

# 嵌入式重难点总结

## 软中断流程

1. 通过 `__swi` (中断号) function 给中断函数指定中断号 (一般在C代码中撰写)
2. 调用function时, 相当于执行了 `SWI 中断号` 指令
3. 再执行 `SWI` 之前需要先注册中断程序, 即把处理软中断的程序 (`SWI_handler`) 地址放到 `0x08` 软中断向量表地址上
4. 注册之后执行 `SWI` 的时候, 会执行 `0x08` 中断向量地址上存放的跳转指令 (`B SWI_handler`), 即跳转到 `SWI_handler`; 同时将function的参数传递到寄存器内 (如果参数不超过4个), 超过4个应该存放到内存

```
// 中断注册程序
unsigned Install_Handler (unsigned *handlerloc, unsigned *vector)
{
    unsigned vec, oldvec;
    vec = ((unsigned)handlerloc - (unsigned)vector - 0x8)>>2;    // 确保中断处
    理函数地址偏移在26位以内: 正负32M
    if ((vec & 0xFF000000) != 0)    // 取低24位, 即中断服务程序地址
    { return 0; }
    vec = 0xEa000000 | vec;    // 0xEa 应该时跳转指令B的操作码, 这里表示跳转指
    令: "B handlerloc"
    oldvec = *vector;
    *vector = vec;    // 将中断向量表中中断向量内写入跳转指令的编码, 即
    vec
    return (oldvec);
}
```

5. 跳转后, `SWI_handler` 需要干五件事:
  - 1) 将保存函数参数的寄存器以及 `lr` 寄存器存放到堆栈里保存,
  - 2) 通过 `BIC` 指令获取中断号放置在 `r0` 寄存器,
  - 3) 将堆栈指针保存到 `r1`,
  - 4) 跳转到要执行function的处理函数中 (`C_SWI_handle`) 加密, 并将 `r0`, `r1` 寄存器作为函数参数传递
  - 5) 处理完中断函数后恢复现场

```
SWI_Handler
    STMFD    sp!, {r0-r12, lr}    ; 保存现场
    LDR      r0, [lr, #-4]    ; 获取 SWI 指令
    BIC      r0, r0, #0xff000000    ; 参数1, NUM
    MOV      R1, SP    ; 参数2, 传递堆栈指针
    BL      C_SWI_Handler    ; To Function
    LDMFD    sp!, {r0-r12, pc}^    ; 处理完中断恢复现场, 将最初的lr->pc, 继续执行
SWI指令的下一条指令
    END
```

6. `C_SWI_handle` 接受中断号和堆栈指针, 通过堆栈指针开始从堆栈中获取参数数据, 根据中断号选择计算的程序逻辑, 计算完成将结果再次写入堆栈

```

void C_SWI_handler (int swi_num, int *reg )
{
    switch (swi_num)
    {
        case 0 :
            ..... /* SWI number 0 code */
            break;
        case 1 :
            ..... /* SWI number 1 code */
            break;
        .....
        default :
            break; /* Unknown SWI - report error */
    }
    return;
}

```

或

```

C_SWI_Handler
    STMFD    sp!, {r0-r12, lr}
    CMP     r0, #MaxSWI          ; Range check
    LDRLE   pc, [pc, r0, LSL #2] ; (PC -> DCD SWInum0)
    B       SWIOutOfRange

SWIJumpTable
    DCD     SWInum0
    DCD     SWInum1

SWInum0    ; SWI number 0 code
    B      EndofSWI

SWInum1    ; SWI number 1 code
    B      EndofSW

EndofSW
    SUB     lr, lr, #4
    LDMFD   sp!, {r0-r12, pc}^
    END

```

7. 最后从堆栈中取出计算结果放到寄存器r0中
8. 中断结束，从堆栈中恢复现场，继续执行 SWI 指令后的指令

## 系统启动流程

以 S3C2410A 处理器的启动文件 [startup.s](#) 为例。

1. 系统上电后，先进行复位Reset
2. 设置PC的初始值0，执行Reset\_Handler
3. Reset\_Handler初始化需要干以下几件事
  - 1) 设置WT\_Setup (Watchdog Timer) 、CLK\_Setup (Clock) 、MC\_Setup (Memory Controller) 、PIO\_Setup (IO port) 、Stack\_Setup (Stack/Heap)
  - 2) 设置异常中断的模式栈指针，包括Undefined、Abort、FIQ、IRQ、Supervisor、User
  - 3) 设置SRAM

BL INISDRAM

- 4) 堆栈空间的设置
- 5) 进入C程序代码(\_\_main -> init -> main)

```

IMPORT __main

LDR     R0, =__main
BX      R0

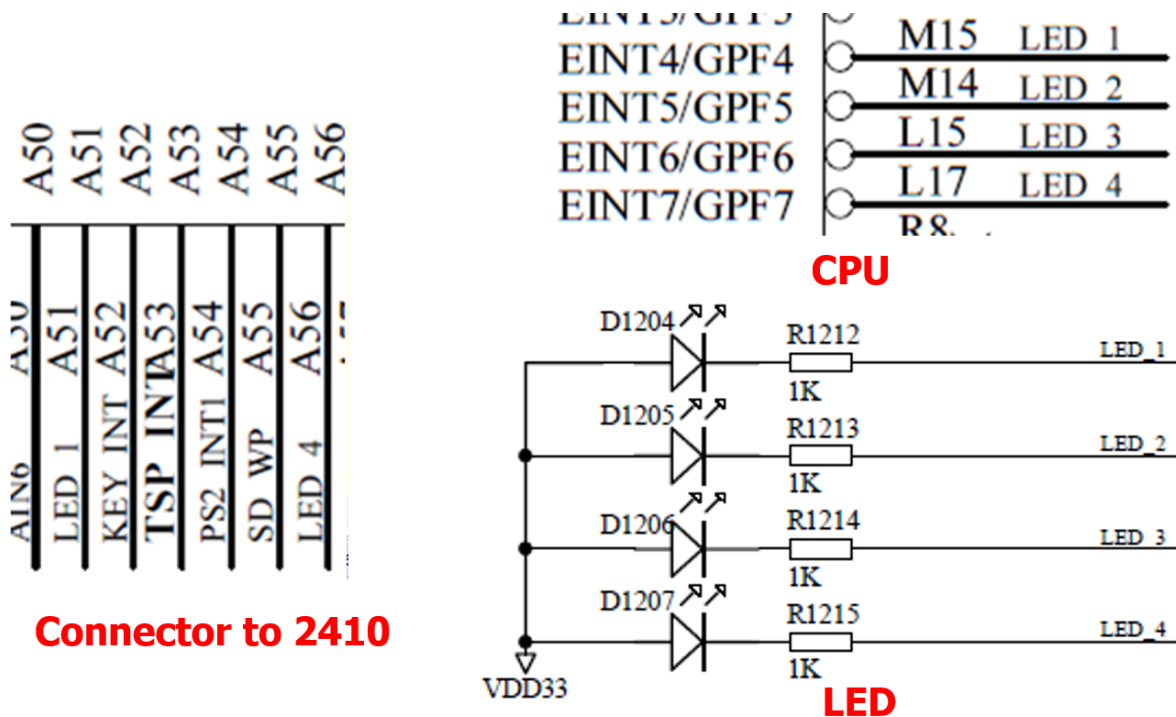
```

- 如果调试时无法进入main，可能的原因是什么？
  - main单词拼写错误
  - 启动代码不对
  - 没有将main.c文件添加到工程文件中
- Startup.s文件的功能和作用？
  - 设置中断向量与中断服务程序地址
  - 分配堆栈空间
  - 设置时钟，看门狗Timer，内存控制，IO端口
  - 设置中断入口IRQ\_Entry
  - 实现Reset\_Handler
- BootLoader移植需要修改什么代码？
  - Start.s是采用汇编语言编写的U-Boot程序入口代码，完成对底层硬件的初始化，其中有一个很重要的功能是从NAND FLASH中把 Stage2阶段的代码复制到 SDRAM中。在此阶段，涉及到对 NAND FLASH的读操作，在U-Boot中，没有对 NANDFLASH读操作的驱动，采用以下方法实现：通过调用 board/smdk2410/ nand\_read.C中的 nand\_read\_11函数将 Stage2阶段的代码复制到ram中。

## LED显示实验

### 硬件配置

- LED连接



- LED连接
  - LED1 <--> GPF4
  - LED2 <--> GPF5
  - LED3 <--> GPF6
  - LED4 <--> GPF7

- LED 控制
  - 0 -> on
  - 1 -> off
- GPF控制(GPIO 寄存器)

Register	Address	R/W	Description	Reset Value
GPFCON	0x56000050	R/W	Configure the pins of port F	0x0
GPFDAT	0x56000054	R/W	The data register for port F	Undefined
GPFUP	0x56000058	R/W	Pull-up disable register for port F	0x0

GPFCON	Bit	Description	
GPF7	[15:14]	00 = Input 10 = EINT7	01 = Output 11 = Reserved
GPF6	[13:12]	00 = Input 10 = EINT6	01 = Output 11 = Reserved
GPF5	[11:10]	00 = Input 10 = EINT5	01 = Output 11 = Reserved
GPF4	[9:8]	00 = Input 10 = EINT4	01 = Output 11 = Reserved

GPFDAT	Bit	GPFUP	Bit	Description
GPF[7:0]	[7:0]	GPF[7:0]	[7:0]	0: enabled 1: disabled

GPFCON (4-7 Output):0b 0101 0101 XXXX XXXX
GPFUP (4-7 No concern):0b XXXX XXXX
GPFDAT (4-7 on):0b 0000 XXXX
GPFDAT (4-7 off):0b 1111 XXXX

C代码

- 4个LED等循环闪烁（周期2秒），详细解释见注释

```

#include<stdlib.h>
#include<stdio.h>
int delay(int times);    // 声明延时函数

// 注意这两个地址是根据所给的板子型号的手册表格上查找，根据表格赋值
int *rGPFCON = (int *) 0x56000050;  // rGPFCON配置哪些引脚需要使用，采取的模式是什么
                                     (00:Input,01:Output)，LED显示即采用输出01
int *rGPFDAT = (int *) 0x56000054;  // rGPFDAT配置相应引脚输入的信号是高电平还是低电平，
                                     具体是高电平亮还是低电平亮看LED电路连接，本次实验的LED电路可以看出，LED引脚低电平时灯亮，要根据
                                     具体电路分析

void main( void)
{
    *rGPFCON=0x5500;    // 0x5500 = 0b0101_0101_0000_0000;高八位的四个引脚（GPF4-7）
    都是01模式，即输出模式，结合CPU电路可知，LED与引脚的连接（例如：LED1对应GPF4）

    while(1)
    {
        *rGPFDAT = 0x00;    // 0x00 = 0b0000_0000;高4位对应GPF引脚的输入Data信号，0表
        示输入低电平，此时灯亮
        delay(1000);    // 延时1000ms，即1s
    }
}

```

```

        *rGPFDAT = 0xf0;    // 0xf0 = 0b1111_0000;高4位对应GPF引脚的输入Data信号，1表示输入高电平，此时灯灭
        delay(1000);
    }
}

int delay(int times)    // 定义延时函数
{
    int i,j;
    for(i=0;i<times;i++)
        for(j=0;j<times;j++)
        {

        }
    return 1;
}

```

- 知道每个引脚作用后，很容易实现想要的灯闪方式：

```

// 知道每一位的具体作用，那么想让等怎么亮就让那一位数据（Data）信号给0即可
// 例如：实现一个简单的流水main函数
int main(void)
{
    *rGPFCON=0x5500;
    while(1)
    {
        int i;
        *rGPFDAT = 0x10;
        delay(500);

        for(i=1; i<=4; i=i+1){
            *rGPFDAT <<= 1;
            delay(500);
        }
        for(i=1; i<=4; i=i+1){
            *rGPFDAT >>= 1;
            delay(500);
        }
    }
    return 1;
}

```

## 注意事项

- 一定要结合电路图判断时**高电平有效**还是**低电平有效**，同时看**LED连接的是哪些引脚**，并在表格中**确定这些引脚是 GPFCON 的哪些位**
- 一定要在给出的表格确定 \*rGPFCON 和 \*rGPFDAT 的**初始地址**，不同板子地址不一样
- 一般LED显示相应的 GPFCON 使用的**引脚位置 01**，表示输出，但是具体还是要看一下表格确认一下，以免有的板子 01 就不是输出
- 最后通过 GPFDAT （8位）传送数据信号，注意这个变量位数一般是 GPFCON （16位）的一半，**引脚关系也是对应的**，例如 GPFCON 最高两位和 GPFDAT 最高一位都表示 GPF7 引脚，再根据高低电平，相应位置传递0或1即可。
- 注意C语言指针变量赋值的格式。

## OpenMP

- #pragma
  - parallel: 表示这段代码将被并行执行;
  - for: 表示将循环计算任务分配到多个线程中并行执行;
  - sections: 用于实现多个结构块语句的任务分担;
  - parallel sections: 类似于parallel for;
  - single: 表示一段只被单个线程执行的代码;
  - critical: 保证每次只有一个OpenMP线程进入;
  - flush: 保证各个OpenMP线程的数据映像的一致性;
  - barrier: 用于并行域内代码的线程同步, 线程执行到barrier时要停下等待, 直到所有线程都执行到barrier时才继续往下执行;
  - atomic: 用于指定一个数据操作需要原子性地完成;
  - master: 用于指定一段代码由主线程执行;
  - Thread private: 用于指定一个或多个变量是线程专用。

- 基本使用

- #pragma omp parallel for ☆☆☆☆☆

```
int step = 100;
int _tmain(int argc, _TCHAR* argv[])
{
    int i; // 注意int i 不要再for循环里面写int i = 0
    #pragma omp parallel for
    for (i = 0; i < step; i++)
    {
        printf("i = %d\n", i);
    }
    return 0;
}
```

- 为了防止共享数据导致并行抢占改写, 需要使用:

- #pragma omp parallel for private(共享变量) ☆☆☆

```
ifirst = 10;
int k[600];
for(j = 0; j <= 60; j++)
{
    #pragma omp parallel for private(i2)
    for(i=0;i<6000000;i++){
        i2 = i-(i/600)*600;
        k[i2] = ifirst + i;
    }
}
```

- share 共享不能用于类似sum求和的方式

```

float sum = 0.0;
float a[10000], b[10000];
for(int j=0; j<10; j++)
{
    sum=0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<10000; i++) { // 注意int不要写在for循环, "int i;"在for外面声明
        sum += a[i] * b[i];
    }
    printf("j=%d, sum=%f\n", j, sum);
}
return sum;

```

- `critical` 每次让一个线程去操作, 即让下一行代码串行

```

float sum = 0.0;
float a[10000], b[10000];
for(int j=0; j<10; j++)
{
    sum=0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    printf("j=%d, sum=%f\n", j, sum);
}
return sum;

```

- `reduction` 让所有线程在最后统一执行某一操作, 例如: 求和 "+: " ☆

```

float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}

```

- `barrier` 用于并行域内代码的线程同步, 线程执行到barrier时要停下等待, 直到所有线程都执行到barrier时才继续往下执行

```

#pragma omp parallel private(myid, istart, iend)
myrange(myid, istart, iend);
for(i=istart; i<=iend; i++){
    a[i] = a[i] - b[i];
}
#pragma omp barrier
myval[myid] = a[istart] + a[iend]

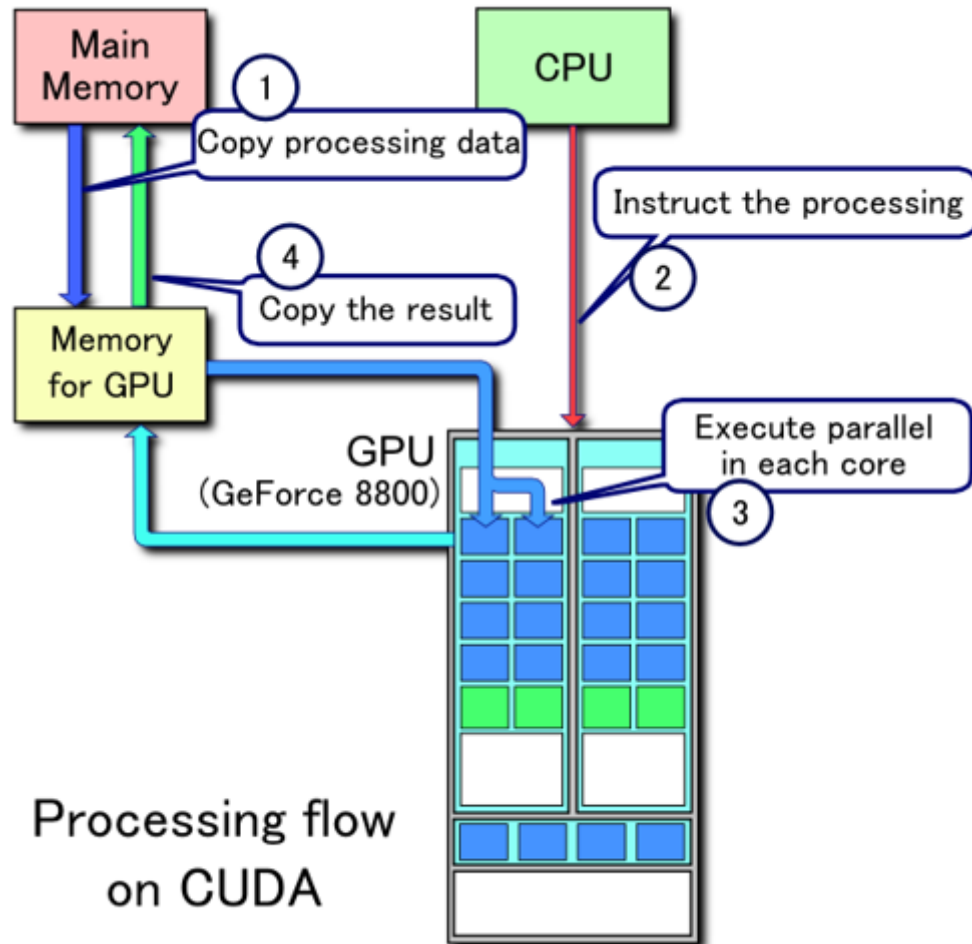
```

- `schedule` 分配线程执行
  - `static`: 线程按顺序分配, 线程内数据处理顺序也按照for循环顺序分配

- `dynamic`: 线程随机分配, 线程内数据处理顺序也按照for循环顺序分配
- `guided`: 线程和线程内数据处理顺序都是随机分配

## CUDA

- CUDA处理数据过程
  - Copy data from main mem to GPU mem.
  - CPU instructs the process to GPU.
  - GPU execute parallel in each core.
  - Copy the result from GPU mem to main mem.



- CUDA编程基础代码
  - `<<<blk,thr>>>` 中, `blk` 表示需要使用的block数目, `thr` 表示每个使用的block用多少个线程使用

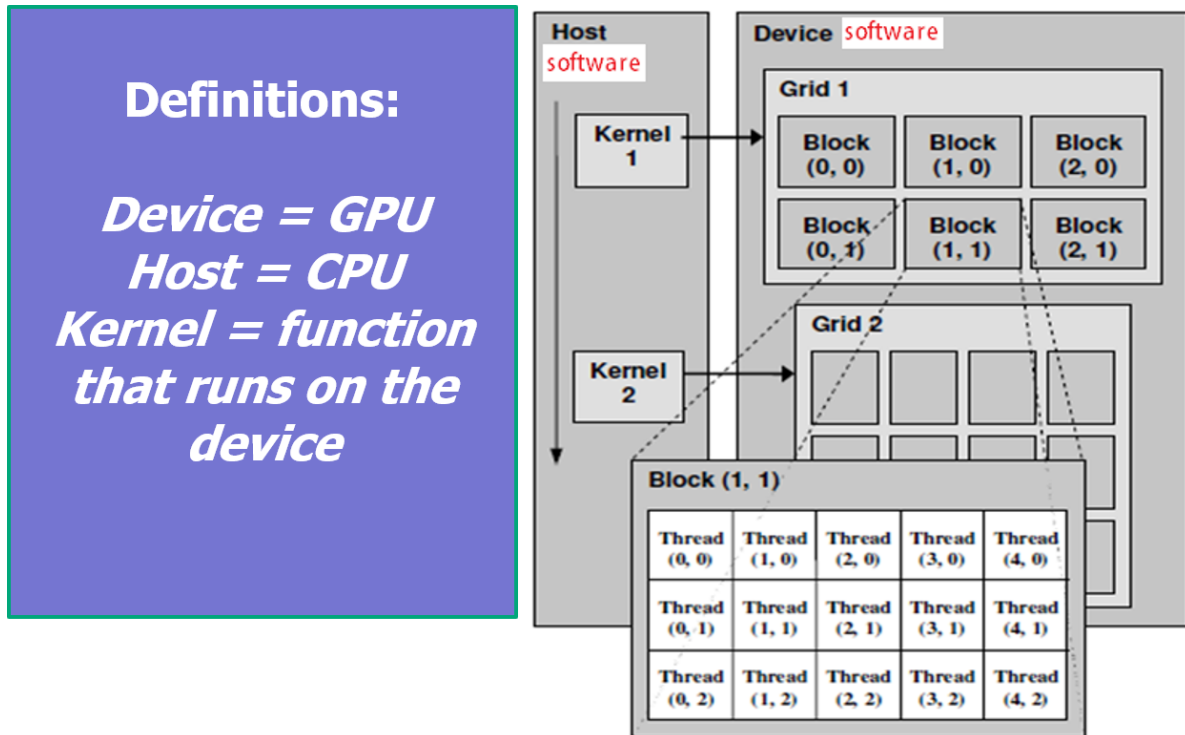
```
//hello_world.cu:
#include <stdio.h>

__global__ void hello_world_kernel()    // CUDA执行的核(kernel)函数需要加__global__
{
    printf("Hello world\n");
}

int main()
{
    hello_world_kernel<<<1,1>>>();    // 函数调用需要指定<<<Blk, Tid>>>
}
```



- 基本概念



BlockID= blockIdx.y\*gridDim.x+blockIdx.x

ThreadID=BlockID\*blockDim.x\*blockDim.y\*blockDim.z  
 +threadIdx.z\*blockDim.x\*blockDim.y\*  
 +threadIdx.y\*blockDim.x  
 +threadIdx.x

- 例子

Grid 0

Block (0,0)	Block (1,0)	Block (2,0)
Block (0,1)	Block (1,1)	Block (2,1)

```
dim3 grid(3,2) ; dim3 block(12,1,1);
gridDim.x=3; gridDim.y=2;
blockDim.x=12; blockDim.y=1; blockDim.z=1;
blockIdx.x=1; blockIdx.y=1;
threadIdx.x=4, threadIdx.y=0; threadIdx.z=0;
threadID=(1*3+1)*12+4=52
```

- 一维CUDA模板代码

```
// kernel definition
__global__ void VecAdd(float* A, float* B, float* C)    // 注意数组参数用指针形式传递
{
    int i = threadIdx.x;    // 线程只需要用x维度
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);    // 使用一个block和N的线程
    ...
}
```

- 二维CUDA一个Block模板代码

```
// kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) // 注意二维数组的参数
{
    int i = threadIdx.x;
    int j = threadIdx.y;    // 需要指定线程的两个维度
    C[i][j] = A[i][j] + B[i][j];
}

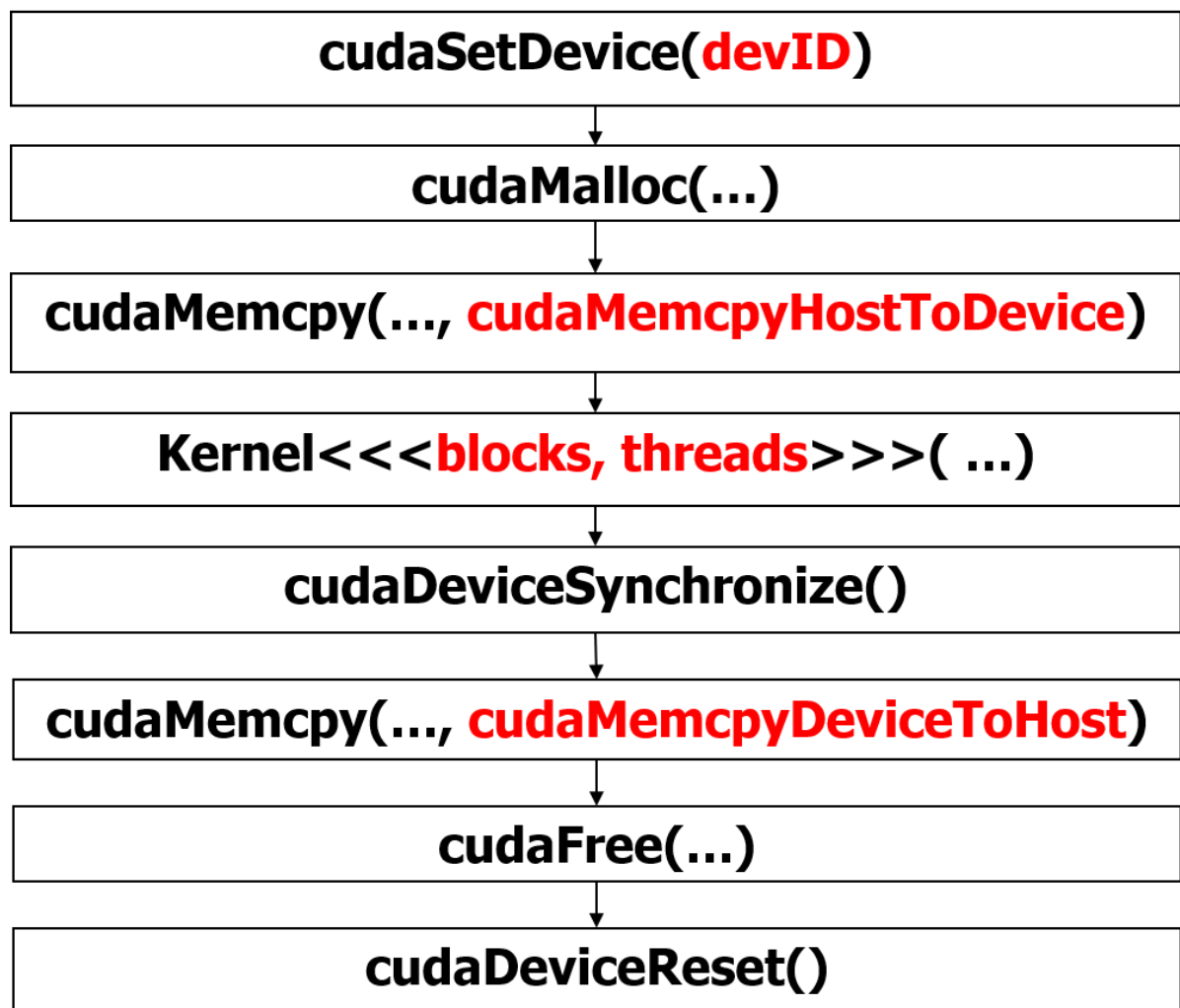
int main()
{
    ...
    // kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;        // 使用的Block数目
    dim3 threadsPerBlock(N, N); // 注意声明dim3类型，每个block采用N*N个线程
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

- 二维CUDA多个Block模板代码 ☆☆☆☆☆

```
// kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y; // 采用blockIdx * blockDim +
threadIdx 的方式赋值变量
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // kernel invocation
    dim3 threadsPerBlock(16, 16); // 数据类型都采用dim3，每个Block使用后16*16个
线程
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y); // 计算每个维度
需要多少个Block，返回Block数目
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

- Host Call Device



## 一些问题

**Q：影响处理器性能（CPI等）的因素有哪些？**

A：指令集设计，流水线深度，缓存设计，系统结构，机器周期

**Q：最小ROM空间与RAM空间如何计算？**

A：ROM = Code + RO + RW；RAM = RW + ZI；

**Q：如果调试过程中程序工作正常，但写入ROM后上电程序不能正确执行，可能的原因是什么？**

A1：电源有问题；复位电路有问题（缺少reset复位程序）；BOOT启动有问题（启动模式选择错误）；复位以后，中断向量表不在程序起始位0x00。

A2：复位时中断向量表是空的，0x00000000没有中断向量表；写了初始化程序，初始化程序错误；链接顺序出错，这是导致没有中断向量表的原因之一；模式选择错误，启动模式选择错误。

**Q：如果实现上电后能够自动执行，需要哪些条件？**

A：①Keil环境中Linker的target地址配置选项应该为：RO Base = 0x00000000 RW Base = 0x30000000

②Output中勾选“Create HEX file”

③DRAM初始化：Initialization by code Debug Init：需要在代码中加入初始化，上电后才能正常访问RAM map 0x48000000, 0x60000000 read write；

④硬件无问题（电源，复位电路，BOOT引导程序等）

**Q：能否将中断向量表中的跳转指令“B”改成“BL”？请说明理由。**

A：不能。BL和B两个指令的区别在于BL指令将当前的PC保存到LR。对于软中断来说，触发中断后。SP和LR切换到相应的模式下的SP和LR，此时LR中保存的是触发中断前的PC，如果中断向量表中的B改成BL，那么LR寄存器就会被修改，中断服务程序执行后就无法返回到触发中断的程序。

**Q：影响Cache命中率的因素有什么？**

A：Cache 大小，替换策略（直接相连等），程序的结构和数据的访问方式

**Q：某系统上电后不能自启动，可能的原因是？**

A：没有装在Flash里；缺少Reset向量；缺少启动程序；板子坏了。

## 常见报错信息解析

---

- “Error: A163E: Unknown opcode Mov, expecting opcode or Macro”
  - 因为 arm 汇编器对标志符的大小写敏感，因此书写标志符及指令时，大小写要一致。在arm 汇编程序中，指令、寄存器名可以全部为大写，也可以全部为小写，但是不能大小写混合使用
  - 编译阶段 compile，编译错误
- “Error: L6406E: No space in execution regions with. ANY selector matching lab1.o(.text)”
  - 提示空间不够
  - 链接阶段错误
- “Error: L16281E: undefined symbol main (referred from \_rtentry2.o)”
  - 错误提示为“main”这个符号未定义
  - 链接阶段错误
- “Error: #20: identifier “i” is undefined”
  - 没有定义标识符“i”
  - 编译阶段错误
- “Error: Target DLL has been canceled. Debugger aborted”
  - 目标 DLL（动态链接库） 已被取消，调试器终止
  - 下载调试阶段错误