

ChipON

KungFu32 嵌入式

编译工具链

用户手册 V1.2

修订记录

版本	日期	描述
V1.0	2026.02.28	初稿,kf32r 框架适用
V1.1	2026.04.08	增加 2.8 使用工具内部库
V1.2	2026.07.31	部分描述与文件格式更新

目 录

修订记录.....	2
目 录.....	3
1 概述.....	7
1.1 简介.....	7
1.2 支持的架构.....	8
1.3 数据类型.....	8
1.3.1 字节序.....	8
1.3.2 整型类型与范围.....	8
1.3.3 默认字节 char 变量的符号说明.....	9
1.3.4 浮点类型与长度.....	9
1.3.5 指针类型与长度.....	9
2 快速入门.....	10
2.1 运行平台.....	10
2.2 编译器目录结构.....	10
2.3 寄存器规格说明.....	10
2.4 编译器典型工作流程.....	11
2.5 使用编译器.....	12
2.5.1 示例代码.....	12
2.5.2 构建可执行文件.....	12
2.5.3 属性支持.....	12
2.6 使用汇编器.....	14
2.6.1 示例代码.....	14
2.6.2 汇编源文件.....	15
2.6.3 C/C++代码调用汇编函数.....	15
2.6.4 编译和链接 C 源文件.....	15
2.7 使用链接器.....	16
2.8 使用工具内部库.....	16
2.8.1 C 库方法示例与使用.....	16
2.8.1.1 stdio 打印输出.....	17

2.8.1.2 类型判断库函数.....	17
2.8.1.3 系统标准库函数.....	17
2.8.2 数学运算库函数.....	18
2.8.3 运行时库.....	18
2.8.4 可变参数列表.....	18
3 工具链选项说明.....	19
3.1 强制性选项.....	19
3.2 编译器工具链常用选项.....	19
3.2.1 kf32clang 常用选项.....	19
3.2.2 kf32as 常用选项.....	20
3.2.3 kf32ld 常用选项.....	20
3.3 选择源语言标准.....	20
3.3.1 源语言标准.....	20
3.3.2 语言标准支持的例外情况.....	21
3.3.3 未定义行为.....	21
3.4 选择优化选项.....	21
3.4.1 优化等级介绍.....	22
3.5 调试信息-g.....	24
3.6 从编译器向链接器传递选项-Wl.....	24
3.7 控制诊断消息.....	24
3.8 编译器工具命令行选择规则.....	27
4 优化工作下代码.....	28
4.1 VOLATILE 关键字.....	28
4.1.1 volatile 的含义.....	28
4.1.2 何时使用 volatile.....	28
4.1.3 未使用 volatile 时可能出现的潜在问题.....	29
4.1.4 强制使用特定指令访问内存.....	29
4.1.5 未使用 volatile 关键字导致的无限循环示例.....	29
4.2 循环优化.....	30
4.2.1 循环展开.....	30
4.2.2 C 代码中的循环终止.....	30
4.3 函数内联.....	32
4.3.1 函数内联示例.....	33
4.4 C 和 C++中的栈使用.....	34

4.4.1 减少栈使用量的方法.....	34
4.5 紧凑数据空间分配.....	35
4.5.1 打包整个结构体.....	35
4.5.2 在结构体中打包单个成员.....	37
4.5.3 访问结构体中被打包的成员.....	37
4.6 针对代码大小或性能进行优化.....	38
4.7 优化对调试的影响.....	39
5 汇编文件输出.....	40
5.1 汇编示例.....	40
5.2 汇编流程.....	40
6 在 C 或 C++代码中使用内嵌汇编.....	42
6.1 编写内嵌汇编代码.....	42
6.1.1 内嵌汇编语句的结构.....	42
6.1.1.1 普通字符串形式.....	43
6.1.1.2 嵌汇编注意事项.....	44
6.1.2 符号与标签的定义.....	45
6.2 从 C 和 C++调用汇编函数.....	45
7 编译工具汇编器.....	47
7.1 汇编器的主要特性.....	47
7.2 命令行语法.....	47
7.3 常用伪指令介绍.....	47
7.3.1 .section.....	47
7.3.2 .global symbol.....	48
7.3.3 .align abs-expr, abs-expr, abs-expr.....	48
7.3.4 .byte expressions.....	48
7.3.5 .word expressions.....	48
7.3.6 .short expressions.....	48
7.3.7 .int expressions.....	48
7.3.8 .long expressions.....	48
7.3.9 .fill repeat, size, value.....	49
7.3.10 .type name , type description.....	49
7.3.11 .size name , expression.....	49
7.3.12 .if absolute expression.....	49

7.3.13 .else.....	50
7.3.14 .elseif.....	50
7.3.15 .endif.....	50
7.3.16 .set symbol, expression.....	51
7.3.17 .end.....	51
8 编译工具链接器.....	52
8.1 命令行语法.....	52
8.2 链接器的输入.....	53
8.3 链接器的输出.....	53
8.4 链接器输出布局.....	53
9 编译工具库管理.....	54
9.1 命令行语法.....	54
9.2 库管理相关.....	54
9.3 处理库文件的注意事项.....	55
9.4 常用功能示例.....	55
10 编译工具反汇编器.....	56
10.1 命令行语法.....	56
10.2 选项说明与示例.....	56
10.2.1 选项说明.....	56
10.2.2 反汇编示例.....	57
11 编译工具 ELF 转换工具.....	58
11.1 命令行语法.....	58
11.2 常用选项示例与说明.....	58
11.2.1 选项说明.....	58
11.2.2 转换示例.....	58
12 编译工具段大小查看器.....	60
12.1 命令行语法.....	60
12.2 输出字段说明.....	60
12.3 常用选项示例与说明.....	61
12.3.1 选项说明.....	61
12.3.2 功能示例.....	61

1 概述

本章介绍 ChipON KungFu32 嵌入式编译工具链（以下简称 KF32 工具链）的组成，帮助您快速了解 KF32 工具链的功能以及典型的程序开发流程，并介绍 KF32 工具链支持的架构。

1.1 简介

KF32 工具链支持 C/C++ 编译，能为 ChipON KungFu32 位嵌入式 MCU 生成高效可执行文件，支持 C99 与 C++14 标准。

KF32 工具链包括基于 LLVM 的编译器、汇编器、链接器和库等，具体如下：

kf32clang

编译器：基于 LLVM 和 Clang 技术，用于编译 C 源文件使用。

kf32clang++

编译器：基于 LLVM 和 Clang 技术，用于编译 C++ 源文件使用。

kf32as

汇编器：将汇编语言源代码转换为机器可执行的目标代码

kf32ld

链接器：链接器将一个或多个目标文件的内容与一个或多个目标库的选定部分组合起来，以生成目标程序。

kf32ar

库管理工具：kf32ar 支持以标准格式的 ar 库来收集和维护 ELF 目标文件集合。

kf32objdump

反汇编工具：支持解析标准格式的 ELF 目标文件，并将其机器指令转换为可读的汇编语言，同时可展示文件结构与符号信息。

kf32objcopy

ELF 转换工具：支持将链接生成的 ELF 格式目标文件转换为多种可直接下载到芯片存储器的标准格式文件。

kf32size

段大小查看器：用于快速统计和显示编译链接后生成的可执行文件中，各段（section）在存储器（如 Flash 和 RAM）中的占用大小

C 库

提供适用于嵌入式设备的高效 C 库和数学库

C++ 库

提供适用于嵌入式设备的高效 C++ 库

1.2 支持的架构

注：基于 LLVM 的 KF32 工具链与老版本 GCC 的工具链不兼容，请勿混用。历史工具链生成的库不建议在当前工具链使用。

当前，KF32 工具链支持以下架构：

- kf32r(精简指令集)

典型型号 KF32A158

1.3 数据类型

1.3.1 字节序

KF32 编译器以小端字节序格式存储数据。最低有效字节存储在最低地址。

例如，32 位数值 0x12345678 在地址 0x10000000 处开始，按如下格式存储：

表 1.1 存储示例

地址	0x10000000	0x10000001	0x10000002	0x10000003
数值	0x78	0x56	0x34	0x12

1.3.2 整型类型与范围

KF32 编译器中的整型范围如下：

表 1.2 整型类型

类型	位	字节	最小值	最大值
signed char	8	1	-128	127
char,unsigned char	8	1	0	255
short,signed short	16	2	-32768	32767
unsigned short	16	2	0	65535
int,signed int	32	4	-2^{31}	$2^{31}-1$
unsigned int	32	4	0	$2^{32}-1$
long,signed long	32	4	-2^{31}	$2^{31}-1$
unsigned long	32	4	0	$2^{32}-1$
long long,signed long long	64	8	-2^{63}	$2^{63}-1$
unsigned long long	64	8	0	$2^{64}-1$

1.3.3 默认字节 char 变量的符号说明

默认情况下，不带修饰符的 char 类型的值被 KF32 编译器定义为无符号值。同时，可以使用命令行选项 -fsigned-char 将默认类型设置为有符号，-funsigned-char 将默认类型设置为无符号。

1.3.4 浮点类型与长度

KF32 编译器使用 IEEE-754 浮点格式。

表 1.3 浮点类型

类型	位	字节
float	32	4
double	64	8
long double	64	8

1.3.5 指针类型与长度

KF32 编译器中指针长度为 32 位整数。

2 快速入门

本章介绍 KF32 工具链的使用方式，并通过简要的例子帮助您快速开始使用 KF32 工具链进行开发。

2.1 运行平台

本节列出了工具链运行的系统环境。

系统要求

支持以下操作系统：

- 64 位 Windows 系统
- 64 位 Linux 系统

当前已在 Windows10 专业版 22H2 版本与 ubuntu20.04 版本进行测试。

2.2 编译器目录结构

工具链命名与版本：当前发布的命名为 kf32-toolchain，版本号为 vX.X.X。

不同平台工具包名：

kf32-toolchain-vX.X.X-windows（Windows 平台）

kf32-toolchain-vX.X.X-linux（Linux 平台）

关键目录结构及其功能：

\bin 目录：存放基本工具的可执行文件，例如编译器 kf32clang.exe。

\lib\clang-runtimes\kf32r\kf32r_generl\lib 目录：存放芯片相关的算法库与系统库文件，例如 libm.a。

\lib\clang-runtimes\kf32r\kf32r_generl\include 目录：存放相关的头文件，例如 math.h、malloc.h 等

2.3 寄存器规格说明

在 C 语言中寄存器使用规则如表，因此可以编写汇编带参数或返回的库，参数或结果在如下对应功能的寄存器中。另外编写汇编库应该适应 C 规则下占用寄存器的压栈出栈保护。

表 2.1 寄存器约定

寄存器名称	用途	跨函数调用时是否保存
R0	gp/传参/返回值	不保存
R1	gp/传参/返回值	不保存
R2 - R4	gp/传参	不保存
R5	gp/scratch	不保存
R6	gp/fp	
R7 - R12	gp	
R13	lr/gp/函数返回地址	
R14	sp/fixed	否
R15	pc/fixed	否

在表 2.1 中，寄存器支持 R0~R15, gp 表示该寄存为通用寄存器，可参与任何操作，fixed 表示该寄存器为特殊寄存器，具有固定的用途，寄存器分配时，不参与分配。R6-R13 用于保存生命周期跨函数调用的变量。R6 在特殊情况下可替代实现 FP 指针的功能，如临时栈空间开辟。R13 别名为 LR，R14 别名为 SP，R15 别名为 PC。

如果中断和外部函数同时使用 R6 以上的寄存器时，应添加压栈出栈指令。

若中断逻辑中使用 R5，可在中断顶层函数添加压栈出栈保护 R5。

2.4 编译器典型工作流程

一个典型的应用程序开发流程包括如下步骤：

- 为主程序开发 C/C++ 源代码（使用 kf32clang 编译）。
- 为接近硬件底层的组件开发汇编语言代码，例如中断服务程序（使用 kf32clang 编译）。
- 链接所有的目标文件生成可执行文件（使用 kf32ld）。
- 将可执行文件转换为纯二进制的 flash 格式。（使用 kf32objcopy）

下图展示了如何使用编译工具开发的典型应用程序：

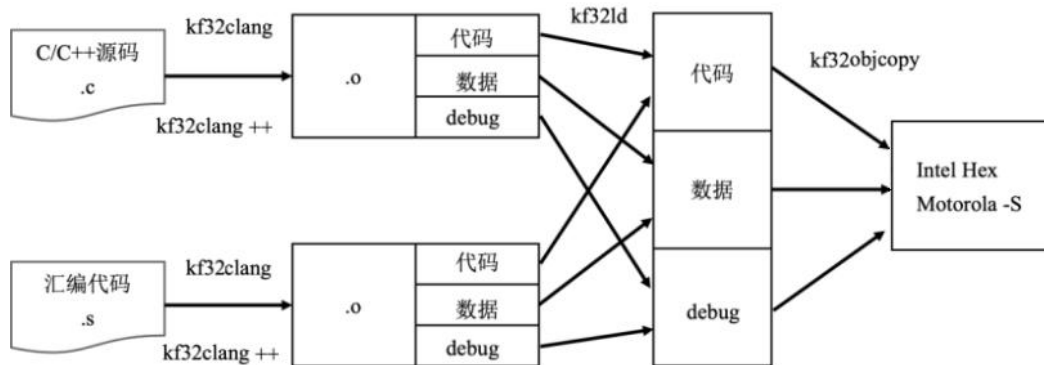


图 2.1 编译器工作流程图

2.5 使用编译器

本节展示如何使用 KF32 工具链从 C/C++源文件构建可执行文件。

2.5.1 示例代码

示例中使用的源代码是一个单独的 C 源文件 `hello.c`：

```
#include <stdio.h>
int main() {
    printf("Hello World\n");
    return 0;
}
```

2.5.2 构建可执行文件

对于简单程序，可以使用单个命令将源代码文件编译为可执行文件。

```
kf32clang --target=kf32r -march=kf32r -T kf32demo.ld hello.c -o hello.elf
```

该命令会创建一个名为 `hello.elf` 的可执行文件。

可通过添加命令行参数 `-c` 或 `-S` 实现编译和汇编或仅输出汇编文件。

2.5.3 属性支持

属性用于向编译器提供关于变量、函数、类型或语句的额外信息，以控制其编译行为、优化或内存布局。

通过属性扩展变量或函数的意义，如中断函数。格式为 `__attribute__((attribute-list))` 其支持多个属性，属性之间用“,”间隔。

也支持该格式单个属性同格式的多次书写表达多属性。

- **Interrupt 中断函数**

用作中断处理程序的函数生成序言和尾声代码。如：使用 `__attribute__((interrupt))` 修饰函数，则该函数即被编译器视为中断函数处理。针对中断向量表入口函数，其为必须声明的属性。

- **section (“name”) 指定类型段**

将函数或变量放入由“name”指定的段。

例如，`void __attribute__((section(“new_sect1”))) foo() {return;}`

函数 `foo` 将被放入 `new_sect1` 段。

`unsigned int var __attribute__((section(“new_sect2”)))`

变量 `var` 将被放入 `new_sect2` 段。

当声明该属性后 `-ffunction-sections`、`-fdata-sections` 命令行选项对该属性定义的函数不起作用。

常见的段定义为 `.text(flash 空间)|.data(RAM 空间)|.bss(RAM 空间)`。同时根据需要将空间进行顺序使用。针对 `flash` 空间，先存放 `text` 段的 `vector` 文件中代码，即中断向量表；紧接着存放默认的函数代码段 `.text`；跟随存放常量属性的 `.rdata(readonly flash 空间)` 和 `.rodata(readonly flash 空间)` 代码信息；工具控制将 `ram` 的初始化值跟随结尾存放。针对 `ram` 空间，设计 `ram` 函数的存放的定义 `.indata` 段名，定义初始化的数据 `.data` 段名，定义无初始化的数据 `COMMON` 或 `bss` 段名，

RAM 的空间默认设计顺序：.indata .date .bss .comm,剩余空间为堆栈空间。

- **packed 对齐**

具有该属性的变量或结构成员将具有所可能的最小对齐值。即，将不为声明分配任何对齐填充存储空间。与 **aligned** 属性联合使用时，**packed** 可以用于设置任意的对齐限制，即大于或小于变量或结构成员的类型所具有的默认对齐值。

需要注意的是，即使声明了该对齐模式，但针对结构体、联合体，其起始仍将按照对齐存放。

如：

```
struct name1 {
    char order1;
    struct name2 {
        char order2;
    } __attribute__((packed)) names;
} __attribute__((packed)) name ; //其 order2 偏移地址是 4。
```

注：对齐通常控制不同数据类型的联合。同时应考虑芯片对齐属性的要求进行数据类型的合理使用。如连续 **char** 数据的操作不应通过 **int** 型的指针操作。

注：针对结构体下的位定义，应该使用 **packed** 修饰该结构体，否则其代码联合优化将错误的选用对齐的指令操作，从而编译结果与代码期望不符。

- **常见属性**

- 变量属性支持**

- aligned(alignment)** 定义数据地址对齐配置

- unused** 定义参数未使用的不给予警告

- used** 即使看起来未使用，也对其进行编译

- packed** 一般语言结构体或联合，明确内部连续地址分派。

- section("section-name")**

- 函数属性支持**

- section("section-name")**

- interrupt** 定义该函数为中断服务函数，中断函数必须该属性修饰

- noinline** 强制函数不内联，定义的 **inline** 及默认由编译器确定是否内联

- always_inline** 强制函数内联

- weak** 常定义中断函数，即 即使未发现调用关系仍保留该函数,但值地址为 0。

- alias("fun-name")** 函数为别名,可实现未定义时关联到默认函数

- unused** 未使用也不输出警告信息

2.6 使用汇编器

本节介绍如何使用运行 KF32 汇编器从汇编文件构建为目标文件，以及如何

从 C/C++ 源文件调用该目标文件中的函数。

有时为了操作接近硬件底层的硬件或取得更加极致的性能，您可能需要手动编写汇编代码来实现某些功能，为了编译这部分代码，您需要使用 `kf32clang` 的汇编器功能来进行编译。

2.6.1 示例代码

这个汇编示例是一个单独的汇编源文件 `add.s`，其中包含一个用于执行简单整数加法的函数：

```
kf32clang --target=kf32r -march=kf32r -c add.s
```

```
.text
.file "add.c"
.globl  add
.p2align 3
.type add,@function
add:
    ADD r0, r1, r0
    JMP lr
```

`.text` 是汇编中的段（**section**）标识符，表示接下来的内容属于程序的代码段。

`.file` 表示该汇编代码来自源文件 `add.c`。

`.p2align` 伪指令将后续代码对齐到 8 字节边界。这种对齐是为了符合 KF32 应用程序过程调用标准。

`.global` 指令将符号 `add` 标记为全局符号。这使得该符号可以被外部文件引用。

`.type` 伪指令将符号 `add` 的类型设置为函数。这有助于链接器在从代码跳转到该符号时使用正确的链接方式。

2.6.2 汇编源文件

在汇编代码时，必须首先确定目标文件要在哪个目标平台上运行。`--target` 选项决定了要为哪个目标状态进行编译。该选项是一个必填选项。

要为 `kf32r` 目标汇编上述源文件：

此命令会创建一个目标文件 `add.o`。

`--target` 选项用于指定编译的目标平台。

`-march` 选择用来指定支持的指令集架构。

2.6.3 C/C++代码调用汇编函数

在汇编文件中编写经过优化的函数，并从 C/C++代码中调用它们时，要确保汇编函数使用的寄存器符合调用标准。

以下 C 语言示例是一个单独的 C 源文件 `main.c`，其中包含对 `add` 函数的调用，用于将两个整型数字进行相加：

```
extern int add(int left, int right);
int main(void) {
    int result = add(3, 5);
    return result;
}
```

已为 `add` 函数添加了外部函数声明，返回类型和函数参数必须手动检查。

如果要从 C++源文件调用汇编函数，必须使用 `extern "C"`而非 `extern` 来禁用 C++的名称修饰。

2.6.4 编译和链接 C 源文件

要为 `kf32r` 架构编译上述源文件：

```
kf32clang --target=kf32r -march=kf32r -c main.c
```

此命令会创建一个目标文件 `main.o`。

要将两个目标文件 `main.o` 和 `add.o` 链接起来并生成可执行文件：

```
kf32ld main.o add.o -T kf32demo.ld -o add.elf
```

此命令会创建一个可执行文件 `add.elf`。

2.7 使用链接器

推荐通过编译器驱动（`kf32clang`）来调用 `kf32ld`，也可以直接使用 `kf32ld` 命令。

通过编译或汇编获取 `main.o` 和 `add.o` 两个目标文件，可以使用下面的命令将它们链接成为一个可执行文件：

```
kf32clang main.o add.o -T kf32demo.ld -Wl,-Map=pro.map -o pro.elf
```

该命令会生成一个 `a.elf` 可执行文件：

- `-T kf32demo.ld`: 指定链接脚本。
- `-Wl,-Map=pro.map`: 生成映射文件。
- `-o pro.elf`: 指定生成的可执行文件名称。

2.8 使用工具内部库

在编译器安装路径下：

`\lib\clang-runtimes\kf32r\kf32r_generl\lib` 目录：存放芯片相关的算法库与系统库文件，包括 `libm.a`、`libc.a`、`librt.a`、`libnosys.a`、`libc++.a`、`libc++abi.a`。

`\lib\clang-runtimes\kf32r\kf32r_generl\include` 目录：存放相关的头文件，例如 `math.h`、`malloc.h`、`string.h`、`stdio.h`、`sdtint.h`、`stdarg.h`、`stdlib.h` 等

KF32 工具提供适配嵌入式的 `c` 库和数学库，同时输出裁剪的 `c++` 库以及编译需要的运行时库，使用链接器时应主动传递如下方法库，使用 `-lc++ -lc++abi -lc -lm -lrt` 选项。

2.8.1 C 库方法示例与使用

KF32 编译工具链实现部分 C 标准库，包括字符操作、字符串操作、数学库及相应的常量、类型等定义。用户可通过编译工具头文件路径中的相应头文件进行查看。

2.8.1.1 stdio 打印输出

工具支持轻量文件系统定义，支持常规 C 的打印库，支持 c++打印输出。

提供基于串口映射的底层实现，可参考 `stdio_driver.c` 文件下的示例代码，对应当前的轻量的嵌入式适配方法。

注：历史工具将打印绑定到 `usart` 和基于 `usart` 的外设寄存器基址并作为参数传达的方法不适用，因此 `stdio.h` 的头文件中不再声明 `USARTx_STREAM`

2.8.1.2 类型判断库函数

使用时需要添加 `#include "ctype.h"`

`int isalpha(int c);`

是否为 A-Z,a-z 的字符，是返回自身，否则返回 0

`int isupper(int c);`

是否为 A-Z，是返回自身，否则返回 0

`int islower(int c);`

是否为 a-z，是返回自身，否则返回 0

`int isdigit(int c);`

是否为 '0'-'9'，是返回自身，否则返回 0

`int tolower(int c);`

返回字符对于的小写，或自身。即 A-Z 返回 a-z，其他字符保持自身返回。

`int toupper(int c);`

返回字符对于的大写，或自身。即 a-z 返回 A-Z，其他字符保持自身返回。

2.8.1.3 系统标准库函数

使用时需要添加 `#include "stdint.h"`

定义常规数据类型别名，如：

`typedef signed char int8_t;`

`typedef unsigned char uint8_t;`

`typedef short int16_t;`

`typedef unsigned short uint16_t;`

`typedef int int32_t;`

`typedef unsigned int uint32_t;`

`typedef long long int64_t;`

`typedef unsigned long long uint64_t;`

2.8.2 数学运算库函数

使用时需要添加#include “math.h”

如：

```
float sinf ( float );
```

```
double sin ( double );
```

```
float expf ( float );
```

2.8.3 运行时库

运行时库主要包含补充浮点模拟与 64 位数的移位。浮点库包括单精度、双精度浮点操作及 64 位整型变量运算的一系列函数组成。

这些函数由 C 语言编译器编译生成调用过程，以辅助实现相应功能。当 C 语言定义 float 类型并进行计算时，编译器会自动调用浮点库完成运算。当使用汇编编写代码时，根据参数传递规则调用对于的函数即可调用库实现。

2.8.4 可变参数列表

头文件 stdarg.h 支持带有可变参数列表的函数。参数列表中必须至少包含一个指定的参数。可变参数用省略号 (...) 表示，当宏定义传递时，可以在宏定义带入代码中使用__VA_ARGS__传递。必须在函数中声明一个 va_list 类型的对象用以保存这些参数。va_start 将变量初始化为一个参数列表，va_arg 用来访问这个参数列表，va_end 用来终止参数的使用。

3 工具链选项说明

KF32 工具链有许多选项用于为应用程序生成代码，本节介绍必需的和常用的可选命令行参数，例如用于控制目标选择、优化和调试的参数等。

3.1 强制性选项

使用 `kf32clang` 时，必须在命令行上指定目标。根据所使用的目标，还需要指定架构。

指定目标

使用 `--target` 选项，目前必须填写 `kf32r`。

指定架构

使用 `-march` 选项，为特定架构生成代码，支持的架构因所选目标而异。

3.2 编译器工具链常用选项

本节介绍 KF32 工具链中每个工具最常用的命令行选项。

3.2.1 `kf32clang` 常用选项

表 3.1 `kf32clang` 常用选项

选项	作用描述
<code>--target=name</code>	指定编译的目标平台
<code>-march=name</code>	指定编译架构
<code>-c</code>	只编译，不链接
<code>-std</code>	指定 C / C++ 的语言标准版本
<code>-g</code>	用于生成调试信息的选项
<code>-Onum</code>	用于控制优化等级
<code>-S</code>	编译到汇编
<code>-Wall</code>	启用编译器警告
<code>-ffunction-section</code>	每个函数一个 <code>.text.<name></code> 段
<code>-fdata-sections</code>	每个变量一个 <code>.data.<name></code> 段
<code>-o</code>	用于指定输出文件名
<code>-mmd</code>	自动生成依赖关系（.d 文件），用于 Makefile，便于管理头文件变更
<code>-v</code>	显示详细的编译过程信息（版本、调用的子命令、路径等），用于排查问题
<code>-save-temps</code>	保留编译过程中产生的所有临时文件

3.2.2 kf32as 常用选项

表 3.2 kf32as 常用选项

选项	作用描述
--arch=name	指定编译架构
-o	用于指令输出文件名

3.2.3 kf32ld 常用选项

表 3.3 kf32ld 常用选项

选项	作用描述
-T	使用自定义链接脚本，控制内存布局
-L	传递库搜索路径
-l	传递库
--gc-sections	删除未使用的函数和数据节
-o	用于指定输出文件名
-map	用于生成含最终可执行文件或库内部详细布局的文本报告

3.3 选择源语言标准

KF32 编译器对不同的源语言标准提供不同程度的支持。编译器会根据文件扩展名推断源语言（例如 C 或 C++）。推荐使用 -std 选项强制编译器选择的适用于嵌入式领域特定的源语言标准进行编译，如 C99，C++14。

3.3.1 源语言标准

KungFu32 编译器支持如下表所示的标准和 GNU 变体的源语言。

表 3.4 KungFu32 编译器支持标准

Standard C	GNU C	Standard C++	GNU C++
C89	–	C++04	–
C90	GNU90	C++11	GNU++11
C99	GNU99	C++14	GNU++14
C11	GNU11	C++17	GNU++17
C17	GNU17		

C 代码的默认语言标准是 C17。C++代码的默认语言标准是 C++17。要指定不同的源语言标准，请使用 -std=name 选项。

3.3.2 语言标准支持的例外情况

KF32 C/C++库对不同的源语言标准提供不同程度的支持。下表列出了其对每种语言标准的支持例外情况：

表 3.5 对语言标准的支持存在例外情况

语言标准	对语言标准的支持存在例外情况
所有支持的标准版本	对于所有支持的标准，libc++ 库与标准库存在以下差异： ● 对于 <code>std::vector<bool>::const_reference</code>，标准要求 <code>const_reference</code> 类型为 <code>bool</code>。然而，在 libc++ 中，<code>const_reference</code> 类型是一个实现定义的只读位引用类。 ● 对于 <code>std::bitset<N></code>，标准要求 <code>bool operator[] (size_t pos) const;</code> 返回 <code>bool</code>。然而，在 libc++ 中，<code>bool operator[]</code> 返回 <code>bool</code>。
C90	C90 完全支持
C99	C99 完全支持
C11	C11 完全支持
C++11	以下库不受支持 ● <code><experimental></code> ● <code><localization></code> ● <code><pstl></code> ● <code><thread></code>
C++14	以下库不受支持 ● <code><experimental></code> ● <code><localization></code> ● <code><pstl></code> ● <code><thread></code>

3.3.3 未定义行为

C 和 C++标准将那些使用不可移植、错误的程序或数据结构的代码视为未定义行为。对于存在未定义行为的程序，ChipON 不对 KF32 工具链的行为提供任何信息或保证，若 KF32 工具链未做特殊说明，则表示与通用 LLVM 编译器行为保持一致。

3.4 选择优化选项

KF32 工具链支持多种优化级别，以控制不同的优化目标。适合您应用程序

的最佳优化级别取决于您的应用程序和优化目标。

3.4.1 优化等级介绍

表 3.6 优化等级与优化目标

优化等级	优化目标
-O0	少量优化
-O1	比-O0 略多一些优化
-O2	最大化运行速度，且不显著增加代码体积或编译时间。
-O3	在-O2 基础上，启用更激进、可能增加代码体积的优化，以进一步挖掘性能。
-Ofast	追求极致性能，且不考虑体积
-Os	优化体积
-Oz	追求最小体积，且不考虑性能

开启优化后会改变调试信息和源码关系，可能造成调试体验变差。

优化级别-O0

-O0 会启用一小部分优化。此优化级别是默认设置。在-O0 级别下，代码大小和堆栈使用量略高于其他优化级别。若想强制禁用所有优化，可使用选项-O0 -native-O0。

优化级别-O1

-O1 启用编译器中的核心优化。此优化级别提供了良好的调试体验，且代码质量优于 -O0。此外，与 -O0 相比，栈使用情况也有所改善。推荐此选项以获得良好的调试体验。

与-O0 相比，使用-O1 时的差异如下：

- 已启用优化，这可能会降低调试信息的准确性。
- 内联和尾调用已启用，这意味着回溯可能不会提供从源代码中预期的开放函数激活栈。
 - 如果不需要结果，一个没有副作用的函数可能不会在预期的位置被调用，或者可能会被省略。
 - 变量的值在不再使用后，可能在其作用域内无法获取。例如，它们的栈位置可能已被重新使用。

优化级别 -O2

与-O1 相比，-O2 是更高的性能优化。它新增的优化很少，但与-O1 相比，优化的启发式方法有所改变。这一级别是编译器可能自动生成向量指令的首个优化级别。它还会降低调试体验，并且与-O1 相比，可能会导致代码大小增加。

使用 `-O2` 与 `-O1` 相比，差异如下：

- 编译器认为内联调用点有利可图的阈值可能会提高。
- 执行的循环展开量可能会增加。
- 向量指令可能会为简单循环以及独立标量操作的相关序列生成。

优化级别 `-O3`

与 `-O2` 相比，`-O3` 是更高阶的性能优化。这一优化级别启用了那些需要大量编译时分析和资源的优化，并改变了与 `-O2` 相比的优化启发式方法。`-O3` 指示编译器针对生成代码的性能进行优化，而不考虑生成代码的大小，这可能会导致代码大小增加。与 `-O2` 相比，它还会降低调试体验。

使用 `-O3` 与使用 `-O2` 相比，差异如下：

- 编译器认为内联调用点有利可图的阈值提高了。
- 执行的循环展开量有所增加。
- 在编译器流水线的后期会启用更激进的指令优化。

优化级别 `-Ofast`

`-Ofast` 是基于 `-O3` 的扩展优化级别，它在 `-O3` 的所有优化基础上，进一步放宽了对语言标准合规性的限制，以追求极致的运行时性能。

与 `-O3` 相比，`-Ofast` 的主要差异如下：

- 放宽标准合规性：允许编译器进行不严格遵循语言标准（如 ISO C/C++ 标准）的优化，特别是浮点运算方面。编译器可能重新排列或简化浮点操作，这可能改变计算结果（例如，忽略 NaN 或无穷大的处理），从而显著提升数值计算密集型代码的性能。
- 更激进的内联与循环变换：在 `-O3` 的基础上，进一步放宽对代码膨胀的限制，可能执行更激进的内联、循环展开和向量化。
- 数学优化：启用更激进的数学近似优化，例如用更快的近似计算替换某些数学函数（如 `sin`、`exp`），可能导致精度损失。
- 内存访问优化：可能进行更激进的内存访问模式优化，如忽略严格的指针别名规则（除非使用 `-fno-strict-aliasing` 限制）。
- 调试与可预测性：与 `-O3` 相比，`-Ofast` 进一步降低了调试体验和代码行为的可预测性，并使浮点结果更难以符合 IEEE 标准。

优化级别 `-Os`

与 `-O3` 相比，`-Os` 能减小代码大小。但与 `-O1` 相比，它也会降低调试体验。

使用 `-Os` 与使用 `-O3` 相比，差异如下：

- 编译器认为内联调用点有利可图的阈值降低了。
- 执行的循环展开量显著降低。

优化级别 -Oz

-Oz 旨在减小代码大小。如果追求最小体积，建议使用此选项以获得最佳的代码大小。与-O1 相比，此优化级别会降低调试体验。

使用-Oz 与使用-Os 相比，不同之处在于：

- 编译器仅针对代码大小进行优化，而忽略性能优化，这可能会导致代码运行速度变慢。
- 函数内联未被禁用。在某些情况下，内联可能会总体减小代码大小，例如当一个函数只被调用一次时。内联启发式算法经过调整，仅在预计会减小代码大小的情况下才进行内联。
- 可能会增加代码大小的优化（如循环展开和循环向量化）已被禁用。
- 循环生成为 while 循环而非 do-while 循环。

3.5 调试信息-g

在应用程序开发过程中，通常需要调试所构建的目标文件。KungFu32 编译器工具具有多种特性，可提供良好的源代码级调试。

可用的命令行选项

要构建用于调试的镜像，必须使用 -g 选项进行编译。此选项允许你指定要使用的 DWARF 格式。-g 选项等效于 -gdwarf-5。例如：

```
kf32clang -gdwarf-5
kf32clang -gdwarf-3
```

3.6 从编译器向链接器传递选项-Wl

默认情况下，运行 kf32clang 时，常规选项可以直接传递给链接器，如-l 选项，允许从编译器命令行直接向链接器传递选项，使用的语法为：-l，该选项指定一个由逗号分隔的选项和参数列表，或选项=参数对，例如：

```
kf32clang main.o -Wl,-T,kf32demo.ld -o program
```

3.7 控制诊断消息

KF32 工具链以警告和错误的形式提供诊断信息。您可以使用选项来抑制这些信息，或将其设置为警告或错误。

KF32 工具链会列出其在编译和链接过程中遇到的所有警告和错误。但是，如果你指定了多个源文件，该编译器仅会报告它在其中遇到错误的第一个源文件的诊断信息。

kf32clang 的消息格式

kf32clang 生成的消息格式如下：

文件：行：列：类型：消息

文件

包含错误或警告的文件名

行号

包含错误或警告的行号

列

生成消息的列号

类型

消息的类型，例如错误或者警告

消息

消息文本。

一个警告诊断消息示例如下：

```
test.c:4:3: error: use of undeclared identifier 'x'
  x = 5;

1 error generated.
```

此警告消息包括：

- 包含问题的文件名为 `test.c`
- 问题出在 `test.c` 的第 4 行，从第 3 个字符开始
- 该警告与变量 `x` 有关

以下是控制 `kf32clang` 诊断输出的常用选项。

表 3.8 常用诊断选项

选项	描述
<code>-Wall</code>	开启大部分常见警告
<code>-Werror</code>	将所有警告转为错误
<code>-Werror=foo</code>	只针对特定类型的警告
<code>-Wno-error=foo</code>	把特定警告恢复为普通警告，即不当作错误
<code>-Wfoo</code>	用来开启特定类型的警告
<code>-Wno-foo</code>	禁用 <code>foo</code> 类型的警告
<code>-w</code>	所有警告信息都会被完全屏蔽
<code>-Weverything</code>	开启 <code>kf32clang</code> 支持的所有警告
<code>-pedantic</code>	使用了非标准 C/C++ 时产生警告

以下是控制 `kf32clang` 诊断输出的常用选项

将以下代码示例复制到 `test.c` 中，并使用 `KungFu32` 编译器进行编译，以查看示例诊断消息。

```
#include <stdlib.h>
#include <stdio.h>
void function (int x) {
    int i;
    int y = i + x;
    printf("Result of %d plus %d is %d\n");
    call();
    return;
}
```

使用以下命令编译 `test.c`：

```
kf32clang --target=kf32r -march=kf32r -c test.c
```

默认情况下，kf32clang 会检查 printf() 语句的格式，以确保%格式说明符的数量与数据参数的数量相匹配。因此，kf32clang 会生成此诊断消息：

```
test.c:6:24: warning: more '%' conversions than data arguments
[-Wformat-insufficient-args]
printf("Result of %d plus %d is %d\n");
      ~^
```

默认情况下，kf32clang 针对.c 文件采用 gnu17 标准进行编译。该语言标准不允许隐式函数声明。因此，kf32clang 会生成此诊断消息：

```
test.c:7:5: warning: implicit declaration of function 'call' is invalid in C99
[-Wimplicit-function-declaration]
call();
  ^
```

要抑制所有警告，请使用-w:

```
kf32clang --target=kf32r -march=kf32r -c test.c -w
```

要仅抑制-Wformat 警告，请使用-Wno-format:

```
kf32clang --target=kf32r -march=kf32r -c test.c -Wno-format
```

默认情况下，有些诊断消息会被抑制。要查看所有诊断消息，请使用-Weverything:

```
kf32clang --target=kf32r -march=kf32r -c test.c -Weverything
```

3.8 编译器工具命令行选择规则

可以使用命令行选项来控制编译工具操作的多个方面。

kf32clang 与 kf32ld 选项规则

kf32clang 遵循与 clang 相同的语法规则。有些选项前加单横线-，另一些则加双横线--。有些选项要求在选项和参数之间使用=字符，另一些则要求使用空格字符。

以下规则适用，具体取决于选项的类型：

- 单横线选项

所有单字母选项（包括带有参数的单字母选项）前都有一个单横线-。选项和参数之间可以用空格分隔，也可以让参数直接跟在选项后面。例如：

```
-o
```

多字符单横线选项也存在，主要用于架构或 CPU，相关设置，例如：

```
-march=kf32r
```

- 双横线选项

关键字或平台相关设置使用双横线--，选项和参数之间需要有一个 = 或空格字符。例如：

```
--target=kf32
```

示例：

```
kf32clang --target=kf32r -march=kf32r test.c -c -o test.o
```

4 优化工作下代码

为了更好的使用 KF32 工具链的优化能力，本章将对一些 KF32 工具链可用的选项、属性以及一些编程技巧进行说明。

4.1 VOLATILE 关键字

当你使用一个不能被编译器优化的变量时，需要使用 `volatile` 关键字，如果在需要使用 `volatile` 的地方没有使用它，编译器可能会优化对变量的访问，从而生成非预期的代码，甚至移除程序中原本预期的功能。

4.1.1 volatile 的含义

将变量声明为 `volatile`，告诉编译器该变量可能在任何时间被当前实现代码之外的代码修改。例如：

- 由操作系统修改
- 由硬件修改

这种声明保证编译器不会基于“变量未被使用或未被修改”的假设对其进行优化。你也可以使用 `volatile` 来告诉编译器：某个包含内联汇编（inline assembly）的代码块具有输入列表、输出列表和 `clobber` 列表所不能完全描述的副作用。

4.1.2 何时使用 volatile

当一个变量可能会在其定义作用域之外被修改时，应当使用 `volatile` 关键字。例如，如果程序在某次计算中使用了一个全局变量，编译器通常会生成指令，将该变量的值加载到寄存器中以执行计算。如果该全局变量随后在另一段计算中再次使用，编译器可能会复用寄存器中已有的值，而不是重新从内存加载。这种复用行为的原因在于：

编译器优化器默认假设非 `volatile` 变量不会被外部实体修改。但这个假设对于内存映射外设（memory-mapped peripherals）是不成立的—— 外设寄存器的值经常会在程序未知的情况下被硬件修改。具体请参考章节“未使用 `volatile` 关键字导致的无限循环示例”。

另一个例子是：某个变量可能被用来实现 `sleep` 或定时延迟。如果该变量看起来未被使用，编译器可能会移除这段用于计时延迟的代码，除非该变量被声明为 `volatile`。

在实践中：

- 当访问内存映射外设时，必须将变量声明为 `volatile`。即使在 `-O0` 优化

等级下，也不能保证编译器会默认把所有相关变量视为 `volatile`。

- 中断处理程序中会修改的变量并在外部使用时，应使用 `volatile` 关键字修饰

4.1.3 未使用 `volatile` 时可能出现的潜在问题

当一个需要声明为 `volatile` 的变量没有声明为 `volatile`，编译器会假定该变量的值不会在其定义作用域之外被修改。因此，编译器可能会执行一些不希望发生的优化。这个问题可能以多种形式表现出来，例如：

- 在轮询硬件时，代码可能会卡在循环中
- 优化可能导致实现刻意延迟（`timing delay`）的代码被移除

4.1.4 强制使用特定指令访问内存

将变量声明为 `volatile` 并不能保证编译器在访问该变量时使用某条特定的机器指令。仅仅在变量或指针类型上添加 `volatile` 修饰符，并不能保证编译器会自动选择该寄存器所要求的访问方式。如果你的代码需要特定访问方式的内存映射位置，那么 仅声明为 `volatile` 是不够的。你还必须使用 `__asm__` 内联汇编语句，并在其中显式写入你所需要的加载或存储指令。

4.1.5 未使用 `volatile` 关键字导致的无限循环示例

下表中的两个示例说明了 `volatile` 关键字应该如何使用。

表 4.1 `buffer` 循环的非 `volatile` 和 `volatile` 版本 C 语言代码

buffer 循环的非 <code>volatile</code> 版本	buffer 循环的 <code>volatile</code> 版本
<pre>int buffer_full; int read_stream(void) { int count = 0; while (!buffer_full) { count++; } return count; }</pre>	<pre>volatile int buffer_full; int read_stream(void) { int count = 0; while (!buffer_full) { count++; } return count; }</pre>

这两个例程都在循环中递增计数器，直到状态标志 `buffer_full` 被设置为 `true`。
`buffer_full` 的状态可能会异步地随程序流程变化。

左边的示例没有将变量 `buffer_full` 声明为 `volatile`，所以编译器只会从内存中获取一次该变量，然后将其存储在寄存器中，接下来每次访问该变量直接使用目标寄存器中的值，因此是错误的。右边的示例将变量 `buffer_full` 声明为 `volatile`，编译器每次获取该变量都会重新从内存中加载该变量，因此是正确的。

4.2 循环优化

循环的执行时间可能会随着迭代次数的增加而显著增长。如果每次迭代都要检查循环条件，这种开销可能会降低循环的执行性能。

4.2.1 循环展开

你可以通过展开部分循环迭代来减少循环条件检查带来的开销，从而降低条件判断的次数。在源代码中，可以使用如下指令来展开时间关键的循环：

```
#pragma unroll (n)
```

然而，展开循环的缺点是会增加代码体积。这些 `#pragma` 指令仅在 `-O2`, `-Os` 优化等级下有效。

表 4.2 循环展开的 `pragma` 指令

Pragma	描述
<code>#pragma unroll (n)</code>	展开循环的 n 次迭代
<code>#pragma unroll_completely</code>	展开循环的全部迭代

4.2.2 C 代码中的循环终止

如果编写不当，循环的终止条件可能会导致显著的开销。尽可能地应遵循以下做法：

- 使用简单的终止条件
- 编写递减至零（count-down-to-zero）循环，并测试是否等于零
- 使用 `unsigned int` 类型的计数器

遵循其中一条或多条指南（单独或组合使用）通常能够生成更高效的代码。

以下实例展示了循环不展开和循环展开所生成代码的不同：

无循环展开的代码：

```
int countSetBits1(unsigned int n) {  
    int bits = 0;  
    while (n != 0) {  
        if (n & 1)  
            bits++;  
        n >>= 1;  
    }  
    return bits;  
}
```

有循环展开的代码:

```
int countSetBits2 (unsigned int n) {  
    int bits = 0;  
    #pragma unroll (4)  
    while (n != 0) {  
        if (n & 1)  
            bits++;  
        n >>= 1;  
    }  
    return bits;  
}
```

使用下面的命令编译上面的代码:

```
kf32clang --target=kf32r -march=kf32r -O2 -S file.c
```

生成的汇编语言文件如下所示：

未循环展开的代码：

```
countSetBits1:
    CMP R0, #0
    JZ .LBB0_4
    MOVL R4, #0
    MOVL R2, #1
    MOV R3, R0
.LBB0_2:
    ANL R5, R0, R2
    ADD R1, R4, R5
    LSR R3, #1
    CMP R0, #1
    MOV R4, R1
    MOV R0, R3
    JHI .LBB0_2
    MOV R0, R1
    JMP LR
.LBB0_4:
    MOVL R0, #0
    JMP LR
```

循环展开后的代码：

```
countSetBits2:
    MOVL R1, #0
    CMP R0, #0
    JZ .LBB0_6
    MOVL R2, #1
.LBB0_2:
    ANL R4, R0, R2
```

```

ADD R3, R1, R4
CMP R0, #2
JNC .LBB0_7
MOV R1, R0
LSR R1, #1
ANL R4, R1, R2
ADD R1, R3, R4
CMP R0, #4
JNC .LBB0_6
MOV R3, R0
LSR R3, #2
ANL R4, R3, R2
ADD R3, R1, R4
CMP R0, #8
JNC .LBB0_7
MOV R1, R0
LSR R1, #3
ANL R4, R1, R2
ADD R1, R3, R4
MOV R3, R0
LSR R3, #4
CMP R0, #15
MOV R0, R3
JHI .LBB0_2
.LBB0_6:
MOV R0, R1
JMP LR
.LBB0_7:
MOV R0, R3
JMP LR

```

通过比较可以发现，使用循环展开的函数 `countSetBits2`，生成的代码运行速度更快但是体积也更大。

4.3 函数内联

`kf32clang` 会判断内联某个函数是否可以提升性能，当可以提升性能时会自动对该函数进行内联。这种自动内联通常不会显著增加代码体积，并且你可以使用编译器提示（`hints`）和选项来影响或控制函数是否被内联。

表 4.3 函数内联

内联选项，关键字，或者属性	描述
<code>__inline__</code>	在函数定义或声明中使用该关键字，可作为提示（ <code>hint</code> ），告诉编译器优先考虑将该函数内联。然而，对于每次函数调用，编译器仍然会自行决定是否真正进行内联。
<code>__attribute__((always_inline))</code>	在函数定义或声明中使用此函数属性，可告诉编译器总是内联该函数，但存在某些例外情况，例如递归函数可

	能无法内联。该属性会覆盖 <code>-fno-inline-functions</code> 选项的设置。
<code>__attribute__((noinline))</code>	在函数定义或声明中使用此函数属性,可告诉编译器不要内联该函数。该属性等效于 <code>__declspec(noinline)</code> 。
<code>-fno-inline-functions</code>	这是一个编译器命令行选项。在编译时指定该选项,可以禁止函数内联。该选项会覆盖 <code>__inline__</code> 提示。

4.3.1 函数内联示例

此示例展示了 `__attribute__((always_inline))` 和 `-fno-inline-functions` 在 C99 模式下的效果 (C 文件的默认行为)。将以下代码复制到 `file.c`。

```
int bar(int a)
{
    a=a*(a+1);
    return a;
}
__attribute__((always_inline)) static int row(int a)
{
    a=a*(a+1);
    return a;
}
int foo (int i)
{
    i=bar(i);
    i=i-2;
    i=bar(i);
    i++;
    i=row(i);
    i++;
    return i;
}
```

在示例代码中,函数 `bar` 和 `row` 是相同的,但函数 `row` 总是被内联。使用以下编译器命令,在 `-O2` 优化级别下分别编译:使用 `-fno-inline-functions`,不使用 `-fno-inline-functions`。

```
kf32clang --target=kf32r -march=kf32r -S file.c -O2 -o file_no_inline.s
-fno-inline-functions
```

```
kf32clang --target=kf32r -march=kf32r -S file.c -O2 -o file_inline.s
```

生成的代码如下表所示：

表 4.4 -fno-inline-functions 编译结果对比

使用-fno-inline-functions 的编译结果	不使用-fno-inline-functions 的编译结果
<pre>foo: PUSH LR LJMP bar SUB R0, R0, #2 LJMP bar ADD R1, R0, #1 ADD R0, R0, #2 MULS R2, R0, R1 ADD R0, R2, #1 POP LR JMP LR</pre>	<pre>foo: ADD R1, R0, #1 MULS R2, R1, R0 SUB R0, R2, #2 SUB R1, R2, #1 MULS R2, R1, R0 ADD R0, R2, #1 ADD R1, R2, #2 MULS R2, R1, R0 ADD R0, R2, #1 JMP LR</pre>

使用 -fno-inline-functions 编译时，编译器不会内联函数 bar。不使用 -fno-inline-functions 编译时，编译器会内联函数 bar。编译器总是内联函数 row，即使它与函数 bar 内容相同。

4.4 C 和 C++中的栈使用

C 和 C++ 都会使用栈（stack）。例如，栈中通常保存以下内容：

- 函数的返回地址。
- 必须保存的寄存器，这些由 KF32 编译器的调用约定决定。
- 局部变量，包括局部数组、结构体和联合体。
- C++ 中的类实例

有些栈使用情况并不直观，例如：

- 如果局部整型或浮点变量溢出寄存器（spilled），也就是未分配到寄存器，它们会分配到栈上。
- 结构体通常分配在栈上。栈上会保留一块大小为 sizeof(struct) 的空间。但是，编译器也可能尝试将结构体分配到寄存器中。
- 如果数组的大小在编译时已知，编译器会将其分配到栈上。栈上会保留一块大小为 sizeof(array) 的空间
- 一些优化可能会引入新的临时变量，用于保存中间结果。这些优化包括：公共子表达式消除（CSE, Common Subexpression Elimination）、活跃区间拆分（Live Range Splitting）、结构体拆分（Structure Splitting），编

译器会尝试将这些临时变量分配到寄存器中。如果寄存器不足，则会将其溢出到栈上。

- KF32 编译器要求某些函数参数通过栈传递，而不是寄存器传递。具体取决于参数的类型、大小以及参数顺序。

4.4.1 减少栈使用量的方法

在嵌入式系统中，栈空间有限，因此在编写应用程序时，尽量减少栈使用量可以提高系统的可靠性和性能。以下是常用方法：

- 编写只需要少量变量的小函数
- 避免使用大型局部结构体和数组
- 避免递归
- 尽量减少函数中每个时刻正在使用的变量数量
- 使用 C 语言的块作用域语法，并仅在需要时声明变量，以便不同作用域可以共享同一块内存

4.5 紧凑数据空间分配

通过将数据打包到结构体中，可以减少应用程序所需的内存。这在嵌入式系统中尤其重要，尤其是当你需要存储和访问大型数据数组时。

如果结构体中的各个成员没有打包，编译器可能会根据每个成员的自然对齐方式，在结构体内部添加填充字节，以加快对单个成员的访问速度。

kf32clang 提供了 `pragma` 和属性，用于在结构体或联合体中打包成员，而不添加任何填充字节。

表 4.5 在结构体或联合体中打包成员

Pragma 或 attribute	描述
<code>#pragma pack (n)</code>	对于每个成员，如果 <code>n</code> 字节小于该成员的自然对齐方式，则将对齐方式设置为 <code>n</code> 字节；否则，使用该成员的自然对齐方式。
<code>__attribute__((packed))</code>	这等同于 <code>#pragma pack(1)</code> 。不过，该属性也可以用于结构体或联合体中的单个成员。

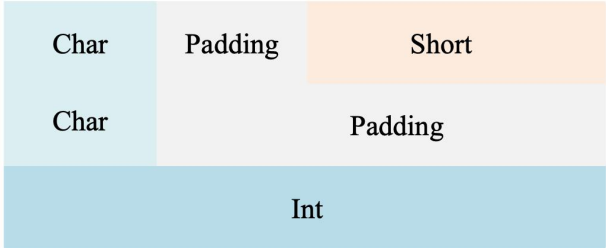
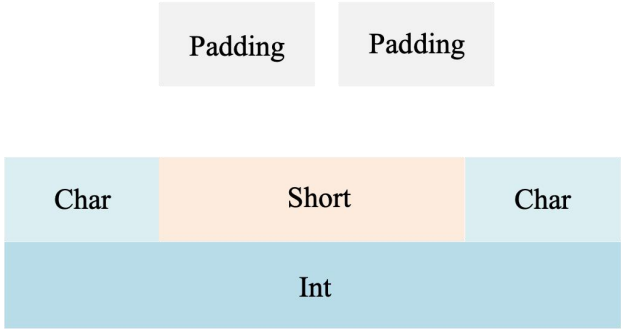
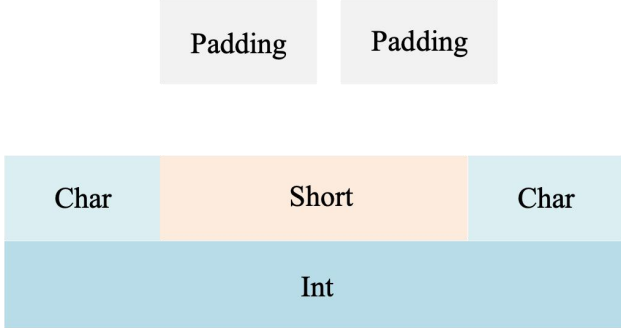
4.5.1 打包整个结构体

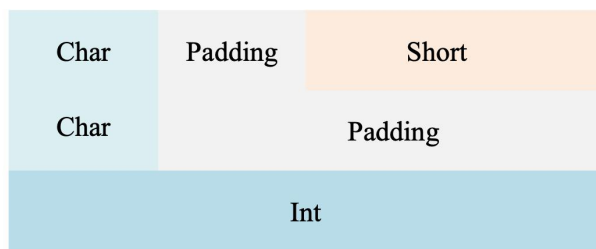
要打包整个结构体或联合体，可以在结构体声明中使用 `__attribute__((packed))` 或 `#pragma pack(n)`，如代码示例所示。该属性和 `pragma` 会应用于结构体或联合体的所有成员。如果某个成员本身是结构体，则该结构体的对齐方式为 1 字节，但该结构体内部的成员仍保持它们的自然对齐

方式。

使用 `#pragma pack(n)` 时，结构体的对齐方式为应用 `#pragma pack(n)` 后结构体中最大成员的对齐方式。

表 4.6 打包结构体

代码	打包结构体
<pre> struct stc { char one; short two; char three; int four; } c,d; int main (void) { c.one=1; return 0; } </pre>	打包:  结构体大小: 12.结构体的对齐方式为最大成员的自然对齐方式。在此示例中，最大成员是一个 int。
<pre> struct __attribute__((packed)) stc { char one; short two; char three; int four; } c,d; int main (void) { c.one=1; return 0; } </pre>	打包:  结构体大小: 8.这个结构体的对齐为 1。
<pre> #pragma pack (1) struct stc { char one; short two; char three; int four; } c,d; int main (void) { c.one=1; return 0; } </pre>	打包:  结构体大小: 8.这个结构体的对齐为 1。
<code>#pragma pack (2)</code>	打包:

<pre> struct stc { char one; short two; char three; int four; } c,d; int main (void) { c.one=1; return 0; } </pre>	 结构体大小: 10.这个结构体的对齐是 2 对齐。
<pre> #pragma pack (4) struct stc { char one; short two; char three; int four; } c,d; int main (void) { c.one=1; return 0; } </pre>	打包:  结构体大小: 12.这个结构体的对齐是 4 对齐。

4.5.2 在结构体中打包单个成员

要打包结构体的单个成员，可以在该成员上使用 `__attribute__((packed))`。这会将该成员对齐到字节边界，从而减少整个结构体所需的内存。它不会影响其他成员的对齐方式。因此，整个结构体的对齐方式仍等于最大成员在未使用 `__attribute__((packed))` 时的自然对齐方式。

表 4.7 打包结构体中单个成员

代码	打包结构体
<pre> struct stc { char one; short two; char three; int __attribute__((packed)) four; } c,d; int main (void) { c.one=1; return 0; } </pre>	打包:  结构体大小: 10. 结构体的对齐方式为 2 字节，因为最大成员在未使用 <code>__attribute__((packed))</code> 时的自然对齐方式为 2 字节。

4.5.3 访问结构体中被打包的成员

如果结构体或联合体的某个成员被打包，因此不具有其自然对齐方式，那么

访问该成员时必须通过包含它的结构体或联合体进行。不能获取该打包成员的地址作为指针使用，因为指针可能未对齐。即使目标平台支持未对齐访问，解引用这样的指针也可能不安全，因为某些指令始终要求地址按字对齐。

```
struct __attribute__((packed)) foobar
{
    char x;
    short y;
};
short get_y(struct foobar *s)
{
    // Correct usage: the compiler will not use unaligned accesses
    // unless they are allowed.
    return s->y;
}
short get2_y(struct foobar *s)
{
    short *p = &s->y; // Incorrect usage: 'p' might be an unaligned pointer.
    return *p; // This might cause an unaligned access.
}
```

4.6 针对代码大小或性能进行优化

KF32 工具链会采用多种技术进行代码优化，其中一些技术可提升代码性能，而另一些技术则能减小代码体积。不同的优化存在一定的冲突，也就是说，提高代码性能的技术可能会导致代码体积增大，而减小代码体积的技术可能会降低性能。例如，编译器可以展开小循环以获得更高的性能，但缺点是代码体积会增加。

- 默认优化级别为-O0，在-O0 级别下，kf32clang 引入少量优化。
- 代码性能优化相关选项：

表 4.8 代码性能优化相关选项

优化选项	作用
-O1 -O2 -O3	指定编译源文件时要使用的优化级别。数字越大，表示为提升性能而进行的优化级别越高。

- 代码体积优化相关选项：

表 4.9 代码体积优化相关选项

优化选项	作用
-Os	以可能增加执行时间为代价，对代码进行优化以减小其大小。此选项旨在实现代码大小缩减与快速性能之间的平衡。
-Oz	旨在减小代码大小。
-fshort-enums	允许编译器将枚举类型的大小设置为能够容纳所有枚举值的

	最小数据类型。
<code>-fno-exceptions</code>	仅适用于 C++。禁用生成支持 C++ 异常所需的代码。
<code>-fno-rtti</code>	仅支持 C++。禁用生成支持运行时类型信息（RTTI）功能所需的代码。

此外，在编码过程中所做的选择可能会影响优化。例如：

（1）优化循环终止条件可以同时改善代码大小和性能。特别是，使用计数器递减到零的循环通常比使用递增计数器的循环生成的代码更小、运行更快；

（2）通过减少循环迭代次数来手动展开循环，同时增加每次迭代的工作量，这种方式可以以增大代码规模为代价提升性能；

（3）使用内联函数需要在代码大小和性能之间进行权衡；

4.7 优化对调试的影响

优化处理可能导致调试视图不友好，目标代码到源代码的映射存在多处映射。根据需求可以合理选择优化等级与优化选项。

5 汇编文件输出

本章介绍如何使用 clang 汇编源代码。

5.1 汇编示例

KF32 工具链兼容 GNU 语法的汇编伪指令。关于更多的伪指令，可参考 GNU Binutils 文档。

以下示例展示了汇编代码，用于计算两个变量的和。

```
.text
.file "add.c"
.globl add
.p2align 3
.type add,@function
add:
    ADD r0, r1, r0
    JMP lr
```

使用 kf32clang 来汇编语言源代码。通常，调用 kf32clang 的步骤如下：

```
kf32clang --target=kf32r -o file.o file.s
```

5.2 汇编流程

默认情况下，clang 使用汇编代码源文件后缀来决定是否运行 C 预处理器：

- `.s`（小写）后缀表示不需要预处理的汇编代码。
- `.S`（大写）后缀表示需要预处理的汇编代码。`-x` 选项允许显式指定后续输入文件的编程语言来覆盖默认配置，而不是通过文件后缀进行推断。具体来说，`-x assembler-with-cpp` 表示汇编代码包含 C 预处理器指令，clang 必须运行 C 预处理器。`-x` 选项只适用于命令行中在该选项之后的输入文件，一旦使用了 `-x` 选项，它就会为其后的所有输入文件设置编程语言，直到遇到下一个 `-x` 选项或文件列表结束。

注意：不要将 `.ifdef` 汇编器指令与预处理器 `#ifdef` 指令混淆：

- 预处理器 `#ifdef` 指令检查是否存在预处理器宏。这些宏通过 `#define` 预处理器指令或 clang 命令行选项 `-D` 定义。
- clang 汇编 `.ifdef` 指令检查代码符号。这些符号通过标签或 `.set` 指令定义。

汇编 `.S` 文件时，预处理器先运行，对源代码进行文本替换，对 `#ifdef` 指令的处理阶段。之后源代码会传递给汇编器，对 `.ifdef` 指令进行处理。

要预处理汇编代码源文件，请执行以下其中之一：

1. 确保汇编代码文件名为 `.S` 后缀，然后使用：

```
kf32clang --target=kf32r -march=kf32r test.S -c -o test.o
```

2. 使用 `-x assembler-with-cpp` 选项来告诉 clang 汇编源文件需要预处理。
当你已有带有小写后缀的源文件时，这个选项非常有用。例如：

```
kf32clang --target=kf32r -march=kf32r -x assembler-with-cpp test.s -c -o test.o
```

6 在 C 或 C++代码中使用内嵌汇编

6.1 编写内嵌汇编代码

编译器支持在 C 或 C++ 源代码中编写汇编代码，例如访问 C 或 C++ 无法提供的目标处理器特性。

`__asm` 关键字可以使用内嵌汇编语法将汇编代码集成到函数中。例如：

```
#include <stdio.h>
//高级嵌汇编
int add(int i, int j) {
    int res = 0;
    __asm("ADD %0, %1, %2]")
    : "=r" (res)    // 输出操作数
    : "r" (i), "r" (j) // 输入操作数
    return res;
}
//普通嵌汇编
int add1 __attribute__((noinline))(int i, int j) {
    __asm("ADD r0, r0, r1");
}
int main(void) {
    int a = 1;
    int b = 2;
    int c = 0;
    c = add(a, b);
    printf("Result of %d + %d = %d\n", a, b, c);
}
```

使用内嵌汇编而非单独编写 `.s` 文件，可以实现嵌汇编指令完成与 c 代码中的变量等快速进行交互。

6.1.1 内嵌汇编语句的结构

在 C 语言的函数中嵌入一段汇编语言程序段是编译器提供的“asm”功能，形式看起来陌生，它不是标准 C 所定义的形式，而是编译器对 C 语言的扩充，支持传递变量，比单纯的汇编语言代码要复杂，涉及到怎么分配和使用寄存器，以及 C 代码中的变量应该存放在哪个寄存器中。

嵌汇编的一般形式为：

```
__asm__ __volatile__("<asm routine>":output:input:modify);
```

其中__asm__表示汇编代码的开始，__volatile__是可选项，作用为避免asm指令被删除、移动或组合。<asm routine>为汇编指令部分，支持数字前加前缀“%”，如%0，%1，%2等表示使用寄存器的样板操作数。指令中有几个操作数，就说明有几个变量需要与寄存器结合，由编译器根据后面输出部分和输入部分的约束条件进行相应的处理。

输出部分：用于规定输出变量如何与寄存器结合的约束，可以有多个约束，互相以逗号分开。每个约束以“=”开头，接着用一个字母表示操作数类型，然后是变量的结合约束，如：“=r”(__dummy)，r表示目标操作数可以使用任一个通用寄存器，__dummy为定义的C变量。如果为：“=m”(__dummy)表示目标操作数是存放在内存单元__dummy中。

输入部分：与输出部分相似，但没有“=”。如果输入与输出部分约束使用同一个寄存器，将对于的操作数编号（如“0”，“1”，“2”等）放在约束条件中。存在多个输入使用逗号间隔。

修改部分：通常以“memory”作为约束条件，表示操作完成后内存中的内容已有改变。如果原来某个寄存器的内容来自内存，那么现在内存中这个单元的内容已经改变。当输入“r0”，“r1”时，即当前语句分配寄存器不使用R0和R1，针对语句中使用固定寄存器时有作用。

注意：指令部分为必选项，其他为可选项，当输入部分存在，而输出部分不存在时，分号：要保留，当仅修改部分“memory”存在时，3个分号都要保留。因为宏定义的关系，__asm__可以只写asm，__volatile__只写volatile。

主要约束字母及含义如下：

表 6.1 主要约束字母及含义

字母	含义
m	表示内存单元
r	表示任何通用寄存器
l、h	表示直接操作数
E、F	表示浮点数
g	表示“任意”
I	表示常数（0~31）

6.1.1.1 普通字符串形式

第一种，编译结果所有汇编指令为一行，每条指令需用分号分隔。该形式通过连接符告知功能代码在1行。如果需要占用的不同的行参加下面示例。

```
asm(  
    asm_instruct1;\n    asm_instruct2;\n    asm_instruct3;\n    ");
```

第二种，如果需要占用的不同的行参加下面示例。

```
sm(  
    "asm_instruct1"  
    "\n\t"  
    "asm_instruct2"  
    "\n\t"  
    "asm_instruct3"  
    );
```

第三种编译结果每条汇编指令为一行，\n\t 用于每一指令的分隔，也可为如下格式：

```
asm(  
    "asm_instruct1\n\t"  
    "asm_instruct2\n\t"  
    "asm_instruct3\n\t"  
);
```

6.1.1.2 嵌汇编注意事项

1.用户在使用 R6-R13 等前，必须对其进行入栈处理，结束时再出栈，以保护其中数据不丢失(确认工具默认已保护的可忽略)，使用时用户可参考使用 PUSH list/POP list 指令。POP list 不能使 LR(即 R13)出栈，应先使用 POP list 将 LR 之外的寄存器出栈，再 POP LR(若 LR 被入栈)。

2.KF32 编译器中在 O2 等级优化下(默认为 O2 等级)，R0-R5 中可能存有中间数据，为防止数据丢失，建议将嵌汇编内容放置在一个新的子函数中，因此子函数内 R0-R5 可自由使用，R6-R13 等根据需要使用前需保护，否则 R0-R5 也可能需要保护，否则会存在运行错误。

3.嵌汇编中可通过 C 语言的变量名、函数名直接获取变量地址、函数地址。汇编文件访问 C 文件中的全局变量、函数时，需要先在汇编文件中使用 .extern 声明外部变量，而后可直接使用变量名、函数名作为其地址。

4.汇编文件中未初始化的全局变量定义为 .common a,1，其中数值 1 表示一个字节长度。C 文件可通过 extern 声明汇编文件的变量为外部变量后使用。

5.汇编文件中函数需使用 .export 声明，如：.export function，C 文件可通过 extern 声明外部变量后使用。

6.C 文件中的宏定义与汇编中 EQU 无法通用，需分别定义。

7.若在嵌汇编中使用芯片特殊功能寄存器，则需引用汇编头文件，嵌汇编语句可在函数外使用，示例如：`asm(".include \"KF32XXXX.inc\"");`// 汇编包含头文件

包含时需要增加选项传递头文件路径，否则需要全路径 include。工具选项 -Wa,-I"XXXXX/include"负责 C 项目将编译选项传递头文件路径给汇编器。在使用 ChipON IDE 的开发环境时，默认已经添加了路径传递。如果使用外设库开发模式，不建议使用该头文件中定义，根据需要可仿照该头文件进行内容的声明，否则符号定义可能与外设库冲突。

8.嵌汇编中使用全局变量时，先使用 LD R5,#global 获取变量的内存地址，再使用 LD/ST.[whb]指令进行加载/存储；

9.函数功能参数规则：R0-R4 (堆栈)用于传递参数，R0、R1（堆栈）存放返回值，LR 自动跟踪函数返回地址。内嵌函数可根据情况自定义；

10.嵌汇编的 sp 操作(修改,如申请空间)，若汇编嵌入在 c 函数中，c 函数的局部变量仍基于 sp 和编译时分配的偏移值进行操作。即嵌汇编 sp 的临时修改与申请须在 c 作用时还原。不支持使用伪指令 MOV SP,#address 的修改，拆分 MOVL 和 MOVH 实现赋值时，若中间被中断打断，中断的现场保护和中断下的函数的栈使用均会出现异常。即 SP 高位为 0 待更新的非法栈区。

6.1.2 符号与标签的定义

内嵌汇编可以用来定义符号。例如：

```
__asm (".global __use_no_semihosting\n\t");
```

要定义标签，请在标签名称后使用：例如：

```
__asm ("my_label:\n\t");
```

6.2 从 C 和 C++调用汇编函数

通常，单个应用程序的所有代码都用相同的源代码编写。这通常是高级语言，如 C 或 C++。这些代码随后被编译成 KF32 汇编代码。

不过，在某些情况下，你可能想从 C/C++ 代码中调用汇编代码。例如：

- 如果你想利用现有的汇编代码，但项目的其他部分是用 C 或 C++编写的。
- 如果你想直接用汇编代码手动编写关键函数，这样可以生成比编译 C 或 C++代码更优化的代码。
- 如果你想直接与设备硬件接口，并且在底层汇编代码中比高级 C 或 C++ 更容易。

要从 C 或 C++ 中调用汇编函数：

1. 在汇编源代码中，将代码声明为全局函数，使用 `.global` 和 `.type`：

```
.globl    add
.p2align  3
.typeadd,@function
add:      // 函数入口
    ADD r0, r0, r1    // 函数参数在 r0 和 r1 寄存器中, 将它们加在一起并将结果放入 r0
    JMP lr           // 跳转到链接寄存器中保存的返回地址
```

在 C 代码中, 使用 `extern` 声明外部函数:

```
#include <stdio.h>
extern int add(int a, int b);
int main() {
    int a = 4;
    int b = 5;
    printf("Adding %d and %d results in %d\n", a, b, add(a, b));
    return (0);
}
```

在 C++ 代码中, 使用 `extern "C"` :

```
extern "C" {int add(int a, int b);}
```

1. 确保您的汇编代码符合 KF32 架构的函数调用约定；
2. 编译两个源文件：

```
kf32clang --target=kf32r -march=kf32r main.c myadd.s -T kf32demo.ld -o main.o
```

7 编译工具汇编器

本章介绍 KF32 工具链提供的 clang 的汇编器功能。

7.1 汇编器的主要特性

clang 汇编器支持指令、伪指令以及用户定义的宏，具体包括以下内容：

- KungFu32 通用指令集的汇编语言
- KungFu32 指令集提供的伪指令
- GNUC 常规通用伪指令及 Clang 汇编伪指令
- 用户定义的宏

7.2 命令行语法

```
kf32as [option...] [asmfile...]
```

7.3 常用伪指令介绍

7.3.1 .section

section 用于指定类型。

在 C 语言中，将函数或变量放入由 “name” 指定的段。

例如，`void __attribute__((section( “.new_sect1 ” ))) foo() {return;}`

函数 foo 将被放入.new_sect1 段。

`unsigned int var __attribute__((section( “.new_sect2 ” )))`

变量 var 将被放入.new_sect2 段。

常见的段定义为 text (flash 空间)， data(RAM 空间)。

在汇编语言中，直接使用伪指令 section 进行段类型定义，如：

```
.section .text$_NMI_exception  
.section .text  
.section .indata
```

需要注意的是汇编语言没有准确的函数概念，即 `section` 对后面的代码仍然有效，除非遇到新的 `section` 声明。

7.3.2 `.global symbol`,

`.global` 使符号 `symbol` 对连接器 `ld` 可见。如果您在局部过程中定义符号 `symbol`，其它和此的局部过程都可以访问它的值。另外，`symbol` 从连接到本过程的另一个文件中的同名符号获取自己的属性。另外可以使用 `.export` 进行辅助变量的描述。

7.3.3 `.align abs-expr, abs-expr, abs-expr`

增加位置计数器(在当前的子段)使它指向规定的存储边界。

第一个表达式参数(结果必须是纯粹的数字)是必需参数：边界基准,见后面的描述。

第二个表达式参数(结果必须是纯粹的数字)给出填充字节的值，用这个值填充位置计数器越过的地方。这个参数(和逗号)可以省略，如果省略它，填充字节的值通常是 0。但在某些系统上，如果本段标识为包含代码，而填充值被省略，则使用 `no-op` 指令填充这个空间。

第三个参数表达式的结果也必须.是纯粹的数字，这个参数是可选的。如果存在第 3 个参数，它代表本对齐命令允许越过字节数的最大值。如果完成这个对齐需要跳过的字节比指定的最大值还多,则根本无法完成对齐。您可以在边界基准后简单地使用两个逗号，以省略填充值参数(第二参数)；如果您想在适当的时候，对齐操作自动使用 `no-op` 指令填充，这个方法将非常奏效。

第一个表达式是边界基准，单位是字节。例如 `‘.align 8’` 向后移动位置计数器至 8 的倍数。如果地址已经是 8 的倍数，则无需移动。

7.3.4 `.byte expressions`

`.byte` 可不带参数或者带多个表达式参数，表达式之间由逗号分隔。每个表达式参数都被汇编成下一个字节。

7.3.5 `.word expressions`

本命令可不带表达式或带多个表达式，这些表达式可以属于任意段，每个表达式由逗号分隔。

汇编生成的数字的大小，字节顺序视生成程序运行的目标机器而定。

7.3.6 `.short expressions`

本命令通常和 `‘.word’` 命令一样。

7. 3. 7 .int expressions

可以不带参数或带多个 expressions,参数之间由逗号分隔。每个 expressions 都生成一个数字,这个数字等于表达式在目标机器运行时的值。字节顺序和数字的位数视汇编的目标机器而定。

7. 3. 8 .long expressions

.long 是.int 的等价命令。

7. 3. 9 .fill repeat, size, value

该汇编指令会产生数个(repeat 个)大小为 size 字节的重复拷贝。大小值 size 可以为 0 或者某个值,但是若 size 大于 8,则限定为 8。每个重复字节内容取自一个 8 字节数。高 4 字节值为 0,低 4 字节的数值 value。这 3 个参数都是绝对值, size 和 value 是可选的。如果第二个逗号和 value 省略, value 默认值为 0 值;如果后连个参数都省略,则 size 默认值为 1。如.fill 1,1 对应一个字节的 0 值,往往对应全局未初始化变量的内容表达。

7. 3. 10 .type name , type description

本命令经常用来设置符号 name 的类型(属性)为函数符号或是目标符号两者之一。type description 部分允许使用 5 种不同的语法,以兼容众多的汇编器。这些语法是:

.type <name>,#function

.type <name>,#object

.type <name>,@function

.type <name>,@object

.type <name>,%function

.type <name>,%object

.type <name>,"function"

.type <name>,"object"

.type <name> STT_FUNCTION

.type <name> STT_OBJECT

7. 3. 11 .size name , expression

本命令经常用来设置符号 name 的内存大小。内存大小的单位是字节,通过计算参数 expression 得到,参数 expression 中可以使用标签进行计算。本命令常用来设置函数符号的长度。

7. 3. 12 .if absolute expression

.if 标志着一段代码的开始,这段代码只有在参数 absolute expression(必须

是一个独立的表达式)不为 0 时才进行汇编。这段条件汇编代码必须使用`.endif` 标志结束。另外,可以使用`.esle` 来标记一个代码块,这个代码块与前面那段代码只有一个会进行汇编。如果您需要检查数个汇编条件,可以在使用`.elseif` 命令,以避免在`.else` 代码块中进行 `if/else` 语句块的嵌套。

同样可以使用下面`.if` 的变体:

`.ifdef symbol`

如果指定的符号 `symbol` 已经定义过,汇编下面那段代码。

`.ifc string1,string2`

如果两个字符串相同的话,汇编下面那段代码。字符串可以可选地使用单引号。如果不使用引号则第 1 个字符串在逗号处结束。第 2 个字符串在本行末结束。包含空白的字符串应该使用引号标注。字符串比较时是区分大小写的。

`.ifeq absolute expression`

如果参数的值为 0,汇编下面那段代码。

`.ifeqs string1,string2`

这是`.ifc` 的另一种形式,字符串必须使用双引号标注。

`.ifge absolute expression`

如果参数的值大于等于 0,汇编下面那段代码。

`.ifgt absolute expression`

如果参数的值大于 0,汇编下面那段代码。

`.ifle absolute expression`

如果参数的值小于等于 0,汇编下面那段代码。

`.iflt absolute expression`

如果参数的值小于 0,汇编下面那段代码。

`.ifnc string1,string2.`

类似与`.ifc`,不过使用反向的测试: 如果两个字符串不相等的话,汇编下面那段代码。

`.ifndef symbol`

`.ifnotdef symbol`

如果指定的符号 `symbol` 不曾被定义过,汇编下面那段代码。上面两种写法是等效的。

`.ifne absolute expression`

如果参数的值为不等于 0,汇编下面那段代码。(换句话说,这是`.if` 的另一种写法)。

`.ifnes string1,string2`

类似与`.ifeqs`,不过使用反向的测试: 如果两个字符串不相等的话,汇编下面那段代码。

7.3.13 `.else`

`.else` 是支持 `as` 进行的条件汇编指令之一;如果前面`.if` 命令的条件不成立,则表示需要汇编`.else` 后的一段代码。

7. 3. 14 .elseif

.elseif 是支持 as 进行的条件汇编指令之一。它可以在.esle 段中快速产生一个新的.if 块。

7. 3. 15 .endif

.endif 是支持 as 进行的条件汇编的指令之一.它标志着条件汇编代码块的结束。

7. 3. 16 .set symbol, expression

设置 symbol 为 expression。这将改变 symbol 的值域和类型领域以符合 expression 参数。如果 symbol 已被标志为 external，则 symbol 保持它的标志。

您可以在同一个汇编程序中多次使用.set 命令来设置同一个符号。

如果设置一个全局符号，该符号在目标文件中值为最后设定的值。

7. 3. 17 .end

.end 标记着汇编文件的结束。as 不处理.end 命令后的任何语句。

8 编译工具链接器

链接器将一个或多个目标文件的内容与一个或多个目标库的选定部分组合起来，以生成目标程序。

链接器支持以下功能：

- 链接 KungFu32 内核专用指令集代码；
- 按需生成交互过渡代码，用于函数调用时在不同指令集模式之间切换；
- 按需生成范围扩展过渡代码，突破分支指令和地址访问的默认偏移限制，扩展访问范围；
- 根据待链接目标文件的构建属性（如调试 / 发布模式、浮点支持情况），自动选择适配的标准 C/C++ 库变体（针对 KungFu32 的内存占用和能效进行优化）；
- 通过命令行选项或带 KungFu32 专用内存区域定义的链接脚本，链接脚本组织详见[脚本组成说明文档](#)，将代码和数据定位到 KungFu32 系统内存映射的指定位置（如 Flash、SRAM、外设寄存器区域）；
- 对读写（RW）数据进行压缩，并对只读（RO）数据进行去重，最大限度减少 Flash 内存占用；
- 剔除未使用的段（死代码 / 数据消除），并合并重复段，减小输出目标文件的体积；
- 控制输出文件中调试信息（如 DWARF 3/4 格式）的生成，兼容 KungFu32 调试工具；
- 控制输出目标文件中符号表的内容（如保留 / 移除局部符号），平衡可调试性和目标文件大小；
- 显示输出文件中代码（TEXT 段）、数据（DATA/RW 段）和未初始化数据（BSS 段）的详细大小，并按 KungFu32 专用内存区域进行拆分展示；
- 支持 checksum。

8.1 命令行语法

该链接器命令行可接收多个输入文件，以及用于指定文件处理方式的选项。调用的命令格式为：

```
kf32ld [options] file
```

其中：

[options]

命令行选项，用于配置链接行为（如内存布局控制、优化策略、输出格式指定等）。

file

以空格分隔的目标文件、库文件或符号定义（symdefs）文件列表。

8.2 链接器的输入

链接器可接受库文件、脚本文件、对象文件。

8.3 链接器的输出

链接器输出 elf 文件与 map 文件。后续可通过 kf32objcopy 提取 hex 文件或 s19 文件，通过 kf32objdump 获取 elf 内容或反汇编。

8.4 链接器输出布局

支持通过参数-T 传递脚本文件，组织文件或段名或输出顺序。可通过 map 文件进行观察。脚本组成语法见[脚本组成说明文档](#)。

9 编译工具库管理

本章介绍 KF32 工具链提供的 kf32ar 库管理工具。

9.1 命令行语法

kf32ar 命令提供了用于指定为文件及库处理方式的选项。
其命令格式为：

```
kf32ar [options] [-]<operation>[modifiers] [relpos] [count] <archive> [files]
```

options

kf32ar 命令行选项

Operation modifiers

操作与修饰符，指定执行什么操作以及如何执行

d - delete [files] from the archive
m - move [files] in the archive
p - print contents of [files] found in the archive
q - quick append [files] to the archive
r - replace or insert [files] into the archive
s - act as ranlib
t - display list of files in archive
x - extract [files] from the archive

relpos

库文件现有成员的名称，用于指定操作的（相对）位置

count

指定最多提取多少个与 `fies` 匹配的成员

archive

指定待操作的库文件文件名

files

指定受操作影响的文件列表

9.2 库管理相关

KF32 工具链的库管理工具 `kf32ar` 支持以标准格式的 `ar` 库来收集和维护 ELF 目标文件集合。用户可以将库文件传递给链接器以替代多个 ELF 目标文件。

`kf32ar` 能够实现以下功能：

- 创建新的库；
- 向库中添加文件；
- 替换库中的单个文件；
- 通过单次操作将库中所有文件替换为指定文件；
- 控制文件在库中的排列位置；
- 显式指定库的相关信息（例如列出库中的所有成员）。

需要注意的是，当在库中新增或替换文件时，每个文件都会附带对应的时间戳。此外，在创建、添加或替换库中的目标文件时，`kf32ar` 默认会创建符号表，但默认情况下不会包含调试符号。

9.3 处理库文件的注意事项

在使用库文件时需要注意以下事项：推荐使用同一套工具链进行输出，减少

版本间的差异。

9.4 常用功能示例

- 创建静态库

```
kf32ar rc libtest.a file1.o file2.o
```

- 查看库内容

```
kf32ar t libtest.a
```

- 添加或替换文件

```
kf32ar rc r libtest.a file3.o
```

- 提取库文件

```
kf32ar x libtest.a file2.o
```

10 编译工具反汇编器

10.1 命令行语法

KF32 工具链的 ELF 反汇编工具 `kf32objdump` 支持解析标准格式的 ELF 目标文件，并将其机器指令转换为可读的汇编语言，同时可展示文件结构与符号信息。其命令行语法：

```
kf32objdump <option(s)> <file(s)>
```

其中

option(s)

操作与修饰符，指定执行什么操作以及如何执行。多个选项可以组合使用。

file(s)

指定待反汇编的目标文件文件名。

10.2 选项说明与示例

10.2.1 选项说明

-G

生成包含调试信息的反汇编列表。适用于需要查看源代码与汇编代码对应关系的场景。

-S

与-G 类似，但反汇编时尽可能穿插显示源代码。提供更直观的源码级调试视图。

-l

在输出中包含源代码的文件名和行号信息。

-z

即使某个段或节区只包含零数据，也显示其内容。

--kf32arch=<arch>

指定目标文件的 CPU 架构，例如 `kf32r`。通常可自动识别，在特殊情况下用于手动指定。

--section=<name>

仅反汇编指定的节区（section），如 `.text`（代码段）、`.data`（已初始化数据段）、`.bss`（未初始化数据段）。可多次使用以指定多个节区。

10. 2. 2 反汇编示例

```
kf32objdump -G TestElf1.elf > TestElf1.lst
```

11 编译工具 ELF 转换工具

kf32objcopy 支持将链接生成的 ELF 格式目标文件转换为多种可直接下载到芯片存储器的标准格式文件。链接器在生成最终映像文件时，可通过参数自动调用此工具。

11.1 命令行语法

其命令行语法为：

```
kf32objcopy [options] input-file [output-file]
```

其中。

[options]

操作与修饰符，指定执行的文件格式转换操作及其他处理选项。多个选项可以组合使用。

input-file

指定输入的 ELF 格式目标文件名。

[output-file]

指定输出的目标格式文件名。如果未指定，或使用 > 重定向符，则输出到指定文件，而非控制台。

11.2 常用选项示例与说明

11.2.1 选项说明

-O <format>

指定输出文件的格式。这是最核心的转换选项。

支持的 <format> 包括：

ihex：Intel HEX 格式，文件扩展名常为 .hex。

srec 或 srec：Motorola S-record 格式 (S19)，文件扩展名常为 .s19 或 .srec。

-j <section>

--only-section=<section> 仅从输入文件中复制指定的节区 (section) 到输出文件。可多次使用以指定多个节区。

11.2.2 转换示例

将 ELF 文件转换为 Intel HEX 格式：

```
kf32objcopy -O ihex project.elf project.hex
```

将 ELF 文件转换为 S-record 格式:

```
kf32objcopy -O srec application.elf application.s19
```

12 编译工具段大小查看器

kf32size 用于快速统计和显示编译链接后生成的可执行文件中，各段（section）在存储器（如 Flash 和 RAM）中的占用大小，便于开发者评估和分析代码与数据的存储空间使用情况。

12.1 命令行语法

其命令行语法为：

```
kf32size [option(s)] [file(s)]
```

其中

[option(s)]

显示格式选项，用于指定输出信息的格式。

[file(s)]

指定一个或多个待分析的 ELF 格式目标文件名。

12.2 输出字段说明

执行 kf32size XXX.elf 的典型输出格式及各字段含义如下：

text	data	bss	dec	hex	filename
8684	0	0	12780	31ec	XXX.elf

text: 代码段大小（单位：字节）。通常存放程序代码和常量，占用 Flash 存储器。

data: 已初始化的全局/静态变量大小（单位：字节）。在程序加载时，这部分数据需从 Flash 复制到 RAM 中，因此同时占用 Flash（存储初始值）和 RAM（运行时的存储空间）。

bss: 未初始化或初始化为 0 的全局/静态变量大小（单位：字节）。程序运行时在 RAM 中分配空间，不占用 Flash。

dec: 十进制（Decimal）表示的存储器总使用量估计（text+ data+ bss+ eep）。

hex: 十六进制（Hexadecimal）表示的存储器总使用量估计。

filename: 被分析的文件名。

12.3 常用选项示例与说明

12.3.1 选项说明

`-A/--format=sysv`

以 System V 风格显示更详细的段信息，包括每个节区的具体大小。

`-B/--format=berkeley`

以 Berkeley 风格显示，即上文示例的默认格式。

`--format=<format>`

显式指定输出格式，如 `sysv` 或 `berkeley`。

12.3.2 功能示例

查看单个 ELF 文件的段大小（默认格式）：

```
kf32size project.elf
```

```
kf32size project.elf
```

以 System V 详细格式查看文件：

```
kf32size -A project.elf
```