Assembly and C

R. Ferrero

Politecnico di Torino

Dipartimento di Automatica e Informatica (DAUIN)

Torino - Italy

Ways to mix assembly and C

混合汇编和C的方法

- calling a C function from assembly file
- calling an assembly subroutine from C file
- adding assembly code in a C file
  - inline assembly
  - embedded assembly

1，在一个汇编文件中调用C函数

2，在汇编文件中调用一个C的子程序

3，添加汇编代码在C文件中
内联汇编 和 嵌入汇编

C function call from assembly

在汇编中调用C函数

- visibility of the C function 可见性的C函数
  - how can the assembler know the name of the function? 汇编器如何知道这个函数的名字
- exchanging data 交换数据
  - how are parameters passed to the C function?
  - where is the result stored?

参数怎么传递到C函数中去

结果存储在哪里

Visibility of the C function

C函数的可见性

- Two directives notify the assembler of the name of a symbol defined in another file:

两个指令通知汇编程序，在另外一个文件中定义的符号的名字

```
IMPORT symbolName {[options]}
```

导入

```
EXTERN symbolName {[options]}
```

外部的

- `IMPORT` always imports the symbol.

IMPORT总是导入符号

  - The linker generates an error if the symbol is not defined elsewhere.

如果符号没有被定义，那么连接器会产生一个错误

- `EXTERN` imports the symbol only if it is used.

只有使用符号时才导入他

  - The linker generates an error if the symbol is not defined elsewhere and is used in the assembly file

如果符号没有被定义并且在一个汇编文件中使用，那么连接器会产生一个错误

Options of IMPORT and EXTERN

IMPORT和EXTERN 的选择

- `WEAK`: prevents the linker searching libraries that are not already included 缺点：防止连接器搜索没有包含的库
- `DATA` | `CODE`: treats the symbol either as data or code when source is assembled and linked
  DATE|CODE：当资源被汇编和链接时，把符号也当作数据或者代码
- `SIZE =` *value*: specifies the size. If missing:
  SIZE = 值：指定大小， 如果丢失：
  - for `PROC` symbols: size of the code until `ENDP`
    对于PROC符号，直到结束，代码的规模
  - for other symbols: size of instruction or data on the same source line of the symbol
    对于其他符号：指令或者数据的规模在相同的符号资源线上
  - if there is no instruction or data: size is zero.
    如果这里没有指令或者数据，规模是0

Data from Assembly to C function

数据从汇编到C函数

- Data exchange is regulated by ARM Architecture Procedure Call Standard (AAPCS).

数据交换通过AAPCS受到管控

- The first 4 parameters are passed in `r0-r3`.

前四个参数通过r0，r1，r2，r3传递

- Further parameters are passed in the stack.

更多的变量通过栈来进行传递

  - After returning, they must be removed from stack

在返回之后，他们必须从栈中移除

- Return value of C function is received in `r0-r3`

C函数返回的值依次由R0 - R3 接收

  - 32-bit sized type -> `r0` 32位规模类型的
  - 64-bit sized type -> `r0-r1` 64位规模类型的
  - 128-bit sized type -> `r0-r4` 128位规模类型的

Example: startup code

```
Reset_Handler   PROC
      EXPORT   Reset_Handler [WEAK]
      IMPORT   SystemInit
      IMPORT   __main
      LDR      R0, =SystemInit
      BLX      R0
      LDR      R0, =__main
      BX       R0
      ENDP
```

程序链接到 子程序 所在地址

跳转到 名为_main 的子程序进行执行

BL：带链接的跳转，把当指令的下一条指令地址保存到LR寄存器中，并跳转到指定标签

BX：带状态切换跳转，最低位为1时，切换到Thumb指令执行，为0时，ARM指令执行

BLX：结合了BL和BX的功能，带链接和状态切换的跳转

__main Vs main

_main 和 main ()

- main() is the user-defined function. main () 是用户定义的函数
- Embedded applications need an initialization sequence before main() function starts. This is called the startup code or boot code.

嵌入式应用程序在main () 函数开始之前，需要初始化序列。这称为启动代码后者引导代码

- The ARM C library contains pre-compiled and pre-assembled code sections for startup.

ARM C的库包括预编译和与汇编的代码段，用来启动

- The linker includes the necessary code from the C library to create a custom startup code.

连接器包括从C库中的必要代码，来创建自定义的启动代码

- __main is the entry point to the C library.

_main 是去C库的进入点

Example: square root 二次根

- Square root is not available in the Thumb-2 instruction set. 二次根在Thumb-2指令集中不可用
- Workaround: in the assembly code, a C function that computes the square root is called. 工作区：在汇编代码中，调用C中计算平方根的函数
- The C function receives one parameter and returns the integer square root.

C函数收到一个参数 并返回一个整型平方根

C code

```
#include <math.h>
int intSquareRoot(int intNumber) {
    double realNumber;
    int result;
    realNumber = sqrt(intNumber);
    result = floor(realNumber+0.5);
    return result;
}
```

floor函数 为 向下取整函数

题中是 利于+0.5 达到四舍五入的目的

Assembly code

```
EXTERN intSquareRoot
MOV r0, #26 ; first parameter
BL intSquareRoot
```

因为依次从R0-R3中取值。

R0是低位，R1是高位

- At the end, `r0` = 5.
- Note that other registers may have been changed, e.g., `r1`, `r2`, and `r3`. According to AAPCS, they are scratch registers.

注意，其他的寄存器也可能会改变，例如：R1，R2和R3.根据AAPCS，他们是临时寄存器

Exercise

The 4th degree polynomial equation 多项式方程

$$ax^4 + bx^3 + cx^2 + dx^1 + e = 0$$

has 4 solutions:

$$x_{1,2} = -\frac{b}{4a} - Q \pm \frac{1}{2}\sqrt{-4Q^2 - 2p + \frac{S}{Q}}$$

$$x_{3,4} = -\frac{b}{4a} + Q \pm \frac{1}{2}\sqrt{-4Q^2 - 2p + \frac{S}{Q}}$$

Exercise

$$p = \frac{8ac - 3b^2}{8a^2}$$

$$S = \frac{8a^2d - 4abc + b^3}{8a^3}$$

$$Q = \frac{1}{2}\sqrt{-\frac{2}{3}p + \frac{1}{3a}\left(\Delta_0 + \frac{q}{\Delta_0}\right)}$$

$$\Delta_0 = \sqrt[3]{\frac{s + \sqrt{s^2 - 4q^3}}{2}}$$

$$q = 12ae - 3bd + c^2$$

$$s = 27ad^2 - 72ace + 27b^2e - 9bcd + 2c^3$$

Exercise

- Let `int solution1_grade4(int a, int b, int c, int d, int e)` be a function implemented in a C file that computes the first solution (rounded to the nearest integer value) of a quartic equation.
- Write the assembly code to compute a solution of the equation 一个四次方程

$$x^4 - 10x^3 + 35x^2 - 50x + 32 = 0$$

通过C调用汇编子程序

Assembly subroutine call from C

汇编子程序的可见性

- visibility of the assembly subroutine
  - how can the linker know the name of the subroutine? 连接器如何知道子程序的名字
- exchanging data 交换数据
  - how are parameters received by the assembly subroutine? 如何通过汇编子程序传递参数
  - where is the result stored?

结果存储在哪里

Visibility of the ASM subroutine

ASM子程序的可见性

- In the C file:
  - there must be the prototype of the subroutine

    必须有一个子程序的原型
  - the prototype must begin with `extern`

    这个原型必须以extern 开头
- In the assembly file: 在这个汇编文件中
  - the subroutine is implemented 子程序必须被实现
  - the symbol is exported with one of the two equivalent directives: 该符号是通过两个等价指令之一导出的

`EXPORT` *`symbolName`* `{[option]}`

`GLOBAL` *`symbolName`* `{[option]}`

这两个标识符的用法

Options of EXPORT and GLOBAL

如果另外一个文件导出了相同的符号名称，则这个符号不导出

- `WEAK`: the symbol is not exported if another file exports the same symbol name.
- `DATA | CODE`: treats the symbol either as data or code when source is assembled and linked

当资源被汇编或者链接时，对待这个符号当作数据或者代码

- `SIZE` = *value*: specifies the size. If missing:
  - for `PROC` symbols: size of the code until `ENDP`

    对于PROC标识符：代码的尺寸是直到ENDP
  - for other symbols: size of instruction or data on the same source line of the symbol

    对于其他标识符：
  - if there is no instruction or data: size is zero.

    如果这里没有代码或者数据，尺寸那么是0

Data from C to Assembly routine

数据从C程序到汇编子程序

- Data exchange is regulated by ARM Architecture Procedure Call Standard (AAPCS).
- The first 4 parameters are received in `r0`-`r3`.

  前四个参数是通过R0-R4接收的
- Further parameters are received in the stack.

  跟多的是通过栈来接收的
  - They must not be removed from the stack.

    不能从栈中删除他们
- Return value of the assembly subroutine is passed in `r0`-`r3` (e.g., a word is passed in `r0`)

  汇编子程序的返回值是通过R0-R3来传递的
- The assembly subroutine must preserve the contents of registers `r4`-`r8`, `r10`, `r11` and `SP`.

字符串链接

Example: string concatenation

写个程序来，复制两个字符串的首字母来组成第三个字符串

- Write a program that copies the first characters of two strings into a third string.
- The three strings are defined in the C file.

这三个字符串被定义在一个C文件中

- The copying routine is written in assembly

复制程序是用汇编语言编写的

  - it copies one byte at a time 它一次复制一字节
  - controls are added for robustness, e.g., the destination string is full, there are no more characters to copy in the source strings.

控件是为了健壮性而添加的，例如，最终字符串是空的，这里就会没有字节去复制，在源字符串中

Parameters and return value

参数和返回值

- The copying subroutine receives in input:

  复制子程序通过输入接收：
  - pointer to string1

    字符串1的指针
  - number of characters to copy from string1

    从String1中要复制的字符串数
  - pointer to string2

    String2的指针
  - number of characters to copy from string2

    从String2中要复制的字符串数
  - pointer to string3

    String3的指针
  - maximum length of string3

    String3的最大长度
- The copying subroutine returns the number of characters copied.

  复制子程序返回所复制字节的数目

Example: C code

int类型函数，6个参数，两个常量，四个变量

```
#define MAX_LENGTH 20
extern int concatenateString(const char *,
    int, const char *, int, char *, int);
int main(void){
  const char *string1 = "problem solving";
  const char *string2 = "grammar book";
  char string3[MAX_LENGTH];
  int len1 = 3, len2 = 4, len3;
  len3 = concatenateString(string1, len1,
    string2, len2, string3, MAX_LENGTH);
  while(1);
}
```

因为主程序在子程序下面，所以不用声明子程序的调用

参数都是通过子程序传递

Example: assembly code (I)

```
concatenateString PROC
   EXPORT concatenateString
   MOV r12, sp
   ; save volatile registers
   STMFD sp!, {r4-r8,r10-r11,lr}
   ;extract argument 4 and 5 from stack
   LDR   r4, [r12]
   LDR   r5, [r12, #4]
   SUB r5, r5, #1      ; the last character
must be the zero terminator
   MOV r6, #0    ; num bytes copied to string3
```

sp：Stack Pointer

STMFD：ST-store, M-multiple, F-full, D-descending

因为包含0，所以总个数减一

Example: assembly code (II)

```
string1copy
  LDRB r7, [r0], #1 ;load byte from string1
  CMP r7, #0  ;check for zero terminator
  BEQ string1End
  STRB r7, [r4], #1 ;store byte in string3
  ADD r6, r6, #1
  CMP r6, r5   ;is string 3 full?
  BEQ string2End
  CMP r6, r1   ;other bytes to copy?
  BLO string1copy
string1End     MOV r8, #0;
```

存入String3，存后并重置指针R7（+1）

此时R6 为String1 的计数器

Example: assembly code (III)

```
string2copy
   LDRB r7, [r2], #1 ;load byte from string2
   CMP r7, #0  ;check for zero terminator
   BEQ string2End
   STRB r7, [r4], #1 ;store byte in string3
   ADD r6, r6, #1
   ADD r8, r8, #1         此时的R8 为String2的计数器
   CMP r6, r5   ;is string 3 full?
   BEQ string2End         此时的R6 为R3的计数器
   CMP r8, r3   ;other bytes to copy?
   BLO string2copy
```

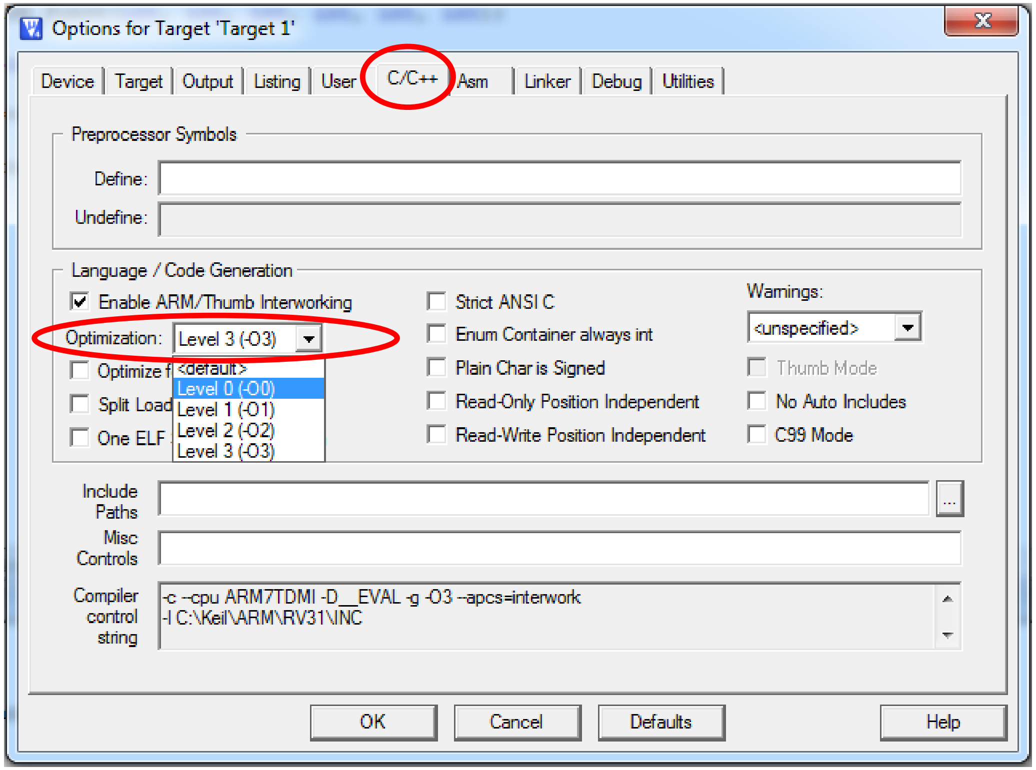
Example: assembly code (IV)

0是取值的终结标识符

```
string2End
  MOV r7, #0    ;insert the zero terminator
  STRB r7, [r4], #1 ;store byte in string3
  MOV r0, r6    ;set the return value
  ;restore volatile registers
  LDMFD sp!,{r4-r8,r10-r11,pc}
  ENDP
```

Compiler optimization

- The compiler can be asked to optimize the machine code generated from the C code.



Compiler optimization levels

- level 0: minimum optimization
  - good for debugging: the structure of generated code directly corresponds to the source code.
- level 1: restricted optimization
  - the generated code can be significantly smaller than level 0: this simplifies analysis of the code.
- level 2: high optimization
  - the compiler automatically inlines functions.
- level 3: maximum optimization
  - loop unrolling, more aggressive inlining.

循环展开，内联更激进

Disassembled code with level 0

```
     8:           const char *string1 = "problem solving";
0x000001B6 A41F       ADR       r4,{pc}+2   ; @0x00000234
     9:           const char *string2 = "grammar book";
    10:           char string3[MAX_LENGTH];
0x000001B8 A522       ADR       r5,{pc}+4   ; @0x00000244
    11:           int len1 = 3, len2 = 4, len3;
0x000001BA 2603       MOVS      r6,#0x03
0x000001BC 2704       MOVS      r7,#0x04
    12:           len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
0x000001BE 2014       MOVS      r0,#0x14
0x000001C0 A903       ADD       r1,sp,#0x0C
0x000001C2 463B       MOV       r3,r7
0x000001C4 462A       MOV       r2,r5
0x000001C6 E9CD1000   STRD      r1,r0,[sp,#0]
0x000001CA 4631       MOV       r1,r6
0x000001CC 4620       MOV       r0,r4
0x000001CE F000F92B   BL.W      concatenateString (0x00000428)
    13:           while(1);
```

- values and pointers are load in registers.

值和指针被载入到寄存器中

- for passing parameters, the registers are exchanged to `r0-r3` or pushed into the stack.

对于传递的参数，寄存器被转换进R0-R3或者压入栈中

Disassembled code with level 1

```
      8:          const char *string1 = "problem solving";
0x000001B6 A011       ADR      r0,{pc}+2   ; @0x000001FC
      9:          const char *string2 = "grammar book";
     10:          char string3[MAX_LENGTH];
0x000001B8 A214       ADR      r2,{pc}+4   ; @0x0000020C
     11:          int len1 = 3, len2 = 4, len3;
0x000001BA 2103       MOVS     r1,#0x03
0x000001BC 2304       MOVS     r3,#0x04
     12:          len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
0x000001BE 2414       MOVS     r4,#0x14
0x000001C0 AD03       ADD      r5,sp,#0x0C
0x000001C2 E9CD5400   STRD     r5,r4,[sp,#0]
0x000001C6 F000F92F   BL.W     concatenateString (0x00000428)
     13:          while(1);
```

- code is shorter: values and pointers are loaded in the proper registers for the subroutine call. 程序更短

返回值丢失

Return value is missing!

在这两段代码中，没有任何指令得到来自子程序concatenateString的返回值

- In both the disassembled codes, there is not any instruction that gets the value returned by the subroutine `concatenateString`.
- This value should be saved in `len3`, but the variable is never used later. 这个值应该存入len3中，但是这个在后来并没有被使用
- Therefore, the compiler does not save the value and consider the subroutine as void.
因此，编译器没有保存值和把子程序认为是空的
- In fact, there is a warning when compiling:

```
Build target 'Target 1'
compiling C_functions.c...
C_functions.c(11): warning:  #550-D: variable "len3" was set but never used
        int len1 = 3, len2 = 4, len3;
C_functions.c: 1 warning, 0 errors
linking...
Program Size: Code=2732 RO-data=220 RW-data=0 ZI-data=608
".\ARMandC.axf" - 0 Error(s), 1 Warning(s).
```

Obtaining the return value

- There are two ways for forcing the acquisition of the return value: 强制获取返回值有两种方法
  - add some instructions that use `len3` 加入使用len3的指令
  - declare `len3` as `volatile`. 声明len3是volatile类型的
- The keyword `volatile` may appear before or after the data type in the variable definition:
  - `volatile int len3;`
  - `int volatile len3;`

volatile：告知编译器不要优化，重新从内存导入数据

Use of the variable: an example

```
int main(void){
  const char *string1 = "problem solving";
  const char *string2 = "grammar book";
  char string3[MAX_LENGTH];
  int len1 = 3, len2 = 4, len3;
  len3 = concatenateString(string1, len1,
string2, len2, string3, MAX_LENGTH);
  for (; len3 > 0; len3 --)
     string3[len3 - 1] += 'A' - 'a';
  while(1);
}
```

Disassembled code with level 0

```
     8:          const char *string1 = "problem solving";
0x000001B6 A427       ADR      r4,{pc}+2   ; @0x00000254
     9:          const char *string2 = "grammar book";
    10:          char string3[MAX_LENGTH];
0x000001B8 A52A       ADR      r5,{pc}+4   ; @0x00000264
    11:          int len1 = 3, len2 = 4;
    12:          volatile int len3;
0x000001BA 2603       MOVS     r6,#0x03
0x000001BC 2704       MOVS     r7,#0x04
    13:          len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
0x000001BE 2014       MOVS     r0,#0x14
0x000001C0 A903       ADD      r1,sp,#0x0C
0x000001C2 463B       MOV      r3,r7
0x000001C4 462A       MOV      r2,r5
0x000001C6 E9CD1000   STRD     r1,r0,[sp,#0]
0x000001CA 4631       MOV      r1,r6
0x000001CC 4620       MOV      r0,r4
0x000001CE F000F931   BL.W     concatenateString (0x00000434)
0x000001D2 9002       STR      r0,[sp,#0x08]
    14:          for (; len3 > 0; len3 --)
0x000001D4 E009       B        0x000001EA
    15:                  string3[len3 - 1] += 'A' - 'a';
0x000001D6 9902       LDR      r1,[sp,#0x08]
0x000001D8 1E49       SUBS     r1,r1,#1
0x000001DA AA03       ADD      r2,sp,#0x0C
0x000001DC 1888       ADDS     r0,r1,r2
0x000001DE 7801       LDRB     r1,[r0,#0x00]
0x000001E0 3920       SUBS     r1,r1,#0x20
0x000001E2 7001       STRB     r1,[r0,#0x00]
0x000001E4 9802       LDR      r0,[sp,#0x08]
0x000001E6 1E40       SUBS     r0,r0,#1
0x000001E8 9002       STR      r0,[sp,#0x08]
0x000001EA 9802       LDR      r0,[sp,#0x08]
0x000001EC 2800       CMP      r0,#0x00
0x000001EE DCF2       BGT      0x000001D6
    16:          while(1);
```

`r0` contains the return value. It is saved in the stack to be used later.

Disassembled code with level 1

```
     8:         const char *string1 = "problem solving";
0x000001B6 A016      ADR      r0,{pc}+2  ; @0x00000210
     9:         const char *string2 = "grammar book";
    10:         char string3[MAX_LENGTH];
0x000001B8 A219      ADR      r2,{pc}+4  ; @0x00000220
    11:         int len1 = 3, len2 = 4, len3;
0x000001BA 2103      MOVS     r1,#0x03
0x000001BC 2304      MOVS     r3,#0x04
    12:         len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
0x000001BE 2514      MOVS     r5,#0x14
0x000001C0 AC03      ADD      r4,sp,#0x0C
0x000001C2 E9CD4500  STRD     r4,r5,[sp,#0]
0x000001C6 F000F92F  BL.W     concatenateString (0x00000428)
    13:         for (; len3 > 0; len3 --)
0x000001CA E005      B        0x000001D8
    14:                 string3[len3 - 1] += 'A' - 'a';
0x000001CC 1821      ADDS     r1,r4,r0
0x000001CE F8112D01  LDRB     r2,[r1,#-0x01]!
0x000001D2 3A20      SUBS     r2,r2,#0x20
0x000001D4 700A      STRB     r2,[r1,#0x00]
0x000001D6 1E40      SUBS     r0,r0,#1
0x000001D8 2800      CMP      r0,#0x00
0x000001DA DCF7      BGT      0x000001CC
    15:         while(1);
```

optimization: `r0` is directly used, without push into and pop from the stack.

Use of volatile

- A volatile variable may change at any time, without any action from the code where the variable is currently used.
- Value may change due to: 值可能因为一下原因而改变
  - peripheral registers 外围寄存器
  - interrupt service routine 中断服务程序
  - another task in a multi-threaded application.

多线程应用中的另外一个任务

Use of volatile: an example

```
int main(void){
  const char *string1 = "problem solving";
  const char *string2 = "grammar book";
  char string3[MAX_LENGTH];
  int len1 = 3, len2 = 4;
  volatile int len3;
  len3 = concatenateString(string1, len1,
string2, len2, string3, MAX_LENGTH);
  while(1);
}
```

Disassembled code with level 0

```
     8:          const char *string1 = "problem solving";
0x000001B6 A420      ADR      r4,{pc}+2  ; @0x00000238
     9:          const char *string2 = "grammar book";
    10:          char string3[MAX_LENGTH];
0x000001B8 A523      ADR      r5,{pc}+4  ; @0x00000248
    11:          int len1 = 3, len2 = 4;
    12:          volatile int len3;
0x000001BA 2603      MOVS     r6,#0x03
0x000001BC 2704      MOVS     r7,#0x04
    13:          len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
0x000001BE 2014      MOVS     r0,#0x14
0x000001C0 A903      ADD      r1,sp,#0x0C
0x000001C2 463B      MOV      r3,r7
0x000001C4 462A      MOV      r2,r5
0x000001C6 E9CD1000  STRD     r1,r0,[sp,#0]
0x000001CA 4631      MOV      r1,r6
0x000001CC 4620      MOV      r0,r4
0x000001CE F000F92B  BL.W     concatenateString (0x00000428)
0x000001D2 9002      STR      r0,[sp,#0x08]
    14:          while(1);
```

`r0` is saved in the stack.

Disassembled code with level 1

```
     8:          const char *string1 = "problem solving";
0x000001B6 A012      ADR      r0,{pc}+2   ; @0x00000200
     9:          const char *string2 = "grammar book";
    10:          char string3[MAX_LENGTH];
0x000001B8 A215      ADR      r2,{pc}+4   ; @0x00000210
    11:          int len1 = 3, len2 = 4;
    12:          volatile int len3;
0x000001BA 2103      MOVS     r1,#0x03
0x000001BC 2304      MOVS     r3,#0x04
    13:          len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
0x000001BE 2414      MOVS     r4,#0x14
0x000001C0 AD03      ADD      r5,sp,#0x0C
0x000001C2 E9CD5400  STRD     r5,r4,[sp,#0]
0x000001C6 F000F92F  BL.W     concatenateString (0x00000428)
0x000001CA 9002      STR      r0,[sp,#0x08]
    14:          while(1);
```

`r0` is saved in the stack.

C file with Assembly code inside

C文件中内嵌汇编代码

- There are two possibilities:

有两种可能性：

1. inline assembly

内联汇编

   - ARM instructions are inserted into a C function.

   ARM指令被插入到一个C程序中

   - Goal: operations which are not available in C can be accomplished in assembly.

   目标：C中没有的操作可以在汇编中实现

2. embedded assembly

嵌入式汇编

   - a whole function in the C file is written in assembly

   整个C文件中的函数是用汇编编写的

   - Goal: optimization by hand, in order to produce more efficient code than the compiler.

   目标：手工优化，以产生比编译器效率更高的代码

Inline assembly

内联汇编

- Assembly code in a single line:

汇编代码在一行

```
__asm("instruction[;instruction]");
__asm{instruction[;instruction]}
```

- Assembly code in multiple lines:

汇编代码在多行

```
__asm
{
    instruction      /* comment */
    instruction    //another comment
}
```

Example

- In C, usually the opposite of a number is obtained by multiplying the number to -1

在C中，我们通过乘以-1的方式来获取一个数的相反数

```
int myNumber = 10;

myNumber = myNumber * (-1);
```

- The multiplication can be avoided by using the `MVN` instruction:

这个乘法可以避免，通过使用MVN指令

```
__asm("MVN myNumber, myNumber");

__asm{MVN myNumber, myNumber}
```

Example

```
int inlineAssembly(int value) {
  int var1, var2, res;
  __asm
  {
     LSR var1, value, 24
     AND var2, value, 0x000000FF
     ADD res, var1, var2
  }
return res;
}
```

内联汇编就是把汇编程序写到内部

Registers in inline assembler

- The inline assembler does not provide direct access to the physical registers.

内联汇编器，不提供直接访问物理寄存器

- Register names are treated as C variables.

寄存器的名字被当作C的变量一样看待

- If a register is used in the inline assembler but not declared as a C variable before, then the compiler generates a warning.

如果一个寄存器被使用在内联汇编器中，但是在之前并没有被声明位一个变量，那么编译器会发出警告

- Instead, if a variable is used before declaration, the compiler generates an error.

代替的，如果一个变量在声明之前被使用，这个编译器会产生一个错误

Example: undeclared variable

```
int inlineAssembly(int value) {
  int var2, res;
  __asm
  {
     LSR var1, value, 24
     AND var2, value, 0x000000FF
     ADD res, var1, var2
  }
return res;
}
```

```
Build target 'Target 1'
compiling C_functions.c...
C_functions.c(56): error:  #20: identifier "var1" is undefined
                LSR var1, value, #24
C_functions.c: 0 warnings, 1 error
".\ARMandC.axf" - 1 Error(s), 0 Warning(s).
Target not created
```

Example: undeclared "register"

```
int inlineAssembly(int value) {
  int var2, res;
  __asm
  {
     LSR r0, value, 24
     AND var2, value, 0x000000FF
     ADD res, r0, var2
  }
retur
}
```

```
Build target 'Target 1'
compiling C_functions.c...
C_functions.c(56): warning:  #1267-D: Implicit physical register R0 should be defined as a variable
                LSR r0, value, #24
C_functions.c: 1 warning, 0 errors
linking...
Program Size: Code=2788 RO-data=220 RW-data=0 ZI-data=608
".\ARMandC.axf" - 0 Error(s), 1 Warning(s).
```

Example: declared "register"

```
int inlineAssembly(int value) {
  int r0, var2, res;
  __asm
  {
     LSR r0, value, 24
     AND var2, value, 0x000000FF
     ADD res, r0, var2
  }
return res;
}
```

```
Build target 'Target 1'
compiling C_functions.c...
linking...
Program Size: Code=2788 RO-data=220 RW-data=0 ZI-data=608
".\ARMandC.axf" - 0 Error(s), 0 Warning(s).
```

Unsupported instructions

不支持的指令

- BX and BLX 带有寄存器的跳转
- LDR Rn, = expression
  Use MOV Rn, expression instead.
  LDR 指令被 MOV指令替代
- MUL, MLA, UMULL, UMLAL, SMULL, and SMLAL flag setting instructions
- MOV or MVN flag-setting instructions where the second operand is a constant
- ADR and ADRL.

嵌入式汇编

Embedded assembly

嵌入式程序集由C文件中的汇编函数组成，具有完整的函数原型

- Embedded assembly consists of assembly functions in a C file with full function prototypes, i.e., parameters and return value.

```
__asm type functionName(parameters)
{
    instruction
    instruction         ;comment
    ...
}
```

Parameters 参数

- Functions declared with `__asm` are called in the same way as normal C functions.
- Argument names cannot be used in the body of the embedded assembly function.

内嵌的汇编函数不能使用具有争议的名字

- The AAPCS rules apply:
  - the first 4 parameters are received in `r0`-`r3`.
  - further parameters are received in the stack.
  - the return value is passed in `r0`-`r3`.

Return value

- No return instructions are generated by the compiler for an `__asm` function.
- A return instruction in assembly code must be included in the body of the function, like:
  - a branch to the link register
  - the value of the link register assigned to the PC
- If the return instruction is missing, the program can fall through to the next function.

如果返回指令丢失，程序就会跳转到下一个函数

Example

```
__asm int embeddedAssembly(int value)
{
    LSR r1, r0, #24
    AND r2, r0, #0x000000FF
    ADD r0, r1, r2
    BX lr
}
```

内联和内嵌的区别在于有无高亮部分

- Embedded assembly gives access to the physical registers and the full instruction set.

嵌入式汇编提供直接访问物理寄存器，并且满指令集

Inline Vs embedded assembler

内联汇编抽象程度更高：产生的代码可能会与源代码不同，因为编译器会一起优化C和汇编代码

- Inline assembler works at high-level: generated code may differ from source code because the compiler optimizes C and assembly code together
- Inline assembler can not be used to optimize code. 内联汇编 不能使用优化代码
- Embedded assembly code is assembled separately from the C code. 嵌入式汇编代码与C代码是分开汇编的
- Inline assembly code can be inlined by the compiler, while embedded code cannot be inlined.

内联汇编代码，可以通过编译器内联，然而嵌入式代码则不能