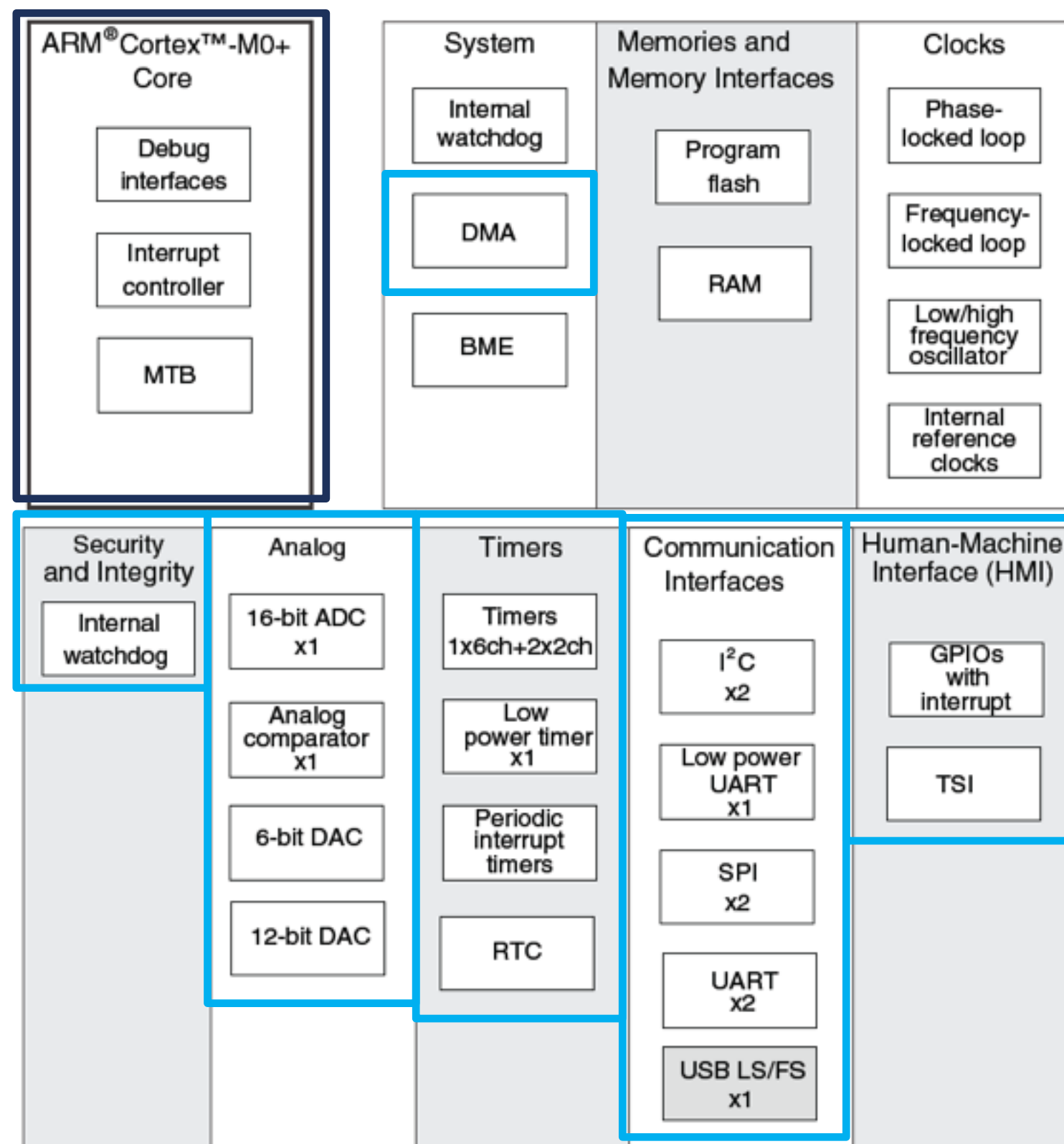


嵌入式软件基础

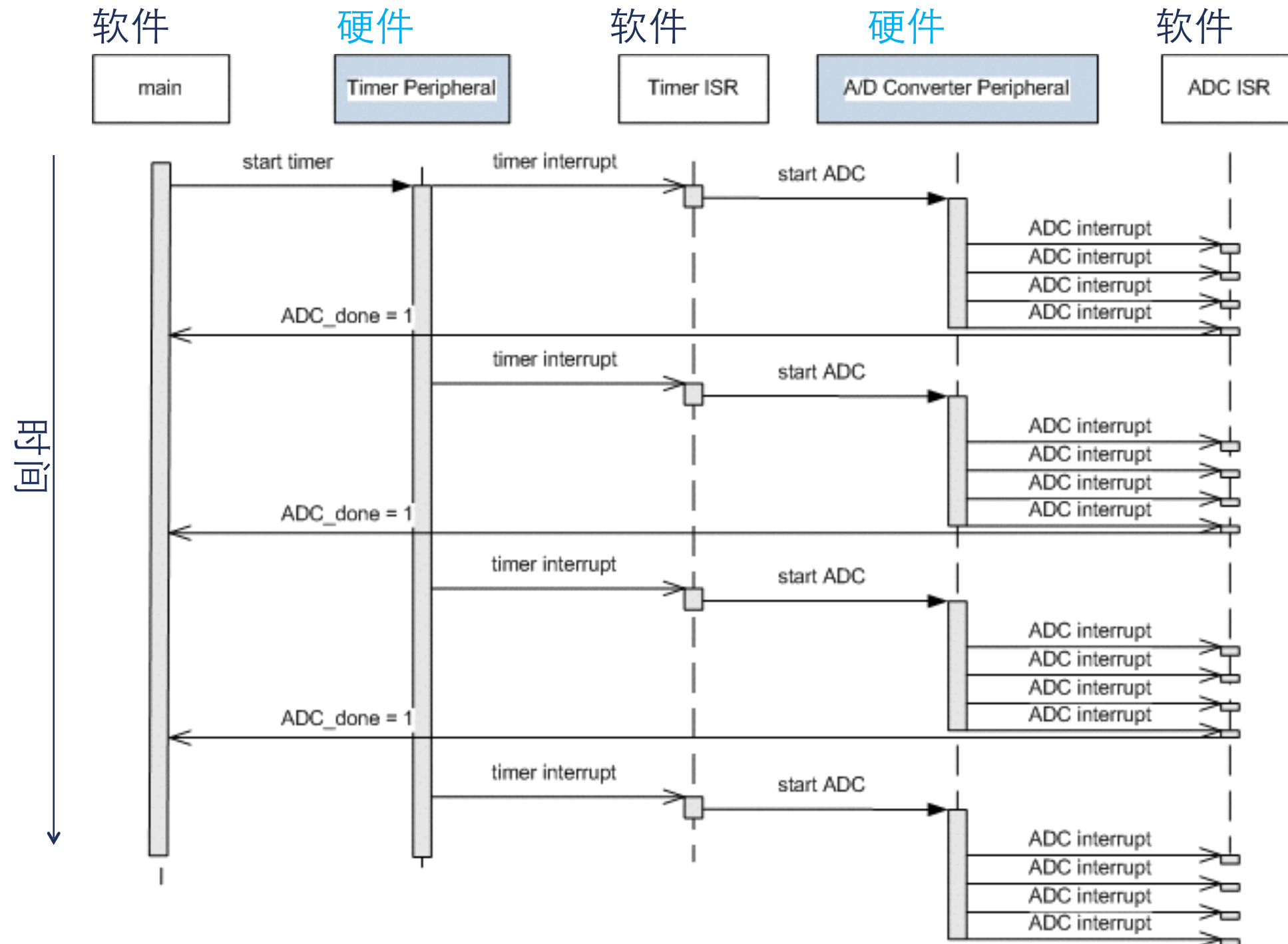
并行

支持并行的MCU硬件和软件

- CPU执行一个或多个线程的指令
- 特殊的硬件外围部件实现专门的并行处理
 - DMA – 在内存和外围器件之间传输数据
 - 看门狗定时器
 - 模拟接口
 - 定时器
 - 与其他器件通信
 - 检测外部信号事件
- 外围部件用中断来通知CPU事件的发生



并行的硬件和软件操作

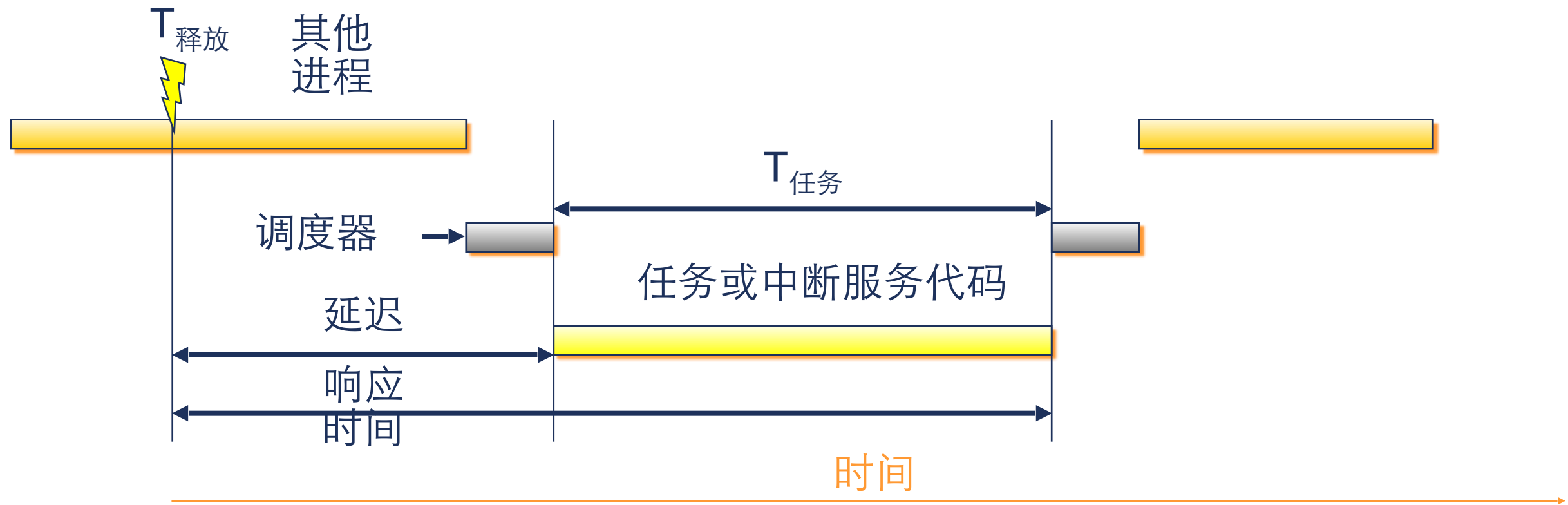


- 嵌入式系统既依赖硬件和外围部件也依赖软件来保证一切按时完成

CPU调度

- **MCU的中断系统实现了基本的CPU调度方法**
 - “每次这个硬件事件发生的时候就运行这个子程序”
 - 适用于简单系统
- **更复杂的系统需要支持多个独立的线程并行执行**
 - 用任务调度器来共享CPU
 - 有不同的任务调度方法
- **如何让处理器是有响应的？（如何能让它在正确的时间做正确的事情？）**
 - 如果要运行的软件线程比硬件处理器多，就需要共享处理器了。

定义



- $T_{\text{释放}}(i)$ = 任务（或中断） i 请求服务/已经释放/就绪可以运行的时刻
- $T_{\text{延迟}}(i)$ = 在释放CPU和任务 i 开始运行之间的时间
- $T_{\text{响应}}(i)$ = 任务 i 请求服务到完成之间的时间
- $T_{\text{任务}}(i)$ = 实现任务 i 的计算所需的时间
- $T_{\text{中断服务}}(i)$ = 完成中断服务程序 i 所需的时间

调度方法

- 依赖**MCU**硬件中断系统来运行正确的代码
 - 用中断做事件触发的调度
 - 对于很多简单系统是可行的
- 用软件来调度**CPU**的时间
 - 静态轮流执行
 - 动态优先级
 - 没有优先级的抢占
 - 有优先级的抢占

用中断做事件触发的调度

- 基础体系结构，对于简单低功耗设备有用
 - 非常少的代码或额外时间开销
- 中断系统内置的任务分派机制的作用
 - 可以用输入变化、定时器到时、UART收到数据、模拟数据电平跨越比较器阈值等来触发中断响应程序（ISR）
- 函数类型
 - 主函数配置好系统就去睡眠
 - 如果被中断了，它还会回去继续睡眠
 - 只用中断来做正常的程序操作
- 例子：自行车码表
 - Int1: 钢圈旋转
 - Int2: 模式按钮
 - Int3: 时钟
 - Output: 液晶屏



自行车码表功能

重置

配置定时器、
输入和
输出

```
cur_time = 0;
rotations = 0;
tenth_miles = 0;
```

```
while (1) {
    sleep;
}
```

ISR 1: 钢圈旋转

```
rotations++;
if (rotations >
    R_PER_MILE/10) {
    tenth_miles++;
    rotations = 0;
}
speed =
    circumference/
    (cur_time - prev_time);
计算avg_speed;
prev_time = cur_time;
从中断返回
```

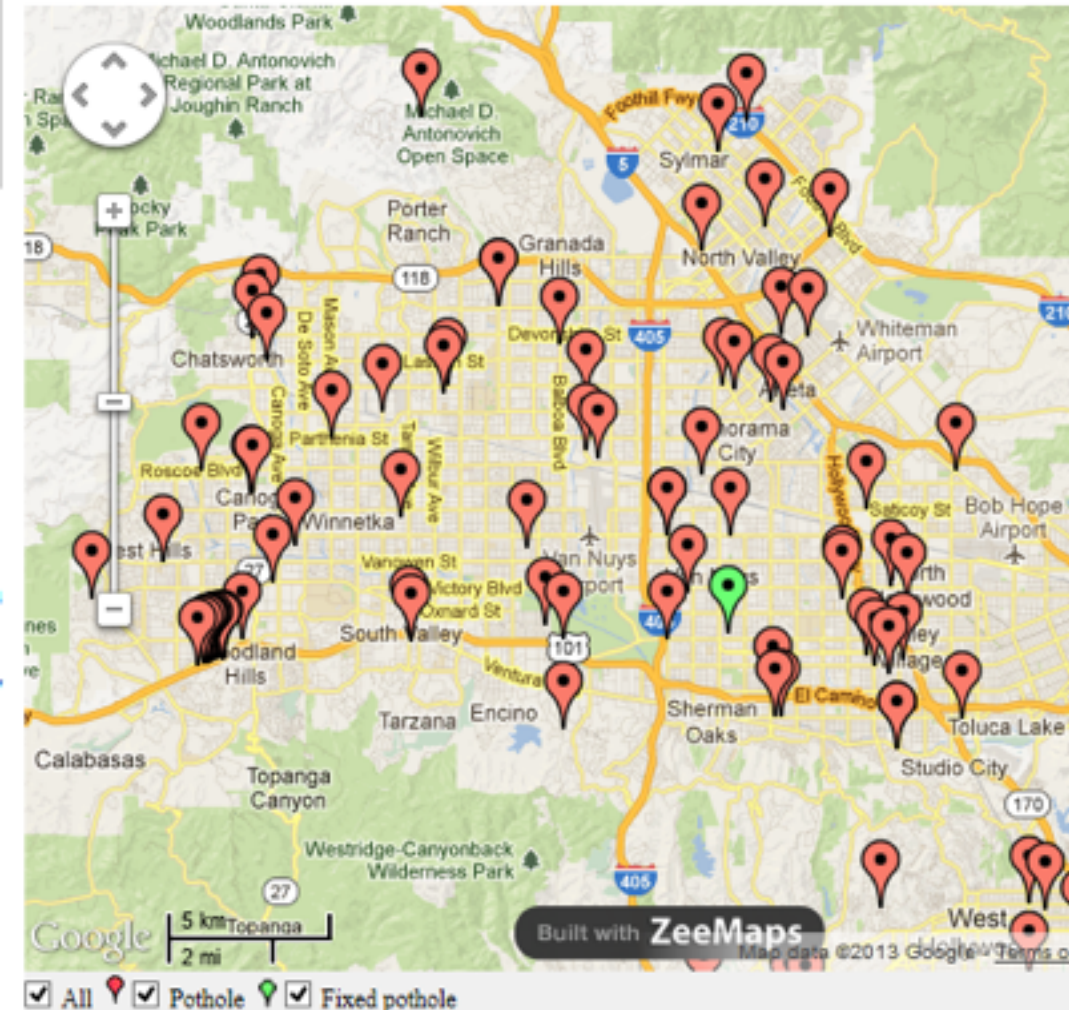
ISR 2: 模式按钮

```
mode++;
mode = mode %
    NUM_MODES;
从中断返回
```

ISR 3: 时间定时器

```
cur_time ++;
lcd_refresh--;
if (lcd_refresh==0) {
    转换tenth_miles
    并显示
    转换speed
    并显示
    if (mode == 0)
        转换cur_time
        并显示
    else
        转换avg_speed
        并显示
    lcd_refresh =
        LCD_REF_PERIOD
}
```

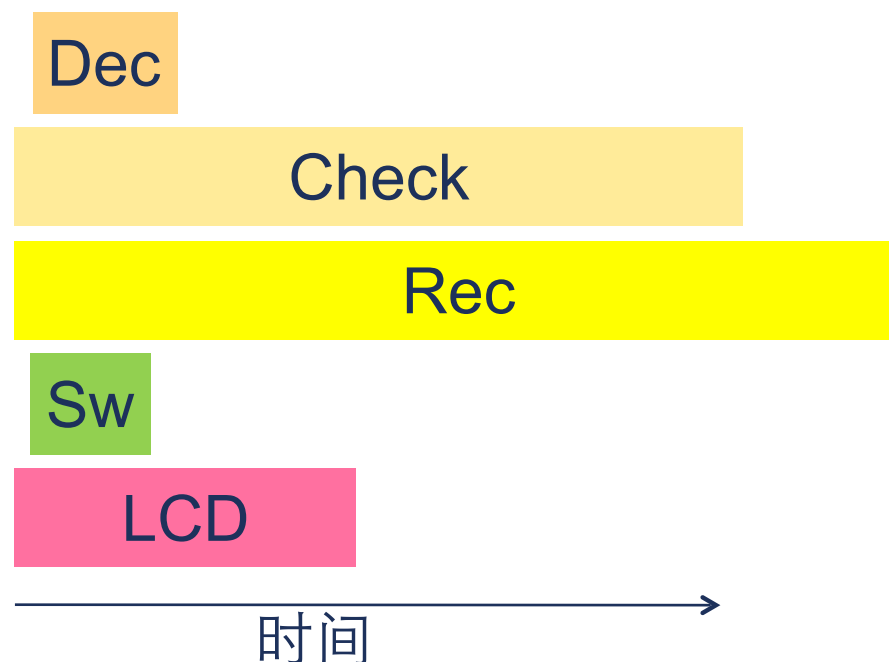
更复杂的应用



- 基于GPS的坑洼告警和移动地图
 - 当靠近坑洼时发出声音报警
 - 在LCD上显示车辆位置
 - 同时记录车辆位置数据
 - 硬件：GPS、用户开关、扬声器、LCD、闪存

应用系统的软件任务

- Dec: 解码GPS输出来获得车辆当前的位置;
- Check: 检查是否靠近某个坑洼的位置。当数据库中的坑洼数量增加时这一步会花更长时间;
- Rec: 把位置记录到闪存中, 如果要删除区块会花更长的时间;
- Sw: 读用户输入开关。每秒运行10次;
- LCD: 更新LCD上的地图, 每秒运行4次。



如何调度这些任务？

Dec

Check

Rec

Sw

LCD

- 任务调度：决定现在应该运行哪个任务
- 两个基础问题
 - 是否每次都以相同的顺序运行这些任务？
 - 是：静态调度（轮流执行、轮换执行）
 - 否：动态、有优先级的调度
 - 一个任务是否可以抢占其他任务，还是必须等待别人完成？
 - 是：抢占式
 - 否：非抢占式（协作、运行到结束再切换）

静态调度（轮流执行）



■ 优点

- 非常简单

■ 缺点

- 总是运行相同的顺序，而不管条件是否变化，也不管任务的重要程度是否不同；
- 所有的任务以相同的速率运行（多久运行一次），要改变速率就需要额外执行某功能；
- 最大的延迟就是所有任务运行的时间。轮换/执行比例就是1/最大延迟。

```
while (1) {  
    Dec();  
    Check();  
    Rec();  
    Sw();  
    LCD();  
}
```

静态调度的例子



- 如果在**Rec**刚好开始运行的时候收到**GPS**数据会怎样？
- 延迟
 - 必须等待**Rec**、**Sw**、**LCD**完成才能在**Dec**阶段开始解码位置数据；
 - 必须等待**Rec**、**Sw**、**LCD**、**Dec**、**Check**完成才能知道是否靠近一个坑！

动态调度

- 让调度可以按照需要执行
 - 基于重要程度或其他因素
 - 使得构建多种速率的系统变得简单
- 基于重要程度的调度
 - 优先级意味着不那么重要的任务不会推迟更重要的任务的执行
- 何时需要决定要运行哪个任务？
 - 粗粒度——任务结束的时候。叫做从运行到完成的工作方式（RTC），也叫做非抢占式；
 - 细粒度——任何时刻。叫做抢占式，因为任何任务都可能抢占其他任务。

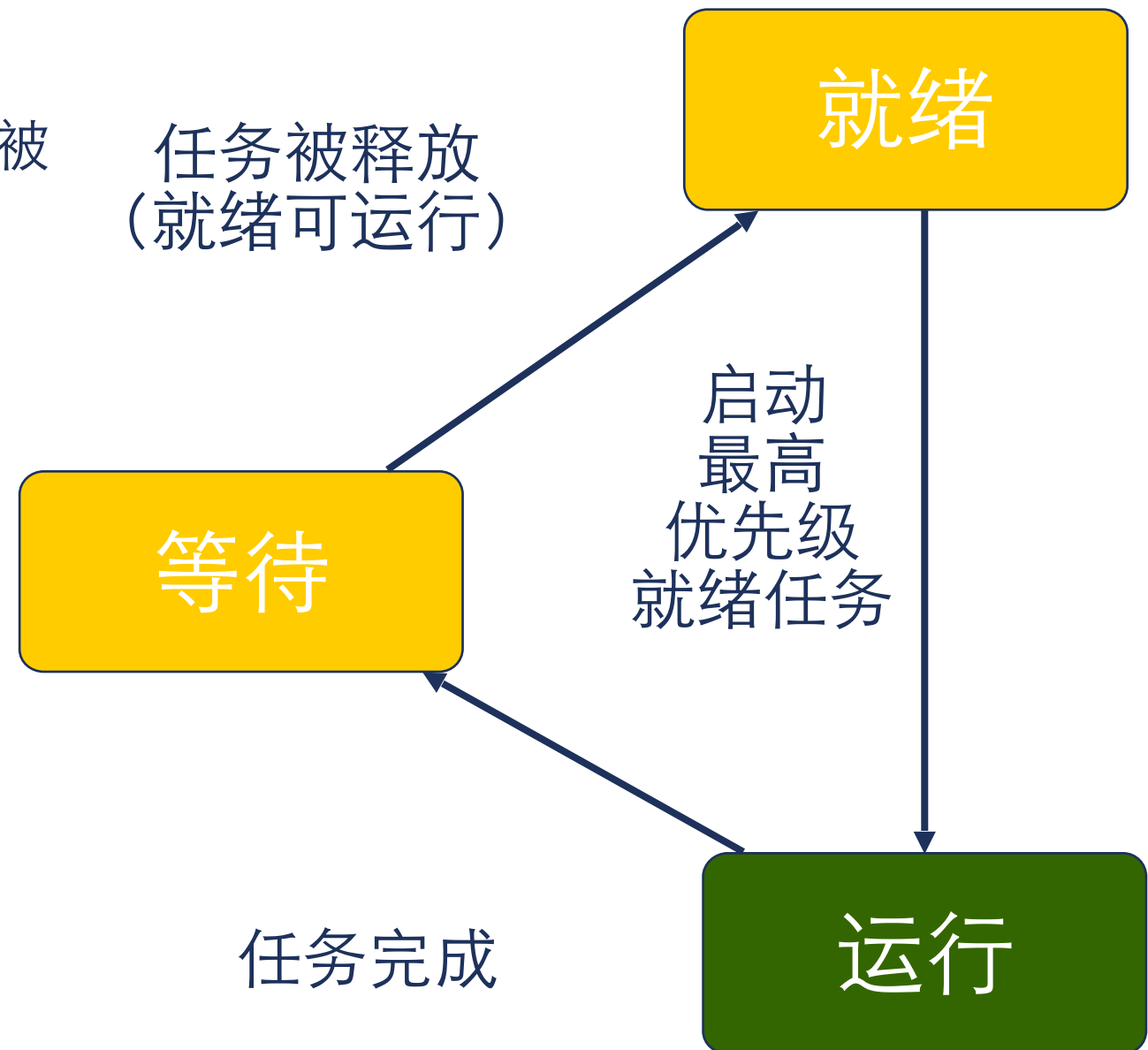
动态RTC调度



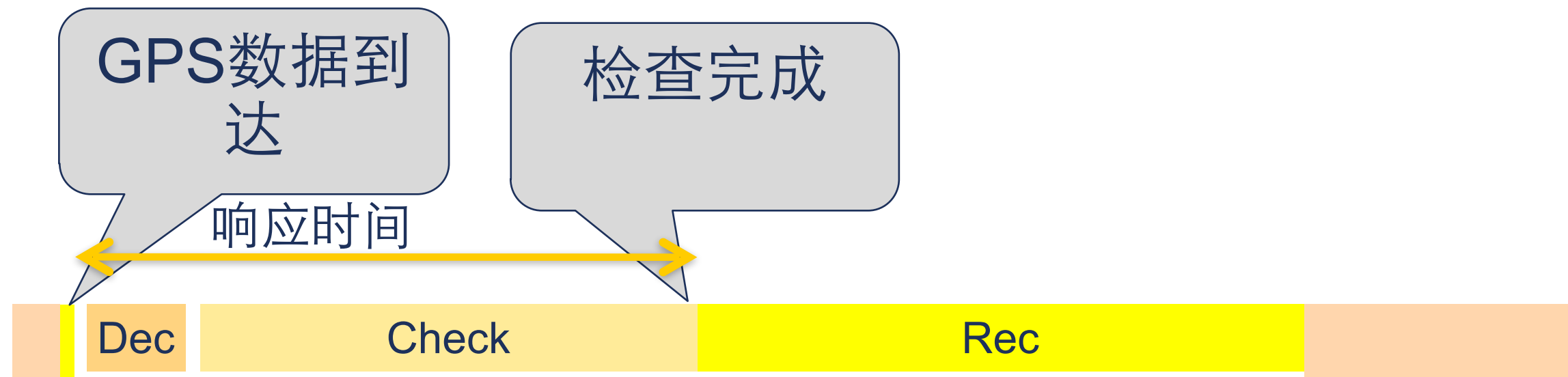
- 如果在**Rec**刚好开始运行的时候收到**GPS**数据会怎样?
- 延迟
 - 必须等待Rec完成才能在Dec阶段开始解码位置数据;
 - 必须等待Rec、Dec、Check完成才能知道是否靠近一个坑

任务状态和调度规则

- 调度器根据优先级在就绪的任务中选择
- 调度规则
 - 如果没有正在运行的任务，调度器启动最高优先级的就绪任务
 - 一旦启动，任务就运行直到完成
 - 然后任务就进入等待状态直到再次被触发或释放



动态抢占式调度



- 如果在**Rec**刚好开始运行的时候收到**GPS**数据会怎样？
- 延迟
 - 调度器把**Rec**切换出去，这样就可以立刻启动**Dec**来解码位置数据了；
 - 必须等待**Dec**和**Check**完成才能知道是否靠近一个坑

响应时间的比较

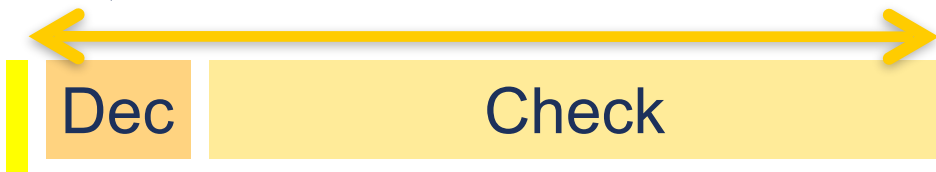
静态



动态非抢占式



动态抢占式



■ 优点

- 抢占式实现了最佳响应时间
 - 可以做更多的处理（支持更多的坑洼数据或更高的车辆速度）；
 - 或可以用更低的处理器速度、节省内存、降低功耗

■ 缺点

- 需要更复杂的编程和更多的内存
- 引入了数据竞争可能造成的不可靠性

常见调度器

- 轮流执行——非抢占而且静态
- 运行到完成——非抢占式而且动态
- 抢占式而且动态

用中断实现轮转执行

- 两个优先级
 - main代码——前台
 - 中断——后台
- 一个前/后台系统的例子
- 主的用户代码在前台运行
- 中断程序在后台运行（高优先级）
 - 一旦触发则运行
 - 处理大多数紧急的工作
 - 设置标志要求前台主循环做处理

```
BOOL DeviceARequest, DeviceBRequest,
DeviceCRequest;
void interrupt HandleDeviceA() {
    /* 做A的紧急工作 */
    ...
    DeviceARequest = TRUE;
}
void main(void) {
    while (TRUE) {
        if (DeviceARequest) {
            FinishDeviceA();
        }
        if (DeviceBRequest) {
            FinishDeviceB();
        }
        if (DeviceCRequest) {
            FinishDeviceC();
        }
    }
}
```

运行到完成的调度器

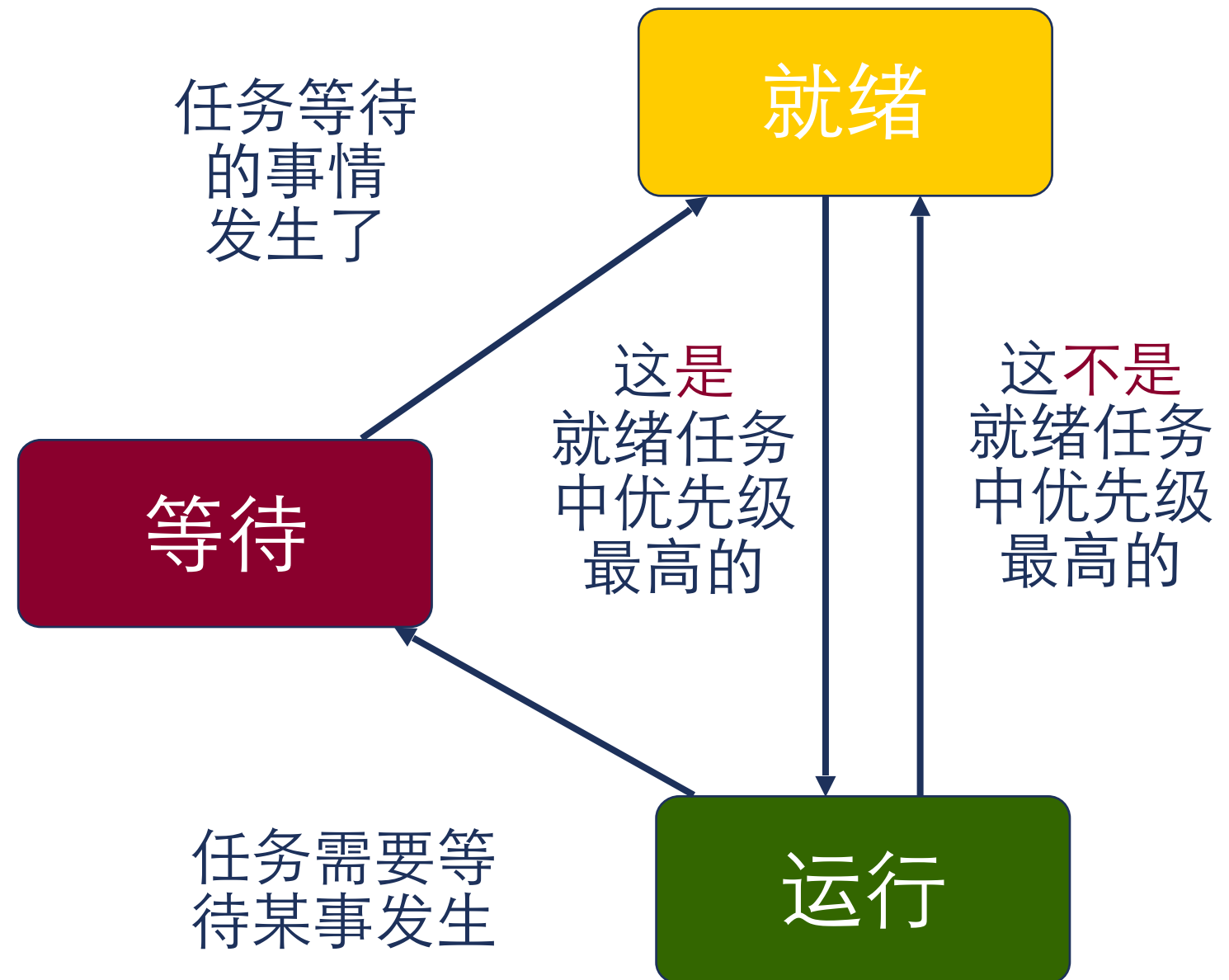
- 用一个调度器函数以适当的速率运行任务函数
 - 存放每个任务的数据的表格
 - Period周期：任务每次释放之间有多少时钟嘀嗒（时间）
 - Release Time释放时间：任务还要多久可以就绪运行
 - ReadyToRun就绪：任务已经就绪马上可以运行
 - 调度器始终运行，检查调度表查看就绪可以运行的任务（已经被“释放”了的）；
 - 由一个定时器中断触发一个ISR，由它更新调度表
 - 递减“直到下一次释放的时间”；
 - 如果这个时间递减为0了，设那个任务的运行标志，然后重置它的周期时间。
- 采用“运行到完成”模式
 - 任务的执行不会和任何其他任务交叠
 - 只有ISR可以中断一个任务
 - 在ISR完成后，会恢复之前正在运行的任务
- 优先级通常是静态的，所以可以用一个最高优先级任务排在前面的表格，以简化调度器的实现

抢占式调度器

- 任务函数不需要一次性运行到结束，但是可能可其他的任务交替运行
 - 简化了软件的编写
 - 改进了响应时间
 - 引入了新的潜在问题
- 最高优先级任务的最劣响应时间与其他任务无关，只受ISR和调度器制约
 - 较低优先级任务只受更高优先级任务制约

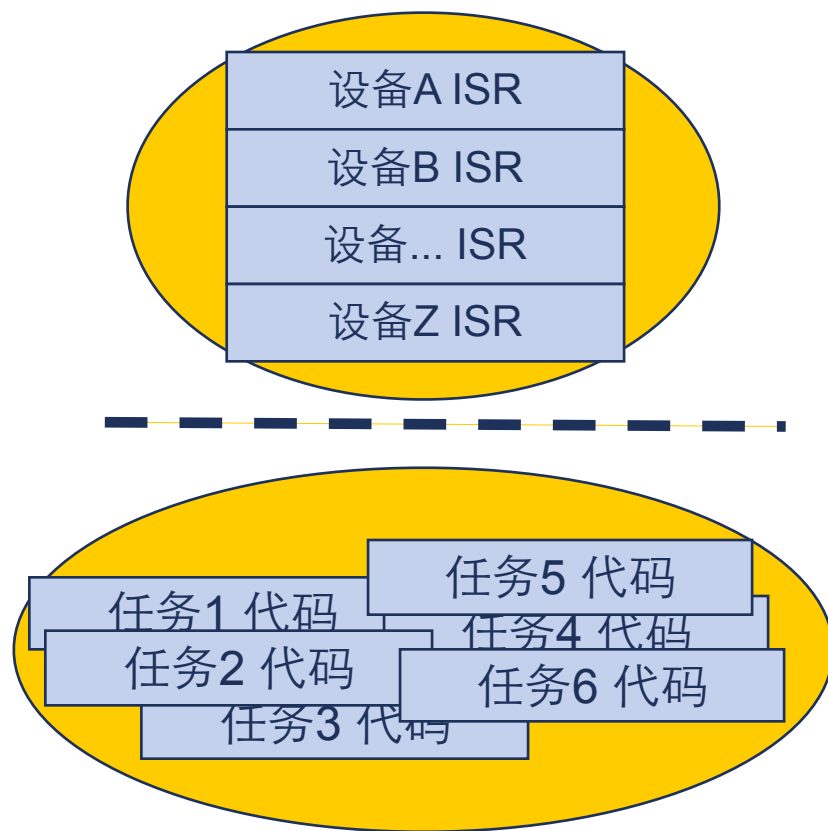
任务状态和调度规则

- 调度器根据优先级在就绪任务中选择
- 调度规则
 - 任务所做的事情可能导致自己等待（阻塞）；
 - 一个等待的任务永远不能得到CPU，必须由ISR或其他任务唤醒；
 - 只有调度器可以将任务在就绪和运行状态之间切换

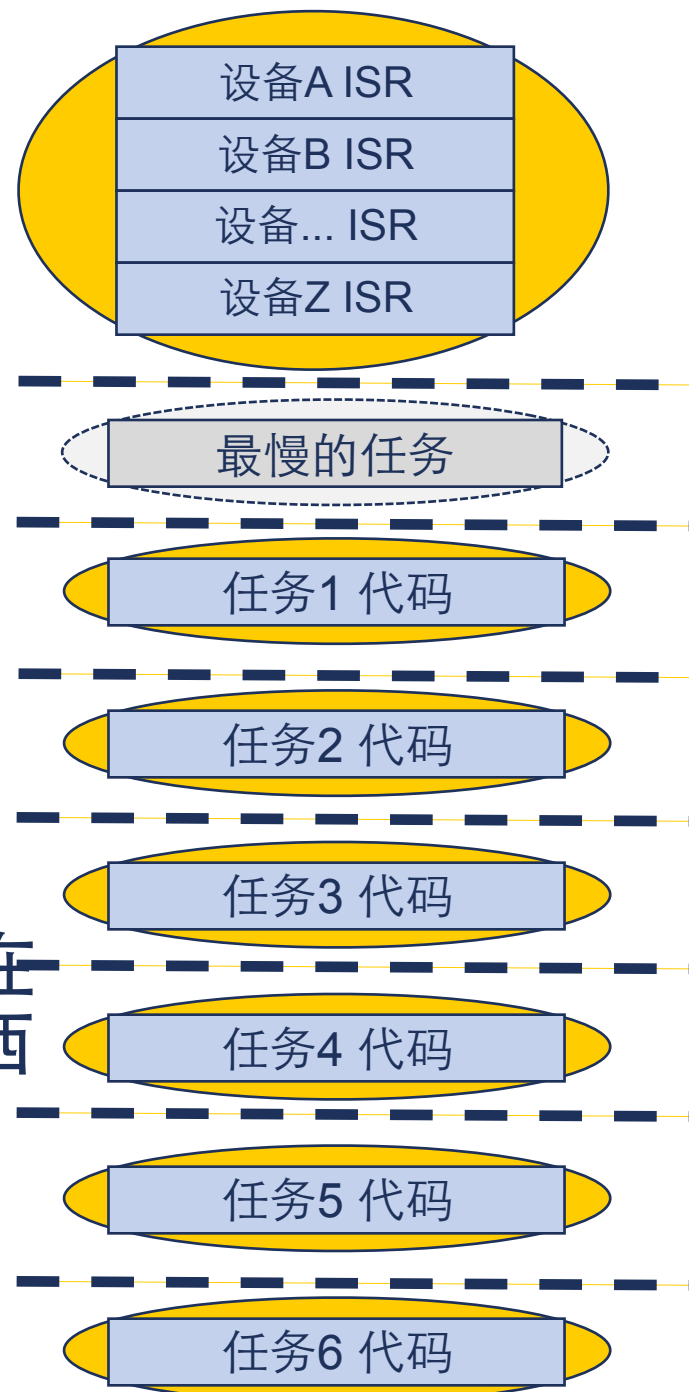


时间依赖性的比较

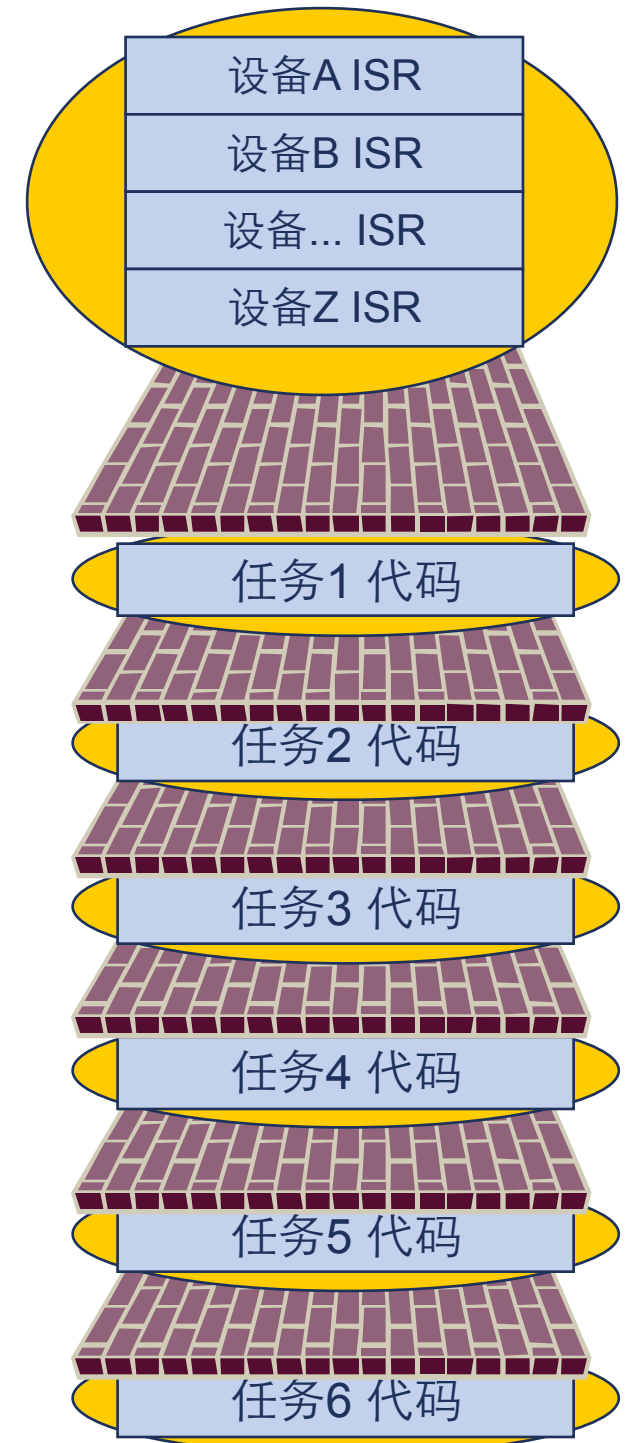
非抢占 静态



非抢占 动态



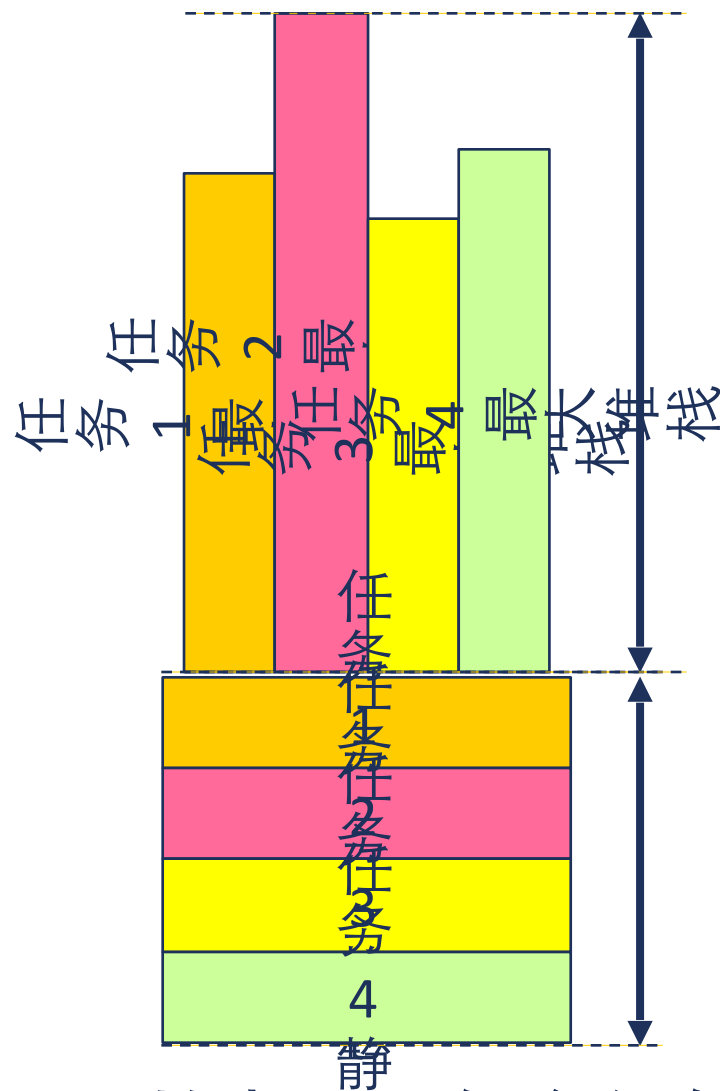
抢占 动态



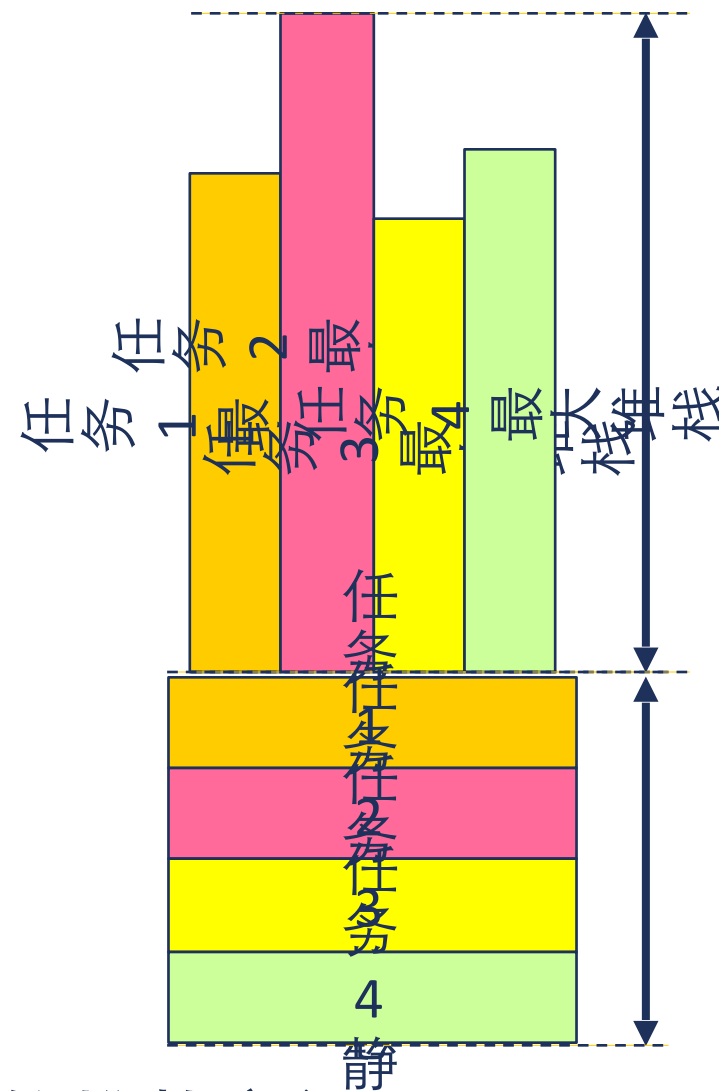
- 任务代码可能被同一级别（在椭圆中）或更上面的任何东西所延迟

RAM使用的比较

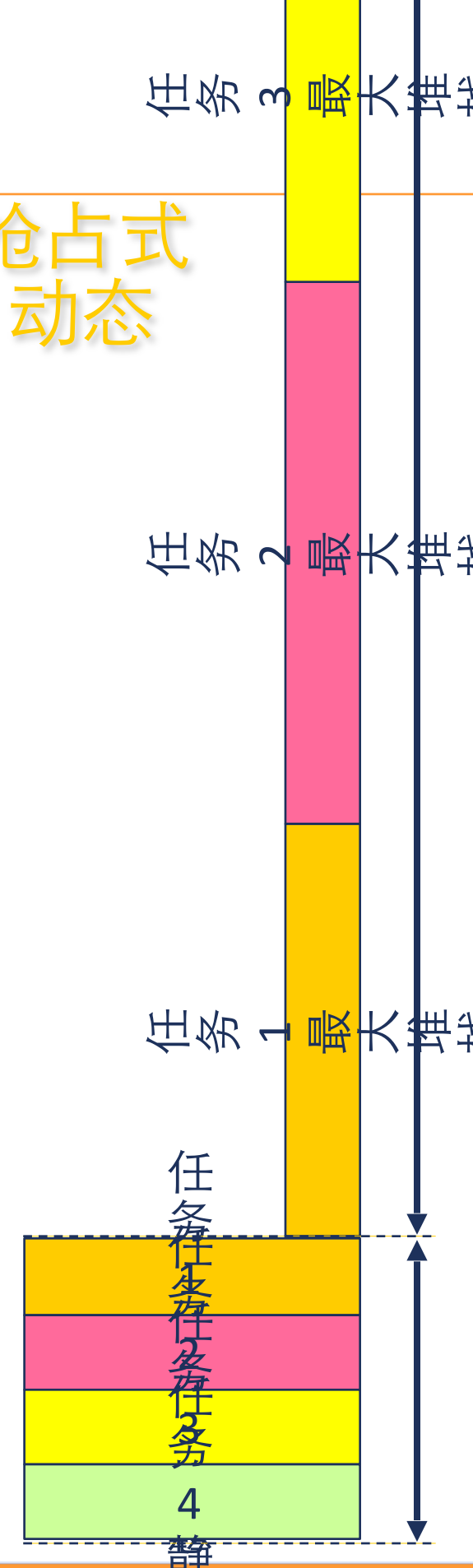
非抢占式
静态



非抢占式
动态



抢占式
动态



- 抢占需要每个任务有各自的堆栈空间
 - 都需要静态变量的空间（包括全局变量）
- *除了某些特殊情况

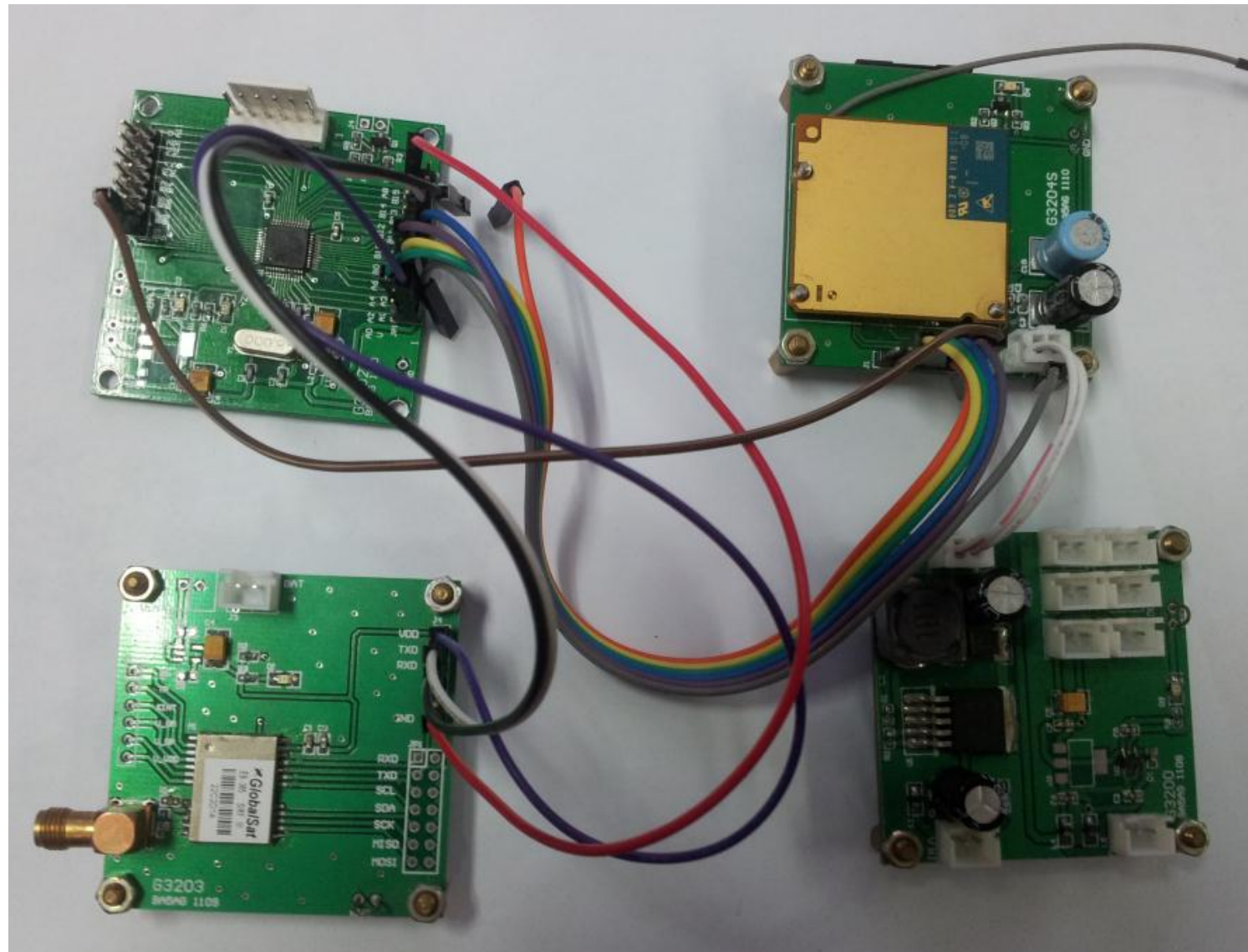
例子：车载定位器

CPU

GPRS

GPS

电源



前后台?

- 前台:
 - 初始化之后—>infinite loop
- 后台:
 - 两个串口中断
 - 定时器中断

前台

- 连接服务器
- 读GPS数据
- 如果GPS数据有效
 - 亮绿灯
 - 符合发送数据条件？ 发送
- 如果各种时间到，做相应的操作
- 喂狗

后台

- 串口中断1：读GPS数据
- 串口中断2：GPRS模块
- 定时器中断：1秒一次

GPS数据

- NMEA-1083
- 每秒一次数据
- 文本，\$开头
- 有多种语句，一般只考虑GPRMC

\$GPGLL,3015.68511,N,12006.96848,E,043352.00,A,A*68
\$GPRMC,043353.00,A,3015.68510,N,12006.96849,E,0.028,,280414,,,A*71
\$GPVTG,,T,,M,0.028,N,0.051,K,A*2D
\$GPGGA,043353.00,3015.68510,N,12006.96849,E,1,03,3.19,-6.9,M,7.2,M,,*43
\$GPGSA,A,2,25,31,14,,,,,,,,,3.34,3.19,1.00*0D
\$GPGSV,1,1,03,14,66,081,36,25,43,042,44,31,51,310,37*45
\$GPGLL,3015.68510,N,12006.96849,E,043353.00,A,A*69
\$GPRMC,043354.00,A,3015.68506,N,12006.96848,E,0.057,,280414,,,A*78
\$GPVTG,,T,,M,0.057,N,0.106,K,A*26
\$GPGGA,043354.00,3015.68506,N,12006.96848,E,1,03,3.19,-7.4,M,7.2,M,,*4E
\$GPGSA,A,2,25,31,14,,,,,,,,,3.34,3.19,1.00*0D
\$GPGSV,1,1,03,14,66,081,36,25,43,042,44,31,51,310,37*45
\$GPGLL,3015.68506,N,12006.96848,E,043354.00,A,A*68
\$GPRMC,043355.00,A,3015.68501,N,12006.96845,E,0.527,,280414,,,A*71
\$GPVTG,,T,,M,0.527,N,0.976,K,A*2B
\$GPGGA,043355.00,3015.68501,N,12006.96845,E,1,03,3.19,-6.8,M,7.2,M,,*48
\$GPGSA,A,2,25,31,14,,,,,,,,,3.34,3.19,1.00*0D
\$GPGSV,1,1,03,14,66,081,36,25,43,042,44,31,51,310,37*45
\$GPGLL,3015.68501,N,12006.96845,E,043355.00,A,A*63
\$GPRMC,043356.00,A,3015.68502,N,12006.96846,E,0.652,,280414,,,A*73
\$GPVTG,,T,,M,0.652,N,1.207,K,A*26
\$GPGGA,043356.00,3015.68502,N,12006.96846,E,1,03,3.19,-7.0,M,7.2,M,,*42
\$GPGSA,A,2,25,31,14,,,,,,,,,3.34,3.19,1.00*0D
\$GPGSV,1,1,03,14,66,081,36,25,43,042,44,31,51,310,37*4□

1秒的数据

\$GPGGA,043434.00,3015.68487,N,12006.96918,E,
1,03,3.15,-12.0,M,7.2,M,,*7F

\$GPGSA,A,2,25,31,14,,,,,,,,,3.30,3.15,1.00*05

\$GPGSV,1,1,03,14,66,082,36,25,43,042,43,31,51,310,37*41

\$GPGLL,3015.68487,N,12006.96918,E,043434.00,A,A*65

\$GPRMC,043435.00,A,3015.68486,N,12006.96917,E,
0.408,,280414,,,A*74

\$GPVTG,,T,,M,0.408,N,0.756,K,A*2B

GPRMC

- <消息 ID>,<UTC 时间>,<数据有效性状态>,<纬度>,<南北指示>,<经度>,<东西指示>,<运动速度>,<运动方向>,<UTC 日期>,<磁场变动>,<数据模式>,<校验和><行尾>
- \$GPRMC,004319.00,A,3016.98468,N,12006.39211,E,0.047,,130909,,,D*79

在PC上如何做？

- \$GPRMC,004319.00,A,3016.98468,N,12006.39211,E,0.047,,130909,,,D*79

- scanf

```
if ( strncmp(buf, "$GPRMC,",7) == 0 )
```

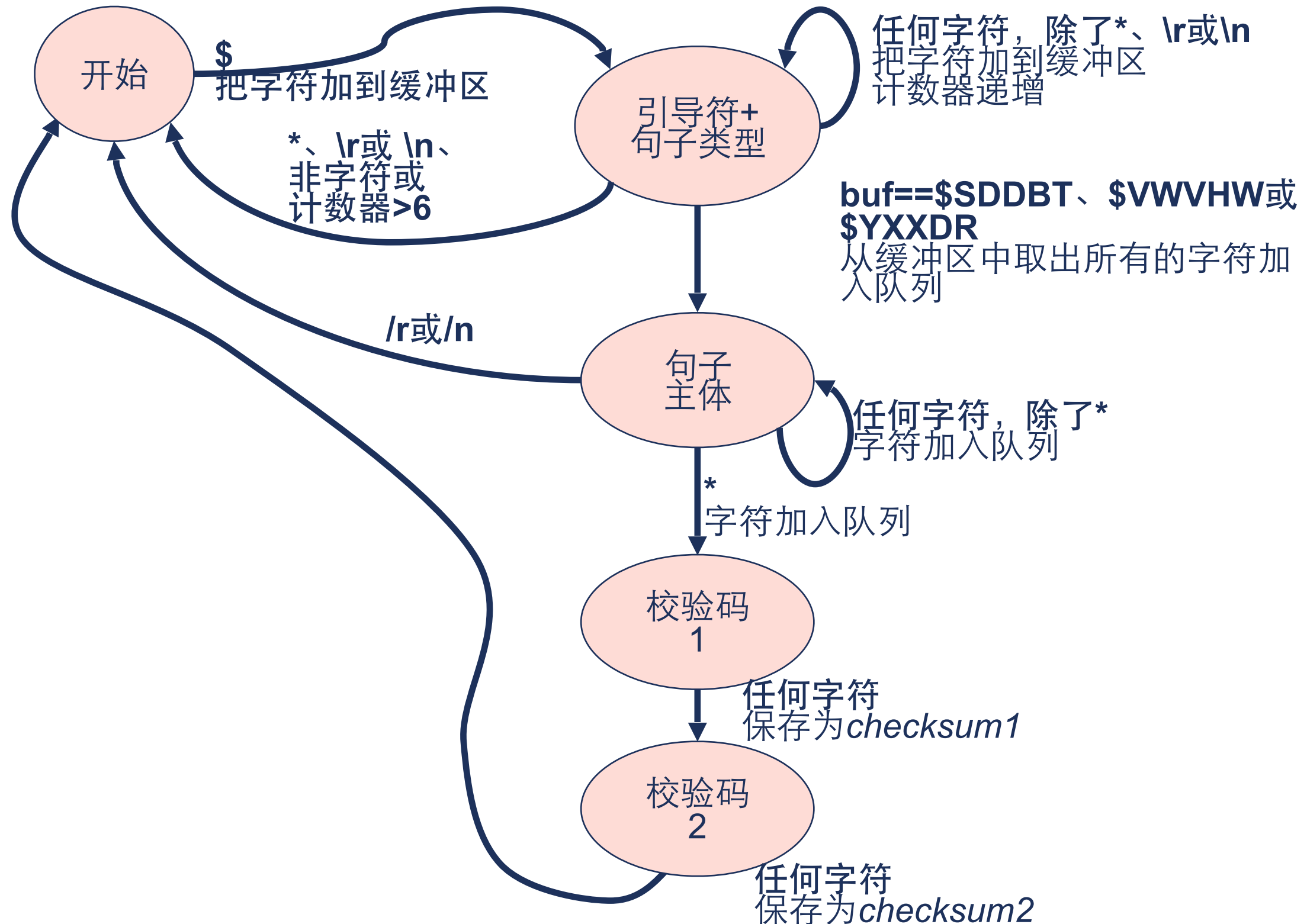
```
sscanf(buf,"%*[^,],%[^,],%[^,],%[^,],%[^,],%[^,],%[^,],%[^,],%[^,],%[^,],%[^,]",
sTime,sAV,sLati,&sNW,sLong,&sEW,sSpeed,sAngle,sDate);
```

- ## ● 从键盘到你的程序的scanf, 究竟有多远?

嵌入式系统如何做？

- 是否可以和PC一样，实现一个环形缓冲区，中断往环缓里填，主程序在需要的时候读
- 问题：
 - 多大合适？GPS数据大量是无用的
 - 大了浪费，小了，很容易读不到所需的头

解析NMEA-0183的状态机



串口状态机

- 是很多嵌入式系统的选择
 - 所需内存小，运行效率高
 - 需要为每种协议设计专门的中断响应程序
 - 代码较为复杂

GPRS接口

- 西门子MC52i
- 内置TCP/IP协议栈
 - GPRS上的socket通信是一种代理方式
 - tcp的链路保持是由GSM信令实现而非socket层
- AT指令接口

AT指令

- DTE向DCE主动发送指令
- 指令以AT开头
 - AT+CSMS=1
- DCE回答结果，并以OK或ERROR结束
- 所有的通信都是文本的，以0D0A结束

状态机 vs 环缓

- 通信具有明显的write-read模式
- DCE的回答是可控可预期的，而且没有不需要的内容
- 采用环缓能有助于简化代码

环缓代码

```
uartBuf[1][uartPtrPut[1]] = U1RBR;  
if ( uartIsEcho[1] )  
    U0THR = uartBuf[1][uartPtrPut[1]];  
  
PTR_INC(uartPtrPut[1]);  
if ( uartPtrPut[1] == uartPtrGet[1] ) {  
    PTR_INC(uartPtrGet[1]);  
}
```

有了环缓就像PC

```
int uartGetChar(int port);  
void uartPutChar(int port, char c);  
int uartPeek(int port);  
void uartPushback(int port);  
int uartHasNewLine(int port);  
int uartReadLine(int port, char* buf, int maxlen);  
void uartWriteLine(int port, char* buf);  
void uartPrintln(int port, char* buf);  
void uartPrintf(int port, const char *fmt, ...);
```

轮询 vs 主动通知

- GPRS模块可以配置为具有主动通知功能，在特定的事件发生时会自动向DTE发出文本
 - 收到短信、网络连接断开
- 传统设计认为主动通知比轮询好，因为轮询不够及时
- 但是通知会夹杂在AT和回答之间，使得DTE的逻辑复杂
- 所以我们宁愿采用发送AT指令轮询来获得这些事件的消息

定时器

- 硬件上通常具有多个定时器
- 程序逻辑上需要定时器用于：
 - 等待一定时间后再做某事
 - 某个等待（如串口的回答）不能超时
 - 一定时间后需要做某事（不等待）

EE vs CS

- EE习惯为每一个定时需要开启一个硬件定时器
 - 定时器时间到了产生中断，中断响应程序去做事，或设标志，在主程序中读到标志做事
- CS习惯用一个硬件定时器产生时间，在每个时间节拍判断有什么挂着要做的事情的时间到了
- 因为1970年代的PDP11只有一个硬件定时器！

STM32 F0 SERIES - ARM CORTEX™-M0 ENTRY-LEVEL MCUS

Part number	Flash size (Kbytes)	Internal RAM size (Kbytes)	Package	Timer functions		ADC	DAC	I/Os	Serial interface						
				16-/32-bit timers	Others				SPI	I²S	I²C	USART	CEC	USB FS	CAN 2.0B
STM32F051C4	16	4	LQFP48	7x16-bit / 1x32-bit	2 x WDG, RTC, 24-bit downcounter	13x12-bit	1x12-bit	39	1	1	1	1	1		
STM32F051K4	16	4	UFQFPN32	7x16-bit / 1x32-bit		13x12-bit	1x12-bit	27	1	1	1	1	1		
STM32F051R4	16	4	LQFP64	7x16-bit / 1x32-bit		19x12-bit	1x12-bit	55	1	1	1	1	1		
STM32F051C6	32	4	LQFP48	7x16-bit / 1x32-bit		13x12-bit	1x12-bit	39	1	1	1	2	1		
STM32F051K6	32	4	UFQFPN32	7x16-bit / 1x32-bit		13x12-bit	1x12-bit	27	1	1	1	2	1		
STM32F051R6	32	4	LQFP64	7x16-bit / 1x32-bit		19x12-bit	1x12-bit	55	1	1	1	2	1		
STM32F051C8	64	8	LQFP48	7x16-bit / 1x32-bit		13x12-bit	1x12-bit	39	2	1	2	2	1		
STM32F051K8	64	8	UFQFPN32	7x16-bit / 1x32-bit		13x12-bit	1x12-bit	27	1	1	1	2	1		
STM32F051R8	64	8	LQFP64	7x16-bit / 1x32-bit		19x12-bit	1x12-bit	55	2	1	2	2	1		
STM32F071V8	64	16	LQFP100	9x16-bit / 1x32-bit		19x12-bit	1x12-bit	87	2	1	2	4	1		
STM32F071CB	128	16	LQFP48	9x16-bit / 1x32-bit		13x12-bit	1x12-bit	37	2	1	2	4	1		
STM32F071RB	128	16	LQFP64	9x16-bit / 1x32-bit		19x12-bit	1x12-bit	51	2	1	2	4	1		
STM32F071VB	128	16	LQFP100	9x16-bit / 1x32-bit		19x12-bit	1x12-bit	87	2	1	2	4	1		
STM32F0x2 line - 48 MHz CPU with USB															
STM32F042C4	16	6	LQFP48 WLCSP49	5x16-bit / 1x32-bit	2 x WDG, RTC, 24-bit downcounter	10x12-bit		38	2	2	1	2	1	1	1
STM32F042F4	16	6	TSSOP20	5x16-bit / 1x32-bit		10x12-bit		16	2	2	1	2	1	1	1
STM32F042G4	16	6	UFQFPN28	5x16-bit / 1x32-bit		10x12-bit		24	2	2	1	2	1	1	1

```

struct timerr
{
    int raw;           // 设定值，对于动态分配的定时器，0表示未
    分配
    int now;           // 当前计数值，-1表示未启动
    int enable;        // 是否已经被触发（计数到0）
};

```

```

void __irq IRQ_TIMER0 (void)
{
    int i;
    for(i=0;i<NUM_TIMERS;i++) {
        if(freq[i].now>0) {
            freq[i].now--;
            if(freq[i].now==0) {
                freq[i].enable=1;
                freq[i].now=freq[i].raw;
            }
        }
    }
}
TIMER=0x01;

```

```
timerConfig(TIMER_SHELL, 20); // 20s  
timerStart(TIMER_SHELL);
```

```
while ( ( ch = uartGetChar(UART0) ) != '~' ) {  
    if ( ch == '$' )  
        goto EXIT;  
    if ( timerIsSet(TIMER_SHELL) )  
        goto EXIT;  
}
```

OS?

- 程序逻辑上需要定时器用于：
 - 等待一定时间后再做某事
 - 某个等待（如串口的回答）不能超时
 - 一定时间后需要做某事（不等待）

什么是RTOS?

- 实时是什么意思?
 - 每个任务和中断服务程序的最大响应时间是可以计算和得到保证的
 - 响应时间的这个“绑定”使得它可以用在硬实时系统中（具有必须满足的截止时间）
- RTOS里有什么
 - 任务调度器
 - 采用抢占式和优先级策略以使得响应时间最小
 - 支持中断
 - RTOS核心集成的服务
 - 进程间通信和同步（安全的数据共享）
 - 时间管理
 - RTOS集成的可选服务
 - I/O抽象?
 - 内存管理?
 - 文件系统?
 - 网络支持?
 - GUI??