嵌入式系统
实验基础

刘 健 培

ljp@bupt.edu.cn

北京邮电大学 计算机学院

Outline

- 实验简介
  - 实验环境与参考资料
  - 实验设计
- 使用C语言
  - C程序的编译、链接和加载
  - 嵌入式C语言选讲
  - C与汇编
- 使用实验用嵌入式操作系统内核kern
- I/O编程

Outline

- 实验简介
  - 实验环境与参考资料
  - 实验设计
- 使用C语言
- 使用实验用嵌入式操作系统内核kern
- I/O编程

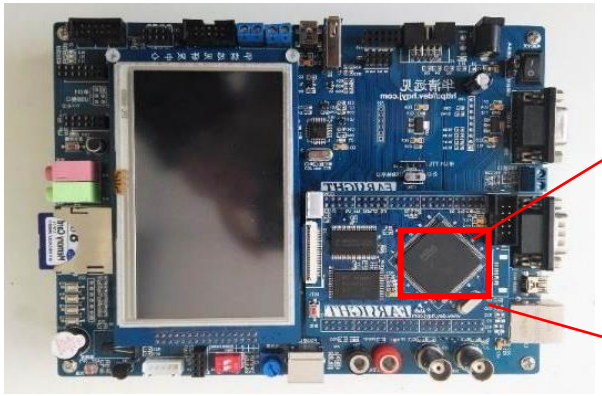
作业提交截止时间

作业	截止时间
Lab1	2021-11-14
Lab2	2021-11-21
Lab3	2021-11-28
Lab4	2021-12-05
Lab5	2021-12-12
Lab6	2021-12-19
Lab7	2021-12-19

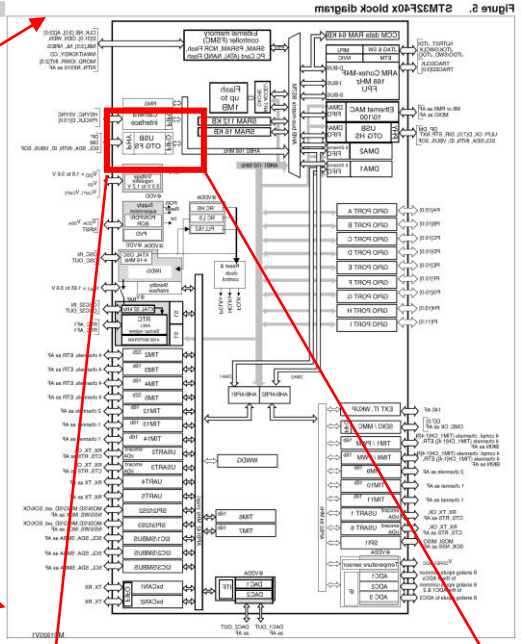
总成绩构成：
作业：实验：期末考试 = 3:4:3

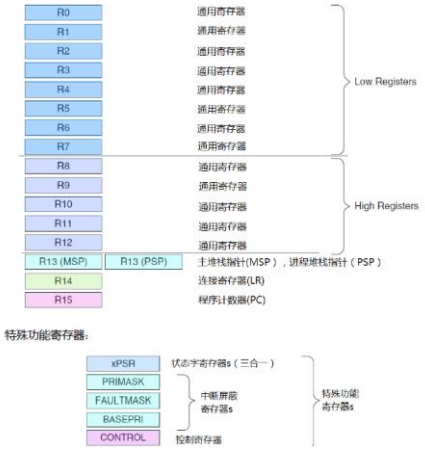
硬件平台

FSM4开发板



STM32F40x MCU





ARMv7-M Architecture

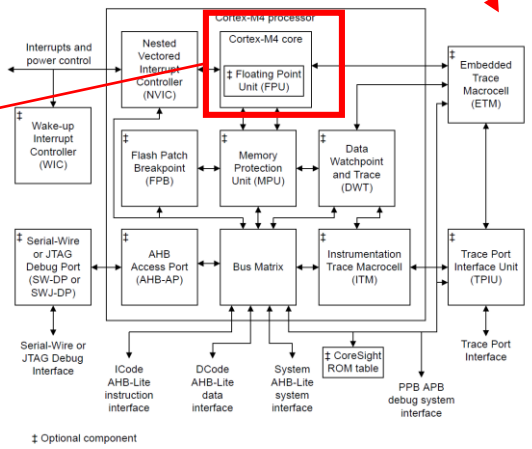
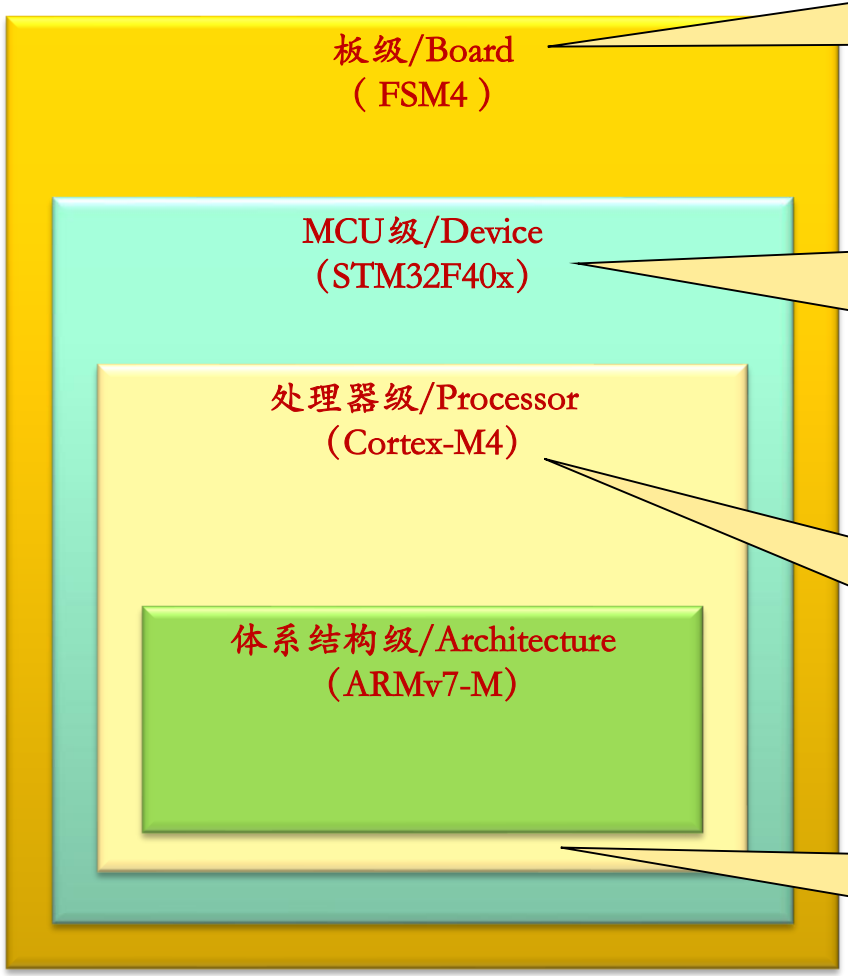


Figure 2-1 Cortex-M4 block diagram

Cortex-M4 processor

查阅资料

- 开发板、微控制器、处理器内核、指令体系结构ISA



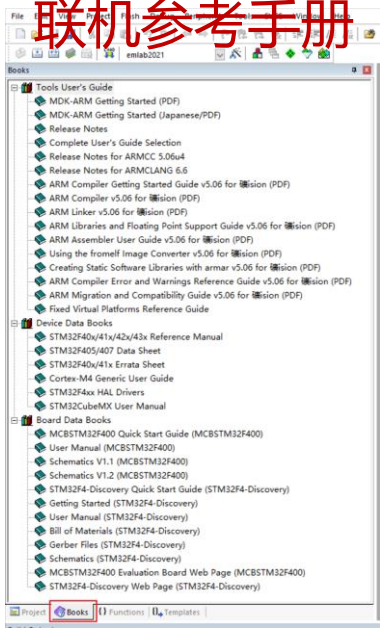
开发板手册：
Cortex-M4实验平台指导书.pdf
FS_STM32F4核心板原理图
FS_STM32F4底板原理图V1.pdf

ST STM32F40x MCU芯片手册：
STM32F4x7-Datasheet.pdf
STM32F4x7-Reference manual.pdf or
STM32F4xx中文参考手册.pdf
图书：嵌入式系统原理及应用—基于ARM CortexM3内核的STM32F407系列微控制器

Cortex-M4处理器手册：
ARM® Cortex®-M4 Processor Technical Reference Manual_trm_100166_0001_00_en r0p1.pdf
图书：ARM Cortex-M3/Cortex-M4权威指南

ARMv7-M体系结构参考手册：
DDI0403E_d_armv7m_arm Armv7-M Architecture Reference Manual.pdf
图书：ARM Cortex-M3/Cortex-M4权威指南

软件开发环境（交叉开发）

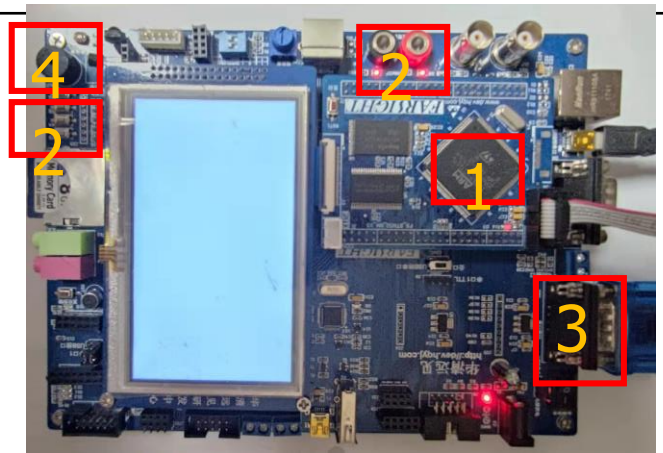

联机参考手册
Books
Tools User's Guide
MDK-ARM Getting Started (PDF)
MDK-ARM Getting Started (Japanese/PDF)
Release Notes
Complete User's Guide Selection
Release Notes for ARMCC 5.06u4
Release Notes for ARMCLANG 6.6
ARM Compiler Getting Started Guide v5.06 for Vision (PDF)
ARM Compiler v5.06 for Vision (PDF)
ARM Linker v5.06 for Vision (PDF)
ARM Libraries and Floating Point Support Guide v5.06 for Vision (PDF)
ARM Assembler User Guide v5.06 for Vision (PDF)
Using the fromelf Image Converter v5.06 for Vision (PDF)
Creating Static Software Libraries with armar v5.06 for Vision (PDF)
ARM Compiler Error and Warnings Reference Guide v5.06 for Vision (PDF)
ARM Migration and Compatibility Guide v5.06 for Vision (PDF)
Fixed Virtual Platforms Reference Guide
Device Data Books
STM32F40x/41x/42x/43x Reference Manual
STM32F405/407 Data Sheet
STM32F40x/41x Errata Sheet
Cortex-M4 Generic User Guide
STM32F4xx HAL Drivers
STM32CubeMX User Manual
Board Data Books
MCBSTM32F400 Quick Start Guide (MCBSTM32F400)
User Manual (MCBSTM32F400)
Schematics V1.1 (MCBSTM32F400)
Schematics V1.2 (MCBSTM32F400)
STM32F4-Discovery Quick Start Guide (STM32F4-Discovery)
Getting Started (STM32F4-Discovery)
User Manual (STM32F4-Discovery)
Bill of Materials (STM32F4-Discovery)
Gerber Files (STM32F4-Discovery)
Schematics (STM32F4-Discovery)
MCBSTM32F400 Evaluation Board Web Page (MCBSTM32F400)
STM32F4-Discovery Web Page (STM32F4-Discovery)

编辑源代码
编辑器
.c/.h源文件
编译
编译器
.o目标文件
链接
链接器
.axf可执行文件
调试
交叉调试器
.axf可调试文件
开发流程
串口终端
显示监控信息
串口
RS232串口线
返回监控信息
串口
执行流程
USB口
ST-Link仿真器
USART
控制器
Jtag/SWD
接口
CPU
执行代码
FLASH
RAM
存储代码
+数据
高地址
栈
stack
堆
heap
未初始化数据段
bss
已初始化数据段
data
代码段
text
低地址
下载程序
设置断点
返回调试信息
调试流程

实验设计

- 1个体系结构实验+3个I/O实验

编号	实验项目	实验内容	实验目的
实验1	异常处理	使用svc异常实现一个系统调用	深入认识Cortex-M体系结构的特点与异常处理的原理、流程
实验2	GPIO操作	读取按键值控制LED灯的状态	深入理解如何通过读写I/O寄存器控制外设。
实验3	基于UART的shell程序	使用轮询、中断、DMA3种方式操作UART，实现一个简单的基于串口的shell	理解I/O编程的3种基本技术：轮询、中断、DMA，以及驱动与上层程序的接口方式。重点掌握中断的处理与中断服务程序的编写。
实验4	基于蜂鸣器的简单音乐播放器	使用定时器的产生PWM驱动蜂鸣器发出不同频率声音，结合按键控制，实现一个简单的音乐播放器应用程序。	理解定时器的使用。了解在操作系统IO子系统下编写标准设备驱动的编程方法。了解基于RTOS的多任务应用程序的编写。



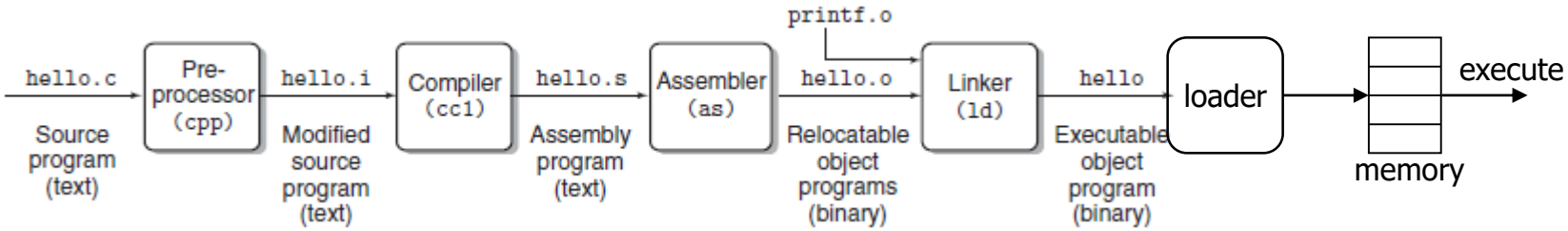
Outline

- 实验简介
- 使用C语言
  - C程序的编译、链接和加载
    - 嵌入式C语言选讲
    - C与汇编
- 使用实验用嵌入式操作系统内核kern
- I/O编程

与C开发有关的问题

- 怎么把源代码(文本)转换成CPU可执行的机器指令码(二进制)?
- 怎么把二进制代码加载到机器运行?
- 二进制代码在内存中是什么状态?
- 代码运行过程中，内存中是什么状态?

C程序的编译、链接和加载



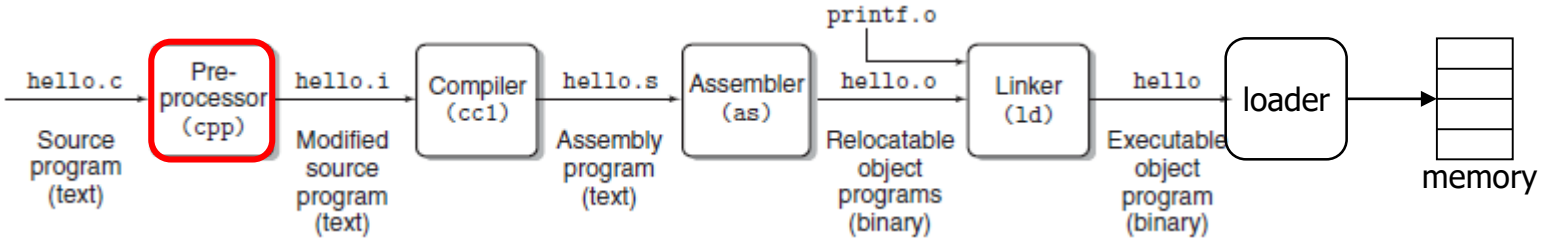
- C编译模型：**分别编译，统一链接**
  - 编译系统分4个阶段：预处理、编译、汇编和链接。
  - 将单个c源文件进行预处理、编译、汇编，生成单个.o目标文件
  - 链接器将多个.o目标文件和库文件彼此相连接，生成可执行文件
  - 加载器将可执行文件加载到内存中执行
  - 示例：源文件hello.c，编译器使用arm gcc

```
cmd> gcc -v -o hello.out hello.c
cpp [args] hello.c /tmp/ccC07Hfe.i
cc1 /tmp/ccC07Hfe.i m.c [args] -o /tmp/ccA4COHB.s
as [args] -o /tmp/cciHpc7F.o /tmp/ccA4COHB.s
ld -o hello.out [system obj files] /tmp/cciHpc7F.o
```

```c
#include <stdio.h>

/*The hello program*/
int main()
{
    printf("hello world\n");
    return 0;
}
```

C程序的编译、链接和加载



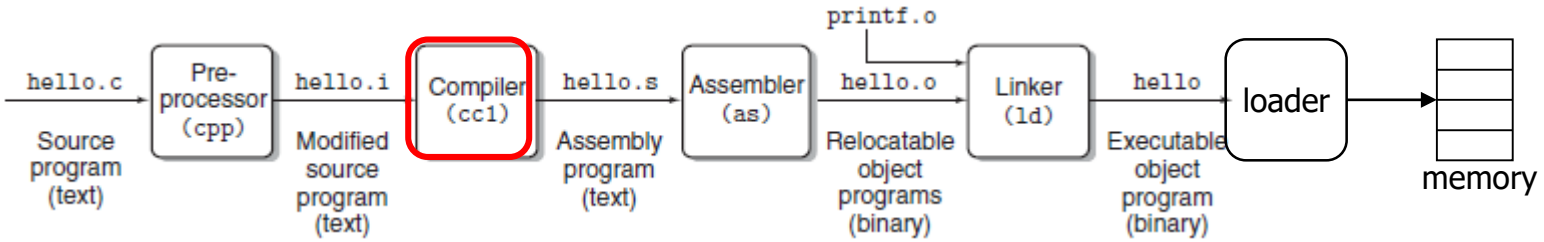
- 预处理阶段preprocessor
  - 预处理器(cpp)根据字符#开头的预编译指令，修改原始的C 程序hello.c，生成另一个C 程序hello.i
  - 主要处理规则:
    - 删除所有的#define, 并展开所有的宏定义;
    - 处理所有的条件预编译指令，如#if、#ifdef等;
    - 处理“#include"预编译指令;
    - 删除所有注释;
    - 添加行数和文件名标识;

```
.....
# 2 "hello.c" 2


int main()
{
 printf("hello world\n");
 return 0;
}
```

gcc -E hello.c -o hello.i
或 cpp hello.c -o hello.i

C程序的编译、链接和加载

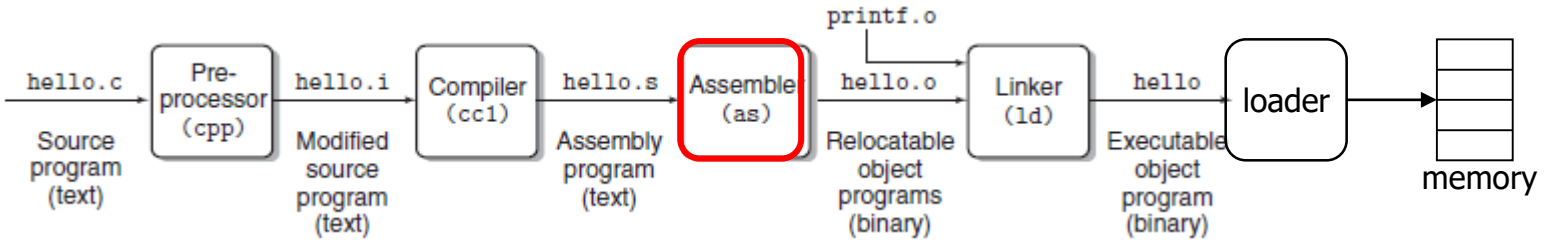


- 编译阶段compiler
  - 编译器（cc1）把预处理完的文本文件hello.i进行一系列词法分析、语法分析、语义分析以及优化后生成相应的汇编代码文本文件hello.s

```
main:
    @ Function supports interworking.
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 1, uses_anonymous_args = 0
    stmfd   sp!, {fp, lr}
    add fp, sp, #4
    ldr r0, .L3
    bl  puts
    mov r3, #0
    mov r0, r3
    sub sp, fp, #4
    @ sp needed
    ldmfd   sp!, {fp, lr}
    bx  lr
```

gcc -S hello.c -o hello.s
或 gcc -S hello.i -o hello.s
或 cc1 hello.c -o hello.s

C程序的编译、链接和加载

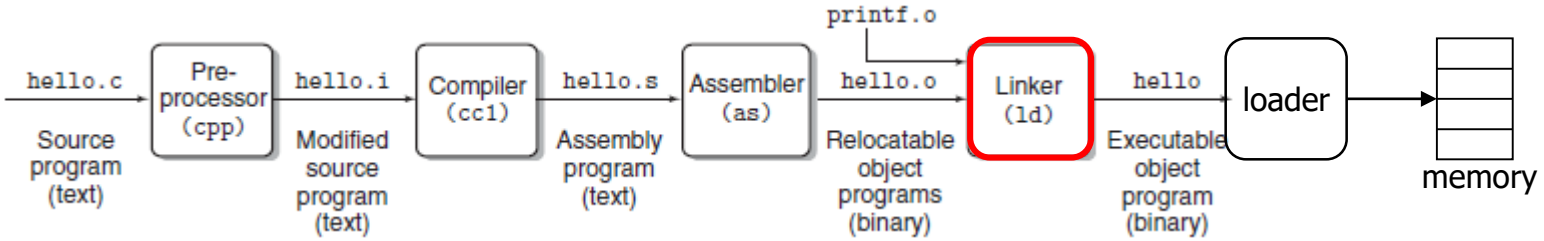


- 汇编阶段assembler
  - 汇编器（as）将汇编代码hello.s转变为机器指令，把这些指令打包成可重定位目标程序（relocatable object program，ELF格式，其中代码起始地址为0），并将结果保存在目标文件hello.o中
  - 程序编译后，主要由2部分构成：代码和数据

```
00000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 .ELF............
00000010 01 00 28 00 01 00 00 00 00 00 00 00 00 00 00 00 ..(.............
00000020 60 01 00 00 00 00 00 05 34 00 00 00 00 00 28 00 `.......4.....(.
00000030 0b 00 08 00 00 48 2d e9 04 b0 8d e2 14 00 9f e5 .....H-???.燴□3鴦
00000040 fe ff ff eb 00 30 a0 e3 03 00 a0 e1 04 d0 4b e2 ?  ?0.?..?廬?□靈
00000050 00 48 bd e8 1e ff 2f e1 00 00 00 00 68 65 6c 6c .H借.  /?...hellD
00000060 6f 20 77 6f 72 6c 64 00 00 47 43 43 3a 20 28 47 o world..GCC: (G
```

gcc -c hello.c -o hello.o
或 gcc -c hello.s -o hello.o
或 as hello.s -o hello.o

C程序的编译、链接和加载

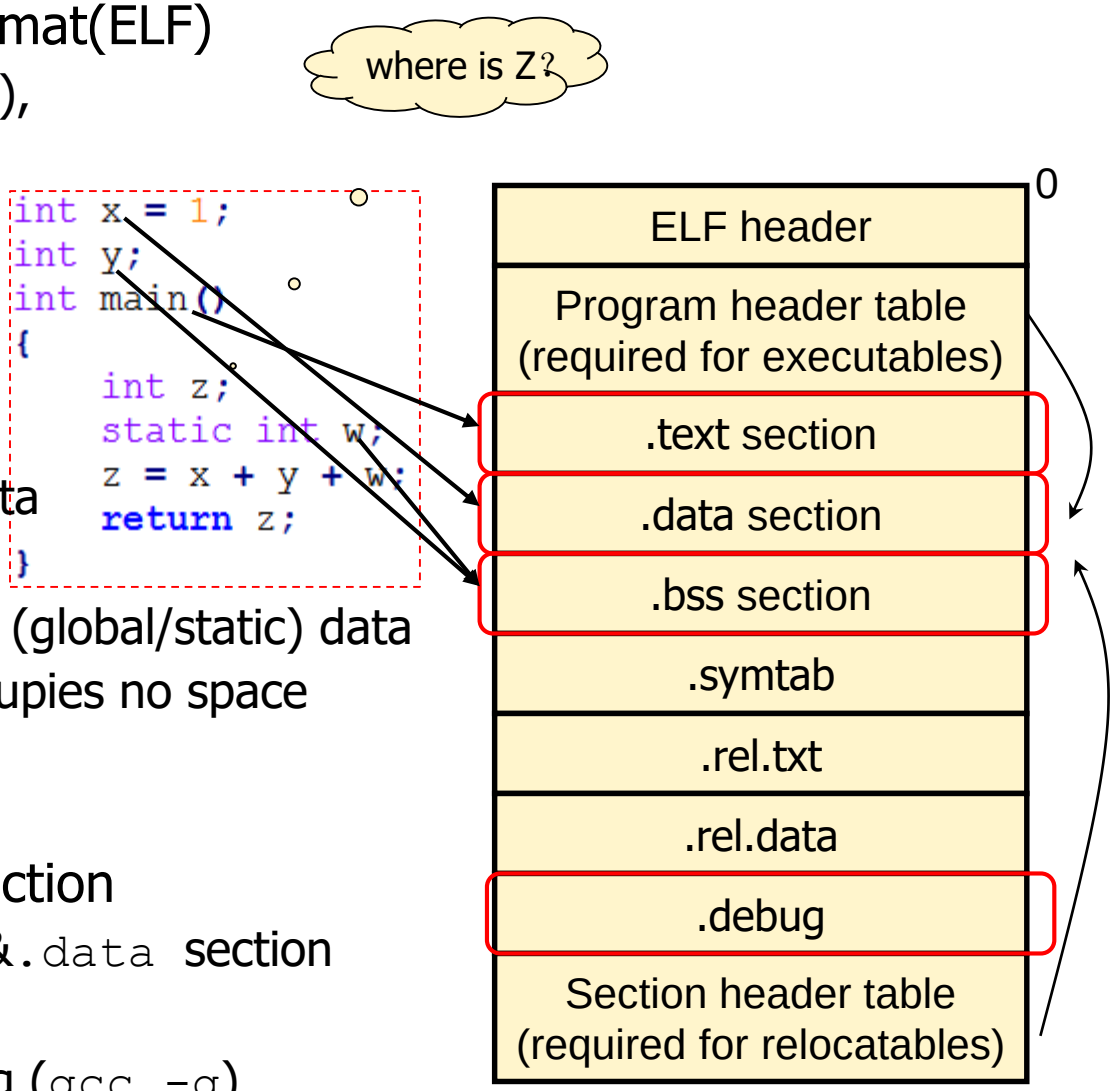


- 链接阶段linker
  - 汇编过程生成的目标文件并不能直接执行（如引用了库函数printf，没有确定绝对地址），链接器(ld) 将有关的目标文件彼此相连接/合并，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够按加载器装入执行的可执行文件(ELF格式).
    - 符号解析 (Symbol resolution)，即将每个符号引用刚好和一个符号定义联系起来
    - 地址重定位(Relocation)，即把每个符号定义与一个存储位置联系起来，然后修改所有对这些符号的引用，使得它们指向这些位置。
    - 一般链接器会在输出文件中额外生成少量的代码和数据，如初始化代码等。
  - 链接可以发生在编译时、加载时、运行时。
  - 在嵌入式开发中，一般需要通过链接脚本指定起始代码起始地址。

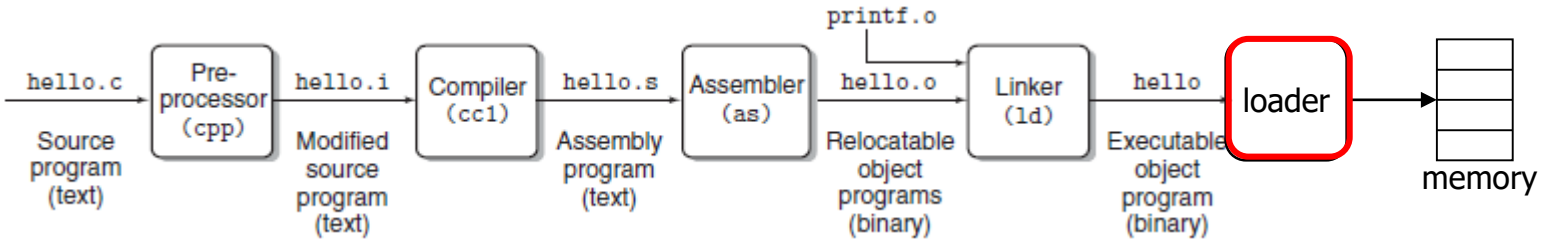
```
ld hello.o –o hello.out –lc –L[lib dir]
```

ELF目标文件格式

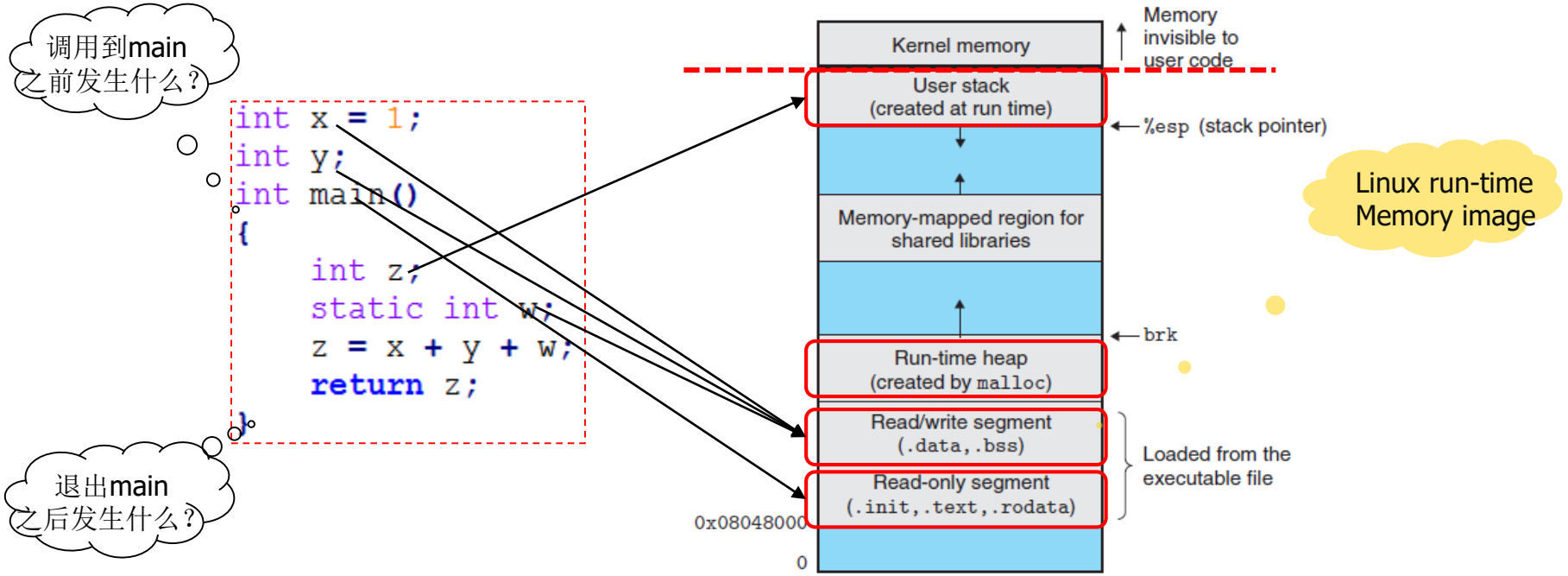
- Executable and Linkable Format(ELF)
  - Relocatable object files (`.o`),
  - Executable object files
  - Shared object files (`.so`)
- `.text` section/节
  - Code
- `.data` section
  - Initialized (global/static) data
- `.bss` section
  - Uninitialized/zero-initialized (global/static) data
  - Has section header but occupies no space
- `symtab` section
  - Symbol table
- `rel.text` & `rel.data` section
  - Relocation info for `.text` &`.data` section
- `debug` section
  - Info for symbolic debugging (`gcc -g`)



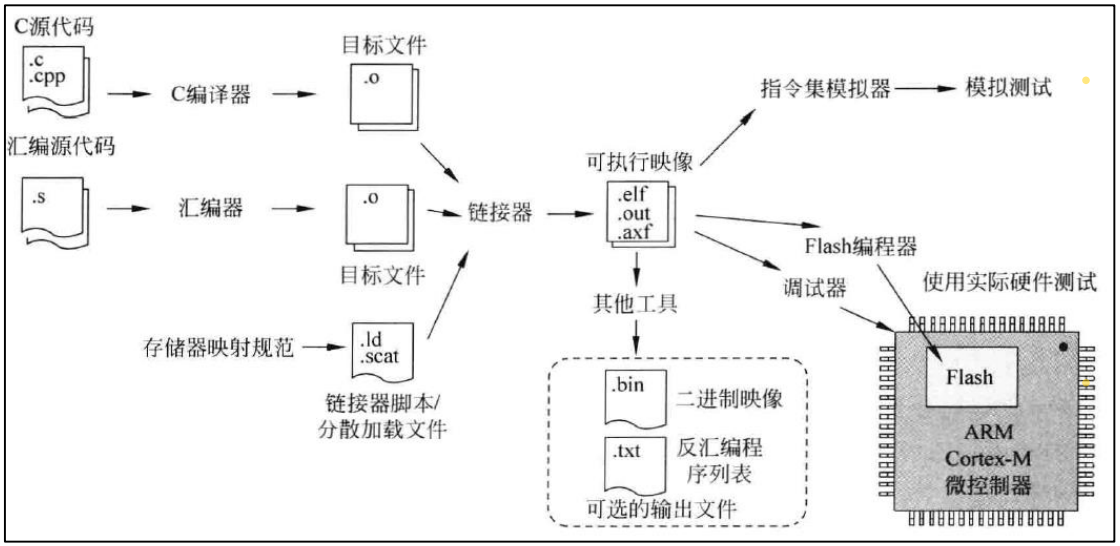
C程序的编译、链接和加载



- 加载器loader
  - 加载器将可执行文件中的代码和数据段从磁盘copy到内存中，然后将程序跳转到程序的入口地址来运行该程序。

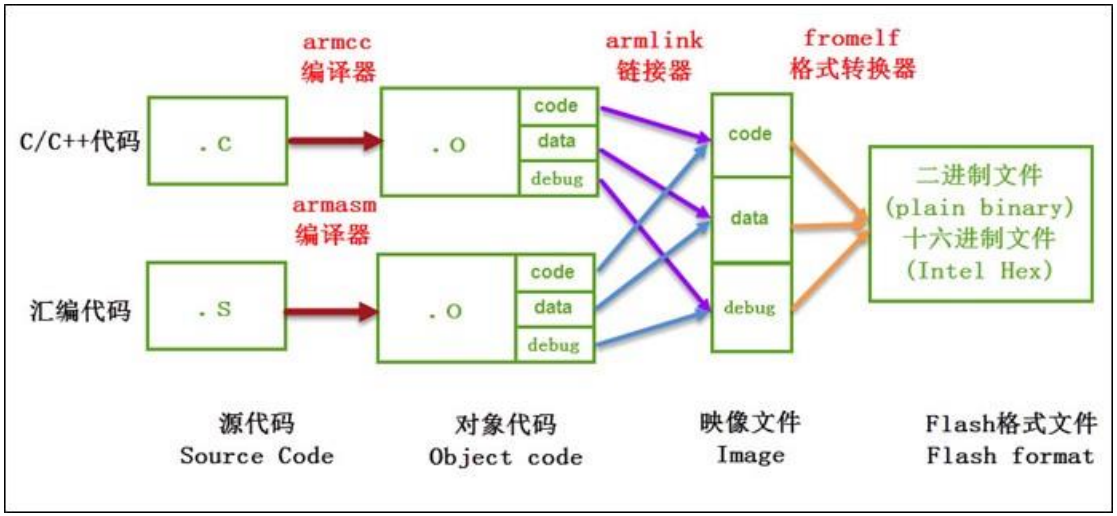


MDK中的编译、链接和调试加载



作业环境

实验环境



Outline

- 实验简介
- 使用C语言
  - C程序的编译、链接和加载
  - 嵌入式C语言选讲
    - 内存的使用
    - 函数指针的使用
    - 宏的使用
    - 头文件的使用
    - Bit操作
    - volatile变量
    - 编写高质量和安全的代码
  - C与汇编

C Memory Model

- Array-of-bytes model of memory
  - Memory is viewed as an array of bytes with addresses
  - Byte Addressing: address points to a byte in memory
- Memory Layout of C-Program consists of 5 Segments/段:
  - Text or Code、Initialized Data、Uninitialized Data、Heap、Stack

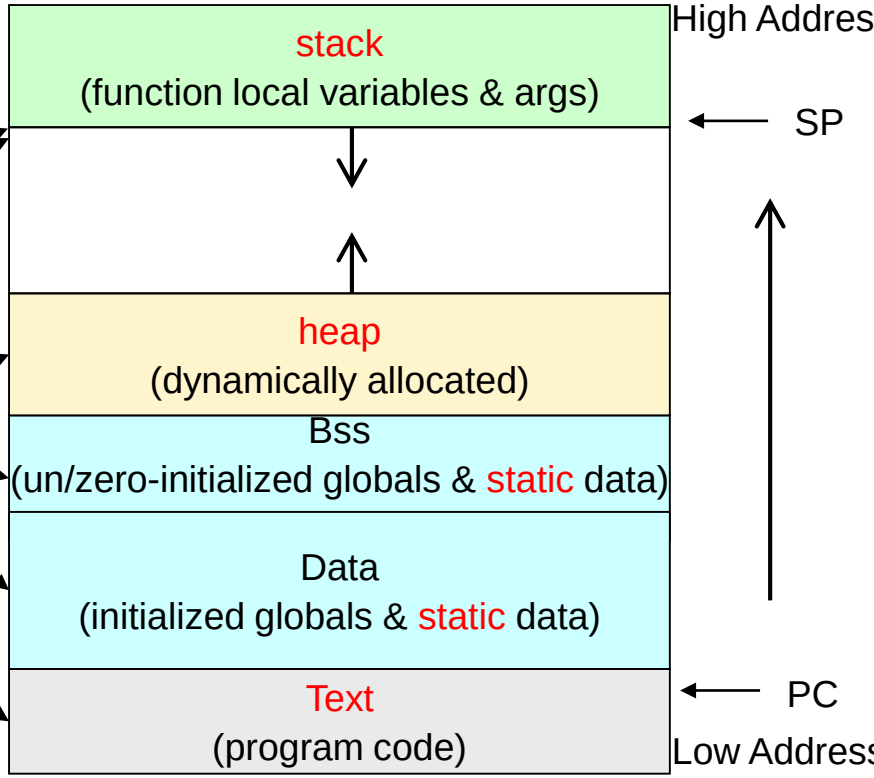
一个合格的嵌入式c程序员应很清楚他所写的代码在内存中的状态(位置与布局)。

```
char *str = "hello world";

int size;

char *f (int x)
{
  char *p;

  size = 4;
  p = (char *)malloc (size);
  return p;
}
```



Variable Initialization & Lifetime

When initialized?

- text:
  - readonly
- data：
  - on program startup
- bss:
  - un-initialized (though some systems initialize with 0)
- heap:
  - un-initialized
- stack:
  - un-initialized

When startup & finish?

- text:
  - program startup
  - program finish
- data, bss:
  - program startup
  - program finish
- heap:
  - dynamically allocated
  - de-allocated (free)
- stack:
  - function call
  - function return

C Memory Usage

- 内存的分配，3种方式
  - 静态存储区static data area：编译时已经分好（即已经编址），应用程序启动前已被初始化好，如global 和 static变量。
  - 堆栈stack/automatic：函数执行时，局部变量和参数在堆栈上创建，在退出函数时自动释放。注意防止堆栈溢出。
  - 堆heap/动态内存分配：运行期间程序员使用malloc或者类似函数申请任意数量的内存，程序员自己掌握内存释放的时机，这是最容易产生bug的一种方式。

启动代码管理

编译器管理

程序员管理

- 动态内存可能的问题
  - 可能出现内存碎片，导致程序变慢或者后续分配失败
  - 分配失败需要检查返回值
  - 动态内存使用完可能忘记释放导致“内存泄漏”，或者使用错误的指针释放、多次释放同一内存、释放后又继续使用，这些都可能导致程序崩溃（一般表现为段错误）。

C Memory Usage

- 内存使用的规则
  - 一般原则：如果使用堆栈和静态存储可以满足要求，就不要使用动态存储。
  - 使用malloc申请动态内存后，应该立即检查返回值是否为NULL
  - 不要忘记初始化指针、数组和动态内存，防止将未初始化的内存作为右值使用。
  - 避免数组或者指针的越界，防止堆栈溢出或者堆溢出。
  - 动态内存的申请和释放必须刚好一一配对，防止内存泄漏。(资源配对不仅仅适用于内存，也适用于系统管理的其余资源，如信号量、文件、socket、数据库连接等，这是一种保持程序状态复原的思想)
  - 用free释放内存时，先检查指针是否为NULL，释放后，立刻将指针置为NULL，防止“野指针”。
  - 注意大小端与内存对齐问题

Function Pointers

- 函数指针=函数名==函数体代码的首地址

```
/* 申明和定义 */
typedef int (*COMPARE_FUNCPTR)( const void *, const void * );
COMPARE_FUNCPTR fp1, fp2;
/* int (*fp1) (const void *, const void *);
 * int (*fp2) (const void *, const void *); */

int compare( const void *a, const void *b ) {
        return(strcmp( (const char *) a, (const char *) b ) );
}

int main() {
        /* 赋值：&可用可不用，建议用 */
        fp1      = compare;          /* short form */
        fp2      = &compare;         /* recommended form */
        /* 调用：对函数指针，*可用可不用，建议用 。以下调用的都是 compare函数 */
        fp1( "ab", "ac" );    /* short form */
        (*fp1)("ab", "ac");  /* recommended form */
        fp2( "ab", "ac" );
        (*fp2)("ab", "ac");
        /* 比较：fp1 == *fp1 == fp2 == *fp2 == compare == &compare */
}
```

Function Pointers

C语言申明的语法较为晦涩(特别是加上函数指针后)，遵循的基本原则是：申明的形式要与使用的形式相似。

The right-left rule:

1. Start with identifier in the innermost parentheses
2. Go right, then go left
3. Reverse direction when parenthesis is encountered

```
void (*fp)(int);
int (*(*foo)(const void *))[3];
void (*signal(int, void (*)(int)))(int);
```

使用typedef可以简化复杂的申明式　　https://cdecl.org/

```
typedef void (* FUNCPTR)(int);
FUNCPTR signal(int, FUNCPTR);
```

/* 示例1：作为参数传给函数。用作回调函数(callback)/钩子函数(hook) *

回调：调用者/Caller传递一个函数指针，被调用者Callee在函数体内“回调”该函数

```
/* 定制快排比较规则 */
void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *));
int namecmp( const void* s1, const void* s2 ){
        return( ( (char *) s1)[0] - ( (char *) s2)[0]);
}
char names[][10] = { "guna", "mccain", "obama", "paul", "barr" };
qsort( *names, 5, 10, namecmp );
```

```
/* 实验中创建任务 */
task_t * task_create(task_func_t func, void *arg, int prio, const char *name);
static void idle_task(void){
    while(1);
}
task_create((task_func_t)idle_task, NULL, PRIO_LOWEST, "idle_task");
```

```
/* 实验3中挂载UART字符收发函数*/
typedef int (*getc_func_t)(void);
typedef int (*putc_func_t)(char);
void kshell_register(getc_func_t, putc_func_t)
kshell_register(uart_getc_IT, uart_putc_IT);
```

/*示例2：存储于结构体。用于实现 接口/多态/ADT抽象数据类型/函数跳转表*/

/* linux内核中设备文件需要实现的文件接口，允许动态添加设备驱动 */

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
        ......
};

static const struct file_operations led_proc_fops = {
        .owner          = THIS_MODULE,
        .open           = led_proc_open,
        .read           = seq_read,
        .llseek         = seq_lseek,
        .release        = single_release,
        .write          = led_proc_write,
};
```

Macro Preprocessor

- 使用#define宏时要非常谨慎，尽量以内联(inline)函数、枚举(enum)和常量(const)代替之。
  - 宏意味着你和编译器看到的代码是不同的，而且宏的展开使用字符串替换，不会经过编译器的语法检查，宏还具有全局作用域，这可能会导致微妙的异常行为。

```
#define max(x, y) (((x) < (y)) ? (y) : (x))
int i = 2;
int j = 3;
int k = MAX(i++, j++);/*k=4, i=3, j=5 !*/
```

- 嵌入式c编程中也经常使用宏对程序进行预处理，达到特殊效果
  - 一般不要直接使用魔数(特定数字，例如寄存器的地址或者某个字段的值，或者固定常量)，而是使用#define将数字定义成宏，这样修改时，只需要改动宏一处，便于重构，也不容易出错。

```
#define TRUE 1
#define rBWSCON    (*(volatile unsigned *)0x48000000)
```

  - 一般c代码的剪裁都是通过#ifdef XXX来实现。
  - 将代码中的调试语句使用#ifdef DEBUG ... #endif包含起来，然后在debug版本中#define DEBUG，在release版本中#undef DEBUG，就能去除release版本中的调试语句。

```
#define debug_printf(fmt, ...) \
        do { if (DEBUG) fprintf(stderr, "%s:%d:%s(): " fmt, __FILE__, \
                                __LINE__, __func__, __VA_ARGS__); } while (0)
```

Header Files

- 通常一个c文件相当于一个模块，每一个.c文件都有一个对应的.h文件，相当于模块的接口。如果文件较复杂，还可以分为一个对外的接口头文件和对内的私有头文件。
- 在 .c 文件中定义一个不需要被外部引用的变量时，建议声明为 static 。但是不要放在 .h 文件中。
- 头文件只申明变量或函数，不定义变量，防止变量重复定义的错误。变量定义应该放在.c文件中。
- 正确使用头文件可令代码在可读性、文件大小和编译性能上大为改观。
- 头文件的包含顺序：一般->特殊 or 特殊 -> 一般。
- 所有头文件都应该使用#define 来防止头文件被多重包含。

foo/bar/baz.h 可按如下方式保护:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
...
#endif // FOO_BAR_BAZ_H_
```

Bitwise Operators

- 嵌入式c中常需要对bit位进行操作

and: &
or: |
xor: ^
not: ~
left shift: <<
right shift: >>

```
#define rGPIOAM (*(volatile unsigned *)0x0x40020000)
#define MASK 0x00000002
if (reg & MASK) { ... }          /* Check bits */
reg |= MASK;                     /* Set bits */
reg &= ~MASK;                    /* Clear bits */
reg ^= MASK;                     /* Toggle bits */
v = (reg & MASK) >> 1;           /* Select field */
reg = reg & ~(1 << n) | (x << n); /* Change bits */
/*set some bits in reg rGPBDAT*/
rGPIOAM = (rGPIOAM & ~(MASK)) | ((data & MASK));
```

```
/*Cheat Sheet*/
/* t=target variable, n=bit number to act upon 0..n */
#define BIT_SET(t,n) ((t) |= (1ULL<<(n)))
#define BIT_CLEAR(t,n) ((t) &= ~(1ULL<<(n)))
#define BIT_FLIP(t,n) ((t) ^= (1ULL<<(n)))
#define BIT_CHECK(t,n) ((t) & (1ULL<<(n)))
/* t=target variable, m=mask */
#define BITMASK_SET(t,m) ((t) |= (m))
#define BITMASK_CLEAR(t,m) ((t) &= (~(m)))
#define BITMASK_FLIP(t,m) ((t) ^= (m))
#define BITMASK_CHECK_ALL(t,m) (((t) & (m)) == (m)) // warning: evaluates m twice
#define BITMASK_CHECK_ANY(t,m) ((t) & (m))
#define MODIFY_REG(reg, clearmask, val) ((reg) = (((reg) & (~(clearmask))) | val)))
```

volatile变量 – 让我保持原样

- volatile：易变的、不稳定的。
  - 在C语言中，volatile关键字用于提醒编译器它后面所定义的变量随时有可能改变，不让编译器进行优化。因此编译后的程序每次需要读取/存储这个变量的时候，都要直接从变量地址中读取/写入数据。
- 嵌入式开发中，volatile主要用于以下场合
  - 中断服务程序中修改的供其他程序检测的变量
  - 多任务环境下各任务间共享的标志
  - 存储器映射的硬件寄存器

Nine ways to break your systems code using volatile
<https://blog.regehr.org/archives/28>

写高质量的代码

- 正确、简单、清晰、可测
- 高质量的代码来源于高质量的设计和良好的编程风格
  - 高质量的设计
    - 设计上追求简单和低耦合
    - 将系统模块化，仔细定义接口，以便能在某段时间内专注于某一部分
  - 良好的编码风格
    - 不要使用过于复杂（人不容易弄懂）或者你不理解的语句和表达式：如多于一次的++、深度嵌套的循环或条件判断、 goto、过长的表达式或者子程序、复杂的指针操作、运算符优先级不加括号等。
    - 用简单、直接、易读的方式写程序，避免“滥用”语言。
    - 使用直观、有意义、无歧义的命名方法。
    - 写有意义的注释，程序注释应该是告诉别人你的意图和想法，简洁/易读的代码比详细的注释更有意义。

KISS：Keep it simple, stupid

DRY: Don’t repeat yourself

Programs must be written for people to read, and only incidentally for machines to execute.
— Harold Abelson 等人[SICP]

养成防御式编程的习惯

- Defensive programming
  - Making the software behave in a predictable manner despite unexpected inputs、user actions or internal errors (bugs).
  - Detect problems as early as possible .
  - Implicit assumptions are tested explicitly.
- Examples
  - Assertion checking (e.g., validate parameters)
  - Parameter Checking
  - Check function return value
  - Test i <= n not i == n
  - Build debugging code into program with a switch to display values at interfaces
  - Error checking codes in data (e.g., checksum or hash)

```
assert(idx <= (len - 1)); /*防止数组越界*/
assert(size >= 0);        /*防止值为负数*/
assert(dividend != 0);  /*防止除数为0*/
assert(NULL != buf);  /*防止野指针*/
```

写安全的代码 - MISRA C (2004)

- MISRA C是由汽车产业软件可靠性协会 (The Motor Industry Software Reliability Association) 提出的C语言开发标准。其目的是在增进嵌入式系统的安全性及可移植性。包括航太、电信、国防、医疗设备、铁路等领域中都已有厂商使用MISRA C。
- 目标：安全第一
- 有强制规则**(required)** 122 条，推荐规则 **(advisory)** 20 条

分类	强制规则	推荐规则	分类	强制规则	推荐规则
开发环境	4	1	表达式	9	4
语言外延	3	1	控制表达式	6	1
注释	5	1	控制流	10	0
字符集	2	0	Switch 语句	5	0
标识符	4	3	函数	9	1
类型	4	1	指针和数组	5	1
常量	1	0	结构体和联合体	4	0
声明和定义	12	0	预处理命令	13	4
初始化	3	0	标准库	12	0
算术类型转换	6	0	运行失败	1	0
指针类型转换	3	2	—	—	—

MISRA C (2004) 部分条例

- 1.2 (req) 不能有对未定义行为或未指定行为的依赖性。
- 6.3 (adv) 应该使用指示了大小和符号的typedef以代替基本类型。
- 8.5 (req) 头文件中不应有对象或函数的定义。
- 8.8 (req) 外部对象或函数应该声明在唯一的文件中。
- 9.1 (req) 所有自动变量在使用前都应被赋值。
- 10.1 (req) 下列条件成立时，整型表达式的值不应隐式转换为不同的基本类型。
- 11.1 (req) 转换不能发生在函数指针和其他除了整型之外的任何类型指针之间。
- 14.1 (req) 不能有不可到达（unreachable）的代码。
- 14.4 (req) 不应使用goto语句。
- 15.3 (req) switch语句的最后子句应该是default子句。
- 13.4 (req) for语句的控制表达式不能包含任何浮点类型的对象。
- 16.2 (req) 函数不能调用自身，不管是直接还是间接的。
- 16.10 (req) 如果函数返回了错误信息，那么错误信息应该进行测试。
- 17.5 (adv) 对象声明所包含的间接指针不得多于2级。
- 20.1 (req) 标准库中保留的标识符、宏和函数不能被定义、重定义或取消定义。
- 20.3 (req) 传递给库函数的值必须检查其有效性。
- 20.4 (req) 不能使用动态堆的内存分配 。

嵌入式环境C编程注意事项

- 编译器优化可能会影响程序行为
  - 优化可能调整代码顺序，如果是I/O操作可能不允许调整代码顺序。
  - 用循环做的delay可能会不准确
  - 优化可能删除死代码，而在嵌入式系统中，这些代码可能是中断服务程序或特定功能的代码
  - 编译器可能认为变量不会被意外改变，从而将变量放到寄存器中，以提高访问速度。但是变量如果是I/O寄存器、中断或者线程间共享的全局变量，就不能使用保存在寄存器里的备份，而是必须每次都访问变量的内存。C中通过volatile关键字强制编译器从内存读（拒绝寄存器缓存）。

```
void DelayMs(int ms)
{
  int j;

  for (j = 0; j < ms*100; j++)
  {}
}
void DelayMs(int ms)
{
  int volatile i, j;

  for (j = 0; j < ms*100; j++)
  {i = i;}
} .
```

```
39: rTCON &= 0xF;    /* stop */
40: rTCFG0 |= 15;    /* set */
41: rTCON = 0x9;     /* start */
```

- C库中不是所有函数都能使用
  - C标准运行库的一部分与OS无关，但是有一部分需要OS的支持，如I/O、内存管理、时间、浮点运算等，使用这些函数可能导致链接错误，也可能不是线程安全的。
    - printf、read、write、malloc、time、etc.

```
#define rTCON   (*(volatile unsigned *)0x51000008)
#define rTCFG0  (*(volatile unsigned *)0x51000000)
```

嵌入式环境C编程注意事项

- 尽量避免全局变量的使用
  - 一是会导致函数不可重入，二是大大增加总是需要兼顾的代码比例。
  - 如果全局变量只在一个文件中使用，定义为static的，便于信息隐藏。
  - 如果全局变量要在多个文件中使用，只能在一个文件中定义。
- 内存地址空间一般是不隔离的
  - 嵌入式系统中一般所有代码都在同一个地址空间，任何代码中的野指针、越界访问等都会导致系统崩溃
- 内存是有限的
  - 不要再堆栈中定义大数组，可能会溢出
  - 每个任务的堆栈都是有限的，过深的函数调用可能导致溢出
  - 动态申请内存要检查返回值
- 特别注意某些容易犯错的语言特性

Global Variable are Evil:

•Global Variables Are Bad on C2 Wiki

•Chapter 19: Global Variables Are Evil from *Better* Embedded System Software

```
void uartPrintf(char *fmt,...)
{
    va_list ap;
    char string[1024];

    va_start(ap,fmt);
    vsprintf(string,fmt,ap);
    sysUartSendString(string);
    va_end(ap);
}
```

```
if ((pMem = malloc (nBytes)) != NULL)
    bzero ((char *) pMem, (int) nBytes);
```

Risky programming constructs

- Pointers
- Dynamic memory allocation
- Floating-point numbers
- Parallelism
- Recursion
- Interrupts

Outline

- 实验简介
- 使用C语言
  - C程序的编译、链接和加载
  - 嵌入式C语言选讲
  - C与汇编
- 使用实验用嵌入式操作系统内核kern
- I/O编程

C与汇编

- 数据的存储与运算
  - 存储、寻址、传输、运算
  - 地址与寻址
- 控制结构
  - 跳转、条件分支、循环、函数调用与返回
- 内联汇编
  - C访问汇编
  - 汇编访问C

Outline

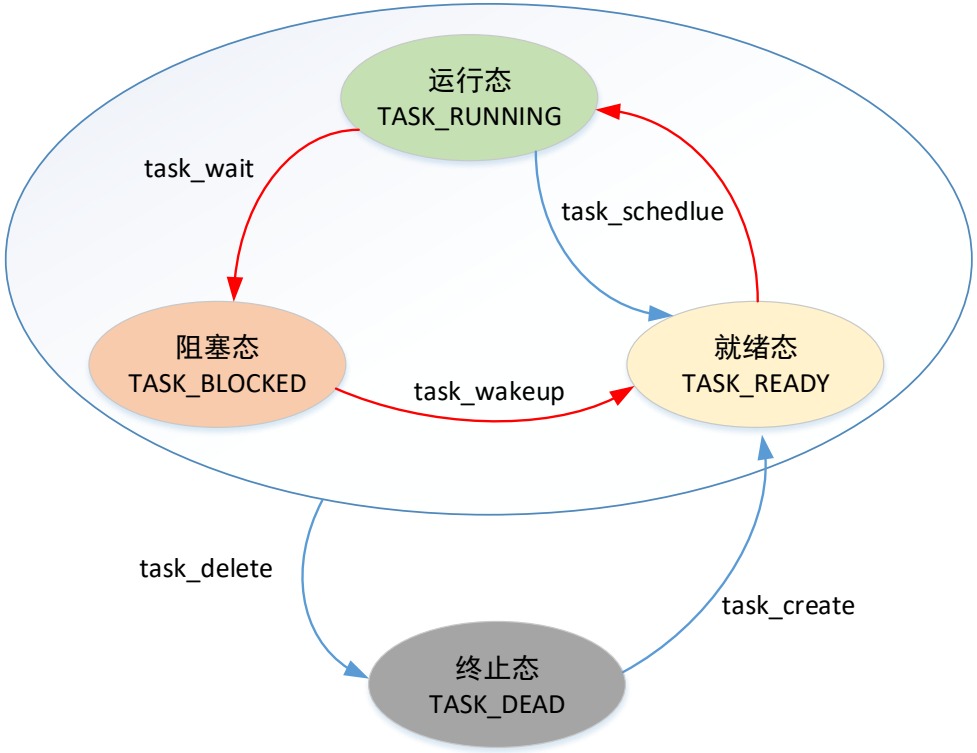
- 实验简介
- 使用C语言
- 使用实验用嵌入式操作系统内核kern
- I/O编程

使用实验用嵌入式操作系统内核kern

- 内核结构

使用实验用嵌入式操作系统内核kern

- 多任务调度模型



Outline

- 实验简介
- 使用C语言
- 使用实验用嵌入式操作系统内核kern
- I/O编程

I/O编程

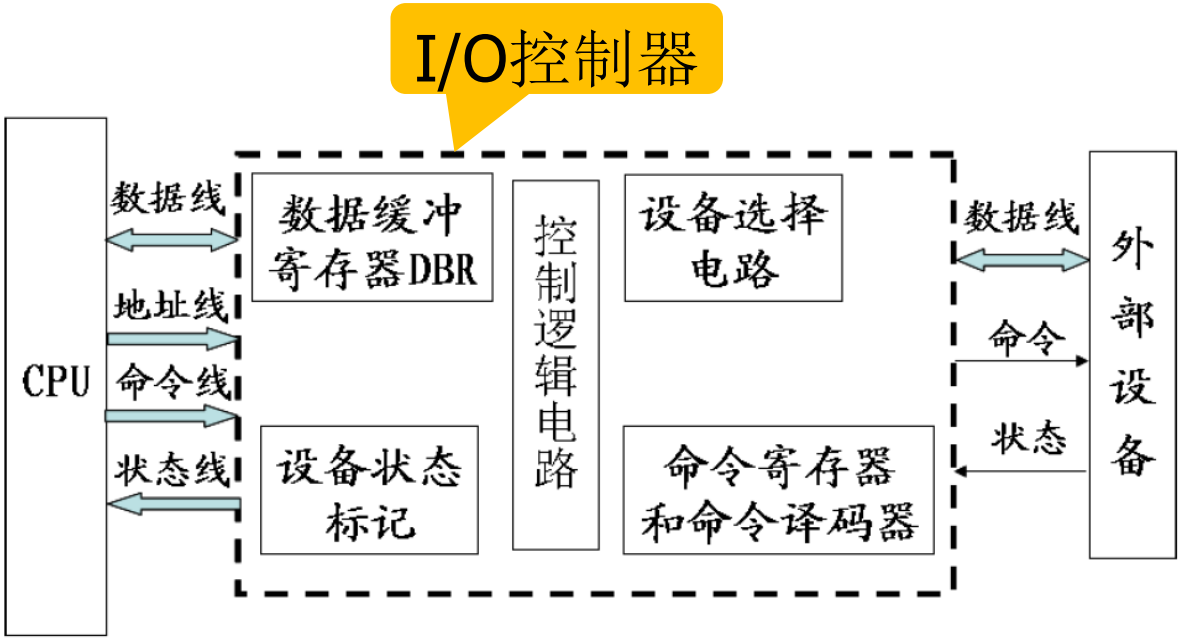
- I/O接口
- 设备驱动

I/O接口

- I/O接口

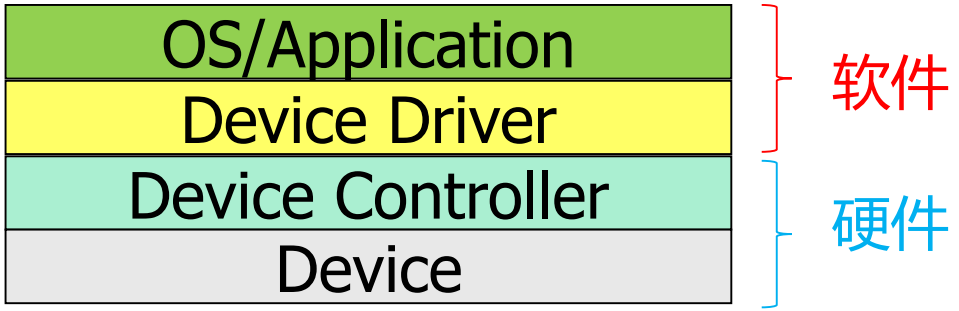
注意区分设备驱动、设备控制器/接口与设备
——软件与设备控制器打交道而不是设备

  - 处理器与外设之间设置的硬件电路及相应的软件控制
  - 端口(port) ＝ 寄存器组（数据、控制、状态...）
  - 接口(interface) ＝ N个端口＋控制逻辑



Device Driver/设备驱动

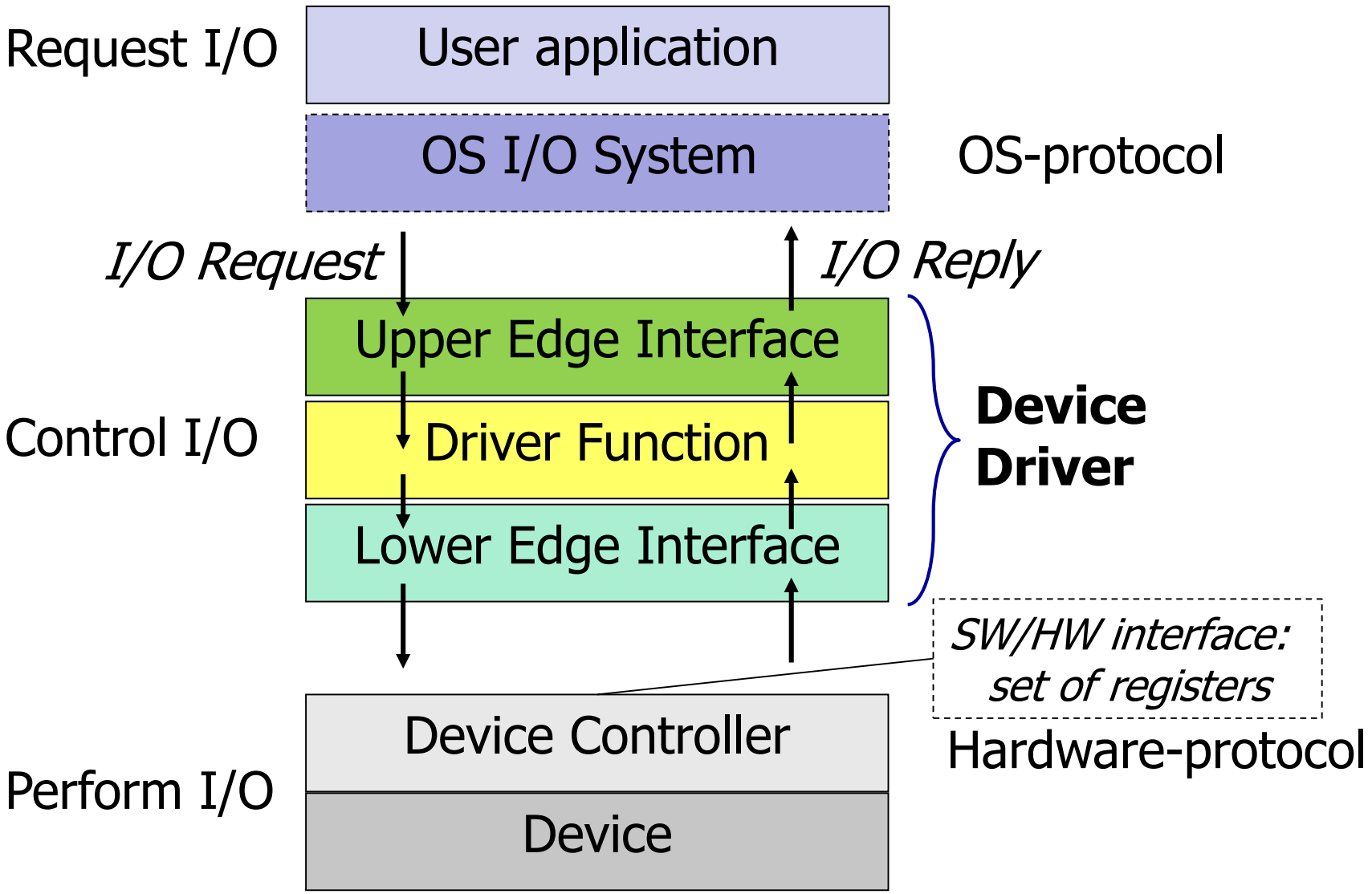
- **基本含义**：驱使硬件设备行动的特定软件
  - Device-specific & OS-specific code to control an IO device
- **主要功能**：负责应用软件和硬件设备之间的沟通，将应用程序的标准调用映射为设备特定的命令序列(操作寄存器)。
  - 向下：直接管理底层硬件设备，按照设备的具体工作方式，读写设备控制器（device controller）的寄存器，完成设备的轮询、中断处理或DMA 通信（ Device-specific ）
  - 向上：为上层程序提供操作设备的编程接口，隐藏硬件设备控制的具体细节（ OS-specific ）
  - 核心操作：发送数据给设备、从设备接收数据、控制设备



Device Driver/设备驱动

- **3种基本的I/O数据传输方式：**
  - 轮询
  - 中断
  - DMA
- **软件结构**：
  - 设备驱动连接系统软件和硬件，受到系统软件接口和硬件接口的规范和限制，会随着系统软件和硬件平台的不同而具有不同的形态。
  - 复杂设备（特别是总线设备，如PCI、USB、SPI等）之间有依赖关系，形成堆栈结构，相应的多个设备驱动也会构成分层的驱动堆栈。
  - 为方便讨论和设计实现，本课把设备驱动本身划分为三层

Layered Device Driver Ref. Model



How to develop device driver

- 1. Check available resources
- 2. Design upper edge interface
- 3. Confirm lower edge interface
- 4. Implement driver function

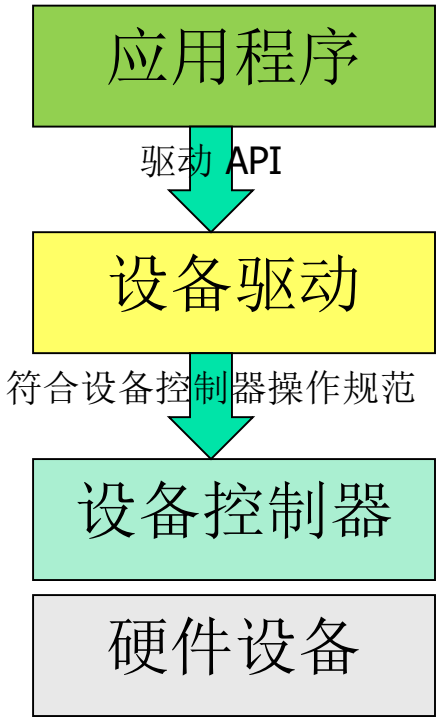
1. Check available resources

- 产品规格书 (Data Sheet)和应用指南 (Application Note)
- 评估板 (Evaluation Board)
- 范例程序代码 (Sample Code)
- 硬件板子设计规格书
- 原理图与layout图
- IC原厂的技术人员或者代理商的FAE (Field Applications Engineer，现场应用工程师)
- 第三方开发工具和软件源码 (如果有)

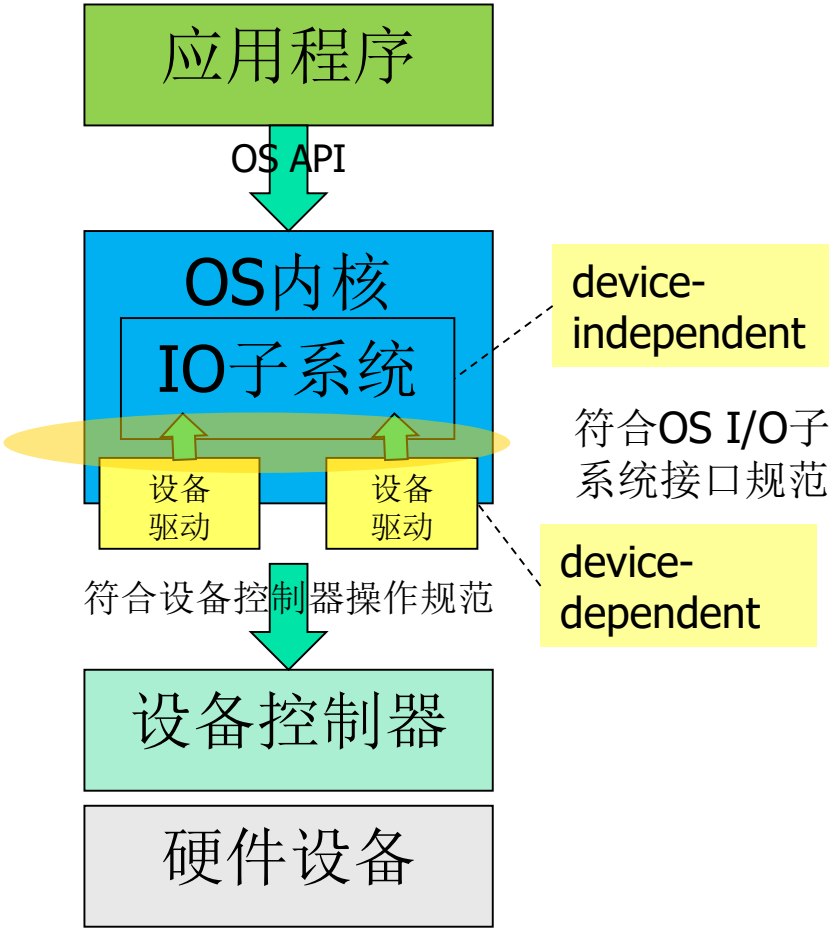
2.Design Upper Edge Interface

- Upper Edge Interface为上层软件提供统一的编程接口/API来调用本驱动的功能。
  - 接口定义的基本原则是设备无关性：隐藏或者封装硬件特性与操作细节，使得硬件变动导致的软件变化不会影响应用程序。
  - 问题：由谁规定该API接口？
- 根据操作系统是否定义了驱动架构，可把驱动分为2类：
  - **不基于OS的设备驱动**
    - 非标准接口。接口可根据硬件设备的特点自行定义或和应用程序协商定义
    - 应用程序直接调用该接口来使用设备。应用与驱动紧耦合。
    - 例如： 无操作系统、uC/OS-II中的设备驱动（本课实验是这一类）
  - **基于OS的设备驱动**
    - 标准接口。OS定义了驱动的架构和独立于具体设备的抽象接口，驱动必须实现这些接口才能融入OS。
    - 应用程序通过OS的API间接调用驱动接口。应用、OS与驱动相对独立。
    - 例如：VxWorks、Linux等操作系统下的设备驱动

2类设备驱动

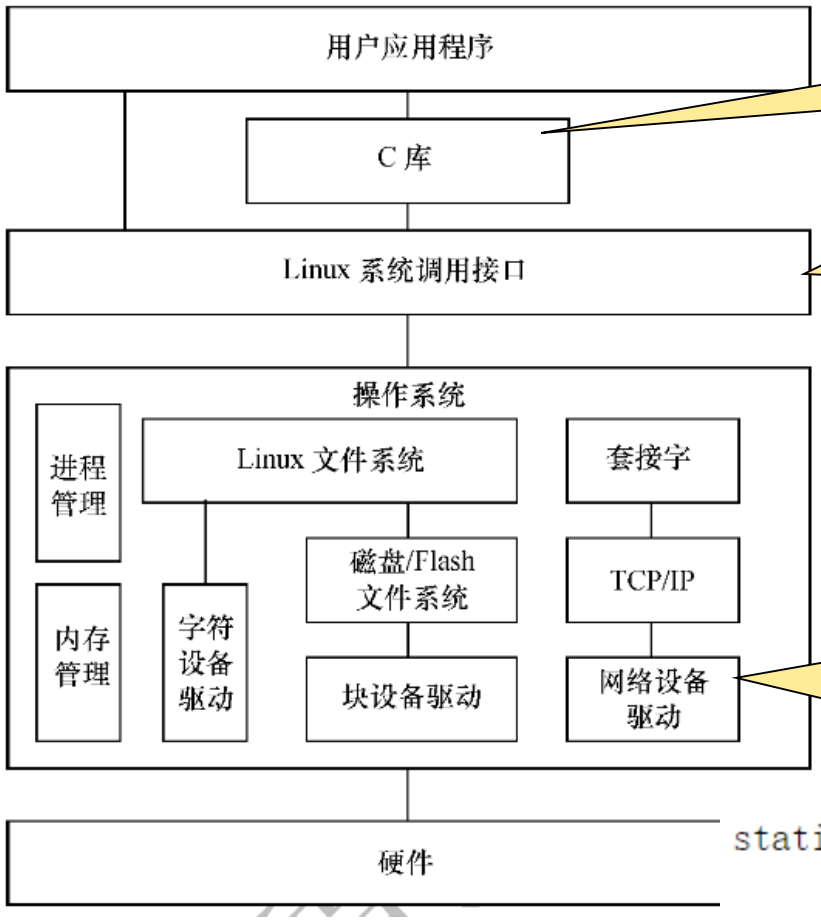


不基于OS的设备驱动（非标准）



基于OS的设备驱动（标准）

E.g. Linux Device Driver Architecture



应用程序也可以通过C库函数fopen()、fwrite()、fread()、fclose()等访问设备功能

应用程序可以直接通过文件系统的系统调用接口open()、write()、read()、close()、ioctl()等函数访问字符设备和块设备，通过socket接口访问网络设备

Linux 将存储器和外设分为3 大类：
字符设备：字节流/Byte-oriented，串行访问
块设备：块数组/ Block-oriented，随机访问
网络设备：数据包/ Packet-oriented，通过socket收发数据

```
static const struct file_operations led_proc_fops = {
        .owner          = THIS_MODULE,
        .open           = led_proc_open,
        .read           = seq_read,
        .llseek         = seq_lseek,
        .release        = single_release,
        .write          = led_proc_write,
};
```

/usr/src/linux/include/linux/fs.h

- 在http://lxr.linux.no/linux 搜索led.c看看

3.Confirm Lower Edge Interface

- Lower Edge Interface处理与设备通信的低层问题
  - 1.设备在整个硬件系统结构中的位置？— 总线
    - 直接：通过设备控制器接到内部总线，一般意味着通过访问设备控制器可以操作设备。
    - 间接：通过外部总线接到内部总线，一般意味着需要通过外部总线驱动间接操作设备。
  - 驱动堆栈
  - 2.处理器怎样访问到设备？— 通过设备控制器
    - I/O instructions（IO映射独立编制）：封装成C代码
    - Memory mapped I/O：设备控制器的地址范围、设备寄存器定义数据结构与读写例程
  - 3.处理器以什么方式与设备通信?
    - Polling：状态寄存器
    - Interrupt：中断服务例程
    - DMA：DMA传输初始化、中断服务例程

4.Implement Driver Function

- Driver Function按照设备操作的规范与流程要求，实现设备数据的输入输出、给设备发送指令等功能
- 主要功能
  - 设备状态的管理与控制：初始化 -> 工作状态 -> 关闭
  - 数据的接收与发送
  - 中断处理
- 关键问题与技术
  - 处理器与设备数据速率不匹配
    - 缓冲/buffering
  - 设备可能被多个任务并发访问，从而导致竞争
    - 使用同步机制管理并发
  - 处理器与设备是异步并行的
    - I/O模型：阻塞与非阻塞、同步与异步

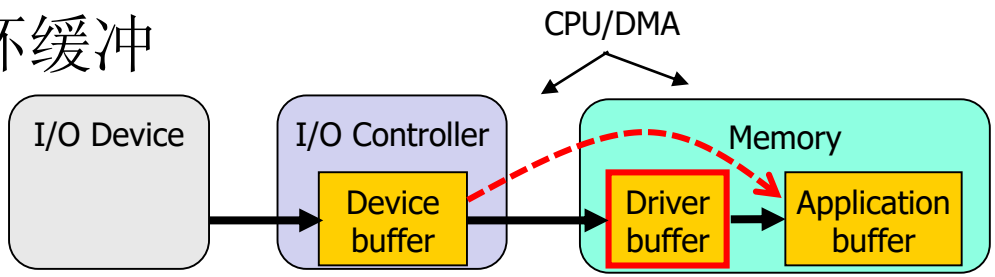
在设备驱动是基于OS的情况下，这些问题主要由OS的IO子系统解决。如果是不基于OS的非标准驱动，在必要的时候，需要考虑这些问题。

Buffering/缓冲管理

- 缓冲：在两个设备 (应用) 间传输数据时，用来临时存放数据的内存区
- 使用缓冲的原因：
  - 缓和CPU和I/O设备之间数据速率不匹配的矛盾
  - 提高CPU和I/O设备之间的并行性
  - 减少对CPU的中断频率
- 缓冲机制：
  - Single buffering 单缓冲
  - Double buffering 双缓冲
  - Circular buffering 循环缓冲
  - Buffer pool 缓冲池

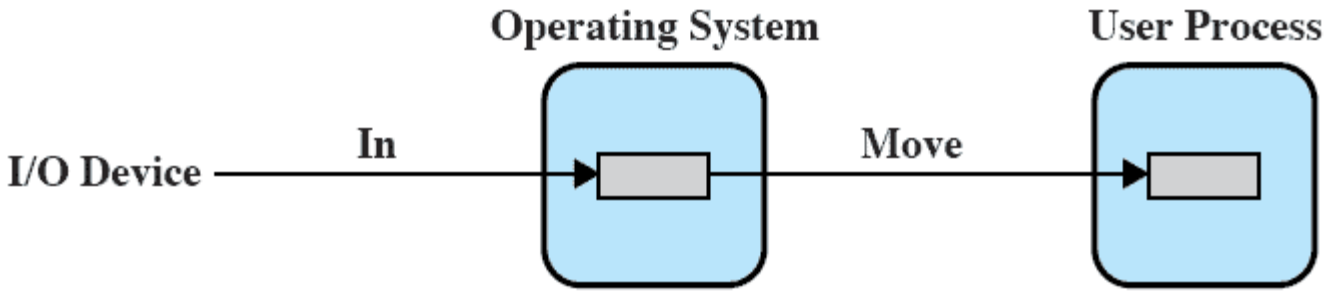
如果设备传输速率高于CPU处理速率，缓冲还能提高效率吗？

zero-copy



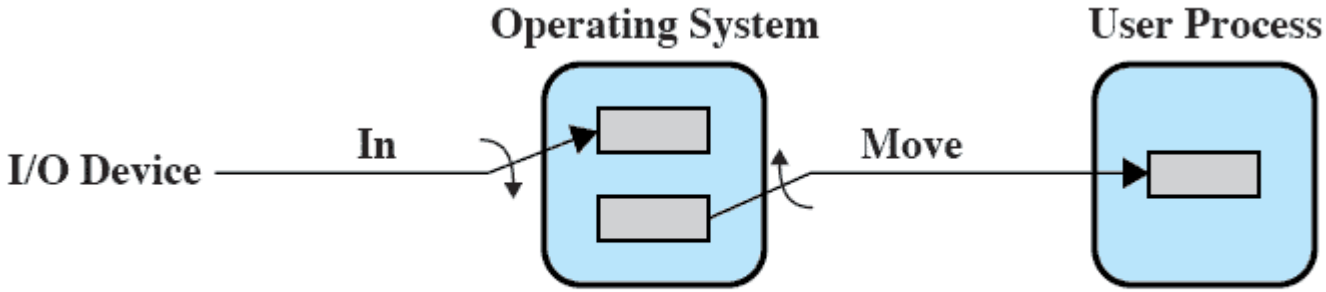
Single buffering

IO传输时间>>(数据copy时间+处理时间)时适用



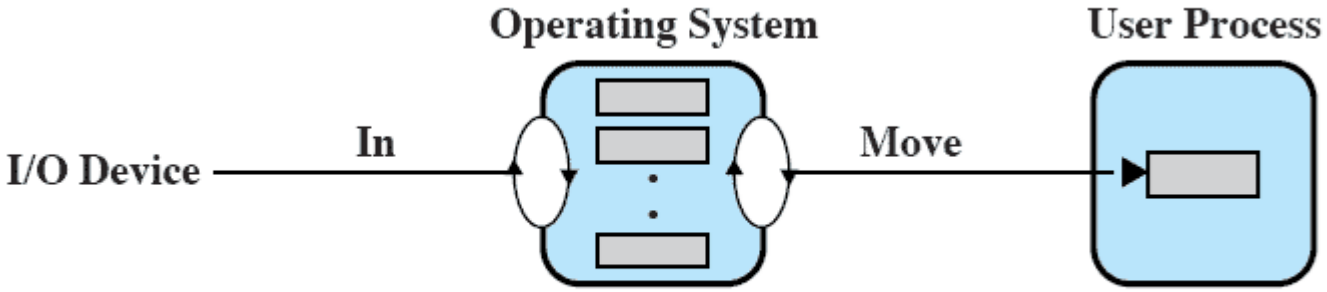
Double buffering

IO传输时间=(数据copy时间+处理时间)时，效率最高



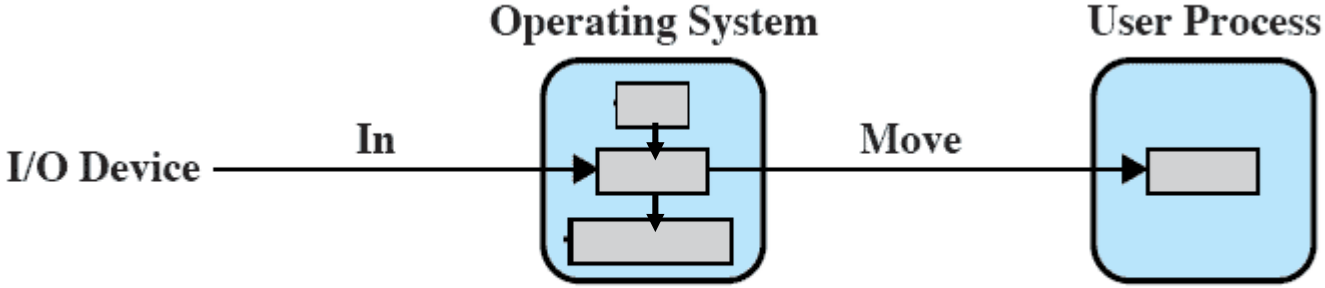
Circular buffering

便于处理突发数据



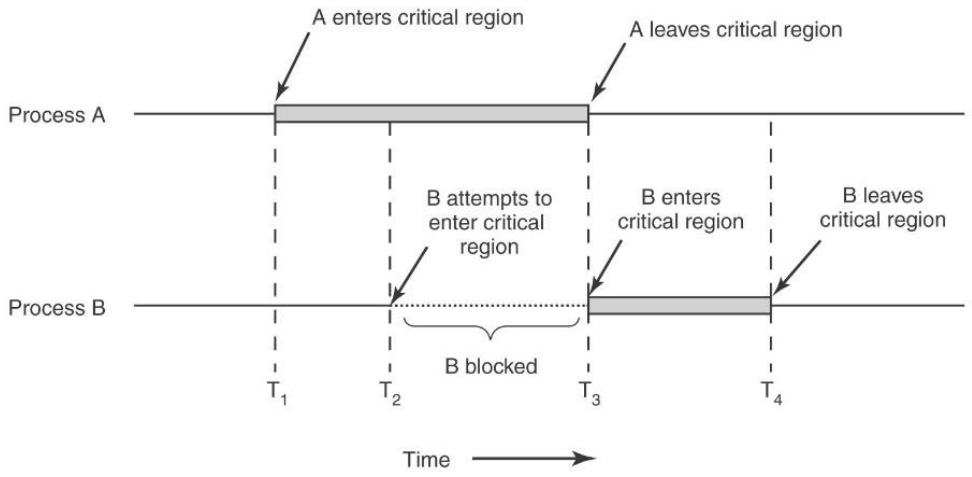
Buffer pool

内存块有不同大小，内存利用率高，使用灵活，需要动态内存管理



Concurrency & Mutual Exclusion/并发与互斥

- I/O设备是系统中的共享资源，如果有**多个执行单元**对设备进行并发访问，则可能导致竞争和数据不一致。
  - 多处理器系统中的多个CPU，使用共同的系统总线，共享外设和储存器。
  - 单CPU 内进程/线程/任务与抢占它的进程/线程/任务
  - 中断与进程/线程/任务之间
- 解决竞争的办法：保证对共享资源的互斥访问。基本手段是使用某种**互斥机制**保护临界区（访问共享资源的代码区域）
  - 中断屏蔽
  - 原子操作
  - 信号量
  - 加锁
    - 自旋锁
    - 读写自旋锁
    - 顺序锁
    - RCU锁（Read-Copy Update，读 - 拷贝 - 更新）



I/O Models

- 阻塞式I/O(blocking I/O)：调用方挂起直到I/O完成
  - 行为：执行设备操作时，如不能获得资源，则挂起调用方，直到等待的条件被满足再返回。
  - 调用方：调用接口一般是同步的，容易理解和使用。但无法满足某些应用要求，如需要处理多个事件的任务
  - 实现：较容易实现，可以利用信号量、等待队列等。
- 非阻塞式I/O(non-blocking I/O)：I/O调用操作立即返回，不会挂起调用方
  - 行为：如果条件满足，则执行完后返回。如果条件不满足，则立即返回错误，而不是等待I/O完成。
  - 调用方：返回错误时，需要调用方做进一步处理，如放弃，或者不停查询直到条件满足。
  - 实现：实现较为复杂，一般需要使用缓冲暂存数据，使用单独的任务/线程完成IO操作

I/O Models

- 异步I/O(Asynchronous I/O)：当I/O运行时，应用程序可以同时进行别的处理
  - 行为：设备就绪或者I/O完成时，通过某种手段 (如回调函数、发信号)，主动通知应用程序。
  - 调用方：应用程序不需要调用通过任何操作来等待条件满足。条件满足后的操作在回调函数或者信号处理函数中进行，可能需要加锁。较难使用。
  - 实现：复杂，除了缓冲和单独的IO线程外，还需要有管理回调或者信号的机制。
- 这些I/O机制可以相互补充，没有优劣之分，具体实现时需要根据设备的特性和应用的要求进行选择。
  - 阻塞I/O：一直等待设备可访问后再访问
  - 非阻塞I/O：一直查询设备是否可以访问
  - 异步I/O：设备通知用户自身可访问，之后用户再进行处理

Thank you!