ARM

ARM
编程技巧

Agenda

- **ARM 编译器优化**

  C/C++和汇编混合模式编程

  使用ARM编译器编码

  局部和全局数据讨论

优化级别

- 使用的编译器优化级别是可选择的

  -O0---DEBUG
  - 关闭大多数优化.
  - 最好的调试信息，最少的优化

  -O1---DEBUGREL
  - 多数优化选项许可
  - 给一个满意的调试，好的代码密度

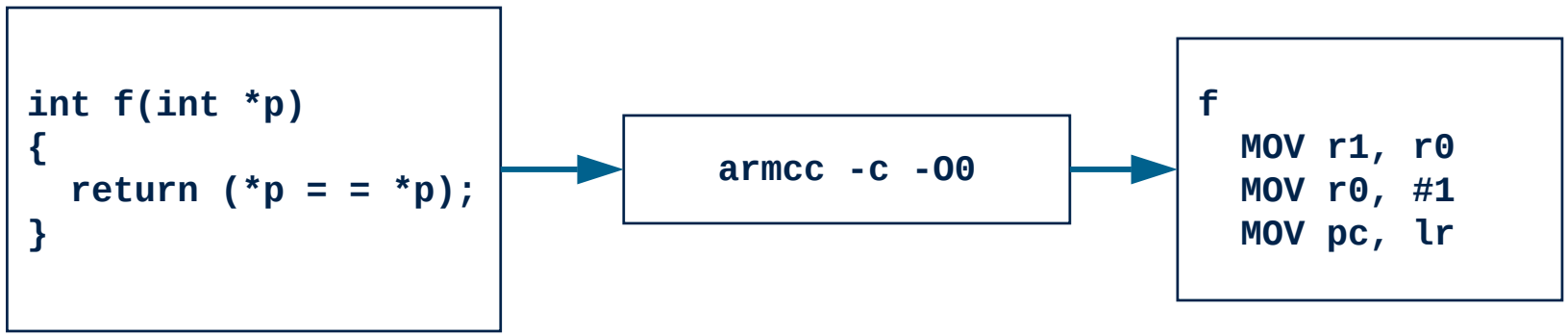
  -O2---RELEASE (default)
  - 完全的优化
  - 有限的调试信息，最好的代码密度

- 为代码大小或运行速度的优化，可选择: **-Ospace** (默认的)或 **-Otime**.
- 使用**-g** 选像可包含源码级调试信息

自动优化

- **ADS 编译器在所有级别中执行一些简单的优化**
  - i.e. **-O0, -O1, -O2**

- **下面是一个例子：即使用-O0,多余的表达式也被清除了：**
  - ATPCS标准中子程序结果返回规则
    - 结果为32位整数，R0返回
    - 结果为64位整数，R0，R1返回
    - 位数更多时，用内存来传递
    - ……



注意：在这种情况下，可使用C的关键字volatile 强制使用这些变量

使用 “volatile”

```
int f(int *p)
{
  return (*p = = *p);
}
```


```
f
  MOV r0,#1
  MOV pc,lr
```

```
int f(volatile int *p)
{
  return (*p = = *p);
}
```


```
f
  LDR   r1,[r0]
  LDR   r0,[r0]
  CMP   r1,r0
  MOVNE r0,#0
  MOVEQ r0,#1
  MOV   pc,lr
```

- 这个代码用的编译级别是：-o2

冗余代码的清除

- 下面是一个冗余代码清除的例子，他只用了**-o1**的优化选项:

```
int dummy()
{
  int a=10, b=20;
  int c;
  c=a+b;
  return 0;
}
```

```
armcc -c -O1
```

```
dummy
  MOV r0, #0
  MOV pc, lr
```

指令编排

- 指令编排在高级优化选项中是有效的(-O1, -O2).
  - 指令的重新编排是为了使要运行的代码更适合对应的核
  - 为arm9和以后的处理器提高吞吐量（一般可达到4%），并防止互锁（interlock）
  - 选择处理器可决定使用的运算法则，在默认情况下，使用针对ARM9的优化方案（对ARM7的运行没有影响）

- 例如:

```
int f(int *p, int x)
{ return *p + x * 3; }
```

没用指令编排 (-O0)

```
ADD r1,r1,r1,LSL #1
LDR r0,[r0,#0]
ADD r0,r0,r1 ; interlock on ARM9
MOV pc,lr
```

armcc –cpu arm7tdmi

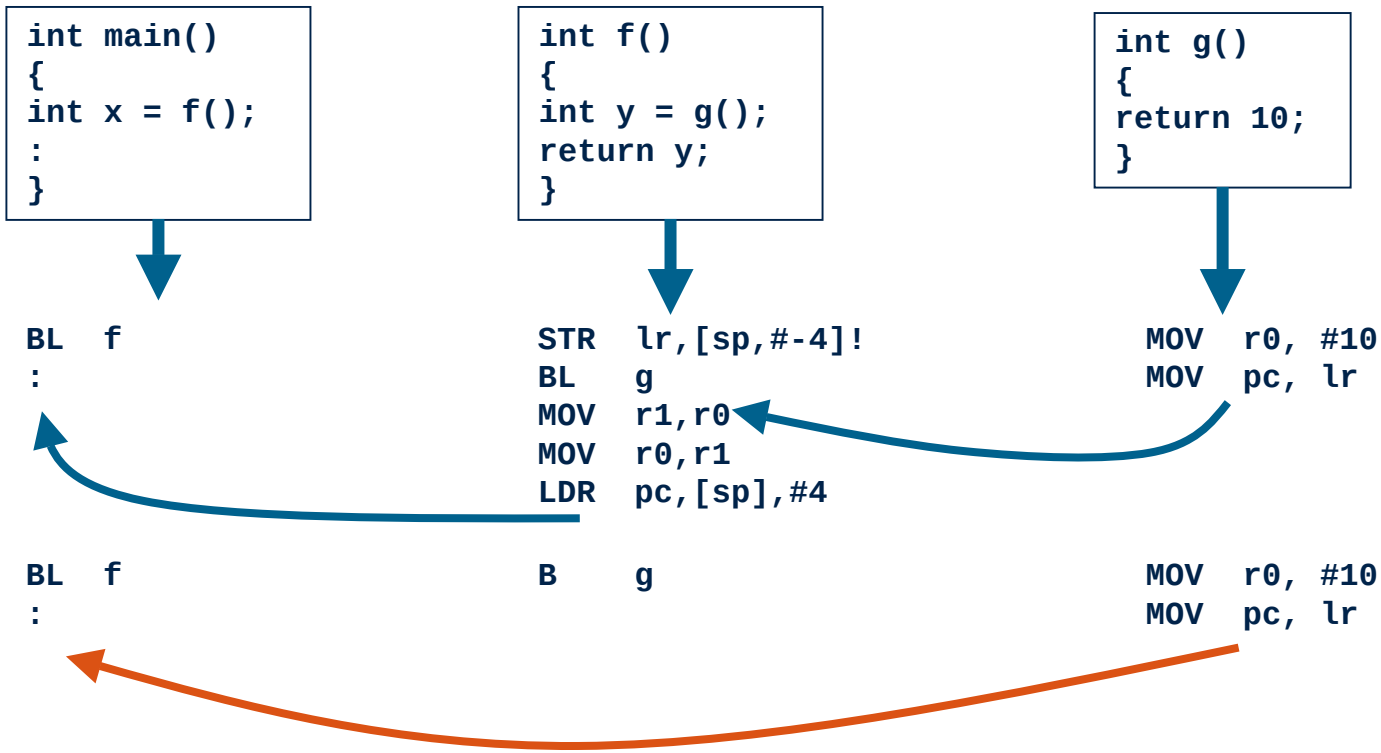
使用指令编排 (-O1,-O2)

```
LDR r0,[r0,#0]
ADD r1,r1,r1,LSL #1
ADD r0,r0,r1
MOV pc,lr
```

armcc –cpu arm9tdmi

嵌套优化

嵌套优化可避免在函数级里的不必要的返回
在可能的情况下BL 译码成B
在高级优化里有效(**-O1**, **-O2**).



内嵌函数（inline）

- 内嵌可通过删除子函数调用的开销来提高性能
- 这个 **`inline`** 关键字显示哪个函数将被内嵌
- 在高级优化选项中，**ADS 1.2** 编译器默认自动内嵌
  - **`-Oautoinline`** (default `-O2`)
  - **`-Ono_autoline`** (default for `-O0`,`-O1`)
- 哪个函数是否被内嵌取决于：
  - 他们是否被 __inline标示
  - 优化的级别
  - **`-Otime`** / **`-Ospace`**
  - 函数被调用的次数
- 如果函数在别的模块中不被调用，一个好的建议是用**static**标识函数，否则，编译器将在内嵌译码里把该函数编译乘非内嵌的
  - 加代码的长度
  - 使调试信息更复杂

Example...

内嵌例子

```
int bar(int a)
{
    a=a+5;
    return a;
}

int foo(int i)
{
    i=bar(i);
    i=i-2;
    i=bar(i);
    i++;
    return i;
}
```

```
bar
    ADD     r0,r0,#5
    MOV     pc,lr
foo
    STR     lr,[sp,#-4]!
    BL      bar
    SUB     r0,r0,#2
    BL      bar
    ADD     r0,r0,#1
    LDR     pc,[sp],#4
```

```
__inline int bar(int a)
{
    a=a+5;
    return a;
}

int foo(int i)
{
    i=bar(i);
    i=i-2;
    i=bar(i);
    i++;
    return i;
}
```

```
foo
    ADD     r0,r0,#5
    SUB     r0,r0,#2
    ADD     r0,r0,#5
    ADD     r0,r0,#1
    MOV     pc,lr
```

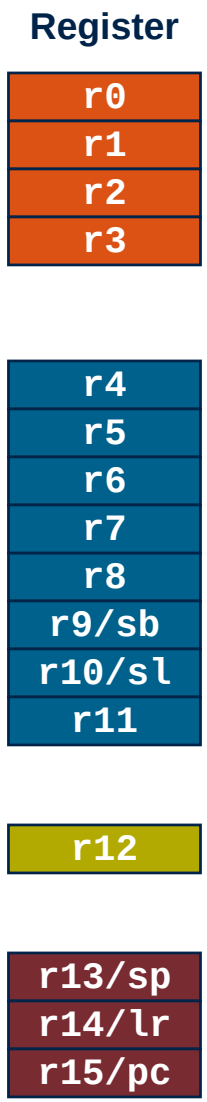
Agenda

- ARM编译器的优化
- **C/C++和汇编混合模式编程**
- 使用ARM编译器编码
- 局部和全局数据讨论

C和汇编的混合编程

- **C/C++ 和汇编能很容易的混合:**
  - 可实现在c中无法实现的处理器功能
  - 使用新的或不支持的指令
  - 产生更高效的代码
- **直接链接变量和程序**
  - 确定符合程序调用规范
  - 输入/输出相关的符号
- **编译器也可包含内嵌汇编**
  - 大多数arm指令集都可实现
  - 寄存器操作数可支持任意的c/c++的表达式
  - 内嵌汇编代码可由编译器的优化器来传递

ATPCS（arm/thumb程序调用规范）



作为函数传递的参数值

寄存器变量
必须保护

- 如果 **RWPI**选项有效，作为栈的基地址
- 如果软件堆栈检查有效，作为栈的限制值

Scratch register (corruptible)

- 子程序内部调用的可改写的寄存器

Stack Pointer
Link Register
Program Counter

- 可作为临时的一个值栈一样来使用
- 程序计数器

编译器使用一套规则的来设置寄存器的用法
ARM-Thumb Procedure Call Standard or
ATPCS (or APCS)
CPSR 标志位可被函数调用所破坏
任何和编译过的代码交互工作的汇编码在接口层必须满足ATPCS的规范

在C程序中调用汇编

- 在汇编程序中用**export name**来定义
- 在**C**程序中直接调用,用**EXTERN**声明
- 正常链接

```
extern void mystrcopy(char *d, const char *s);
int main(void)
{
  const char *src = “Source”;
  char dest[10];

  ...
  mystrcopy(dest, src);
  ...

}
```

CALL

```
  AREA StringCopy, CODE, READONLY
  EXPORT mystrcopy

mystrcopy
  LDRB r2, [r1], #1
  STRB r2, [r0], #1
  CMP r2, #0
  BNE mystrcopy
  MOV pc, lr

  END
```

这里所有的参数都是可以用寄存器来传递的，所以不需要在汇编程序中使用PUSH/POP来保护

内嵌汇编

- 允许使用一些不能由编译器自动生成的指令:
  - MSR / MRS
  - 新的指令
  - 协处理器指令
- 通常在关联的内嵌函数中使用
- 使用C变量代替寄存器
  - 不是一个真正的汇编文件
  - 通过优化器实现
- **ADS FAQ 入口 “Using the Inline Assembler”**

```
#define Q_Flag 0x08000000 // Bit 27

__inline void Clear_Q_flag (void)
{  int temp;
        __asm
        {
          MRS temp, CPSR
          BIC temp, temp, #Q_Flag
          MSR CPSR_f, temp
        }
}

__inline int mult16(short a,
                    short b, int c)
{
        int temp;
        __asm
        {
          SMLABB temp,a,b,c
        }
        return temp;
}
```

Agenda

- ARM编译器的优化
- C/C++和汇编混合模式编程
- **使用ARM编译器编码**
- 局部和全局数据讨论

参数传递

- 开始四个字大小的参数直接使用寄存器的**R0-R3**来传递**(**快速且高效的**)**
  - 更多的信息可参看ATPCS
- 如果需要更多的参数，将使用堆栈。**(**需要额外的指令和慢速的存储器操作**)**
- 所以通常限制参数的个数，使它为**4**或更少。
  - 如果不可避免，把常用的参数前4个放在R0-R3中

Example...

Parameter Passing (4 parameters)

- Parameter Passing (4 parameters)

```
int func1(int a, int b,
                int c, int d)
{
    return a+b+c+d;
}

int caller1(void)
{
    return func1(1,2,3,4);
}
```

```
func1
    0x000000 : ADD   r0,r0,r1
    0x000004 : ADD   r0,r0,r2
    0x000008 : ADD   r0,r0,r3
    0x00000c : MOV   pc,lr

caller1
    0x000014 : MOV   r3,#4
    0x000018 : MOV   r2,#3
    0x00001c : MOV   r1,#2
    0x000020 : MOV   r0,#1
    0x000024 : B     func1
```

Parameter Passing (6 parameters)

- Parameter Passing (6 parameters)

```
int func2(int a,int b,intc,
          int,d,int e,int f)
{
 return a+b+c+d+e+f;
}

int caller2(void)
{
 return func1(1,2,3,4,5,6);
}
```

This code is compiled with “-O2 -Ono_autoinline”

```
func2
  0x000000 : STR      lr, [sp,#-4]!
  0x000004 : ADD      r0,r0,r1
  0x000008 : ADD      r0,r0,r2
  0x00000C : ADD      r0,r0,r3
  0x000010 : LDMIB    sp,{r12,r14}
  0x000014 : ADD      r0,r0,r12
  0x000018 : ADD      r0,r0,r14
  0x00001C : LDR      pc,{sp},#4

caller2
  0x000020 : STMFD    sp!,{r2,r3,lr}
  0x000024 : MOV      r3,#6
  0x000028 : MOV      r2,#5
  0x00002C : STMIA    sp,{r2,r3}
  0x000030 : MOV      r3,#4
  0x000034 : MOV      r2,#3
  0x000038 : MOV      r1,#2
  0x00003C : MOV      r0,#1
  0x000040 : BL       func2
  0x000044 : LDMFD    sp!,{r2,r3,pc}
```

循环终止

- 在**for(), while() do…while()**的循环中，用减到**0**代替加到某个值。

- 比如，用下面的代替：

```
for (loop = 1; loop <= total; loop++) //（ADD, CMP）
```

代替为：

```
for (loop = total; loop != 0; loop--) //（SUBS）
```

- 尽量减少循环的次数
  - 代码小，且使用更少的寄存器

Example...

- **Count up**

```
int fact1(int limit)
{
    int i;
    int fact = 1;

    for (i = 1; i <= limit; i++)
    {
      fact = fact * i;
    }
    return fact;
}
```

```
fact1
  0x000000 : MOV      r2,#1
  0x000004 : MOV      r1,#1
  0x000008 : CMP      r0,#1
  0x00000c : BLT      0x20
  0x000010 : MUL      r2,r1,r2
  0x000014 : ADD      r1,r1,#1
  0x000018 : CMP      r1,r0
  0x00001c : BLE      0x10
  0x000020 : MOV      r0,r2
  0x000024 : MOV      pc,lr
```

- **Count down**

```
int fact2(int limit)
{
    int i;
    int fact = 1;

    for (i = limit; i != 0; i--)
    {
        fact = fact * i;
    }
    return fact;
}
```

```
fact2
  0x000000 : MOVS     r1,r0
  0x000004 : MOV      r0,#1
  0x000008 : MOVEQ    pc,lr
  0x00000c : MUL      r0,r1,r0
  0x000010 : SUBS     r1,r1,#1
  0x000014 : BNE      0x0c
  0x000018 : MOV      pc,lr
```

This code is compiled with “-O2 -Otime”

除法操作(1)

- **ARM核不含除法硬件**
  - 除法通常用一个运行库函数来实现
    - 运行需要很多的周期

```
unsigned div(unsigned a, unsigned b)
{
    return (b / a);
}
```

```
div
    B    __rt_udiv
```

- 一些除法操作在编译时作为特例来处理
  - 除2操作，被左移代替

```
unsigned div2(unsigned b)
{
    return (b / 2);
}
```

```
div2
    MOV  r0,r0,LSR #1
    MOV  pc,lr
```

除法操作 (2)

- 在 -O1和-O2 (使用 -Otime), 其他的常量将使用一个标准的乘法序列来完成
  例如：

```
unsigned div10(unsigned c)
{
    return (c / 10);
}
```

```
div10
    MOV r1,r0
    LDR r0,=0xCCCCCCCD
    UMULL r2,r1,r0,r1
    MOV r0,r1,LSR #3
    MOV pc,lr
```

- 实时除法程序
  - 使用CLZ指令
  - 只有V5te体系结构才有效。
  - 用下面的办法来选择
    - C - #pragma import __use_realtime_division
    - Assembler - IMPORT __use_realtime_division

余数 – 模算法

- 余数的操作符 ‘%’, 通常使用模算法
- 如果这个值的模不是2的n次幂，它将花费大量的时间和代码空间
  - 避免这种情况发生的办法使用if()作状态检查
- 比如说:count的范围是0到59

```
count = (count+1) % 60;
```

用下面的句子代替

```
if (++count >= 60) count = 0;
```

```
modulo
  ADD       r1,r0,#1
  MOV       r0,#0x3c
  BL        __rt_udiv
  MOV       r0,r1
```

```
test_and_reset
  ADD       r0,r0,#1
  CMP       r0,#0x3c
  MOVCS     r0,#0
```

这个代码用“-O1 -Ospace”编译

浮点

- 软件浮点库 **(fplib)**
  - 默认：**`-fpu softvfp`** (or **`softfpa`**)
- 浮点协处理器
  - VFP (ARM10 and ARM9)
    - **`-fpu vfp`** (or **`vfpv1`** or **`vfpv2`**)
  - FPA (eg ARM7 500fe) - now obsolete
    - **`-fpu fpa`**
- 软件浮点仿真 **(FPE)**
  - 通过未定义的异常来捕获协处理器指令
- **VFP (and FPA)** 实际上是硬件协处理器和仿真的混合
  - 要求支持代码去实现混合运算
  - 在AFS 1.3 和以后的版本里有VFP的 支持代码, 在ADS的FPA里.
- 在**thumb**代码使用**fp**处， **vfp**系统用**`-fpu softvfp+vfp`**编译
- 使用 **`-auto_float_constants`** 预防常量被处理为双精度类型，关闭警告用 **`-Wk`.**

Example...

Floating point examples

```
float foo(float num1, float num2)
{
    float temp, temp2;

    temp = num1 + num2;
    temp2 = num2 * num2;
    return temp2-temp;
}
```

armcc float.c

armcc -fpu vfpv2 float.c

```
foo
    STMFD    sp!,{r3-r5,lr}
    MOV      r4,r1
    BL       _fadd
    MOV      r5,r0
    MOV      r0,r4
    MOV      r1,r4
    BL       _fmul
    MOV      r1,r5
    LDMFD    sp!,{r3-r5,lr}
    B        _fsub
```

使用浮点库

```
foo
    FADDS    s0,s0,s1
    FMULS    s1,s1,s1
    FSUBS    s0,s1,s0
    MOV      pc,lr
```

使用协处理器指令

Agenda

ARM 编译器优化

C/C++和汇编混合模式编程

使用ARM编译器编码

局部和全局数据

变量类型

- 全局和静态变量保留在**RAM**里
  - 需使用loads/stores访问外部存储器
- 局部变量通常放在寄存器中，用来快速且高效的处理
  - 如果编译器的寄存器分配算法认为超过现有的寄存器数量，将把变量压入栈中
- 对局部变量，用 **word-sized (int)** 代替 **halfword** 和 **byte:**
  - 为了确保不受其他条件的影响，可特别指定使用32-bit寄存器变量.

变量大小

```
int wordsize(int a)
{
    return (a*2);
}
```

```
wordsize
        0x000000 : MOV      r0,r0,LSL #1
        0x000004 : MOV      pc,lr
```

```
short halfsize(short b)
{
    return (b*2);
}
```

```
halfsize
        0x000008 : MOV      r0,r0,LSL #17
        0x00000c : MOV      r0,r0,ASR #16
        0x000010 : MOV      pc,lr
```

```
char bytesize(char c)
{
    return (c*2);
}
```

```
bytesize
        0x000014 : MOV      r0,r0,LSL #25
        0x000018 : MOV      r0,r0,LSR #24
        0x00001c : MOV      pc,lr
```

堆栈的用法

- **C/C++代码的堆栈使用，堆栈用来保留:**
  - 子程序的返回地址
  - ‘溢出’的局部变量
  - 局部数组和结构体

- **注意:**
  - 函数越小越好： (更少的变量，更少的‘溢出‘);
  - 更少数量的‘live’变量 (比如：函数里每个点保存的有用的数据)
  - 避免使用大的局部结构体或数组 (使用malloc/free代替)
  - 避免递归

堆栈使用估计

- 链接使用 **-callgraph**
  - 显示静态堆栈的开销(html文件).
- 编译时使用软件堆栈检查
  - -apcs /swst
- 在栈结束点设置 **watchpoint**
  - 测试堆栈
- 定义大的栈
  - 填充某个值，看覆盖了多少，从而判定栈的使用情况
- **ARMulator映射文件**
  - 拒绝访问栈下面的区域，栈溢出将导致一个data abort异常
- **stackuse.c**
  - ARMulator模式, 跟踪堆栈的大小，用ARMulator的统计来输出报告

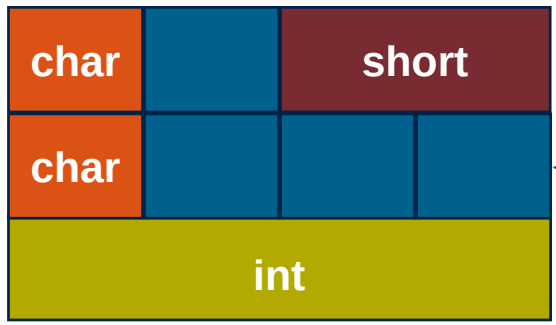
当要对堆栈使用情况进行估计时，使用‘worst case’

全局数据布局

- 全局数据保存在存储器里，不是寄存器
  - 需要load / store指令来访问
  - 用物理尺寸的边界对齐
- **ADS 1.2** 会优化在一个模块里的全局数据的布局
  - 用**-Ono_data_reorder** 将关闭排序

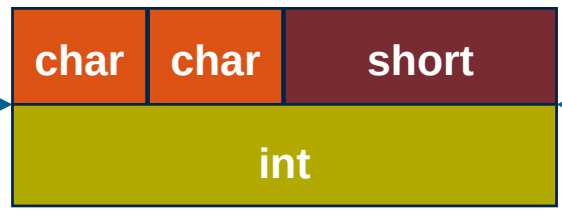
e.g. 声明的数据

```
char one;
short two;
char three;
int four;
```



Declared alignment
12 bytes
(4 bytes of padding)

ADS 1.1+ 将自动
用此风格排序



Optimal alignment
8 bytes
(Zero bytes of padding)

不对齐访问

- **ARM硬件需要在自然尺寸的边界访问内存**
  - Word访问在word尺寸
  - Halfword访问在halfword尺寸
  - Byte访问在byte尺寸
- 不对齐访问
  - 遗留代码
  - 特定协议

  需要必须告诉编译器，让它产生适当的指令序列
- 使用 **__packed** 属性
  - 可能导致多字节访问代替单字节访问
  - 用LDM指令的结果有2 字,转变为生成单字
- 不对齐数据的访问所产生的意外的结果取决于指令的使用
  - 将是不可预知的

指针的对齐

- 必须非常小心指针的对齐
  - 可能导致程序的失败

```
#include <string.h>
int *a = (int *)0x1000;
int *b = (int *)0x2000;
char *c = (char *)0x3001;
__packed int *d;

void foo (void)
{
  memcpy (b,a,12);

  memcpy (c,a,12);

  b = (int *)c;
  memcpy (b,a,12);

  d = (__packed int *)c;
  memcpy ((void *)d,a,12);
}
```

```
STMFD   r13!,{r4,r14}
LDR     r4,0x58

LDR     r1,[r4,#0]
LDR     r0,[r4,#4]
LDMIA   r1,{r2,r3,r12}
STMIA   r0,{r2,r3,r12}

LDR     r0,[r4,#8]
LDR     r1,[r4,#0]
MOV     r2,#0xc
BL      __rt_memcpy

LDR     r0,[r4,#8]
STR     r0,[r4,#4]
LDR     r1,[r4,#0]
LDMIA   r1,{r2,r3,r12}
STMIA   r0,{r2,r3,r12}

LDR     r0,[r4,#8]
LDR     r1,0x5c
MOV     r2,#0xc
STR     r0,[r1,#0]
LDR     r1,[r4,#0]
BL      __rt_memcpy

LDMFD   r13!,{r4,pc}
```

memcpy inline'd

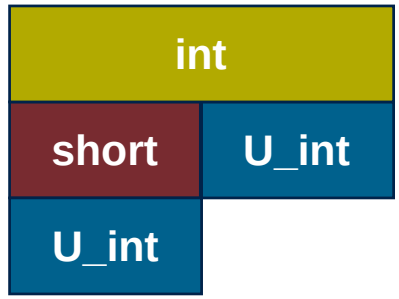
memcpy called

memcpy inline'd unsafely

memcpy called safely

结构的打包

- _packed
  - 限定的数据为1字节对齐
  - 不实现字节对齐调整
  - 很高的访问代价，不会节省存储空间




```
__packed struct mystruct {
  int aligned_i;
  short aligned_s;
  int unaligned_i;
};

extern struct mystruct S;
```

PREFER...

```
struct mystruct {
  int aligned_i;
  short aligned_s;
  __packed int unaligned_i;
};

extern struct mystruct S;
```

- 在结构里定义打包的元素代替结构的打包
  - 他将帮助减小访问输出的结构的开销
  - ADS FAQ 入口: Aligned v. unaligned accesses and use of __packed

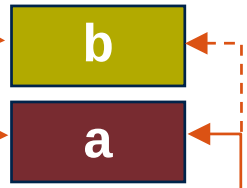
优化的指针基地址

```
extern int a;
extern int b;
void foo (int x, int y)
   {
       a = x;
       b = y;
    }
```

a 和 b 被定义为外部的

```
LDR r2, [pc,#12]
STR r0, [r2,#0]
LDR r3, [pc,#8]
STR r1, [r3,#0]
MOV pc, lr

DCD “address of a”
DCD “address of b”
```




```
int a;
int b;
void foo (int x, int y)
    {
        a = x;
        b = y;
    }
```

a 和 b 被定义为模块内用的数据

```
LDR r2, [pc,#8]
STR r0, [r2,#0]
STR r1, [r2,#4]
MOV pc, lr

DCD “base address of a and b”
```

注意：在用-o0时无效

优化外部全局指针

- 如果全局数据放在结构体里，每个元素的访问将自动的在基指针上偏移
  - 在结构体里的元素将按大小的边界对齐
  - 编译器不对结构体重新排列
- 把数据放在多个逻辑结构体内，代替一个大的结构
- **'#define' 将对主应用代码的改变隐藏起来**
  - `#define value mystruct.value`

Example...

外部全局变量

data.c

```
int a;
int b;
```

code.c

```
extern int a;
extern int b;

int main(void)
{
    return a+b;
}
```

Assembler output

```
main      LDR      r0,0x000080c0
000080ac  LDR      r1,0x000080c4
000080b0  LDR      r0,[r0,#0]
000080b4  LDR      r1,[r1,#0]
000080b8  ADD      r0,r0,r1
000080bc  MOV      pc,lr
000080c0  DCD      0x000083d4
000080c4  DCD      0x000083d8
```

```
struct data
{
    int a;
    int b;
}mystruct;
```

```
extern struct data mystruct;

int main(void)
{
    return mystruct.a+mystruct.b;
}
```

```
main      LDR      r0,0x000080bc
000080ac  LDR      r1,[r0,#0]
000080b0  LDR      r0,[r0,#4]
000080b4  ADD      r0,r1,r0
000080b8  MOV      pc,lr
000080bc  DCD      0x000083cc
```

测验

1) 默认的优化级别是什么？

2）给tail-call优化有什么好处

3) 在函数调用时，管理寄存器用法的标准的名字是什么？

4) 在参数传递时，被推荐的最大的量是多少？

5) 为什么在arm里要尽可能避免使用除法？

6) __packed的效果是什么？

参考

- 需要更多的信息，请看:
  - ADS 1.2 Compilers and Libraries Guide
    - Section 2 : C and C++ Compilers
    - Section 3 : ARM Compiler Reference
  - ADS 1.2 Developer Guide
    - Chapter 4: Mixing C, C++ and Assembly Language
  - Application Note 34, Writing Efficient C
  - Application Note 36, Declaring Global Data in C


ARM®
THE ARCHITECTURE
FOR THE DIGITAL WORLD™