

目录

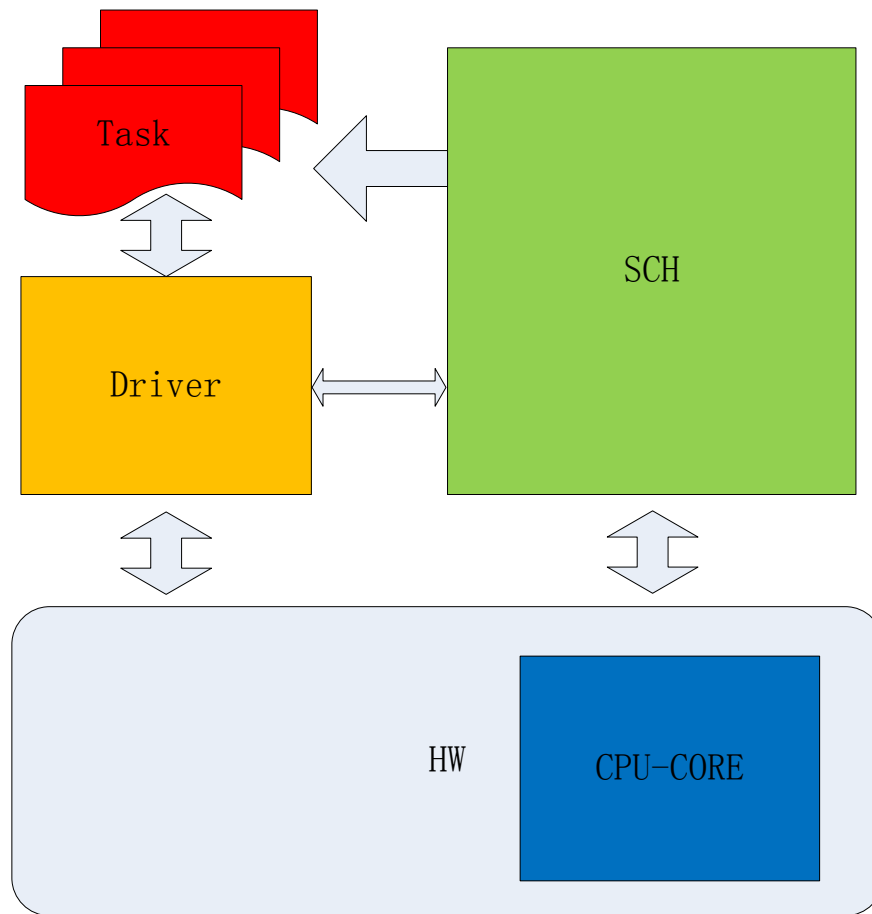
Mini OS(Scheduler)	1
概述.....	1
架构.....	2
内存管理.....	2
栈.....	3
堆.....	3
Task 栈.....	4
Task.....	4
描述.....	5
状态.....	6
数据结构.....	7
调度.....	8
策略.....	8
CPU 轮询	9
Task 切换.....	10
消息.....	11
数据结构.....	12
收发 API	12
编译&下载.....	12
源码.....	12
文件结构&说明	13
编译链接.....	14
下载.....	14

Mini OS (Scheduler)

概述

使用 Linux 下的 gcc 环境进行编译，包括 Bootloader 和 Scheduler 两部分，具体实现可以及使
用可以参考 github 源码，其中主要功能包括，内存管理、task 调度、task 状态管理、消息传
递，时间片管理，task 挂起，关于 Bootloader 可以参考相关源码，可以使用 minicom 提供的
xmodem 协议烧写软件到芯片

架构



RTOS 软件架构

软件主要包括 Task App、Driver、Scheduler 三部分，task 通过 scheduler 提供的 API 接口注册到系统，由 scheduler 基于时间片和优先级进行调度，其中 scheduler 的时间片由 systick 提供，其最小颗粒可以在 inc/systick.h 文件进行配置，每一个 task 均有独立的栈空间，真正完全独立的 task，其大小在注册该 task 时由传入的 size 参数决定

```

23 #define SYSTICK_CVR_PERIOD 0xFFFFFFFF
24
25 #define SYSTICK_RELOAD_VAL 71999 //1K 0.1K
26

```

内存管理

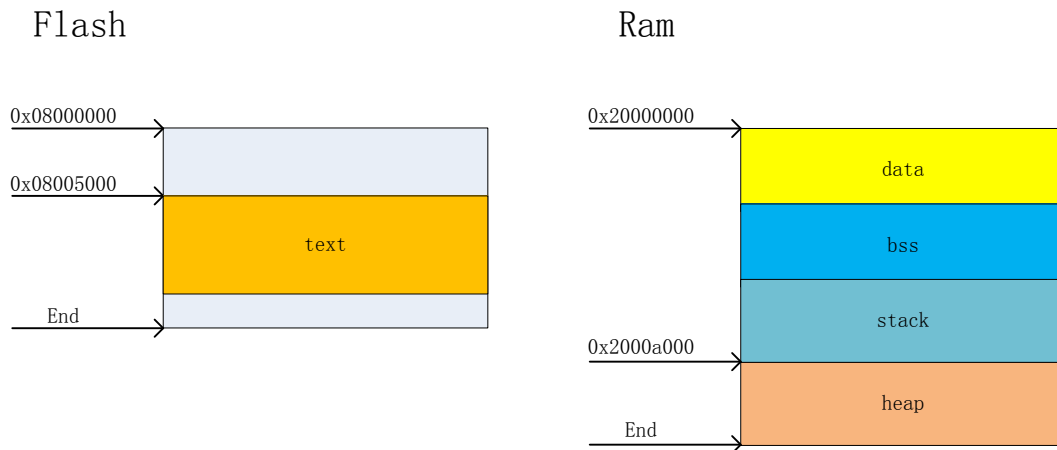
内存管理使用动态和静态两种方式进行管理，由链接文件指定其内存静态分配

```

2
3 MEMORY
4 {
5     RAM(rwx)      : ORIGIN = 0x20000C00, LENGTH = 61K
6     FLASH (rx)    : ORIGIN = 0x08005000, LENGTH = 492K
7 }
8

```

内存分为两部分，需要动态读写的 ram (data、bss)，只读内存 flash (text)，由于 bootlaoder 占用部分 flash 空间所以 flash 起始地址向后偏移 20K，详细分配情况如下：



内存分配参考链接文件（cm3/flash.ld）

栈

栈空间由链接文件指定 **top** 指针（0x2000a000），根据 **crotex-m3** 架构定义，需要将该地址放入中断向量表首地址（参考文件：cm3/start_up.S）

```

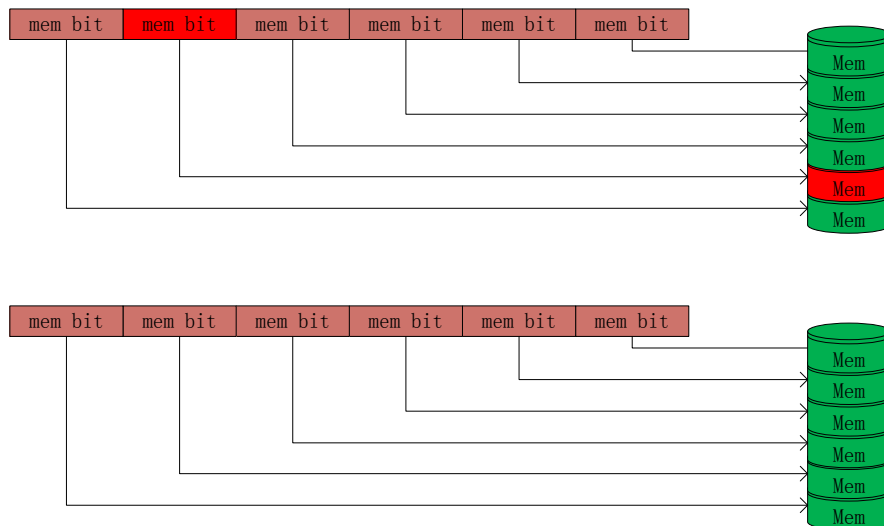
1  .syntax unified
2  .cpu cortex-m3
3  .fpu softvfp
4  .thumb
5
6  Vectros:
7  .word stack_top
8  .word (Reset_Handler + 1)
9  .word NMI_Handler
10 .word HardFault_Handler
11 .word MemManage_Handler
12 .word BusFault_Handler
13 .word UsageFault_Handler
14 .word 0
15 .word 0

```

堆

堆空间使用 **ram** 的后面 20K 空间，由系统进行管理，提供 **malloc**、**free** 内存 API 接口，用于内存的动态管理，同时为调度器提供数据基础

内存管理使用 **bit map** 的方式进行管理，将所有堆的空间划分为若干内存块，使用全局数组对内存块进行使用标记，通过内存块起始位置进行索引，通过块 **size** 进行偏移，当系统需要相应大小内存时，在当前 **map** 中进行查找，找到符合要求的连续内存块，将其对于的 **bit map** 进行标记，并且返回该连续内存的起始地址；释放内存时，将相应的 **bit map** 清除



API:

```
extern void * malloc(int size);
extern int free(void *mem);
extern void memcpy(unsigned char *dec, unsigned char *src, int size);
extern void memset(unsigned char *src, unsigned char val, int size);
```

Task 栈

系统为每一个 task 提供独立的栈空间在创建任务时由参数指定

```
int creat_task(void (*func), int stack_size, int priority)
{
    struct task_list *p;
    struct task_p *task;
```

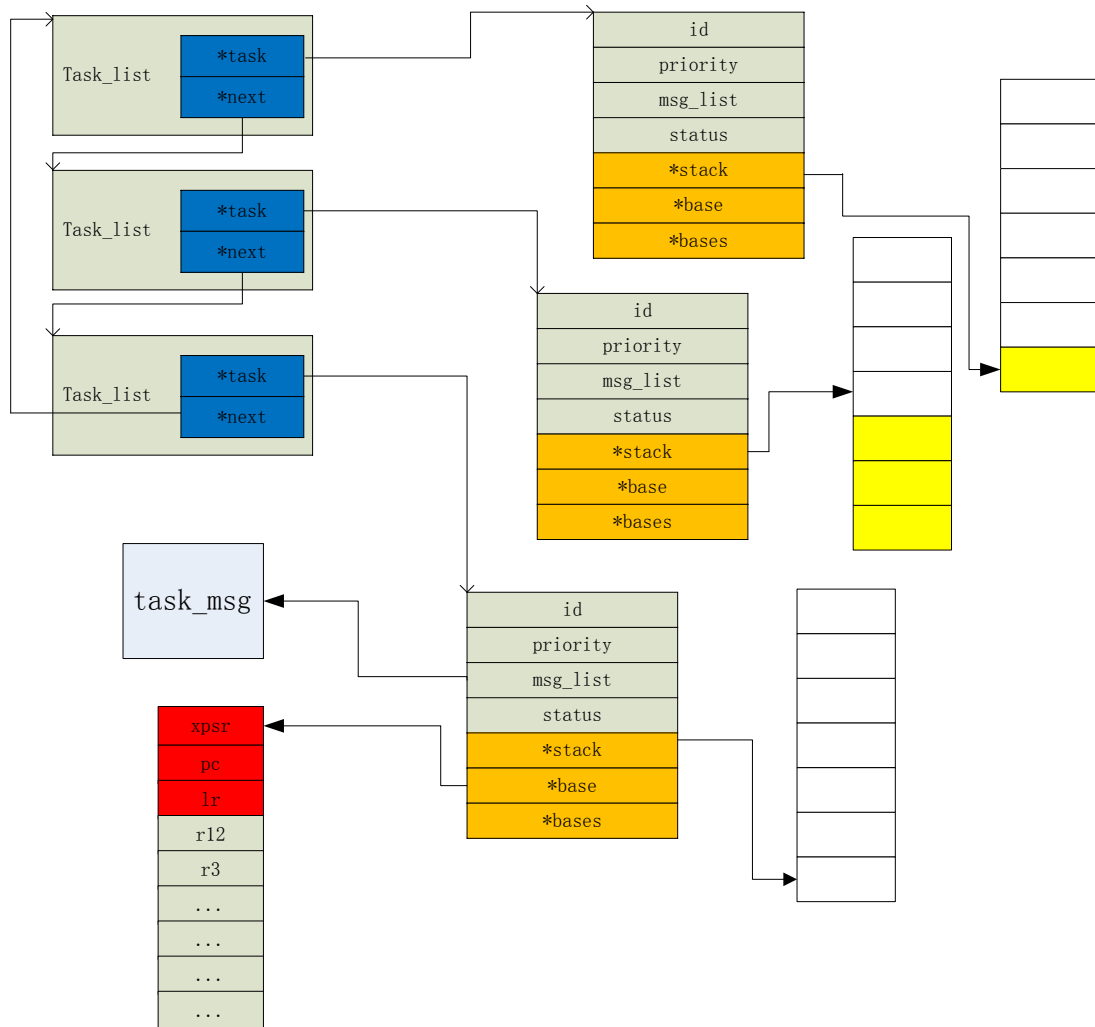
其真实内存位于堆中, 使用 malloc 分配, 在注销 task 时由 free 进行释放, 当调度器在对 Task 进行切换时, 将对于的栈替换到 cpu 栈指针寄存器, 实现 task 的中断上下文还原及为临时数据提供保护

Task

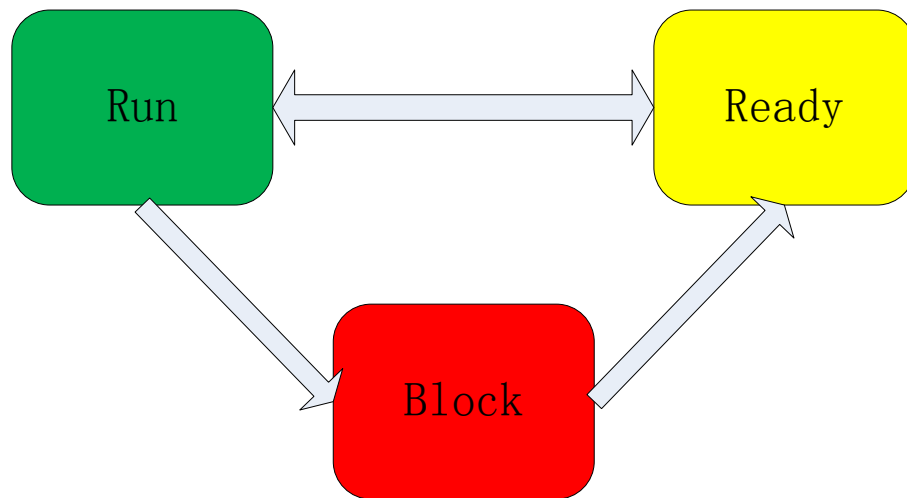
```
38 struct task_p {
39     long id;
40     long priority;
41     struct task_msg msg_list;
42     unsigned long status;
43     unsigned char *stack;
44     struct task_init_stack_frame *base;
45     void *basep;
46 };
47
48 struct task_list {
49     struct task_p *task;
50     struct task_list *next;
51 };
52
```

系统调度器实现的本质就是为 task 轮转，调度器将每一个 task 抽象为任务描述符，注册 task 时将其添加到 task list，在 systick 的驱动下进行栈指针的切换，达到任务调度

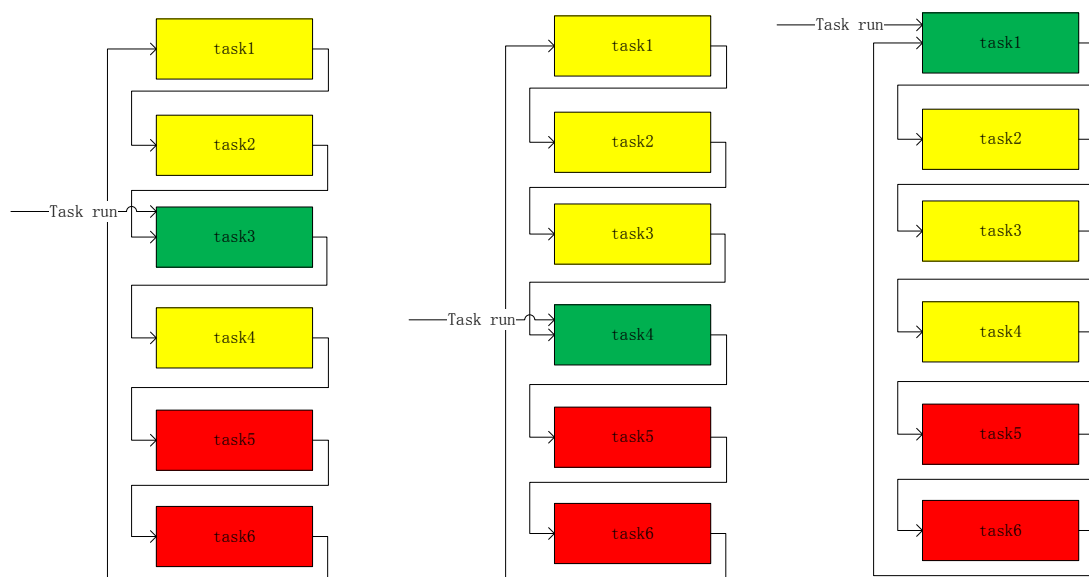
描述

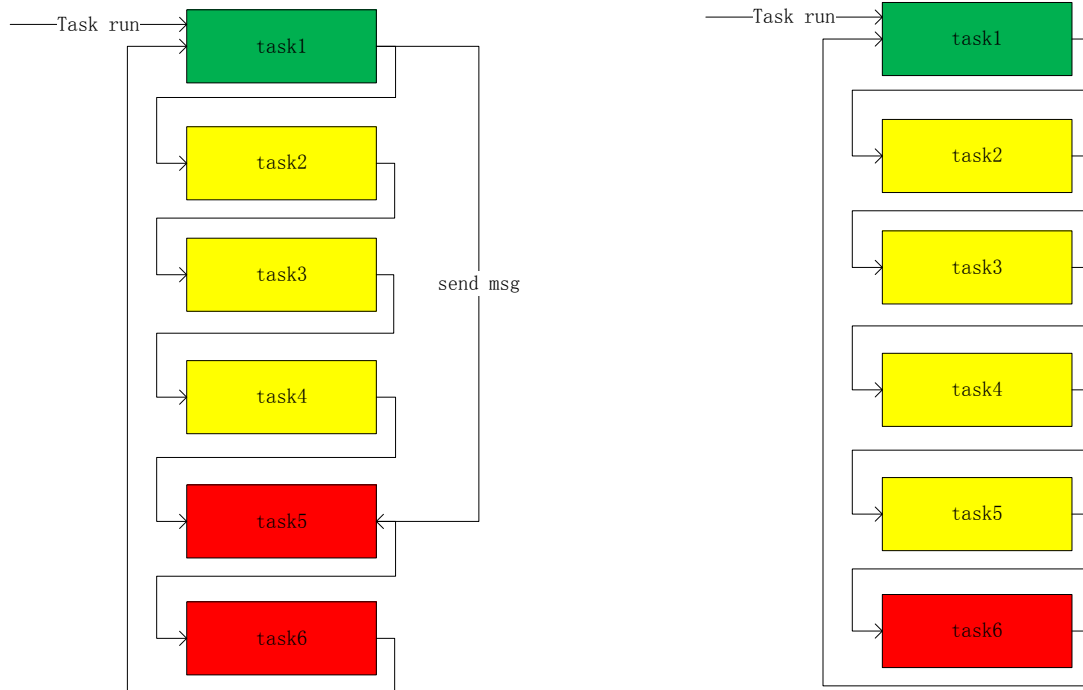


状态



每一个 task 可以分为 Run、Ready、Block 三个状态，当 task 处于 Ready 状态下，并且被装载与 CPU 进行执行时表示 Run 状态，在执行的过程中遇到 wait msg 造成阻塞或者主动使用 delay 函数进行挂起，系统会将该 task 置为 Block，直到收到消息或者主动挂起的时间耗尽，系统将其重新设置为 Ready 等待下一次调度





数据结构

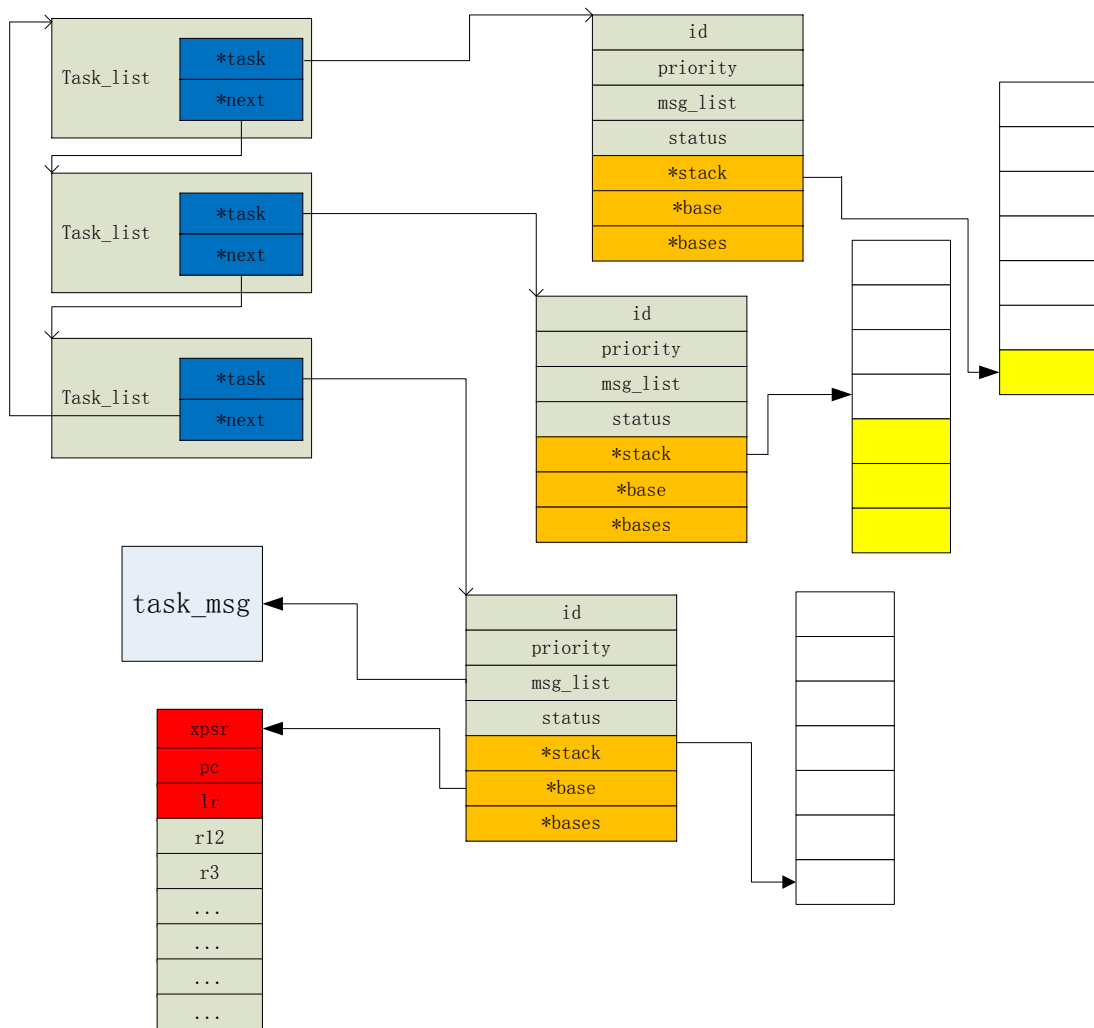
```

22 struct msg_queues {
23     int rcv_id;
24     int send_id;
25     void *data;
26 };
27
28 struct msg_queues_list {
29     struct msg_queues msg;
30     struct msg_queues_list *next;
31 };
32
33 struct task_msg {
34     struct msg_queues_list *head;
35     struct msg_queues_list *last;
36 };
37
38 struct task_p {
39     long id;
40     long priority;
41     struct task_msg msg_list;
42     unsigned long status;
43     unsigned char *stack;
44     struct task_init_stack_frame *base;
45     void *basep;
46 };
47
48 struct task_list {
49     struct task_p *task;
50     struct task_list *next;
51 };
52

```

调度

Task 数据结构



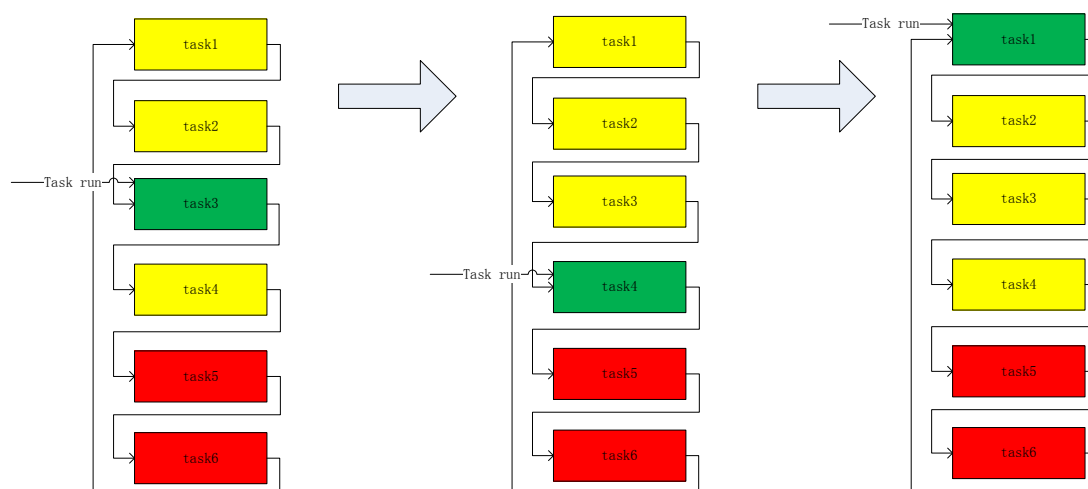
基于上述数据结构，调度器基于某种策略调度 cpu 轮询该 Task list 中所有 task

策略

1. 基于时间片
2. 基于优先级（最大优先级可配置）

```
3 #define PRIORITY_MAX 10
4 #define RT 1
```


CPU 轮询




调度器基于时间片，将 cpu 分配给每一个 ready 状态的 task

Task 切换

```

160
161 void* tick_and_switch(void* cur_stack)
162 {
163     void* temp ;
164     os_delay_clear();           //clear task delay timer
165     run->task->basep = cur_stack;
166     run = find_ready_task(run);
167     temp = run->task->basep;
168     return temp;
169 }
170
171
172 void SysTick_Handler()
173 {
174     __asm volatile (
175         "    mov r0, sp          \n"
176         "    sub r0, #(8*4)      \n"
177         "    push {lr}           \n"
178         "    bl.w tick_and_switch \n"
179         "    pop {lr}           \n"
180         "                        \n"
181         "    cmp r0, #0          \n"
182         "    beq end             \n"
183         "                        \n"
184         "    push {r4-r11}       \n"
185         "    mov sp, r0          \n"
186         "    pop {r4-r11}       \n"
187         "                        \n"
188         "end:                    \n"
189     );
190 }
191

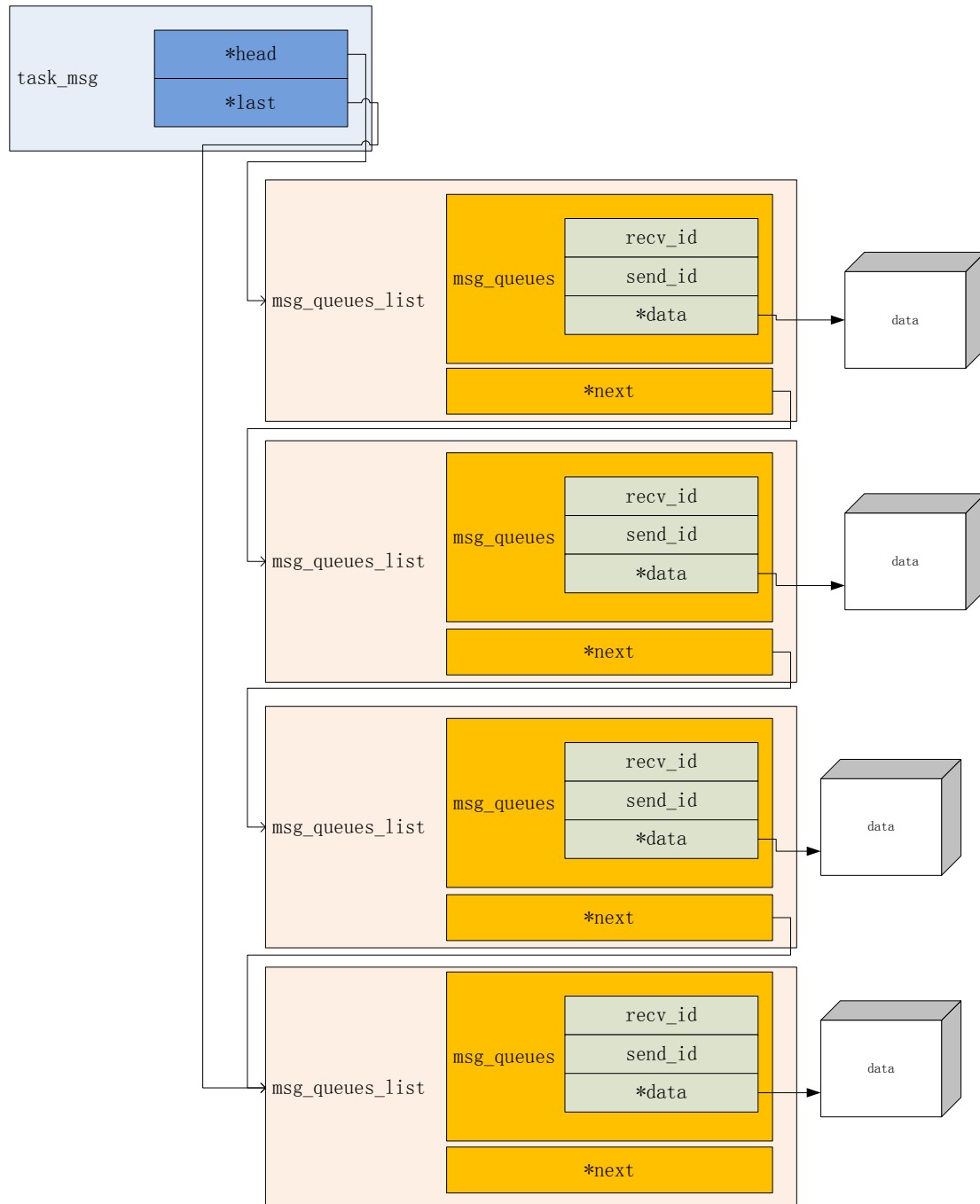
```



在 systick 中断里将 sp 指针保存到 r0, 然后偏移 32 字节, 将 lr push 入栈, 调用 tick_and_switch 进行调度 (保存当前 task 的栈指针到该 task 的描述结构体, 然后找寻下一个需要加载的 task 结构体返回该 task 的栈指针), push r4-r11, 将返回的 task 栈指针还原到 sp, pop r4-r11

注释: 由于 cortex-M3 体系架构中断不会将 r4-r11 进行入栈, 所以需要软件对其进行压栈/出栈

消息



系统在 Task 描述结构体中为每一个 task 分配了一个 msg 指针, 初始值为 NULL, 当有 task send 时:

1. 向系统申请 msg
2. 调用 `msg_send`, 指定 `recv id`、`data`
3. 调度器激活相应的 task 进行 msg 接收
4. 接收端使用 `msg_rcv` 接收消息并释放 msg, 处理 data

数据结构

```
21
22 struct msg_queues {
23     int recv_id;
24     int send_id;
25     void *data;
26 };
27
28 struct msg_queues_list {
29     struct msg_queues msg;
30     struct msg_queues_list *next;
31 };
32
33 struct task_msg {
34     struct msg_queues_list *head;
35     struct msg_queues_list *last;
36 };
37
```

收发 API

```
74
75 extern int send_msg_queues(int recv_id, void *msg);
76 extern void *recv_msg_queues(void);
77
78 /*
```

编译&下载

项目所有源码均上传于 github，相关使用方式如下：

源码

Github 连接：

Bootloader: <https://github.com/ShilinGuo520/bootloader>

Scheduler: <https://github.com/ShilinGuo520/scheduler>

文件结构&说明

```
line:~/scheduler$ ls -hl
4.0K 10月  2 15:22 cm3
4.0K 10月  2 09:46 common
4.0K 10月  2 09:46 driver
4.0K 10月  2 09:46 glib
4.0K 10月  2 16:29 inc
1.5M 10月  2 09:46 libgcc.A
1.4K 10月  2 09:46 Makefile
4.0K 10月  2 09:46 mem
1.1K 10月  2 09:46 README.md
4.0K 10月  2 16:23 rtos
```

Cm3: 与架构相关文件，包括启动文件、链接文件

Common: 包括用到的 libc 函数实现

Driver: 驱动相关文件，包括 uart、flash、gpio 等

Mem: 内存管理，包括 malloc、free、memcpy 等

Rtos: 系统调度，包括 task 切换、task 管理、消息传递等

libgcc.A 编译链接依赖（cm3 软件实现的除法、浮点等需要使用编译器中的软除法、软浮点）

编译链接

```

ALL_LIB = libcommon.a
ALL_LIB += libdriver.a
ALL_LIB += libglib.a
ALL_LIB += librtos.a
ALL_LIB += libmem.a
ALL_LIB += ./libgcc.A

ALL_SRC = start_up.o $(ALL_LIB)

$(TARGET):$(ALL_SRC)
$(LD) $(LDFLAGS) $(ALL_SRC) --output $(TARGET).elf
$(OBJCOPY) -O binary $(TARGET).elf $(TARGET).bin
$(OBJDUMP) -h -S -D $(TARGET).elf > objdump.txt

libglib.a:
$(MAKE) -C glib

librtos.a:
$(MAKE) -C rtos

libcommon.a:
$(MAKE) -C common

libdriver.a:
$(MAKE) -C driver

libmem.a:
$(MAKE) -C mem

start_up.o:
$(AS) $(ASFLAGS) -o start_up.o -c ./cm3/start_up.S

```

直接在目录下输入 make 自动完成如下步骤：

1. Makefile 根据文件依赖，首先将调用每个子目录下的 Makefile 文件
2. 子目录的 Makefile 文件将目录中所有.c 文件编译为.o
3. 将当前目录所有.o 文件打包为对应的 lib 文件
4. 通过 Makefile 指定的连接文件将所有 lib 按照指定的内存分配原则链接为 elf 文件
5. 使用 objcopy 将 elf 转换为 flash 镜像文件.bin

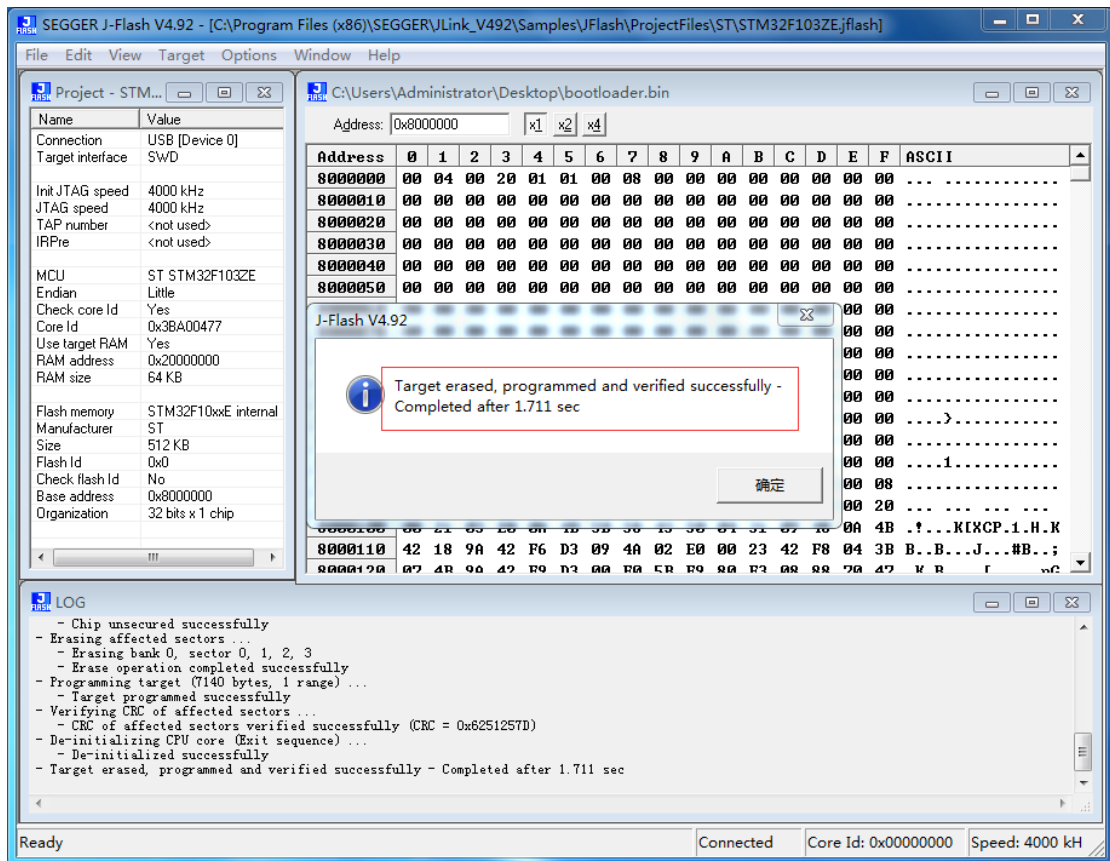
```

arm-none-eabi-objcopy -O binary application.elf application.bin
arm-none-eabi-objdump -h -S -D application.elf > objdump.txt
shilinguo@shilinguo-virtual-machine:~/scheduler$ ls
application.bin  cm3  driver  inc  libdriver.a  libglib.a  librtos.a  map.txt  objdump.txt  rtos
application.elf  common  glib  libcommon.a  libgcc.A  libmem.a  Makefile  men  README.md  start_up.o
shilinguo@shilinguo-virtual-machine:~/scheduler$

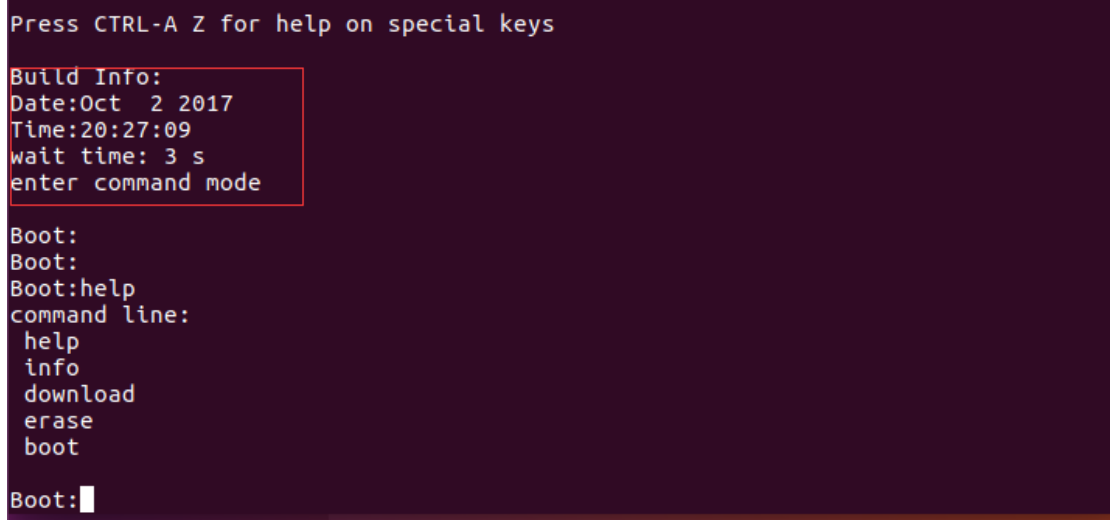
```

下载

1. 安装工具链 git、gcc 编译器
2. 下载源码 Bootloader 和 Scheduler
3. 编译 Bootloader 和 Scheduler, 使用 jtag 将 Bootloader 下载到 flash 起始位置 0x02000000



4. 使用 minicom (windows 下可以选择 Activator.exe) 通过串口连接到芯片
5. 重启, 输入任意键, 让芯片进入 Bootloader 模式



6. 输入 download 芯片等待 25s, 在 25s 内使用 Activator.exe 自动的 xmodem 协议将编译的 Scheduler 下载到芯片

```

shilinguo@shilinguo-virtual-machine: ~/b... x shilinguo@shilinguo-virtual-machine: ~/s... x
Build Info:
Date:Oct  2 2017
Time:20:27:09
wait time: 4 s
enter command mode

Boot:  +-----[xmodem upload - Press CTRL-C to quit]-----+
Boot:  |Sending application.bin, 43 blocks: Give your local XMODEM re|
Boot:  |ceive command now.
Boot:down|Xmodem sectors/kbytes sent:  43/ 5k

```

7. 下载完成，输入 boot 启动或者复位自动运行

```

wait time: 4 s
enter command mode

Boot:  +-----[xmodem upload - Press CTRL-C to quit]-----+
Boot:  |Sending application.bin, 43 blocks: Give your local XMODEM re|
Boot:  |ceive command now.
Boot:down|Bytes Sent:  5632  BPS:1883
        |Transfer complete
        |READY: press any key to continue...

```

```

Boot:
Boot:
Boot:boot

```

8. 运行 test app, Task1、2、3 按照预先的优先级和挂起唤醒关系执行

```

Build Info:
Date:Oct  2 2017
Time:20:27:09
wait time: 1 s
wait time out,try to boot the app.
Build Info:
Date:Oct  2 2017
Time:16:40:56
App Start Add:8005000
Flash Size:512
Ram Size:64
add:0x2000a130
T-3 time:0
T2:2
T1:1
T-3 time:1
T2:4
T1:3
T2:6
T1:5

```