

主要内容

- ❖ 嵌入式软件开发关键技术
- ❖ “ARM”软件设计基础
- ❖ “ARM”软件调试与运行

嵌入式软件开发关键技术

- ❖ 嵌入式软件的启动代码
- ❖ 嵌入式实时操作系统
- ❖ 程序的链接定位
- ❖ 软件调试技术

启动代码说明

❖ 启动代码是用来初始化电路以及用来为高级语言写的软件做好运行前准备的一小段汇编语言，是任何处理器上电复位时的程序运行入口点

➤ 功能

初始化电路

为高级语言编写的软件运行做准备

➤ 特征

汇编语言

处理器上电复位的程序运行入口点

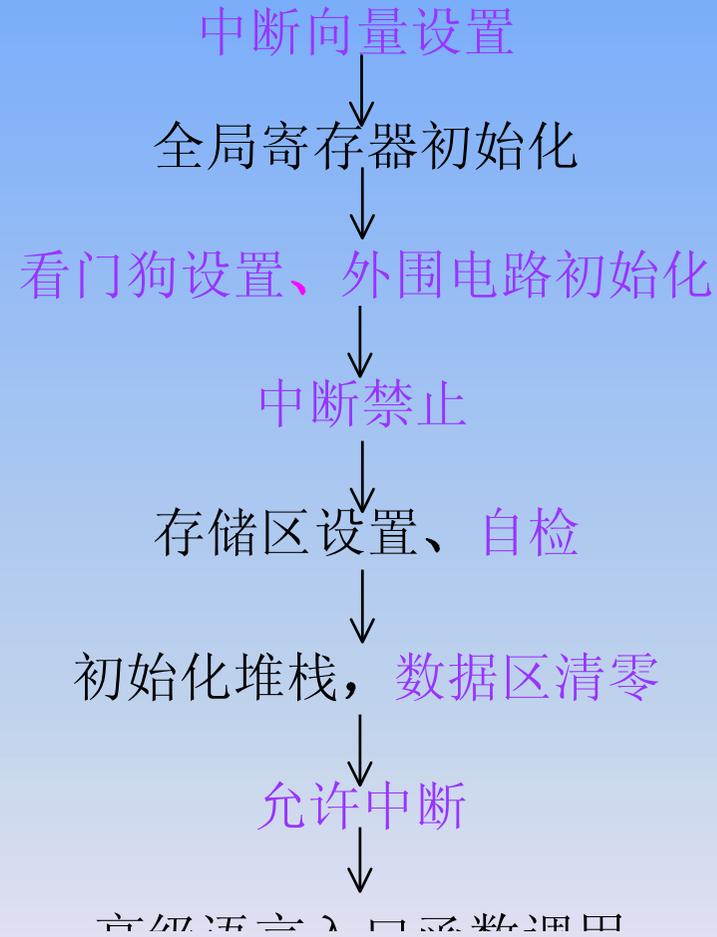
启动代码常见形式

- ❖ 实时操作系统的板基支持包——**BSP**
- ❖ 计算机主板的**BIOS**

启动代码最小流程



启动代码典型流程



程序的链接定位

- ❖ 链接定位是系统级软件开发过程中必不可少的一部分，嵌入式软件开发均属于系统级开发。
- ❖ 链接定位过程一般由链接器根据链接定位文件完成，比较简单的系统可以通过设置链接器开关选项取代链接定位文件。
- ❖ 链接定位的关键是链接定位文件的编写

常见链接程序段

以下程序段存在于各个目标文件中（***.obj *.o**），链接定位后按段的类别收集在一起，同时指定在存储区中的位置

- ❖ **text**：代码段，所有代码块部分
- ❖ **rodata**：已初始化的全局只读数据
- ❖ **data**：已初始化的全局数据
- ❖ **bss**：未初始化的全局变量

简单链接定位文件

```
SECTIONS
{
    . = 0x10000
    .text: {*(.text)}
    . = 0x8000000
    .data: {*(.data)}
    .bss: {*(.bss)}
}
```

典型链接定位文件

SECTIONS

```
{  
    . = 0x02000000;  
    .text : { *(.text) }  
    Image_RO_Limit = .;  
    Image_RW_Base = .;  
    .data : { *(.data) }  
    .rodata : { *(.rodata) }  
    .bss : { *(.bss) }  
    PROVIDE (__stack = .);  
    end = .;  
    _end = .;  
    .debug_info      0 : { *(.debug_info) }  
    .debug_line      0 : { *(.debug_line) }  
    .debug_abbrev    0 : { *(.debug_abbrev) }  
    .debug_frame     0 : { *(.debug_frame) }
```

调试技术

在应用程序的编辑、编译工作完成后，开发人员需要借助一些调试设备或调试模块，对应用程序进行调试，发现程序设计中的缺陷。常用的调试设备或调试模块有以下几种：

- ❖ 指令集模拟器
- ❖ 驻留监控软件
- ❖ JTAG仿真器
- ❖ 在线仿真器

驻留监控软件

驻留监控软件（**Resident Monitors**）是一段运行在目标板上的程序，集成开发环境中的调试模块，通过以太网口、并行端口或者串行端口等通讯端口与驻留监控软件进行交互，由调试模块发送命令通知驻留监控软件，控制程序的执行、读写存储器、读写寄存器、设置断点等。

驻留监控软件是一种比较低廉有效的调试方式，不需要任何其他的硬件调试和仿真设备。**ARM**公司的**Angel**就是该类软件，大部分嵌入式实时操作系统也是采用该类软件进行调试，不同的是在嵌入式实时操作系统中，驻留监控软件是作为操作系统的任务存在的。

驻留监控软件的不便之处在于它对硬件设备的要求比较高，一般在硬件稳定之后才能进行应用软件的开发，同时它占用目标板上的一部分资源，而且不能对程序的全速运行进行完全仿真，所以对一些要求严格的情况不是很适合。

JTAG仿真器

JTAG仿真器也称为**JTAG**调试器，是通过**ARM**芯片的**JTAG**边界扫描口进行调试的设备。

JTAG仿真器比较便宜，连接比较方便，通过现有的**JTAG**边界扫描口与 **ARM CPU** 核通信，属于完全非插入式(即不使用片上资源)调试，它无需目标存储器，不占用目标系统的任何端口，而这些是驻留监控软件所必需的。

另外，由于**JTAG**调试的目标程序是在目标板上执行，仿真更接近于目标硬件，因此，许多接口问题，如高频操作限制、**AC**和**DC**参数不匹配，电线长度的限制等被最小化了。

使用集成开发环境配合**JTAG**仿真器进行开发是目前采用最多的一种调试方式。

在线仿真器

在线仿真器使用仿真头，完全取代目标板上的**CPU**，可以完全仿真**ARM**芯片的行为，提供更加深入的调试功能。但这类仿真器为了能够全速仿真时钟速度高于**100MHz**的处理器，通常必须采用极其复杂的设计和工艺，因而其价格比较昂贵。

在线仿真器通常用在**ARM**的硬件开发中，在软件的开发中较少使用，其价格高昂，也是在线仿真器难以普及的因素。

实时操作系统(RTOS)

- ❖ RTOS选择的原则
- ❖ Embest IDE与RTOS的配合
- ❖ 常见的几种RTOS介绍

RTOS基础知识

RTOS特点:

- 基于优先级的任务调度
 保证优先任务得到优先执行
- 任务间的通信、互斥机制
 实现任务间同步和通讯
- 实时时钟管理
 保证任务在确定时间内执行完成

RTOS选择

❖ 真的需要RTOS?

- 主程序是不是很长? 程序的执行是不是需要判断很多条件参数或资源是否得到?
- 是否发现花费很多时间盘算怎样使一段代码在该执行的时候执行?
- 是否花费太多时间在中断子程序上? 需要编写出所有代码处理中断事件?
- 处理器控制的时间和方式和预想的一样吗?

RTOS选择原则

- ❖
- ❖ **RTOS**性能（包括任务最长切换时间、中断最长延迟时间、可调度的任务数和优先级数等）
- ❖ 软件组件和设备驱动程序是否齐全
- ❖ 开发工具和调试工具是否易用
- ❖ 标准兼容性，是否支持**POSIX**标准
- ❖ **RTOS**发送形式，是源代码还是二进制代码
- ❖ 是否需要许可证以及能否提供及时的技术支持

EmbestIDE和RTOS的配合

- ❖ **EmbestIDE**完全支持源代码方式提供的**RTOS**，只需要将**RTOS**的源代码纳入**EmbestIDE**的工程管理目录中，和应用程序一起编译，即可调试**RTOS**及应用程序。
- ❖ **EmbestIDE**支持二进制码方式提供的**RTOS**，条件是**RTOS**的二进制码是使用**GNU**工具链编译的，将该二进制码文件作为库链接进**EmbestIDE**应用工程，即可调试**RTOS**及应用程序。

uCOS操作系统

- ❖ 简单、高效、易用
- ❖ 完全免费使用，提供全部源代码
- ❖ 支持**64**个任务
- ❖ 支持多种**CPU**
- ❖ 提供邮箱、信号量以及消息队列三种任务间通讯方式
- ❖ 无设备驱动程序，缺乏足够多的应用模块（如**TCP/IP**协议以及**GUI**模块等等）
- ❖ 无技术支持

eCOS操作系统

- ❖ 从**linux**移植而来，复杂但完全功能
- ❖ 完全免费使用，提供全部源代码
- ❖ 支持多种**CPU**
- ❖ 使用**GNU**工具链开发
- ❖ 有完备的设备驱动程序和应用模块（可从**linux**中移植）
- ❖ 技术支持需付费
- ❖ 支持**POSIX**标准
- ❖ 需要较多的系统资源

uCLinux操作系统

- ❖ 从**linux**移植而来，复杂但完全功能
- ❖ 完全免费使用，提供全部源代码
- ❖ 支持多种**CPU**
- ❖ 使用**GNU**工具链开发，提供通用的**linux API**
- ❖ 有完备的设备驱动程序和应用模块（可从**linux**中移植）
- ❖ 使用直接物理内存访问方式
- ❖ 完整的**TCP/IP**协议栈
- ❖ 需要较多的系统资源，内核<**512KB**
- ❖ 无技术支持，但国内使用较多

Vxworks操作系统

- ❖ 销售额最大的实时操作系统，价格昂贵
- ❖ 通常只提供二进制码内核
- ❖ 支持多种**CPU**
- ❖ 完整的开发工具和测试工具
- ❖ 完备的设备驱动程序和应用模块
- ❖ 技术支持需付费
- ❖ 支持**POSIX**标准
- ❖ 需要中等系统资源，性能好，功能齐全

ARM寄存器

- ❖ ARM寄存器组织
- ❖ 各种模式下的寄存器
- ❖ 主要寄存器的用途

ARM寄存器组织

ARM处理器总共有37个寄存器:

- ❖ 31个32位通用寄存器
- ❖ 6个32位状态寄存器

ARM寄存器分类

❖ 通用寄存器

- 不分组寄存器 **R0~R7**
- 分组寄存器 **R8~R14**

(访问的具体物理寄存器取决于当前处理器模式)

- 程序计数器(PC) **R15**

❖ 程序状态寄存器 CPSR

(每种处理器模式都有单独的当前程序状态寄存器)

General Registers and Program Counter Modes

User32	FIQ32	Supervisor32	Abort32	IRQ32	Undefined32
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
------	------	------	------	------	------

特殊功能寄存器

❖ R13

用做堆栈寄存器SP，每种异常模式都有自己单独的分组R13

❖ R14

用做子程序链接寄存器LR，可保存子函数返回地址

❖ R15

用做程序计数器PC

❖ CPSR

保存当前程序状态，包含条件码标志、中断禁止位、当前处理器模式以及其它状态和控制信号

存储区配置

- ❖ 存储区类型
- ❖ 存储区大小和起始地址
- ❖ 访问等待周期
- ❖ 访问时序
- ❖ 数据总线宽度
- ❖ 其他需要设置的内容

存储区配置

- ❖ 启动代码完成
- ❖ 手工实现
- ❖ 利用命令脚本

ARM的存储器组织

ARM的中断向量表分配在0x0到0x20,

ARM启动代码从0地址开始执行

复位的时候, 0地址为ROM区

很多芯片允许存储区的重映射

ARM指令集

- ❖ 寄存器装载和存储指令
- ❖ 算术和逻辑指令
- ❖ 移位操作指令
- ❖ 乘法指令
- ❖ 比较指令
- ❖ 分支指令
- ❖ SWI 指令
- ❖ 协处理器指令
- ❖ 伪指令

Thumb指令集

- ❖ 存储器访问指令
- ❖ 数据处理指令
- ❖ 分支指令
- ❖ 中断和断点指令
- ❖ 伪指令

寄存器装载和存储指令

举例：

STR Rd, [Rbase] ;存储 **Rd** 到 **Rbase** 所包含的有效地址。

STR Rd, place ;存储 **Rd** 到 **PC + place** 所合成的有效地址

STR Rd, [Rbase, Rindex, LSL #2] ;存储 **Rd** 到 **Rbase + (Rindex * 4)**
;所合成的有效地址

算术和逻辑指令

举例：

ADCS R3, R7, R11 ; 加高端的字，带进位

AND R0, R0, #3 ; **R0** = 保持 **R0** 的位 0 和 1，丢弃其余的位。

BIC R0, R0, #%1011 ; 清除 **R0** 中的位 0、1、和 3。保持其余的不变

MOV R0, R0, LSL#3 ; **R0** = **R0** * 8

移位操作指令

LSL	逻辑左移
ASL	算术左移
LSR	逻辑右移
ASR	算术右移
ROR	循环右移
RRX	带扩展的循环右移

乘法指令

MLA : 带累加的乘法(**M**ultiplication with **A**ccumulate)

MLA{条件}{S} <dest>, <op 1>, <op 2>, <op 3>

$$\text{dest} = (\text{op}_1 * \text{op}_2) + \text{op}_3$$

MLA 的行为同于 **MUL**，但它把操作数 3 的值加到结果上。这在求总和时有用。

MUL : 乘法(**M**ultiplication)

MUL{条件}{S} <dest>, <op 1>, <op 2>

$$\text{dest} = \text{op}_1 * \text{op}_2$$

MUL 提供 32 位整数乘法。如果操作数是有符号的，可以假定结

↓

比较指令

举例：

CMN R0, #1 ; 把 **R0** 与 **-1** 进行比较

TST R0, #%1 ; 测试在 **R0** 中是否设置了位 **0**。

分支指令

B : 分支(**B**ranch)指令, 处理器PC立即跳转到给定的地址

B{条件} <地址>

BL : 带连接的分支(**B**ranch with **L**ink)指令, 在分支之前, 在寄存器 14 中装载上 R15 的内容

BL{条件} <地址>

伪指令

- ❖ ADR
- ❖ ADRL
- ❖ LDR
- ❖ ALIGN
- ❖ DCx
- ❖ EQUx
- ❖ OPT

伪指令，它们不是处理器实际上能理解的指令，但可以转换成它能理解的某种东西。它们的存在能使你的程序更加简单。

异常向量

- ARM 的Exception Handler
 - Reset
 - Undefined instruction
 - SWI
 - Prefetch abort
 - Data abort
 - Reserved
 - FIQ
 - IRQ

RESET

- 正常情况下，系统reset后进入的入口
 - ENTRY主入口
 - 驻留于存储系统的0x0地址，占4Byte,机器码通常为EA0000XX
 - 通常在进入时将系统CPSR设为监控模式，退出时改为用户模式
 - 所有必须的BSP必须在此入口内完成
 - 使用汇编语言编写，可控制C代码入口

Undefined Instruction

- 未定义指令
 - 当程序员使用未定义指令时，系统出错处理的入口
 - 驻留于存储空间的0x4,4bytes
 - 处理程序通常做法是首先进行现场保护，然后Do nothing
 - 当出现此错误时，主因大致是仿真器跳入错误地址。

SWI

- 软中断
 - 驻留于0x8, 4Bytes
 - monitor程序的入口
 - 包含有丰富的指令寄存器
 - 适用于ARM和Thumb

Abort

- Perfect abort
 - 预取失败错误
 - 驻留于0x0c,4bytes
- Data abort
 - 取数据失败错误
 - 驻留于0x10,4bytes
 - 通常是保护现场，然后do nothing
 - 出错主因：程序跳飞（查程序）

FIQ和IRQ

- FIQ快速中断请求
 - 驻留于0x18,4bytes
- IRQ中断请求
 - 驻留于0x1c,4bytes
 - FIQ和IRQ处理原理相同
 - 所有的硬件中断源共用一个通道来进行IRQ或FIQ
 - 中断处理支持中断嵌套

编程实例1.数码管显示程序

功能：显示**0 1 2.....F**

lcddemo.c

Isr.c

snds.h

ledDemo.h

segment.h

板卡ROM编程

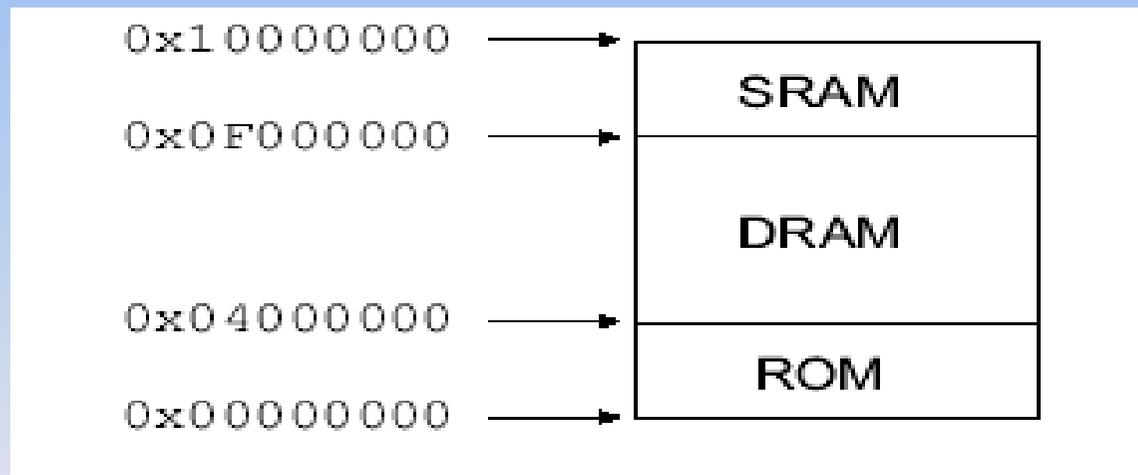
本章包含如下部分:

- 关于写**ROM**代码
- 内存映射的考虑
- 系统的初始化
- 调用地址**0**处的**ROM**镜像

内存映射的考虑

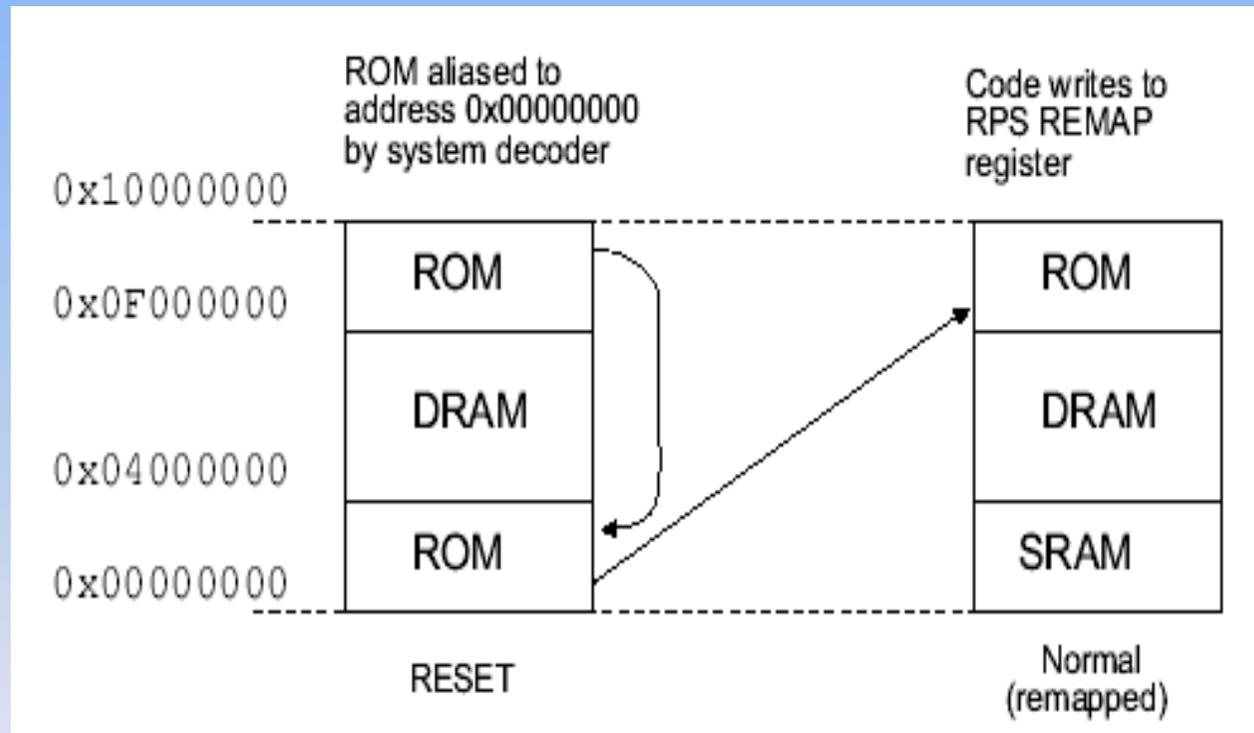
❖ 内存映射的考虑

❖ ROM at 0x0



RAM at 0x0

❖ RAM at 0x0



系统的初始化

- ❖ 系统的初始化
- ❖ 执行环境的初始化
- ❖ 入口点的识别
- ❖ 设置向量表
- ❖ 内存系统的初始化
- ❖ 堆栈指针的初始化
- ❖ 任何关键的I/O设备的初始化
- ❖ 通过中断系统初始化RAM变量
- ❖ 中断的使能
- ❖ 改变处理器模式

应用的初始化

- ❖ 应用的初始化
- ❖ 根据C代码的要求初始化内存
- ❖ 使用主函数

调用地址0处的ROM镜像

调用地址0处的ROM镜像

To build the example from the CodeWarrior IDE:

- 1. Use the CodeWarrior project `embed.mcp`**
- 2. Select Target=Embedded.**

execute build_b.bat

❖ 从命令行编译例子，执行build_b.bat 步骤如下：

1. Assemble the initialization code:

```
armasm -g vectors.s
```

```
armasm -g init.s
```

Compile the main example

2.编译以下程序，步骤如下：

```
armcc -c -g -O1 main.c -DEMBEDDED
```

```
armcc -c -g -O1 retarget.c
```

```
armcc -c -g -O1 serial.c -I. \include
```

Link the image

3. 使用如下命令链接映像 (**all on one line**):

```
armlink vectors.o init.o main.o  
retarget.o serial.o  
-ro-base 0x0 -rw-base 0x00040000  
-first vectors.o(Vect) -entry 0x0  
-o embed.axf -info totals -map -list  
list.txt
```

Run the fromELF

4. 运行ELF，产生目标文件：

where:

-bin Specifies a binary output image with no header.

```
fromelf embed.axf -bin -o embed.bin
```

Use ARMuLator to test

5. 用 ARMuLator 测试目标文件（下载目标文件到 ROM 中并执行）：

• **For armsd use:**

```
getfile embed.bin 0x0
```

```
readsyms embed.axf
```

• **For ADW & ADU, select:**

Use ARMuLator to test

File → **Get File** and specify embed. bin with load address 0x0.

File → **Load symbols only** and specify embed. axf.

- For AXD select:

File → **Load Memory From File** and specify embed. bin with load address 0x0.

File → **Load Debug Symbols** and specify embed. axf.

内存映射

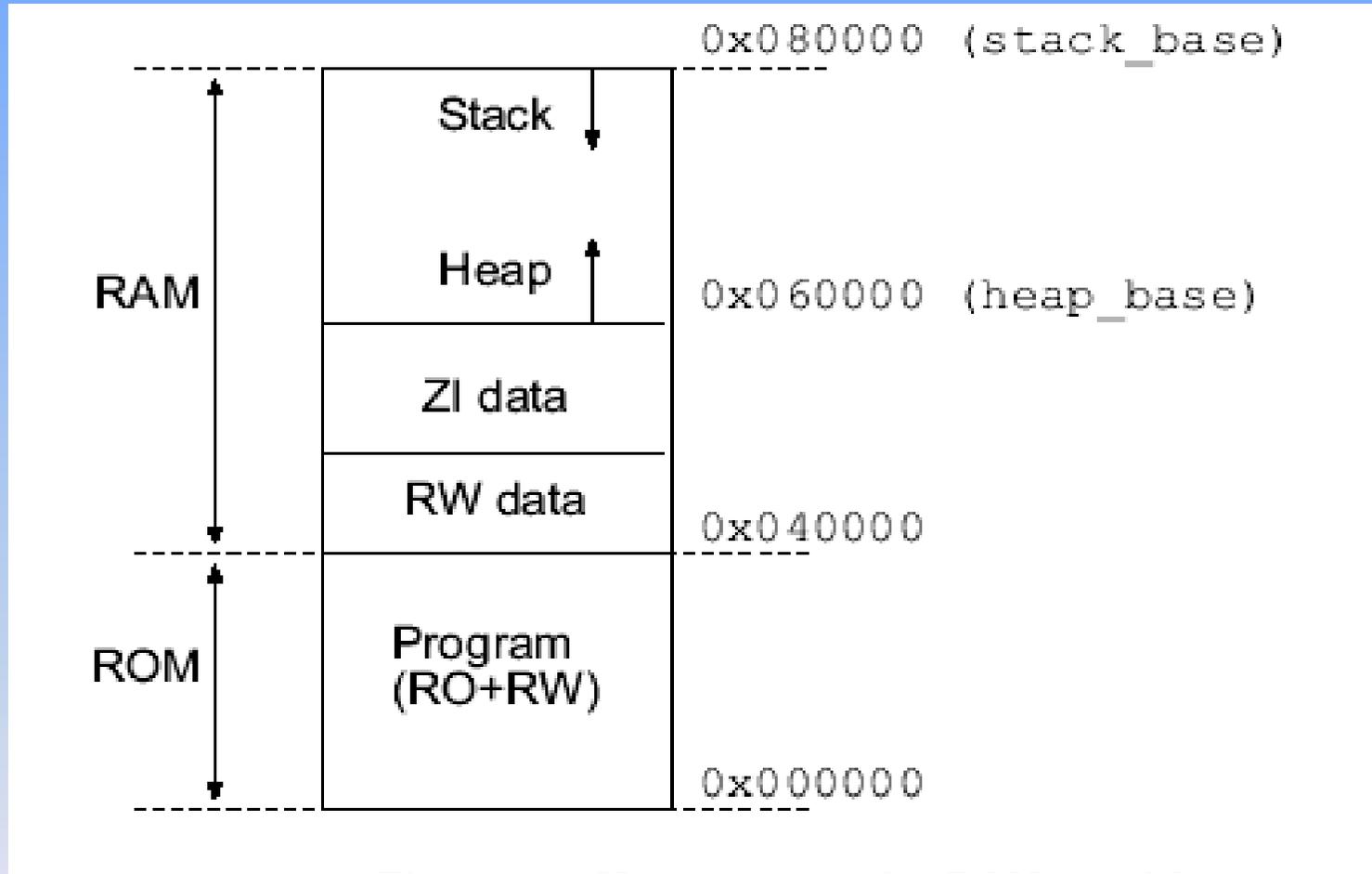
内存映射:

- . **ROM is address 0 as specified by `-ro-base`.**
- . **RAM is at 0x040000, as specified by `-rw-base`, to hold the stack, heap, and data.**
- . **The stack pointer is initialized to 0x80000 in `init.s`.**

Writing Code for ROM

- . **The heap base is initialized to 0x060000 by `__user_initial_stackheap()` in `retarget.c`.**
- . **The stack pointer is inherited from the value set in `init.s`.**

Memory map



列表文件输出

列表文件输出

The file list.txt shows the map (segment listing) for the sample code:

Image component sizes

Code R0 Data RW Data ZI Data Debug

1152 64 8 12 7660 Object Totals

21908 772 0 64 8576 Library Totals

Code R0 Data RW Data ZI Data Debug

23060 836 8 76 16236 Grand Totals

Total R0 Size(Code + R0 Data) 23896 (23.34kB)

Total RW Size(RW Data + ZI Data) 84 (0.08kB)

Total ROM Size(Code + R0 Data + RW Data) 23904 (23.34kB)

BootLoader编程

BootLoader功能:

- ❖ 初始化 CPU 寄存器
- ❖ 初始化系统设备
- ❖ 初始化系统参数。
- ❖ FLASH烧写工具
- ❖ 基本调试工具
- ❖ 初始化系统设备
- ❖ 初始化系统参数，初始化网络 初始化系统设备
- ❖ 其他功能程序

μ HAL:硬件抽象层:

μ HAL:硬件抽象层:

μ HAL allows for rapid prototyping and porting of applications (including RTOSs). On an existing target, an application can be written in 'C' from `main()`. Porting to a new environment is simple because of μ HALs modular design. The file `boot.s` in the `lib` directory performs initial configuration of exception vectors, memory, stack, MMU and cache initialisation, before calling `main()`.

Vectors 向量

向量: **Vectors**

ARM processors have eight types of exception.

Addresses	Exception Type	Mode	Priority
0x00	Reset	Svc	1 (High)
0x04	Undefined instruction	Undefined	6 (Low)
0x08	Software Interrupt	Svc	6
0x0C	Prefetch Abort	Abort	5
0x10	Data Abort	Abort	2
0x14	Reserved	N/A	N/A
0x18	Interrupt IRQ	Irq	4
0x1C	Fast Interrupt FIQ	Fiq	3

Routines过程

Routines

The entry-point to an ARM program is defined by the Assembler/Linker directive 'ENTRY'. μ HAL attaches this directive to the routine name '__main'. By actually executing the reset exception vector as the first instruction, μ HAL can be loaded into ram by another program, or physically stored in static memory at zero. When used in a system with static memory at zero, the default vectors in the file boot.s in the lib directory will need to be changed to correspond to the ones actually used in the high-level application.

LEDS

LEDs

Each target may provide a different number of Light-Emitting Diodes (LEDs)

for use as status indicators. By using a standard access mechanism, an application can be transferred to new platforms without concern about IO

space addressing, or whether writing a 1 turns a LED on or off.

Serial IO

Serial IO

μ HAL currently only provides minimal serial IO support, just enough to reset the defined serial port and to handle polled input and output.

IO Space Addressing

IO Space Addressing

By using routines to access peripherals in the IO Address Space, all regarding different busses and access mechanisms such as ISA/PCI can be completely masked. Such routines also allow easy transfer between multiple target environments.

Software Interrupts (SWIs)

Software Interrupts (SWIs)

Software Interrupts, or SWIs, provide a means for a program running in User mode to request privileged operations which need to be run in Supervisor mode. The only SWI currently handled by μ HAL is SWI_EnterOS, which switches into Supervisor mode. All other SWIs are undefined; they are currently ignored, but no code should rely on this.

Interrupt Support Routines

Interrupt Support Routines

These are the routines used to install, remove, enable and disable interrupts.

API

void __main(void)

void __rt_exit(void)

void *SAir_InitTargetMem(void)

void *SAir_InitStacks(void *TopOfStacks)

void SAir_InitBSSMemory(void)

void *SAr_StartOfRam(void)

void *SAr_EndOfFreeRam(void)

.....