PERIPHERALS

Embedded Software Design

熊博安

國立中正大學資訊工程研究所

pahsiung@cs.ccu.edu.tw

Textbook: Programming Embedded Systems with C and GNU Development Tools, 2nd Edition, Michael Barr and Anthony Massa, O'Reilly

Contents

- Control and Status Registers
- The Device Driver Philosophy
- A Serial Device Driver
- Device Driver Design

Introduction

- Besides processor and memory, there are other hardware devices called "peripherals"
- Type Classification
  - Application domain specific (ASICs)
  - Common ones
    - Timers/counters
    - Serial ports
- Location Classification
  - On-chip or internal (same chip)
  - Off-chip or external (different chips)

Control and Status Registers

- Basic interface between embedded processor and a peripheral device
- Part of peripheral devices
- Register locations, size, and individual meanings are features of the peripherals
- Address mapping:
  - Memory-mapped (popular! easy!)
  - I/O-mapped

Memory-Mapped Device

- Memory-mapped registers
  - Look like ordinary variables (pointers)
  - Example: GPIO registers in PXA255

    ```
    uint32_t *pGpio0Set = (uint32_t *)(0x40E00018);
    ```
  - Difference from ordinary variable
    - Can be changed by hardware
  - Use keyword "**`volatile`**" for register variables
    - Warns compiler not to make any assumptions about the data stored at that address
      - Turns off compiler optimizations on that variable

Use of "volatile" keyword

```
uint32_t volatile *pGpio0Set = (uint32_t
  volatile *)(0x40E00018);
void gpioFunction(void) {
  /* Set GPIO pin 0 high */
  pGpio0Set = 1;      /* First write */

  delay_ms(1000);

  /* Set GPIO pin 1 high */
  *pGpio0Set = 2;     /* Second write */
}
```

Will not be optimized!

Bit Manipulation

- C language operators for bit manipulation
  - &   (AND)
  - | (OR)
  - ~   (NOT)
  - ^   (XOR)
  - <<   (left shift)
  - >>   (right shift)
- How to test, set, clear, toggle individual bits?

Bit Manipulation

- **`pTimerStatus`**
  - A pointer to a timer status register
  - Least Significant Bit (LSB)
    - bit 0
    - represented by 0x01
  - Most Significant Bit (MSB)
    - bit 7
    - represented by 0x80

Testing Bits

- To see whether bit 3 is set in the timer status register using the & operator

```
If (*pTimerStatus & 0x08) {
   /* Do something here … */
}
```

- Suppose *pTimerStatus = 0x4C

```
              0 1 0 0 1 1 0 0 (0x4C)
    AND (&)
              0 0 0 0 1 0 0 0 (0x08)
             =======================
              0 0 0 0 1 0 0 0 (0x08)
```

Setting Bits

- To set bit 4, using | operator

`*pTimerStatus |= 0x10;`

```
            0 1 0 0 1 1 0 0   (0x4C)
OR  (|)
            0 0 0 1 0 0 0 0   (0x10)
           =================
            0 1 0 1 1 1 0 0   (0x5C)
```

Clearing Bits

- To clear bit 2, using & and ~ operators

`*pTimerStatus &= ~(0x04);` ⇦ = 0xFB

```
            0 1 0 1 1 1 0 0   (0x5C)
AND (&)
NOT (~)     1 1 1 1 1 0 1 1   (0xFB)
           =================
            0 1 0 1 1 0 0 0   (0x58)
```

Toggling Bits

- To toggle bit 7, using ^ operator

`*pTimerStatus ^= 0x80;`

```
            0 1 0 1 1 0 0 0   (0x58)
XOR (^)
            1 0 0 0 0 0 0 0   (0x80)
           =================
            1 1 0 1 1 0 0 0   (0xD8)
```

Shifting Bits

- To right shift by 1 bit

  `bitCount >>= 1;`

```
              1  0  1  0  1  1  0  0    (0xAC)
>> by 1
            =================
              0  1  0  1  0  1  1  0    (0x56)
```

- To left shift by 2 bits

  `bitCount <<= 2;`

```
              1  0  1  0  1  1  0  0    (0xAC)
<< by 2
            =================
              1  0  1  1  0  0  0  0    (0xB0)
```

Used when performing an operation on EACH bit of a register:

- Bitmask with one bit set/clear
- Shift it one bit at a time

Bitmasks

- Bitmask
  - A constant used with bitwise operators to manipulate one or more bits in a larger integer field.
  - Used to set, test, clear, toggle bits.
  - Examples

```
#define TIMER_COMPLETE                    (0x08)
#define TIMER_ENABLE                      (0xC0)

if (*pTimerStatus & TIMER_COMPLETE)
{
    /* Do something here... */
}
```

Bitmask Macros

- Handy macro to avoid typos in long hexadecimal literals

```
#define BIT(X)          (1<<(X))
```

- Usage:
- To define a specific register bit in a bitmask, such as bit 22, use this macro

```
#define TIMER_STATUS    BIT(22)
```

Bitfields

- Bitfield
  - A field of one or more bits within a larger integer value.
  - Used for bit manipulations
  - Supported within C struct

```
struct
{
    uint8_t  bit0   : 1;
    uint8_t  bit1   : 1;
    uint8_t  bit2   : 1;
    uint8_t  bit3   : 1;
    uint8_t  nibble : 4;
} foo;
```

Bitfields

- To test bits using bitfield

```
if (foo.nibble == 0x03)
{
    /* Do other stuff. */
}
```

- To set bits using bitfield

```
foo.nibble = 0xC;
```

- To toggle a bit using bitfield

```
foo.bit3 = ~foo.bit3; /* or !foo.bit3 */
```

Bitfield Unions

- Bitfields are not portable
  - Compilers: start either from LSB or MSB!
- Solution
  - Enclose within a “union”

```
union {
        uint8_t byte;
        struct {
                uint8_t bit0   : 1;
                uint8_t bit1   : 1;
                uint8_t bit2   : 1;
                uint8_t bit3   : 1;
                uint8_t nibble : 4;
        } bits;
} foo;
```

Bitfield Unions

- Bitfield unions can be used to
  - Initialize a register

```
foo.byte = (TIMER_COMPLETE | TIMER_ENABLE);
```

  - Still access individual bits

```
foo.bits.bit2 = 1;
```

Struct Overlays

- Overlay a C struct onto a peripheral's control and status registers
- Benefits
  - Read/write through pointer to struct
  - Register described nicely by struct
  - Code can be kept clean
  - Compiler does address construction at compile time

Struct Overlays

Struct member	Offset
count	0x00
maxCount	0x02
_reserved1	0x04
control	0x06

- Example
  - Not properly aligned registers
    - Use reserved members in struct

```
typedef struct
{
    uint16_t count;             /* Offset 0 */
    uint16_t maxCount;          /* Offset 2 */
    uint16_t _reserved1;        /* Offset 4 */
    uint16_t control;           /* Offset 6 */
} volatile timer_t;

timer_t *pTimer = (timer_t *)(0xABCD0123);
```

Struct Overlays

- To test bits

```
if (pTimer->control & 0x08)
{
    /* Do something here... */
}
```

- To set bits

```
pTimer->control |= 0x10;
```

- To clear bits

```
pTimer->control &= ~(0x04);
```

- To toggle bits

```
pTimer->control ^= 0x80;
```

The Device Driver Philosophy

- **Goal**: Hide the hardware completely!!!
- **Device driver module**: the only piece of software that reads or writes registers directly
- **Solution**: Create API that need no change if underlying peripheral is replaced by another from its general class
- **Example**: Flash memory devices all have sectors (but different sizes!), erase an entire sector, write single byte or word, driver should work with all flash memories of different sector sizes

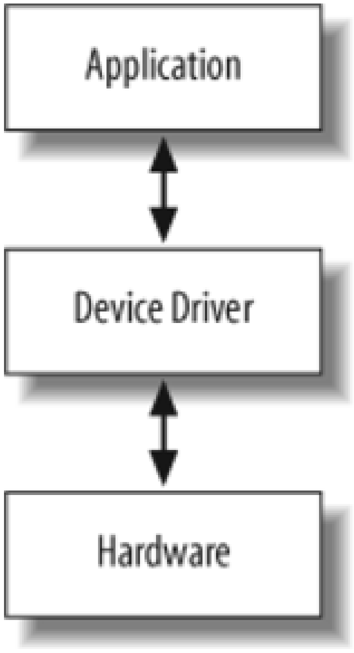
Flash Driver API

```
void flashErase(uint32_t sector);
void flashWrite(uint32_t offset, uint8_t *pSrcAddr,
                uint32_t numBytes);
```

- An erase operation can be performed only on an entire sector.
- Once erased, individual bytes and words can be rewritten.

Benefits of good device drivers

- Modularization: easy to maintain: add or modify features
- Single module with direct access: state of hardware can be more accurately tracked
- Software changes due to hardware changes: localized to device driver



blink.c

led.c

GPIO

Driver Implementation (5 Steps)

- Data Structure: to overlay memory-mapped registers
- State Variables: to track hardware and driver states
- Initialization Routine: to initialize hardware to a known state
- API Routines: for users to use
- Interrupt Service Routines (ISR): for IRQs

1. Data Structure

- Create a C-style `struct` looking like memory-mapped registers (an overlay)
  - study data book for peripheral
  - create table of registers and their offsets
  - begin filling `struct` from lowest offset
  - place dummy variables for unused space

2. State Variables

- Variables to track hardware and driver states:
  - Hardware initialized?
  - Length of timer countdown?
- Multiple software timers using a single hardware timer
  - Length of each timer countdown?

3. Initialization Routine

- After knowing how to check hardware state
- Initialize hardware to a known state
- Good way to learn how to interact with and control hardware

4. API Routines

- To add functionalities to driver
  - Choose names and purposes of various routines
  - Decide on parameters and return values
  - Implement API routines
  - Test API routines

5. Interrupt Service Routines

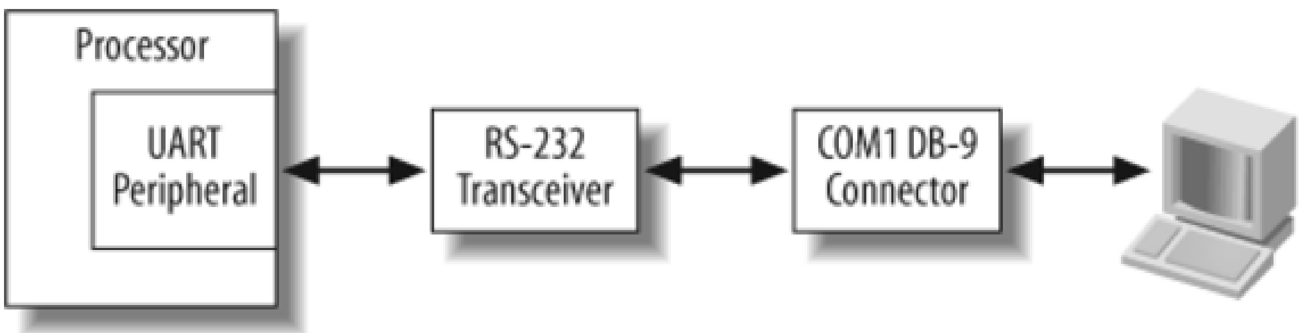
- Best to design, implement, and test most device driver routines BEFORE ENABLING INTERRUPTS for the first time
- Use polling to get the driver working first
- Then, switch to interrupts
- There are often some problems related to interrupts

A Serial Device Driver

- Universal Asynchronous Receiver Transmitter (UART)
  - A serial communication device
  - Transmission
    - A parallel byte is received from processor
    - Byte is serialized
    - Each bit is transmitted at appropriate time

UART

- PXA255 processor has 4 on-chip UARTs
- This example uses the Full Function UART (FFUART) at COM1 port of Arcom's board
  - FFUART registers start at 0x40100000



UART

- Read docs to understand
  - Control register structures
    - How to setup communication?
    - How to get data into and out of the peripheral?
  - Addresses of control and status registers
  - Polling or interrupt?
  - For interrupt-driven communication
    - Interrupt conditions?
    - How is software driver informed of interrupt?
    - How is interrupt acknowledged?

1. Register Interface

- Struct overlay for UART registers (memory-mapped)

```
typedef struct
{
    uint32_t data;
    uint32_t interruptEnable;
    uint32_t interruptStatus;
    uint32_t uartConfig;
    uint32_t pinConfig;
    uint32_t uartStatus;
    uint32_t pinStatus;
} volatile uart_t;
```

- Address:

```
uart_t *pSerialPort = (uart_t *)(0x40100000);
```

2. State Variables

- Serial driver parameters
  - **serialparams_t**
- Initialization tracking
  - **bInitialized**
- Bitmask values: enumerated types for
  - Parity: **parity_t**
  - Data bits: **databits_t**
  - Stop bits: **stopbits_t**

2. State Variables

```
/* UART Config Register (LCR) Bit Descriptions */
#define DATABITS_LENGTH_0                  (0x01)
#define DATABITS_LENGTH_1                  (0x02)
#define STOP_BITS                          (0x04)
#define PARITY_ENABLE                      (0x08)
#define EVEN_PARITY_ENABLE                 (0x10)

typedef enum {PARITY_NONE, PARITY_ODD = PARITY_ENABLE,
              PARITY_EVEN=(PARITY_ENABLE |
              EVEN_PARITY_ENABLE)} parity_t;

typedef enum {DATA_5, DATA_6 = DATABITS_LENGTH_0, DATA_7 = DATABITS_LENGTH_1,
              DATA_8 = (DATABITS_LENGTH_0 | DATABITS_LENGTH_1)} databits_t;

typedef enum {STOP_1, STOP_2 = STOP_BITS} stopbits_t;
```

2. State Variables

```
typedef struct
{
    uint32_t dataBits;
    uint32_t stopBits;
    uint32_t baudRate;
    parity_t parity;
} serialparams_t;

serialparams_t gSerialParams;
```

3. Initialization Routine

```
/*********************************************************************
 *
 * Function:    serialInit
 *
 * Description: Initialize the serial port UART.
 *
 * Notes:       This function is specific to the Arcom board.
 *              Default communication parameters are set in
 *              this function.
 *
 * Returns:     None.
 *
 *********************************************************************/
void serialInit(void)
{
    static int bInitialized = FALSE;

    /* Initialize the UART only once. */
    if (bInitialized == FALSE)
    {
        /* Set the communication parameters. */
        gSerialParams.baudRate = 115200;
        gSerialParams.dataBits = DATA_8;
        gSerialParams.parity = PARITY_NONE;
        gSerialParams.stopBits = STOP_1;

        serialConfig(&gSerialParams);

        bInitialized = TRUE;
    }
}
```

Initialized only once

Programming of registers

4. Device Driver API

- For sending characters
  - **`serialPutChar`**
  - Waits until transmitter is ready
  - Then, sends a single character via serial port
- For receiving characters
  - **`serialGetChar`**
  - Waits until a character is received
    - Data ready bit is checked in UART status register
  - Then, reads a character from serial port

4. Device Driver API (send)

```
#define TRANSMITTER_EMPTY           (0x40)

/**********************************************************************
 *
 * Function:    serialPutChar
 *
 * Description: Send a character via the serial port.
 *
 * Notes:       This function is specific to the Arcom board.
 *
 * Returns:     None.
 *
 **********************************************************************/
void serialPutChar(char outputChar)
{
    /* Wait until the transmitter is ready for the next character. */
    while ((pSerialPort->uartStatus & TRANSMITTER_EMPTY) == 0)
        ;

    /* Send the character via the serial port. */
    pSerialPort->data = outputChar;
}
```

4. Device Driver API (get)

```
#define DATA_READY                    (0x01)

/**********************************************************************
 *
 * Function:     serialGetChar
 *
 * Description: Get a character from the serial port.
 *
 * Notes:        This function is specific to the Arcom board.
 *
 * Returns:      The character received from the serial port.
 *
 *********************************************************************/
char serialGetChar(void)
{
    /* Wait for the next character to arrive. */
    while ((pSerialPort->uartStatus & DATA_READY) == 0)
        ;

    return pSerialPort->data;
}
```

Testing Driver

- Connect COM1 to PC's serial port
- Start HyperTerminal or minicom on PC
  - Use default parameter values
    - Same as those used by RedBoot
- Need a Command Line Interface (CLI) to interact with terminal
  - An indispensable tool commonly implemented in embedded systems

Testing Driver (CLI)

```
#include "serial.h"

/**********************************************************************
 *
 * 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 rcvChar = 0;
```

Testing Driver (CLI)

```
    serialInit( );

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

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

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

    return 0;
}
```

Extending Functionality

- Develop a more robust and useful program
- Selectable configuration
  - **`serialInit`**(initial communication parameters …)
- Error checking
  - Define and return error codes to application
    - parameter error, hardware error, …
- Additional APIs
  - String functions: **`serialGetStr`**, **`serialPutStr`**

Extending Functionality

- FIFO usage
  - FIFOs as buffers for receive and transmit channels ➔ more robust
- Interrupts
  - Better than polling
  - No need of busy waiting for incoming character in **`serialGetChar`**

Device Driver Design

- More than one device driver
- Interrupt priorities
  - Determine and set appropriate priority levels
- Complete requirements
  - Allow peripherals to function fully

Device Driver Design

- Resource usage
  - What resources are needed by a peripheral?
  - Example: Ethernet device needs a large buffer, so don’t use a small buffer, otherwise will affect throughput
- Resource sharing
  - Access to common hardware (e.g. I/O pins) or common memory
  - Think about how to share them!