicial. Por ejemplo, baud rate para el puerto serial.

Verificación de errores

Es importante para los controladores de hardware realizar un adecuado chequeo de errores. Se puede comenzar definiendo una lista de códigos de errores (por ejemplo, errores en los parámetros, errores de hardware, etc) en la API de controlador. Las funciones del controlador utilizarían estos códigos para devolver el estado de la operación si existe un error. De esta manera, la aplicación

embebida utilizando el controlador puede tomar acciones de alto nivel cuando existan a fallas, y por ejemplo, reintentar la operación.

APIs adicionales

Agregando serial_get_str y serial_put_str (el cual requiere buffer para los datos del receptor y transmisor) podría también ser útil. La implementación de las funciones de tratamiento de cadenas podrían hacer uso de las funciones serial_get_char y serial_put_char para implementar el envío individual de cada elemento.

Uso de FIFO

Algunas veces el hardware de dispositivos UARTs contienen FIFOs (buffers en hardware) para los datos recibidos y transmitidos. Utilizando FIFOs agrega buffer a ambos canales de recepción y transmisión, logrando que el controlador UART sea mas robusto. Puede ser útil incorporar su control por software.

Interrupciones

Activar las interrupciones del UART para la recepción y transmisión es usualmente mejor que utilizar E/S programada. Por ejemplo, si en la función `serial_get_char` se utiliza interrupciones se eliminaría la necesidad de que el controlador deba esperar el caracter entrante. La aplicación podría entonces realizar otras tareas, mientras se espera que el dato sea recibido.

4 Consideraciones finales en el diseño de controladores para dispositivos

La mayoría de los sistemas embebidos tienen mas de un controlador de dispositivo. De hecho, algunas veces pueden tener muchos. En cuanto mejore con la experiencia necesita entender la manera en que los diferentes controladores en el sistema interactúan entre ellos. También es importante entender muy bien cómo el software de aplicación utiliza los controladores de dispositivos para que pueda diseñar APIs apropiadas.

Con respecto al manejo de errores es necesario primero tener un buen entendimiento de todo el diseño del software, para conocer los posibles problemas que pueden surgir. Algunas áreas a considerar cuando se realiza el diseño de la arquitectura del software que incluye varios controladores de software son :

Prioridades en las interrupciones Si se utilizan interrupciones en los controladores de dispositivos de un sistema necesita determinar un conjunto apropiado de niveles de prioridades.

Uso de recursos

Es importante comprender qué recursos son necesarios para cada controlador. Por ejemplo, imagine el desarrollo de un controlador ethernet para un sistema con muy poca memoria. Es muy probable que esta limitación afecte al esquema de buffer implementado. El controlador podría manejar el almacenamiento de unos pocos paquetes entrantes, el cual afectaría al rendimiento de la interfaz.

Compartir recursos

Debe tener presente que en ciertas situaciones puede tener múltiple controladores de dispositivos que necesitan acceder a un hardware común (tales como pines de E/S o memoria compartida). Esto hará difícil localizar errores si el esquema de divisiones de responsabilidades y uso de recursos no se piensa a fondo antes.

5 Referencias

Michael Barr. Programming Embedded Systems in C and C++ 1st Edition. ISBN-13: 978-1565923546 ISBN-10: 1565923545. O'Reilly Media; 1 edition (February 9, 1999)

6 Licencia y notas de la traducción

Rafael Ignacio Zurita (rafa@fi.uncoma.edu.ar) 2016-2020

Este apunte es una traducción del libro de referencia, para ser utilizado como apunte en la materia de grado 'Programación de Sistemas Embebidos' de la Facultad de Informática, Universidad Nacional del Comahue. Se han realizado modificaciones al contenido para aclarar ciertos detalles o agregar secciones nuevas. Tambien se han modificado todos los archivos fuentes de código, ya que fueron portados a la plataforma Atmel AVR atmega328.

Esta es una obra derivada del libro de referencia que ha obtenido permiso de O'Reilly para su publicación en PEDCO.

6.0.1 Permiso de publicación

```
From: Teri Finn <teri@oreilly.com>
Date: Mon, 11 Apr 2016 15:48:58 -0700
Message-ID: <CAOgscZ92-G9iT+=+nnuft4LCoHoe5o8SYPfVEKnS6AKXh8TLyQ@mail.gmail.com>
Subject: Re: Question about permission to translate some parts
To: Rafael Ignacio Zurita <rafa@fi.uncoma.edu.ar>
Content-Type: multipart/alternative; boundary=94eb2c094eeedeba0005303d5a31
```

```
--94eb2c094eeedeba0005303d5a31
Content-Type: text/plain; charset=UTF-8

Hi Rafael,

Thank you for providing this further information.  O'Reilly Media is happy
to grant you permission to translation the content you have proposed for
posting to the Moddle site. We're happy you find this information helpful
to the students.

Best regards,

Teri Finn
O'Reilly Media, Inc.

On Thu, Apr 7, 2016 at 10:56 AM, Rafael Ignacio Zurita <
rafa@fi.uncoma.edu.ar> wrote:

[snip]
```