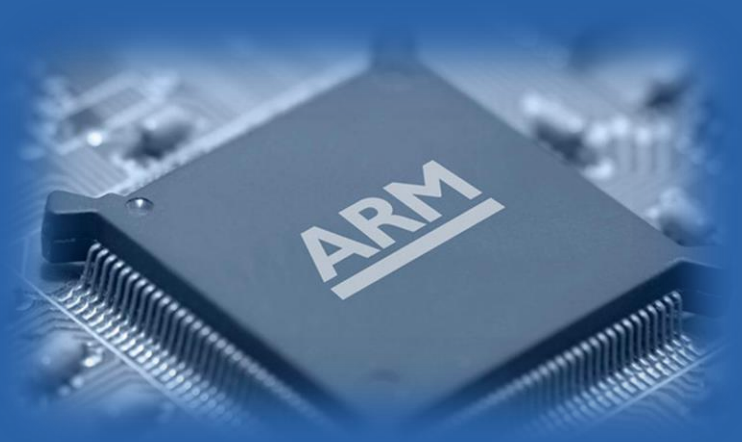
仪器科学与光电工程学院
School of Instrumentation Science and Opto-electronics Engineering

微机原理与接口技术

第九讲 ARM汇编程序设计




```
bl   __isoc99_scanf
ldr r2, [sp, #12]
ldr r3, [sp, #8]
cmp r2, r3
bgt .L7
ldrlt   r1, [sp, #4]
ldrge   r1, [sp, #4]
sublt   r1, r1, #1
strlt   r1, [sp, #4]
```

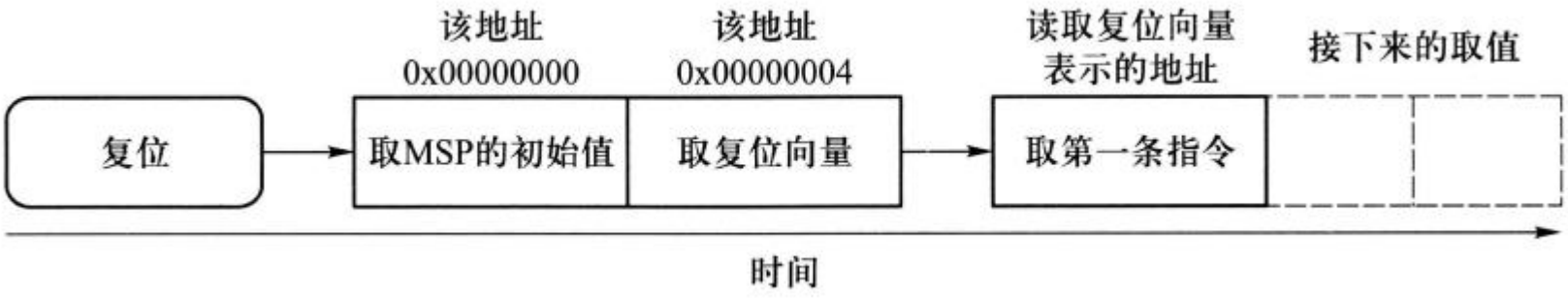
4.2 复位启动和调试

* 复位过程
* 启动模式
* 存储器重映射

4.2 复位启动和调试

复位过程:

Cortex-M4 处理器在复位后以及开始执行程序前，会固定地从地址 0x00000000 处读出第一个字，该字是主栈指针的初始值，从地址 0x00000004 处读出第二个字，该字是复位处理代码起始地址的复位向量。处理器读出这两个字后，分别赋给MSP和程序计数器PC，然后以此来取第一条指令执行。



4.2 复位启动和调试

启动模式选择:

由用户通过硬件配置方式或软件配置方式选择具体映射哪个地址空间到地址**0x00000000** 处。例如：当选择主**flash** 时，就是把主**flash** 的地址**0x08000000** 映射到地址**0x00000000** 处。

地址空间	地址范围	大小
主 flash	0x08000000~0x080FFFFF	1 MB
系统存储器	0x1FFF0000~0x1FFF77FF	30 KB
SRAM1	0x20000000~0x2001BFFF	112 KB

4.2 复位启动和调试

通过硬件配置选择启动模式： STM32F4xx 处理器提供了通过BOOT1和BOOT0 硬件引脚选择启动模式的方式。

BOOT1	BOOT0	启动模式
x	0	主 flash
0	1	系统存储器
1	1	SRAM1

4.2 复位启动和调试

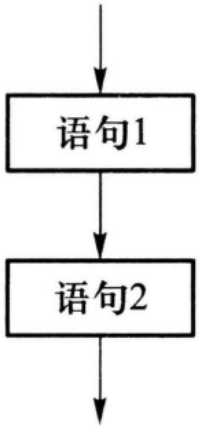
通过软件配置选择存储器重映射：STM32F4xx 处理器有一个SYSCFG存储器重映射寄存器(SYSCFG_MEMRMP)，此寄存器的最低2位bits<l: 0> MEM_MODE 用于对存储器重映射进行配置，由软件置1和0

MEM_MODE		存储器映射选择
bit<1>	bit<0>	
0	0	把主 flash 映射到地址 0x00000000
0	1	把系统存储器映射到地址 0x00000000
1	0	把 FSMC Bank1（NOR/PSRAM 1 和 2）地址 0x60000000 映射到地址 0x00000000
1	1	把 SRAM1 映射到地址 0x00000000

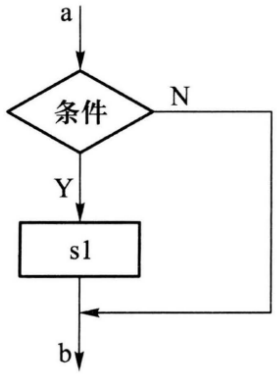
注：可变静态存储控制器（flexible static memory controller，FSMC）用于扩展存储器。

4.3 ARM汇编语言结构化程序设计方法

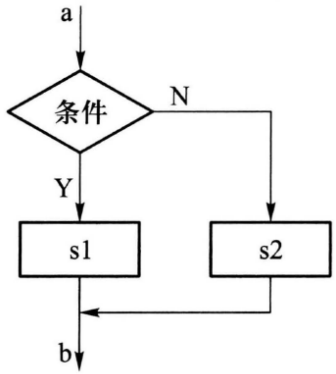
顺序结构设计



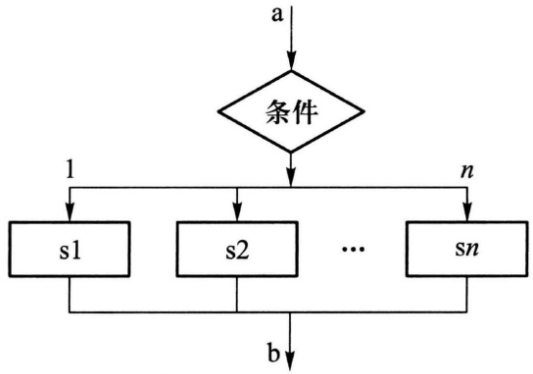
选择结构设计



(a)



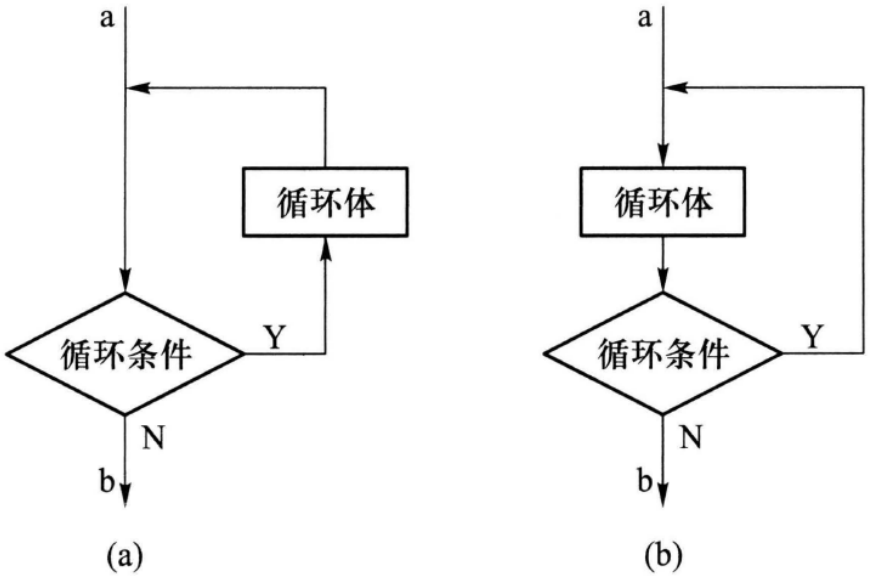
(b)



(c)

4.3 ARM汇编语言结构化程序设计方法

循环结构设计



4.3 ARM汇编语言结构化程序设计方法

子程序结构设计

1、子程序的调用一般是通过BL 指令来实现的，使用方式为：

BL funcname （**funcname** 是调用的子程序的名称）

2、子程序返回

```
BX LR
或          MOV PC, LR
或          STMFD SP!, {R0-R7, LR}    ;把要保护的寄存器内容压入堆栈
LDMFD SP!, {R0-R7, PC}          ;恢复寄存器内容,原 LR 值传送到 PC
```

4.3 ARM汇编语言结构化程序设计方法

汇编语言程序中使用子程序，需要注意两个问题：

1、现场保护， 2、参数传递

现场保护：一般是通过批量存储指令STMFD 和批量加载指令LDMFD完成。在进入子程序后调用STMFD 把要保护的寄存器内容压入堆栈，在子程序返回时调用LDMFD 恢复寄存器内容。

参数传递：可以通过寄存器传递，也可以用堆栈传递。用寄存器传递参数是最简单的方法：可以向子程序传入参数（入口参数），也可以将子程序运行的结果通过寄存器返回（出口参数）。

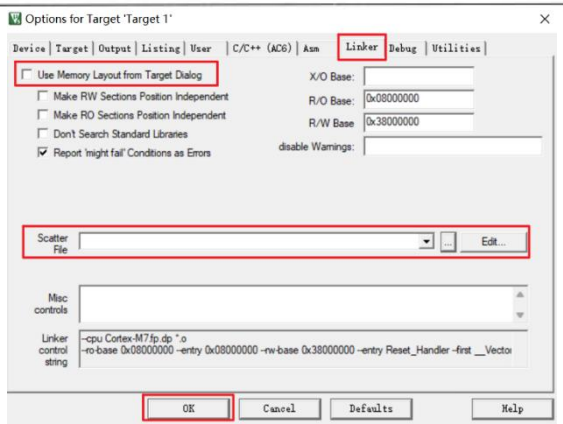
4.3 ARM汇编语言结构化程序设计方法

参数传递：入口参数为R0，出口参数为R1。

```
22                      ;初始化寄存器
23                      LDR    R0,    =N          ;初始化循环计数值
24                      BL     funcadd
25                      ;此时 R1 中是计算结果
26  deadloop
27                      B      deadloop        ;无限循环
28  ;子程序:入口参数 R0,出口参数 R1
29  funcadd
30                      MOV    R1,    #0          ;初始化计算结果
31                      ;将数值 N,N-1,...,2,1 相加,计算结果在 R1 中
32  loop
33                      ADD    R1,    R0    ;R1 = R1 + R0
34                      SUBS   R0,    #1          ;减小 R0,更新标志位('S' 后缀)
35                      BNE    loop        ;若上一条 SUBS 指令计算结果非 0,则跳到 loop
36                      BX     LR          ;子程序返回
37                      NOP
38                      END
```

可执行映像文件的构成及各个段在存储器中的位置（以4-6为例）

```
1  ;asm4-6.s
2  ;栈配置
3  Stack_Size    EQU      0x00000400        ;定义栈空间大小
4                AREA     MyStack,  NOINIT,  READWRITE,  ALIGN=3          ;声明栈段
5  Stack_Mem     SPACE    Stack_Size        ;分配内存空间
6  __initial_sp
7
8  ;异常/中断向量表(复位后,向量表位于地址0处)
9                AREA     Reset,  DATA,  READONLY   ;声明Reset数据段
10 __Vectors     DCD      __initial_sp              ;栈顶地址(MSP初始值)
11               DCD      Reset_Handler             ;"复位"异常处理代码的起始地址
12
13               THUMB          ;表示接下来的代码为THUMB指令集
14               PRESERVE8      ;表示接下来的代码保持8字节栈对齐
15               AREA     Init,  CODE,  READONLY              ;声明代码段
16       IMPORT |Image$$ER_IROM1$$RO$$Base|  ;RO段起始地址
17       IMPORT |Image$$ER_IROM1$$RO$$Limit|  ;RO段末地址后面的地址,即RW段起始地址
18       IMPORT |Image$$RW_IRAM1$$RW$$Base|  ;RW段在RAM里的运行区起始地址
19       IMPORT |Image$$RW_IRAM1$$RW$$Limit|  ;RW段在RAM里的运行区末地址后面的地址
20       IMPORT |Image$$RW_IRAM1$$ZI$$Base|  ;ZI段在RAM里的运行区起始地址
21       IMPORT |Image$$RW_IRAM1$$ZI$$Limit|  ;ZI段在RAM里的运行区末地址后面的地址
22                ENTRY
23 ;"复位"异常处理代码
24 Reset_Handler
25 ;RW段和ZI段初始化代码(第25~41行):完成RW段数据的拷贝和ZI段的清零
```




```
26               ;RW段数据的拷贝:从Image里的加载区拷贝到RAM里的运行区(第26~34行)
27               LDR    R0,    =|Image$$ER_IROM1$$RO$$Limit|
28               LDR    R1,    =|Image$$RW_IRAM1$$RW$$Base|
29               LDR    R3,    =|Image$$RW_IRAM1$$ZI$$Base|
30 Init_RW
31               CMP    R1,    R3
32               LDRCC R2,    [R0], #4
33               STRCC R2,    [R1], #4
34               BCC    Init_RW
35               ;ZI段清零:ZI段在RAM里的运行区全部清零(第35~41行)
36               LDR    R1,    =|Image$$RW_IRAM1$$ZI$$Limit|
37               MOV    R2,    #0
38 Init_ZI
39               CMP    R3,    R1
40               STRCC R2,    [R3], #4
41               BCC    Init_ZI
42
43 ;功能代码:将4位十六进制数转换为相应的ASCII码字符串
44               LDR    R0,    =Number      ;待转换的4位十六进制数的地址
45               LDR    R1,[R0]             ;待转换的4位十六进制数存入R1
46               LDR    R2,    =String      ;把存放字符串的首地址存入R2
47               STMFD  SP!,   {R1-R2}  ;把R1-R2压入堆栈,倒序压入(先R2后R1)
48               BL     Hex2String          ;调用子程序
49               LDMFD  SP!,   {R1-R2}  ;参数出栈(释放保存R1~R2的栈空间)
50               ;此时String中是转换完的字符串
51 deadloop
52               B      deadloop            ;无限循环
53 ;子程序Hex2String:4位十六进制数转换为字符串
54 Hex2String
55               STMFD  SP!,   {R0-R6, LR}  ;保护现场,R0~R6,LR入栈
56               ADD    R6,    SP,    #8*4     ;使SP指向参数(跳过栈顶的8个寄存器)
57               LDR    R0,    [R6], #4     ;取出待转换的4位十六进制数到R0,R6加4
58               LDR    R1,    [R6]  ;取出存放字符串4字节单元的首地址到R1
59               ADD    R1,    #3      ;使R1指向存放字符串4字节单元的最后一个单元
60               MOV    R2,    #4              ;循环计数寄存器R2赋初值为4
61 loop
62               MOV    R3,    R0     ;将待转换的4位十六进制数复制到R3
63               AND    R3,    R3,    #0x0F      ;取出低4位
```

可执行映像文件的构成及各个段在存储器中的位置（以4-6为例）

```
64                BL      Hex2ASCII          ;调用 1 位 16 进制数转换为 ASCII 子程序
65                STRB    R3,  [R1],  #-1    ;保存转换结果 ASCII
66                MOV     R0,  R0,  LSR #4   ;R0 右移 4 位,准备处理下一个 4 位
67                SUBS    R2,  R2,  #1       ;循环计数器减 1
68                BNE     loop               ;循环计数器不等于 0,继续循环
69                LDMFD   SP!,  {R0-R6,  LR}  ;恢复现场,R0~R6,LR 出栈
70                BX      LR       ;子程序返回
71  ;子程序 Hex2ASCII:把 1 位 16 进制数转换为 ASCII,入口参数 R3,出口参数 R3
72  Hex2ASCII
73                CMP     R3,  #9    ;判断 R3 是否已大于 9
74                BLE     Next       ;不大于 9 则跳转
75                ADD     R3,  R3,  #7    ;大于 9,预增 7
76  Next
77                ADD     R3,  R3,  #'0'   ;转换为 ASCII 码
78                BX      LR         ;子程序返回
79                NOP
80
81                AREA    MyData,  DATA,  READWRITE      ;声明 MyData 数据段
82  Number        DCD     0x8AF5          ;待转换的 4 位十六进制数
83  String        DCB     0, 0, 0, 0      ;保存字符串的空间
84                END
```

可执行映像文件的构成及各个段在存储器中的位置

(1) ARM 可执行映像文件的构成

ARM 映像文件就是可执行文件，也称为Image文件或ELF文件，包括bin 或hex 两种格式，可以直接烧入ROM(flash) 中。

映像文件一般由域(region) 组成，域最多由三个输出段(RO, RW, ZI) 组成，RO就是readonly，RW就是read/write，ZI就是zero。RO是程序中的指令和常量；RW是程序中的已初始化变量; ZI是程序中的未初始化的变量.

映像文件所处的区域，又分为加载域(load region) 和运行域(execution region)。加载域：是指程序烧入ROM中的状态；运行域：是指程序运行时的状态。

可执行映像文件的构成及各个段在存储器中的位置

(2) 各个段在存储器中的位置

加载域中输出段的地址位置，一般来说RO段后面紧跟着RW段，通常是地址连续的。

运行域中输出段的地址位置，一般来说RO段在flash地址空间，RW和ZI 段在RAM 地址空间， RO 段与RW/ZI 段地址不连续，但RW和ZI是连续的。

可执行映像文件的构成及各个段在存储器中的位置

==========

Memory Map of the image

Image Entry point : 0x08000009

Load Region LR_IROM1 (Base: 0x08000000, Size: 0x0000009c, Max: 0x00080000, ABSOLUTE)

Execution Region ER_IROM1 (Exec base: 0x08000000, Load base: 0x08000000, Size: 0x00000094, Max: 0x00080000)

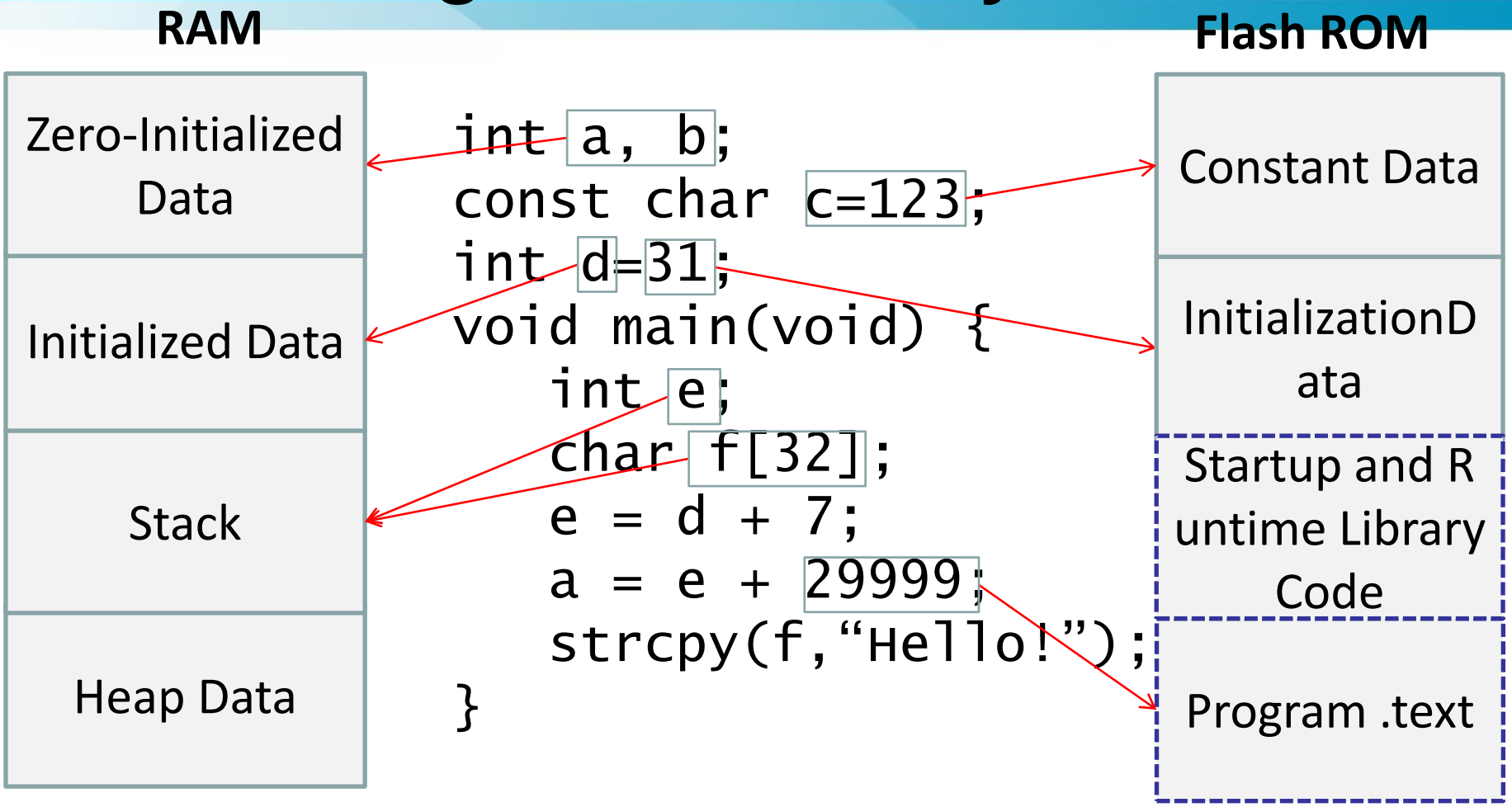
Exec Addr	Load Addr	Size	Type	Attr	Idx	E Section Name	Object
0x08000000	0x08000000	0x00000008	Data	RO	2	Reset	asm4-6.o
0x08000008	0x08000008	0x0000008c	Code	RO	3	* Init	asm4-6.o

Execution Region RW_IRAM1 (Exec base: 0x20000000, Load base: 0x08000094, Size: 0x00000408, Max: 0x00020000)

Exec Addr	Load Addr	Size	Type	Attr	Idx	E Section Name	Object
0x20000000	0x08000094	0x00000008	Data	RW	4	MyData	asm4-6.o
0x20000008	-	0x00000400	Zero	RW	1	MyStack	asm4-6.o

==========

Program Memory Use



调用其他源文件中的符号

在被调用符号所在源文件中使用**EXPORT** 指示符声明该符号可以被其他源文件调用，在调用者源文件中使用**IMPORT** 指示符声明该符号是在其他源文件中定义的，然后就可以调用该符号了.

```
8   ;异常 / 中断向量表(复位后,向量表位于地址 0 处)
9                AREA    Reset,   DATA,   READONLY        ;声明 Reset 数据段
10  __Vectors    DCD     __initial_sp              ;栈顶地址(MSP 初始值)
11               DCD     Reset_Handler             ;"复位" 异常处理代码的起始地址
12
13               THUMB            ;表示接下来的代码为 THUMB 指令集
14               PRESERVE8        ;表示接下来的代码保持 8 字节栈对齐
15               AREA    Init,    CODE,   READONLY          ;声明代码段
16               IMPORT  MyMain
17               ENTRY
18  ;"复位" 异常处理代码
19  Reset_Handler
20               LDR             R0, = MyMain      ;加载 MyMain 的地址
21               BX      R0                        ;调用 MyMain
22               END
```

文件1：start.s

```
12               EXPORT    MyMain
13  MyMain       FUNCTION
14  ;RW 段和 ZI 段初始化代码(第 14~30 行):完成 RW 段数据的拷贝和 ZI 段的清零
15               ;RW 段数据的拷贝:从 Image 里的加载区拷贝到 RAM 里的运行区(第 15~23 行
16               LDR     R0,     =|Image$$ER_IROM1$$RO$$Limit|
17               LDR     R1,     =|Image$$RW_IRAM1$$RW$$Base|
18               LDR     R3,     =|Image$$RW_IRAM1$$ZI$$Base|
19  Init_RW
```

文件2：asm4-7.s

文件1中函数调用文件2中函数

4.4 C语言程序与汇编程序的相互调用

AAPCS 标准

(1) 父函数与子函数间的入口参数依次通过R0-R3 这4个寄存器传递。父函数在调用子函数前先将参数存入到R0-R3 中，若只有一个参数，则使用R0传递， 2个参数则使用R0和R1传递，依次类推，当超过4个参数时，其他参数通过堆栈传递。当子函数运行时，根据自身参数个数自动从R0-R3或者栈中读取参数。

(2) 子函数通过R0寄存器将返回值传递给父函数。子函数返回时，将返回值存入R0；当返回到父函数时，父函数读取R0获得返回值。

4.4 C语言程序与汇编程序的相互调用

AAPCS 标准

(3) 发生函数调用时，R0-R3是传递参数的寄存器，即使是父函数没有参数需要传递，子函数也可以任意更改R0-R3 寄存器，无需考虑会破坏它们在父函数中保存的数值，返回父函数前无需恢复其值。AAPCS规定，发生函数调用前，由父函数将R0-R3中有用的数据压栈，然后才能调用子函数，以防止父函数R0-R3中的有用数据被子函数破坏。

4.4 C语言程序与汇编程序的相互调用

在汇编程序中调用C函数

C函数

```
int my_add_c(int x1, int x2, int x3, int x4, int x5, int x6)
{
    return (x1 + x2 + x3 + x4 + x5 + x6);
}
```

x1=R0，x2=R1，x3=R2，x4=R3；

x5 和x6 通过堆栈传递，入栈顺序是按x6->x5 的顺序入栈的，按自右至左的顺序入栈

汇编程序

```
STMFD   SP!,   {R0-R3,   LR}        ;调用者负责压栈保存 R0~R3,LR 旧值
STMFD   SP!,   {R4-R5}      ;本函数用到了 R4~R5,所以负责压栈保存 R4~R5 旧值
MOV     R0,    #0x11        ;第 1 个参数(x1)
MOV     R1,    #0x22        ;第 2 个参数(x2)
MOV     R2,    #0x33        ;第 3 个参数(x3)
MOV     R3,    #0x44        ;第 4 个参数(x4)
MOV     R4,    #0x55        ;第 5 个参数(x5)
MOV     R5,    #0x66        ;第 6 个参数(x6)
STMFD   SP!,   {R4-R5}      ;注意入栈顺序:R5(x6)先入栈,R4(x5)后入栈
IMPORT  my_add_c            ;引入其他源文件中的函数名 my_add_c
BL      my_add_c            ;调用函数 my_add_c,返回结果位于 R0 中
LDR     R5,    = Result     ;取得变量 Result 的地址
STR     R0,    [R5]         ;把 my_add_c 函数返回结果存入变量 Result
ADD     SP,    SP,    #4    ;清除栈中参数 x5,本语句执行完后 SP 指向栈中参数 x6
ADD     SP,    SP,    #4    ;清除栈中参数 x6,本语句执行完后 SP 指向栈中 R4 旧值
LDMFD   SP!,   {R4-R5}      ;弹栈恢复 R4~R5 旧值
LDMFD   SP!,   {R0-R3, LR}  ;调用者负责弹栈恢复 R0~R3, LR 旧值
```

4.4 C语言程序与汇编程序的相互调用

在C 语言程序中调用汇编函数

汇编程序

```
EXPORT my_add_asm
my_add_asm  FUNCTION
            ADD    R0, R0, R1          ; R0 = x1 + x2
            ADD    R0, R0, R2          ; R0 = x1 + x2 + x3
            ADD    R0, R0, R3          ; R0 = x1 + x2 + x3 + x4
            LDR    R1,  [SP]           ; 从 栈顶 位置取出 x5 到 R1
            ADD    R0, R0, R1          ; R0 = x1 + x2 + x3 + x4 + x5
            LDR    R2,  [SP,   #4]     ; 从 栈顶 +4 位置取出 x6 到 R2
            ADD    R0, R0, R2          ; R0 = x1 + x2 + x3 + x4 + x5 + x6
            BX     LR                  ; 返回结果位于 R0 中
ENDFUNC
```

C函数

```
extern int my_add_asm( int x1,  int x2,  int x3,  int x4,  int x5,  int x6 );
…
int y = 0;
y = my_add_asm( 1, 2, 3, 4, 5, 6 );
…
```

4.4 C语言程序与汇编程序的相互调用

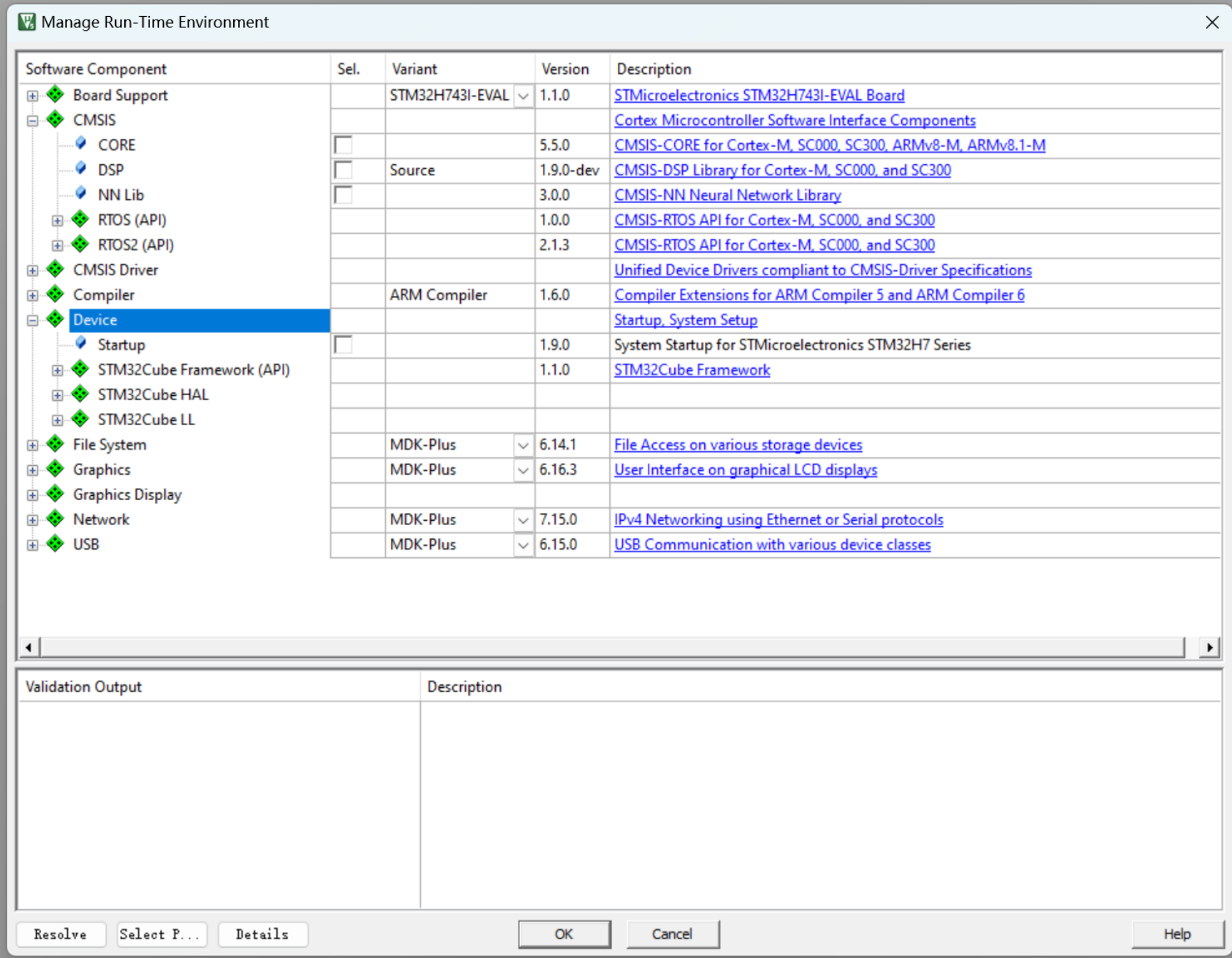
4.4.6内部函数

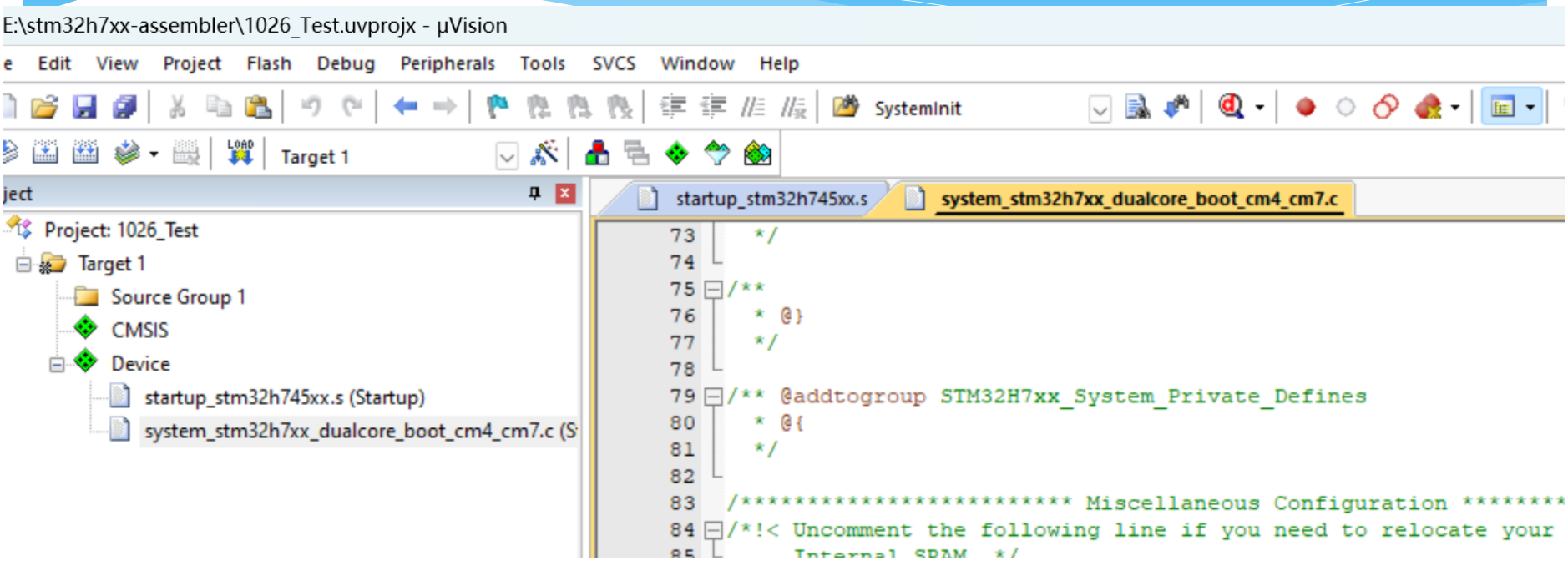
有些情况下,需要使用无法用普通 C 语言程序实现的特殊功能操作。除了使用汇编函数(子程序)、内联汇编、嵌入汇编来进行汇编语言编程外,实现特殊功能操作的另一个方法是使用内部函数(intrinsic functions)。

内部函数有两种:

(1) CMSIS-Core 内部函数;

(2) 编译器相关的内部函数。

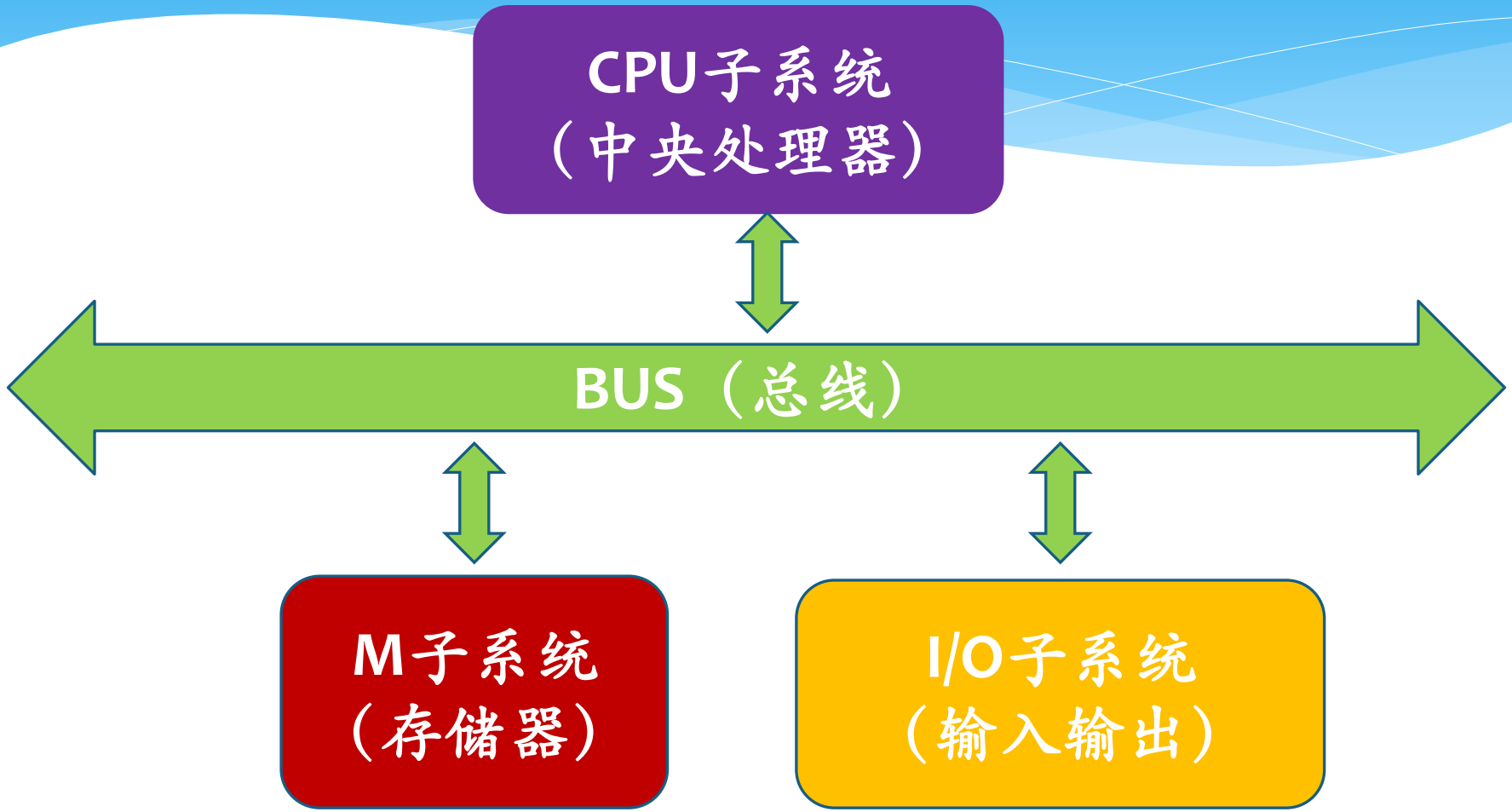

Manage Run-Time Environment
Software Component
Sel.
Variant
Version
Description
Board Support
STM32H743I-EVAL
1.1.0
STMicroelectronics STM32H743I-EVAL Board
CMSIS
Cortex Microcontroller Software Interface Components
CORE
5.5.0
CMSIS-CORE for Cortex-M, SC000, SC300, ARMv8-M, ARMv8.1-M
DSP
Source
1.9.0-dev
CMSIS-DSP Library for Cortex-M, SC000, and SC300
NN Lib
3.0.0
CMSIS-NN Neural Network Library
RTOS (API)
1.0.0
CMSIS-RTOS API for Cortex-M, SC000, and SC300
RTOS2 (API)
2.1.3
CMSIS-RTOS API for Cortex-M, SC000, and SC300
CMSIS Driver
Unified Device Drivers compliant to CMSIS-Driver Specifications
Compiler
ARM Compiler
1.6.0
Compiler Extensions for ARM Compiler 5 and ARM Compiler 6
Device
Startup, System Setup
Startup
1.9.0
System Startup for STMicroelectronics STM32H7 Series
STM32Cube Framework (API)
1.1.0
STM32Cube Framework
STM32Cube HAL
STM32Cube LL
File System
MDK-Plus
6.14.1
File Access on various storage devices
Graphics
MDK-Plus
6.16.3
User Interface on graphical LCD displays
Graphics Display
Network
MDK-Plus
7.15.0
IPv4 Networking using Ethernet or Serial protocols
USB
MDK-Plus
6.15.0
USB Communication with various device classes
Validation Output
Description
Resolve
Select P...
Details
OK
Cancel
Help

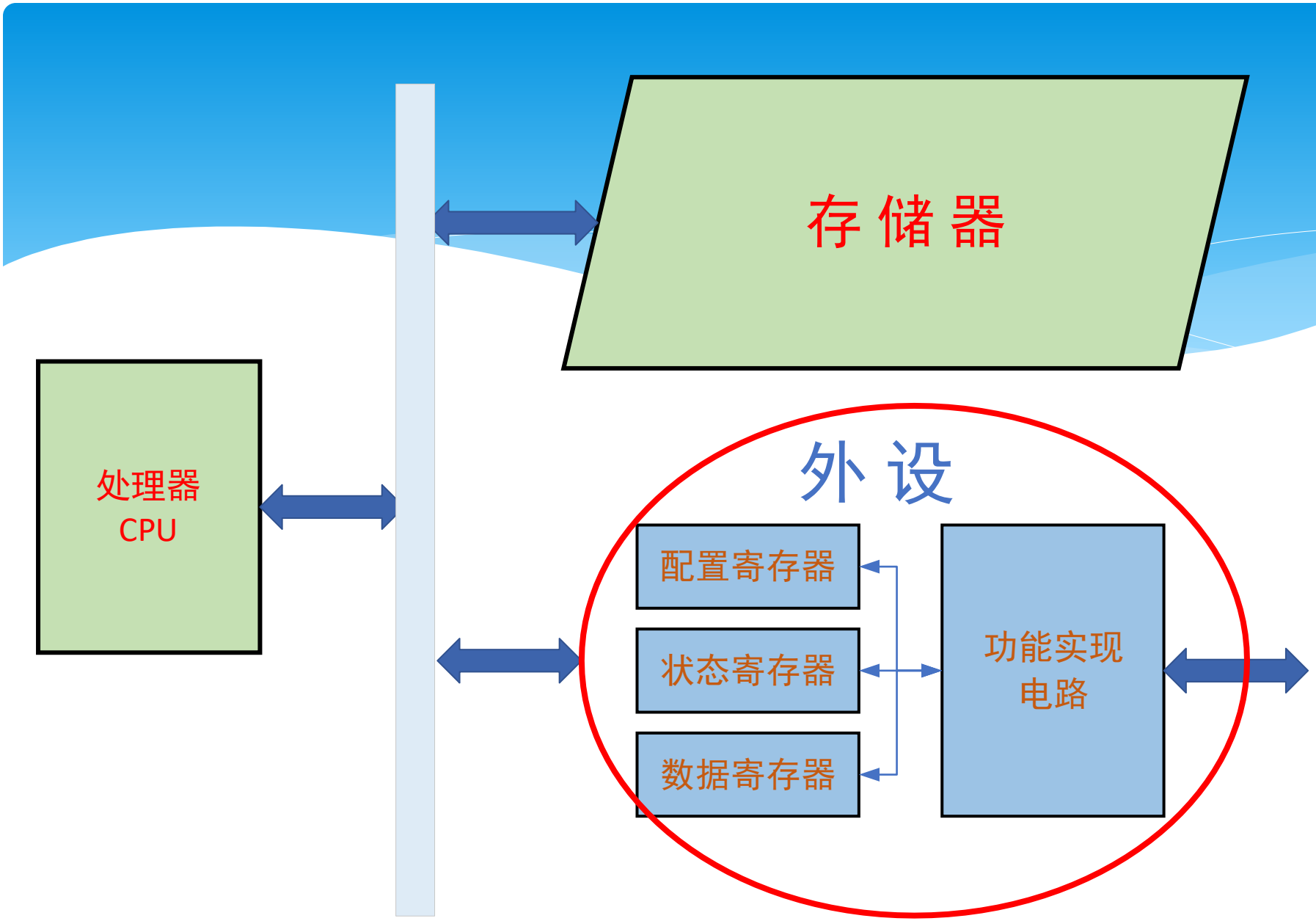

E:\stm32h7xx-assembler\1026_Test.uvprojx - µVision
e Edit View Project Flash Debug Peripherals Tools SVCS Window Help
SystemInit
Target 1
ject
Project: 1026_Test
Target 1
Source Group 1
CMSIS
Device
startup_stm32h745xx.s (Startup)
system_stm32h7xx_dualcore_boot_cm4_cm7.c (S
startup_stm32h745xx.s
system_stm32h7xx_dualcore_boot_cm4_cm7.c
73 */
74
75 /**
76 * @}
77 */
78
79 /** @addtogroup STM32H7xx_System_Private_Defines
80 * @{
81 */
82
83 /************************* Miscellaneous Configuration ********
84 /*!< Uncommente the following line if you need to relocate your

微机原理与接口技术

第8章 I/O接口技术

现代冯·诺依曼计算机架构




存 储 器
处理器
CPU
外 设
配置寄存器
状态寄存器
数据寄存器
功能实现
电路

第一节 I/O接口概述

输入输出、I/O接口电路

I/O接口的功能

① 数据的寄存和缓冲

② 设备的选择

③ 控制及状态信息传送

④ 信号转换

⑤ 时序控制

第一节 I/O接口概述

CPU与I/O设备之间的信息通常包括：

数据信息，状态信息和控制信息

1. **数据信息**：数字量、模拟量、开关量

2. **状态信息**：反映外设的工作状态；

3. **控制信息**：一般是CPU通过接口电路传送给外部设备的，主要用来控制外部设备的动作；

I/O设备的寄存器--》端口

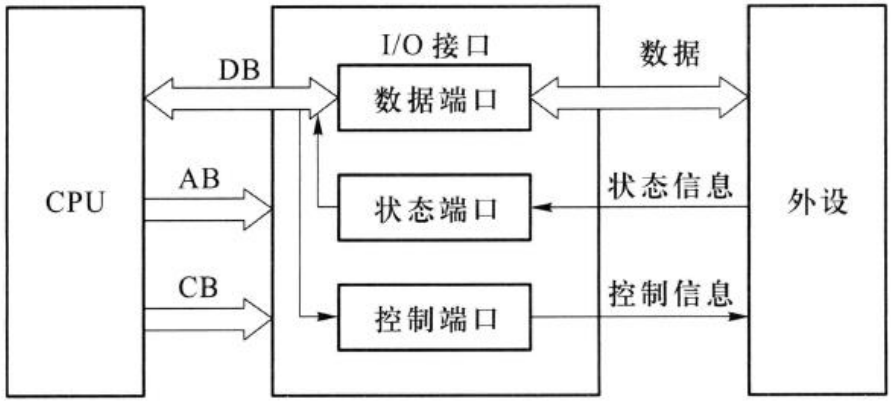


图 8-1-1　通常的 I/O 接口示意图

第一节 I/O接口概述

I/O端口的编址方式

1. 存储器统一编址方式（存储器映像）

在存储空间中划出一部分给外设端口，对端口当作存储器单元一样进行访问，不设置专门的I/O指令；

优点：对存储器操作的指令可以全部用于对端口操作；端口有较大的编址空间；

缺点：使存储器容量变小等；

第一节 I/O接口概述

2. 独立编址方式（I/O映像）

端口地址单独编址，而不和存储空间合在一起；
设置专门的I/O指令来访问端口；

优点：不占用存储空间，端口操作速度快；

缺点：一般对端口只能进行传送操作；

第一节 并行口与串行口

I/O接口的“本质”是电路，这些电路包括：

- I/O端口寄存器
- 地址译码电路
- 传送方式控制电路
- 并-串/串-并变换等转换电路

接口部件，使用本身的寄存器来实现各种接口，这些寄存器就是I/O端口。

接口部件的寄存器---CPU当作RAM单元访问

第一节 并行口与串行口

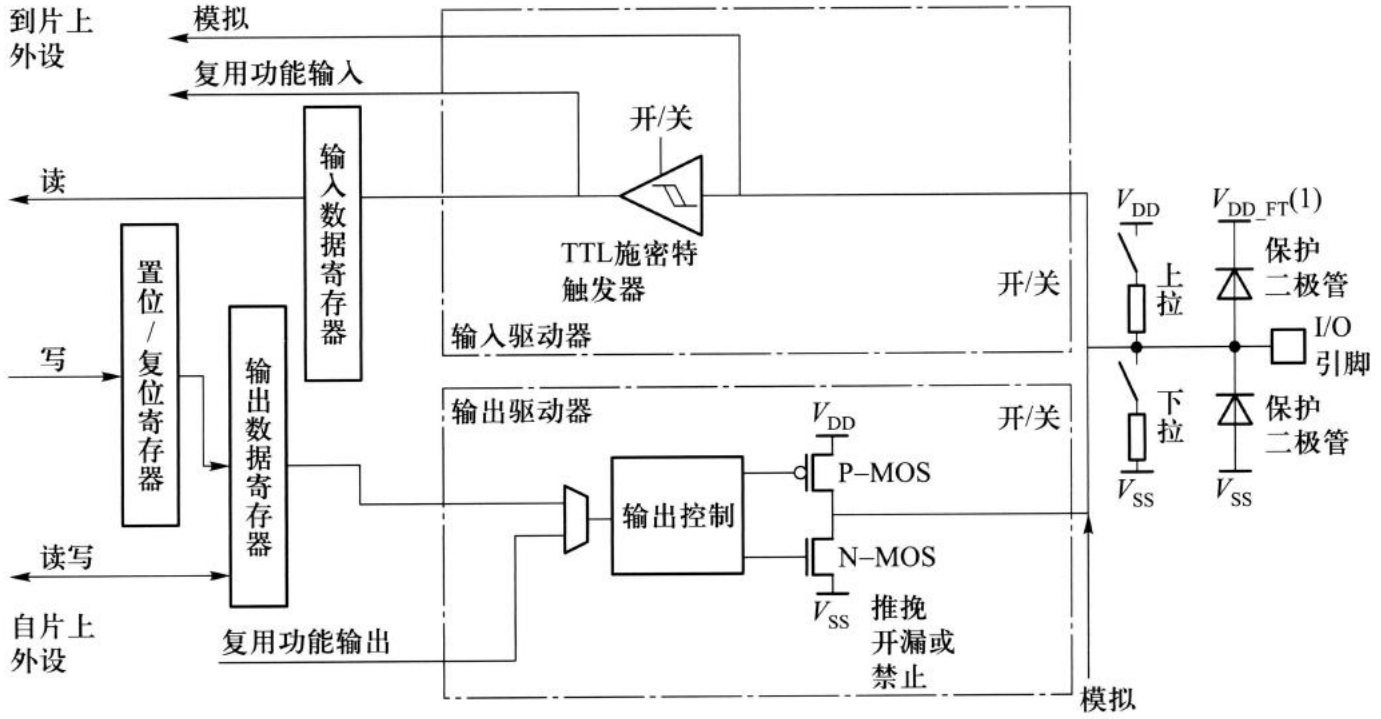
* 并行I/O接口：以并行方式和CPU传送I/O接口，以并行方式和外设交换数据。

* 串行口：数据和控制信息是一位接一位串行按顺序的传送的。因为计算机系统内部的数据是并行传送的，所以要串行传送数据的话，需要进行并---串/串---并变换。

通用I/O-GPIO

GPIO的基本特性

GPIO的基本结构



Input configuration

When the I/O port is programmed as input:

- The output buffer is disabled
- The Schmitt trigger input is activated
- The pull-up and pull-down resistors are activated depending on the value in the GPIOx_PUPDR register
- The data present on the I/O pin are sampled into the input data register every AHB clock cycle
- A read access to the input data register provides the I/O state

Figure 78 shows the input configuration of the I/O port bit.

Figure 78. Input floating/pull up/pull down configurations

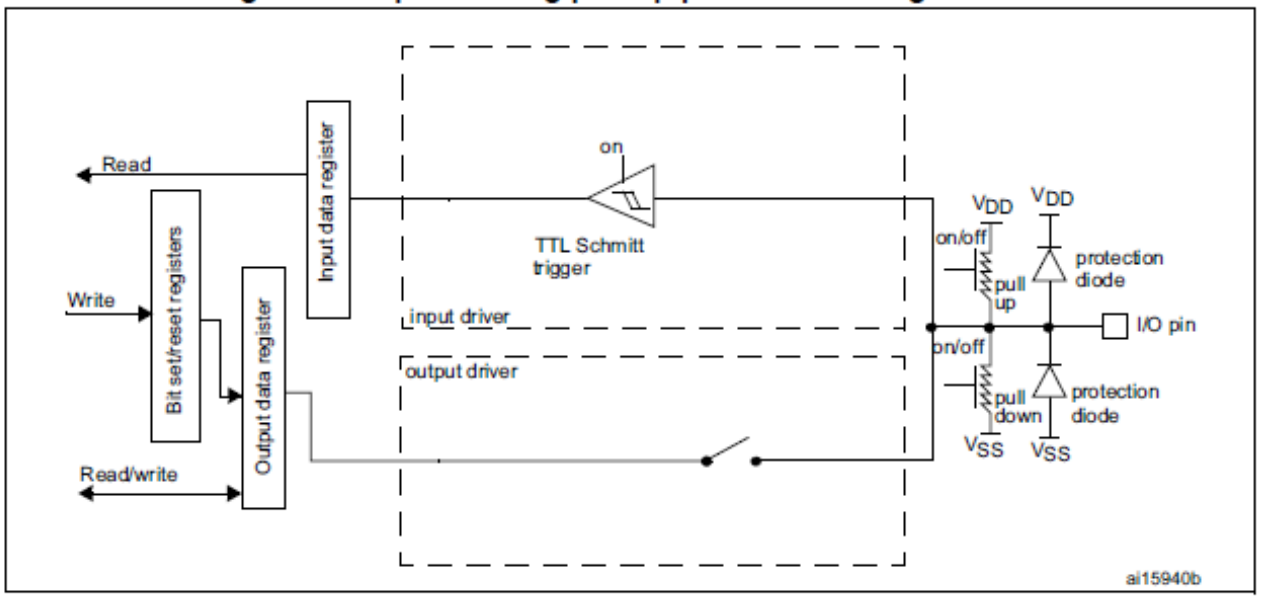


Figure 79. Output configuration

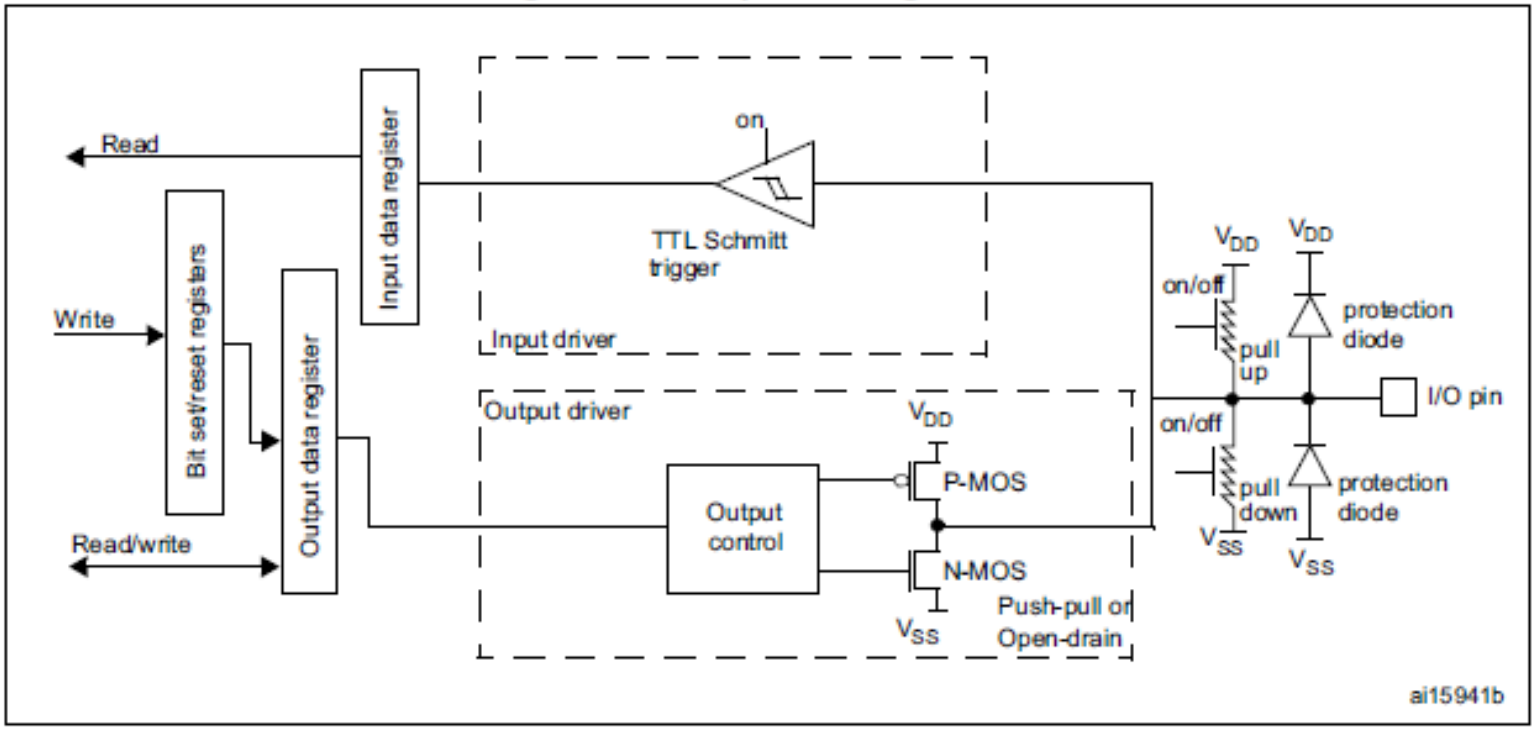


Figure 80. Alternate function configuration

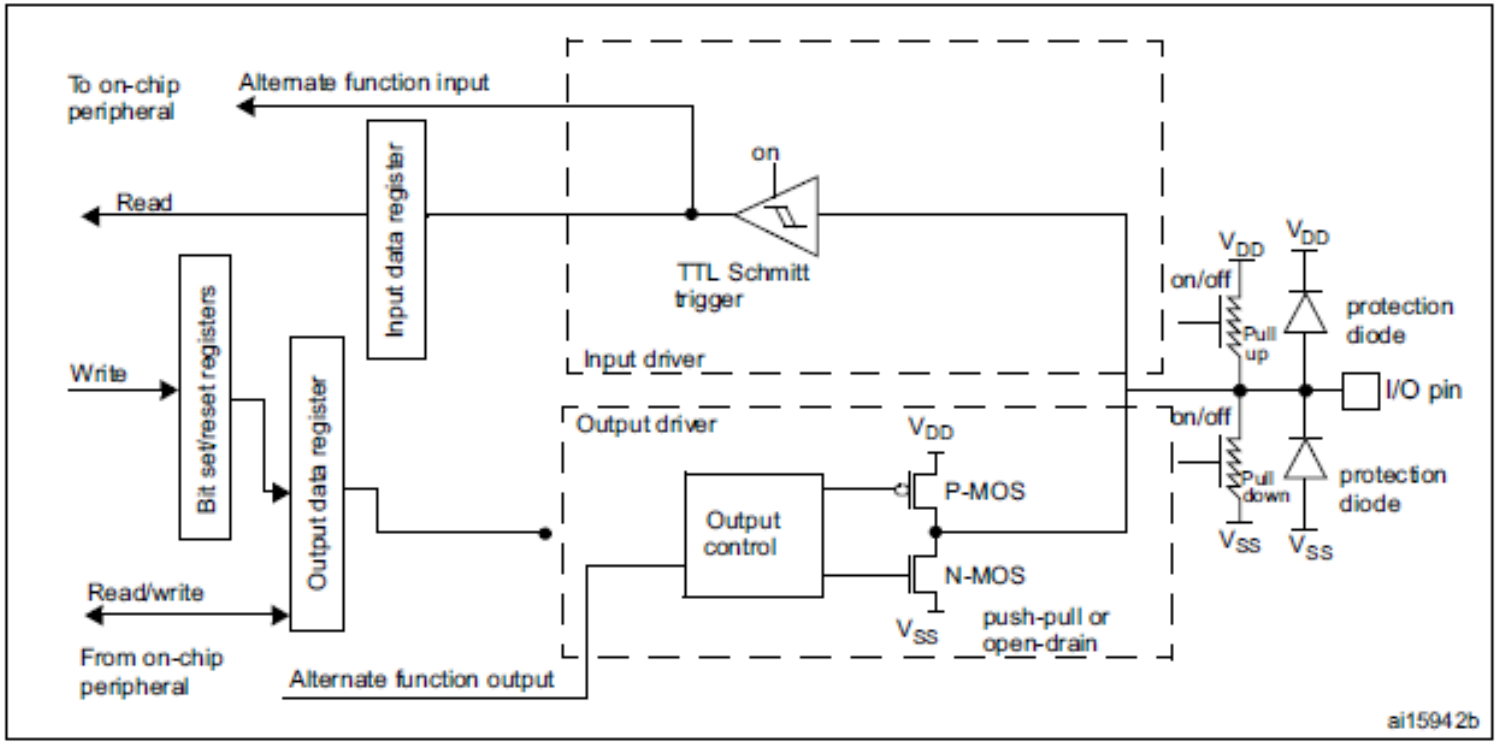
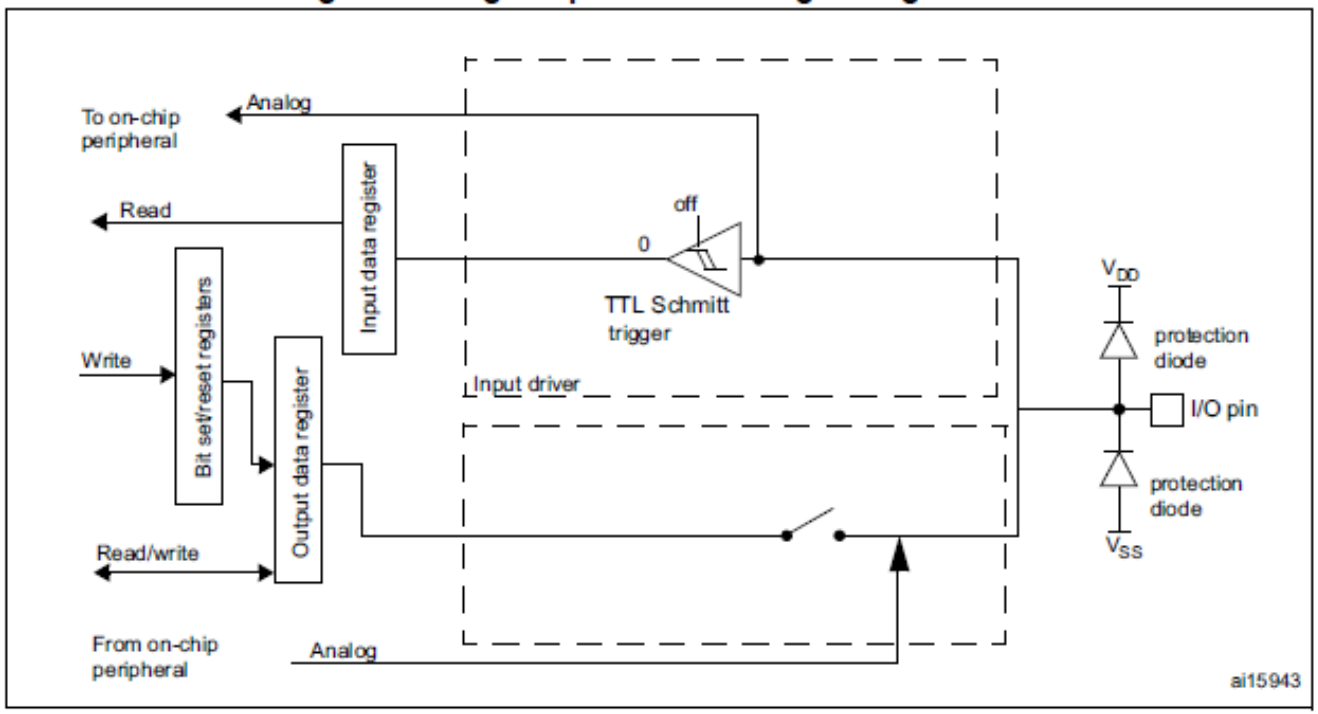


Figure 81. High impedance-analog configuration



通用I/O-GPIO

STM32H7xxd的GPIO寄存器

- 4个32-bit 配置寄存器(MODER,OTYPER, OSPEEDR PUPDR),
- 2个32-bit 数据寄存器(IDR,ODR)
- 1个32-bit 位操作寄存器 (BSRR).
- 1个32-bit 锁存寄存器 (LCKR)
- 2个32-bit 复用功能选择 (AFRH , AFRL).

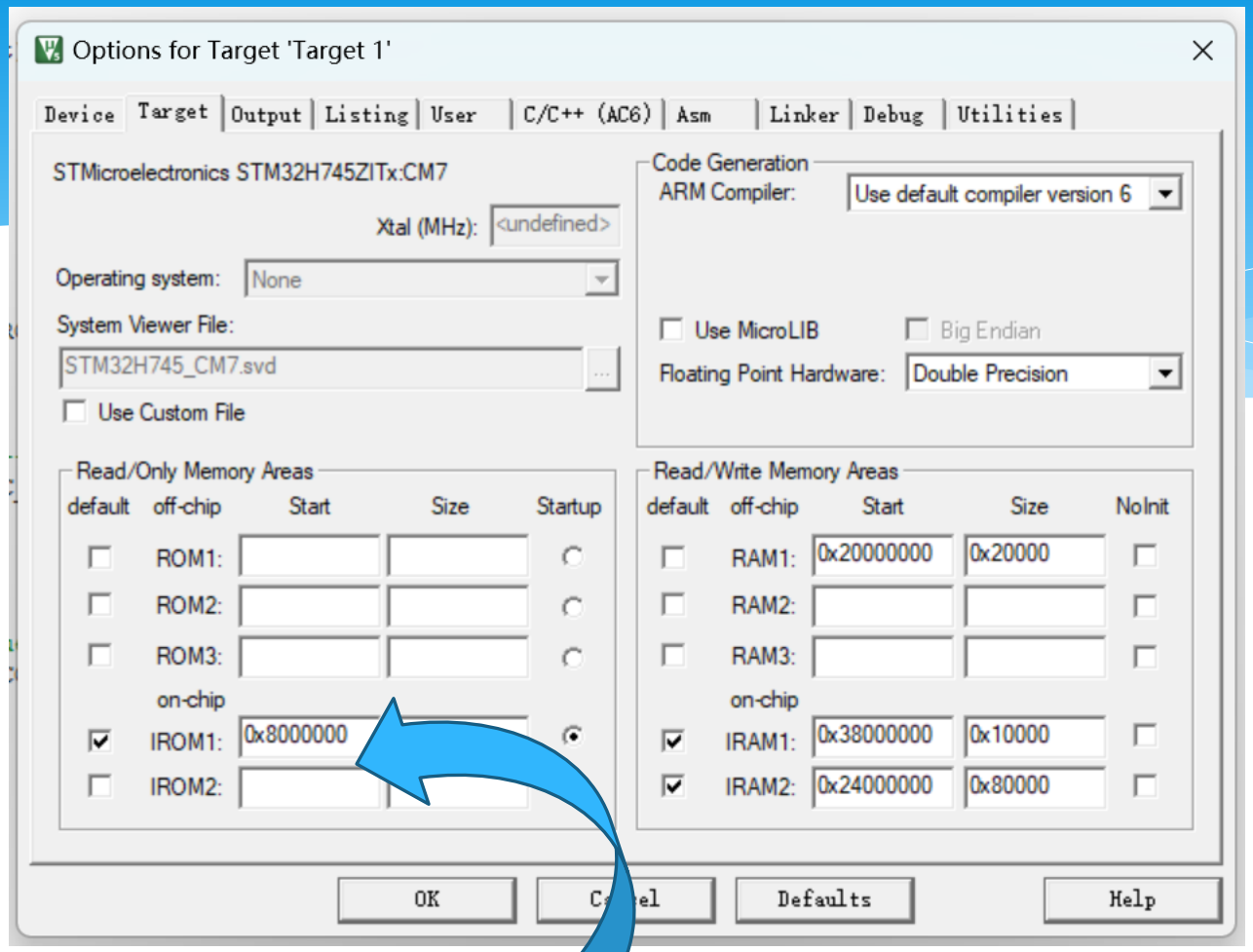
Table 96. Port bit configuration table[1]

MODE(i) [1:0]	OTYPER(i)	OSPEED(i) [1:0]		PUPD(i) [1:0]		I/O configuration	
01	0	SPEED [1:0]		0	0	GP output	PP
	0			0	1	GP output	PP + PU
	0			1	0	GP output	PP + PD
	0			1	1	Reserved	
	1			0	0	GP output	OD
	1			0	1	GP output	OD + PU
	1			1	0	GP output	OD + PD
	1			1	1	Reserved (GP output OD)	
10	0	SPEED [1:0]		0	0	AF	PP
	0			0	1	AF	PP + PU
	0			1	0	AF	PP + PD
	0			1	1	Reserved	
	1			0	0	AF	OD
	1			0	1	AF	OD + PU
	1			1	0	AF	OD + PD
	1			1	1	Reserved	
00	x	x	x	0	0	Input	Floating
	x	x	x	0	1	Input	PU
	x	x	x	1	0	Input	PD
	x	x	x	1	1	Reserved (input floating)	
11	x	x	x	0	0	Input/output	Analog
	x	x	x	0	1	Reserved	
	x	x	x	1	0		
	x	x	x	1	1		

1. GP = general-purpose, PP = push-pull, PU = pull-up, PD = pull-down, OD = open-drain, AF = alternate function.

主要问题

* 移码的计算，正数的反码和补码，
* 采集转换
* 内存中的数存放形式，大小端问题




```
import  |Image$$ER_IROM1$$RO$$Base|
```