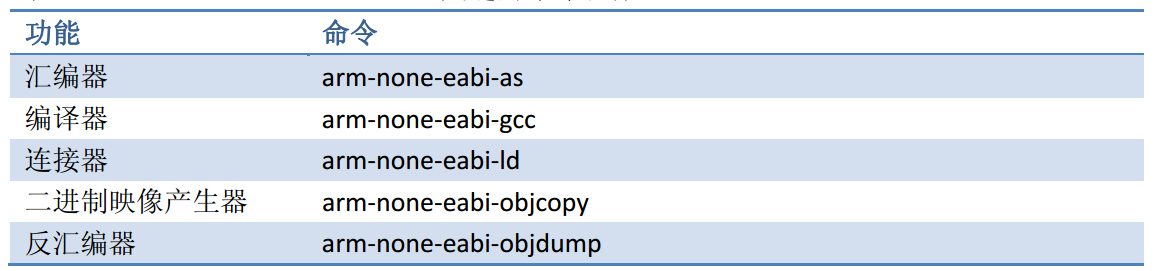
# 概述

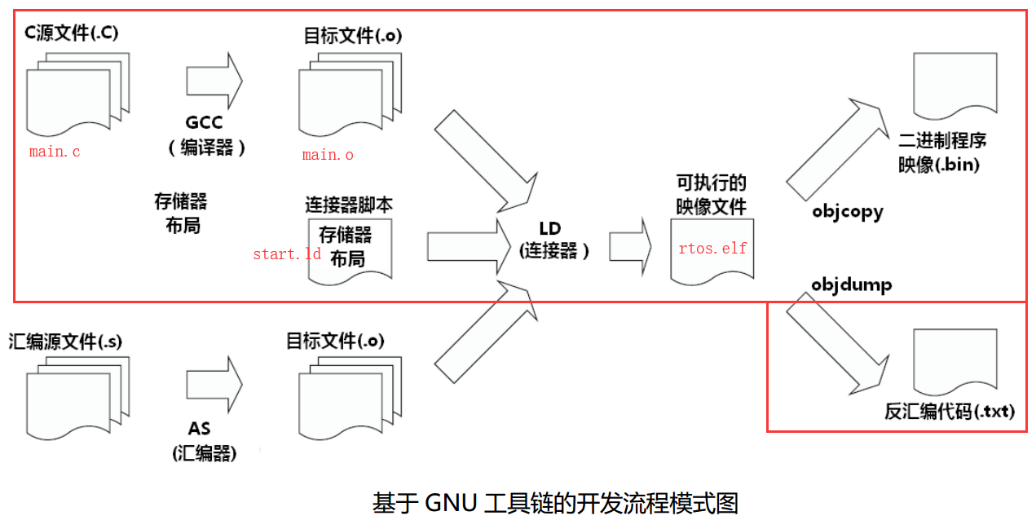
## 工程相关

本项目代码使用ARM指令集汇编编写。分析在ARM汇编中通过现有技术，能否解读出汇编中的函数、全局变量、局部变量以及函数调用关系等信息。 通过分析对获取信息的方法以及难易程度进行做出判断。

## 编译器及开发流程

编译器：

使用 CodeSourcery 的 GNU ARM 工具链，下载地址：<https://www.mentor.com/embedded-software/downloads/>。

开发流程：

puts. o

puts. s

因为是c语言所以只关心红色框里的内容。红色字体为本项目中的文件。

## 工程文件

本项目主要由Makefile、链接脚本、和源码文件（文件内容见[附录](#_附录)）组成。经过编译后可以通过QEMU运行和调试。

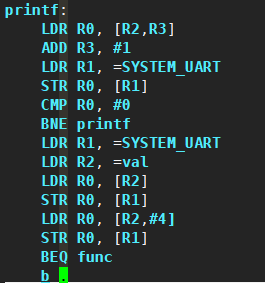
编译生成的文件：start.out（elf格式文件），start.bin（经过压缩的可执行文件），start.dump（反汇编后的文件）。

## 运行和调试

**运行：**执行make run后可以直接在QEMU中进行仿真。

**调试：**执行make debug 后进入调试状态，打开另一个终端输入arm-none-eabi-gdb回车进行远程调试。在gdb输入target remote localhost:1234回车，再输入file start.elf 即可进行调试。

## 函数

汇编函数的一个简单例子：

上图就是一个简单的汇编函数，第一行printf就相当于C语言中的函数名，在汇编中叫做标号，经过编译器编译后它们的表现形式是一样的。下面一部分是函数体。

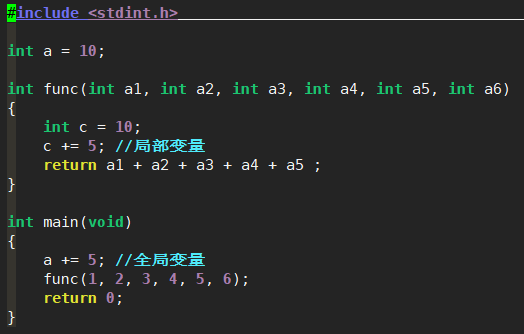
# 函数

## 目的

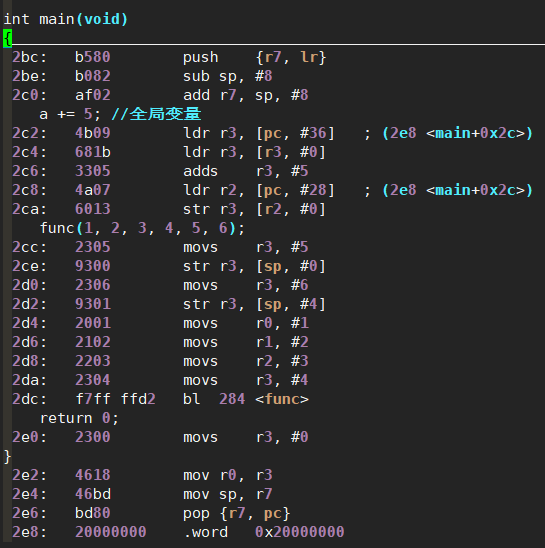
通过objdump（[4.3](#_objdump)）对ELF格式文件反汇编。

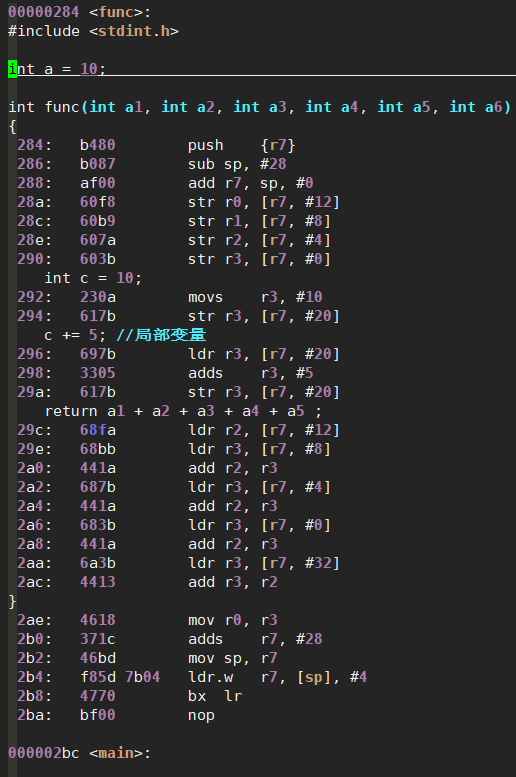
分析反汇编生成的文件能否解析出函数的函数名、参数、和返回值。

## 函数原型

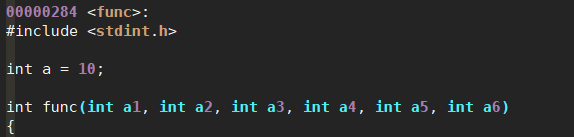
以一个简单的例子来解读ARM上函数在反汇编后的信息。

编译后再反汇编即可生成对应的汇编代码，以下主要对其汇编代码进行分析。

 反汇编后的main函数：

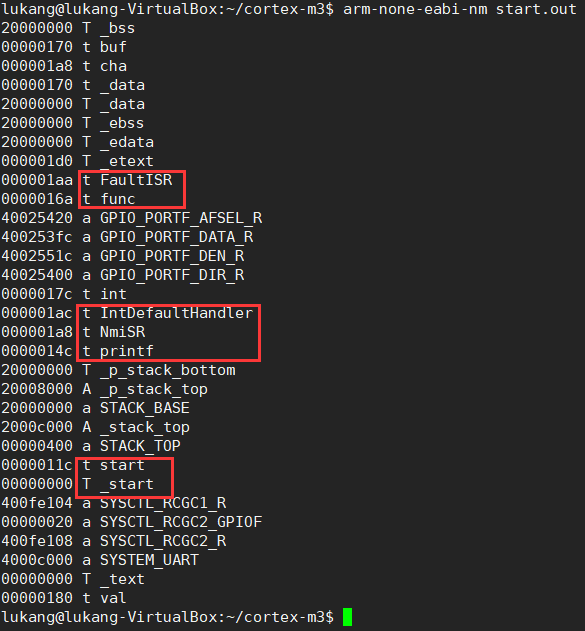
 反汇编后的func函数：

## 函数名

在汇编中函数名仅仅为一个标号，重要的是与标号对应的函数的地址信息。

这是编译时加调试信息，然后反汇编得到的文件。可以发现每个函数的开头都会有一个标号，当其他函数调用它时，bl指令会跳转到标号处执行函数体中的内容。由此可以推断出通过调用objdump命令，然后解析命令生成的反汇编文件，是可以获得函数名的此方法可行。

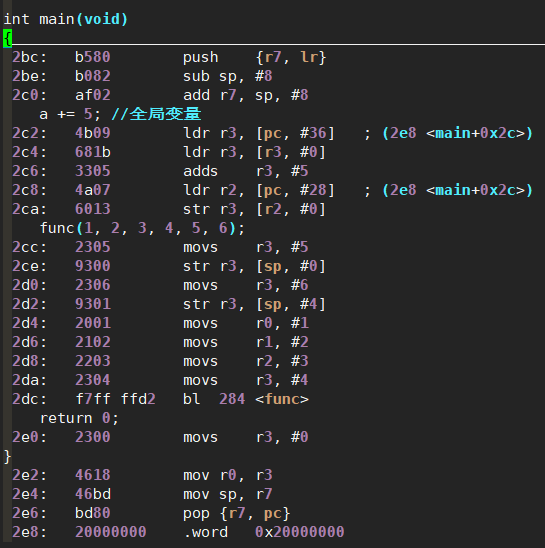
如果通过解析文本的方法来识别函数名工作量较大，还有一种简单的方法。通过命令arm-none-eabi-nm命令读取文件的符号表（[4.4](#_符号表)）来获取函数名信息。

如图：为nm命令生成的目标文件中的各种符号。

**结论**：通过nm命令可以更方便的获取函数名（红色标记部分为函数名），而且符号表中的其他信息还可以用来解析其他信息。（关于nm命令的详细用法在[4.2](#_格式说明)）

## 函数参数

**方法一：**

函数被调用时，主调函数将要传递的参数前4个参数保存到r0-r3寄存器，其他的入栈，然后跳转掉被调函数入口处。被调函数通过读取栈和寄存器的值，获取参数信息。

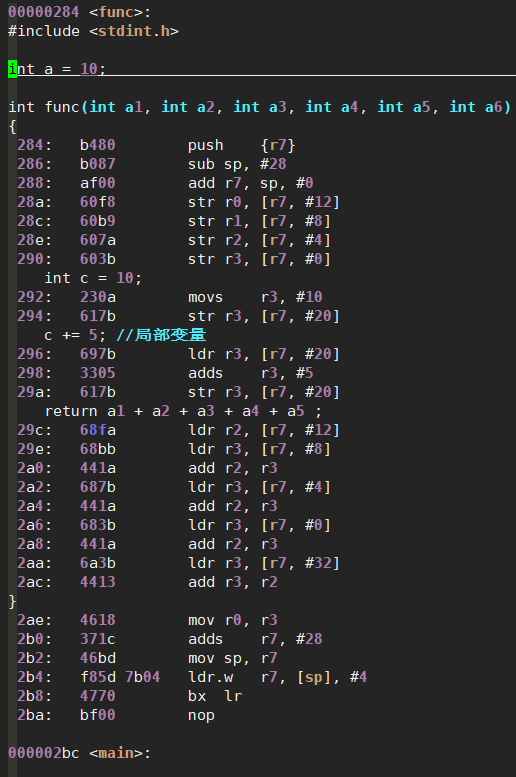
参数大于四个时使用寄存器和栈结合的方式传值。

将5、6入栈。

将1、2、3、4分别存放到r0-r3寄存器。

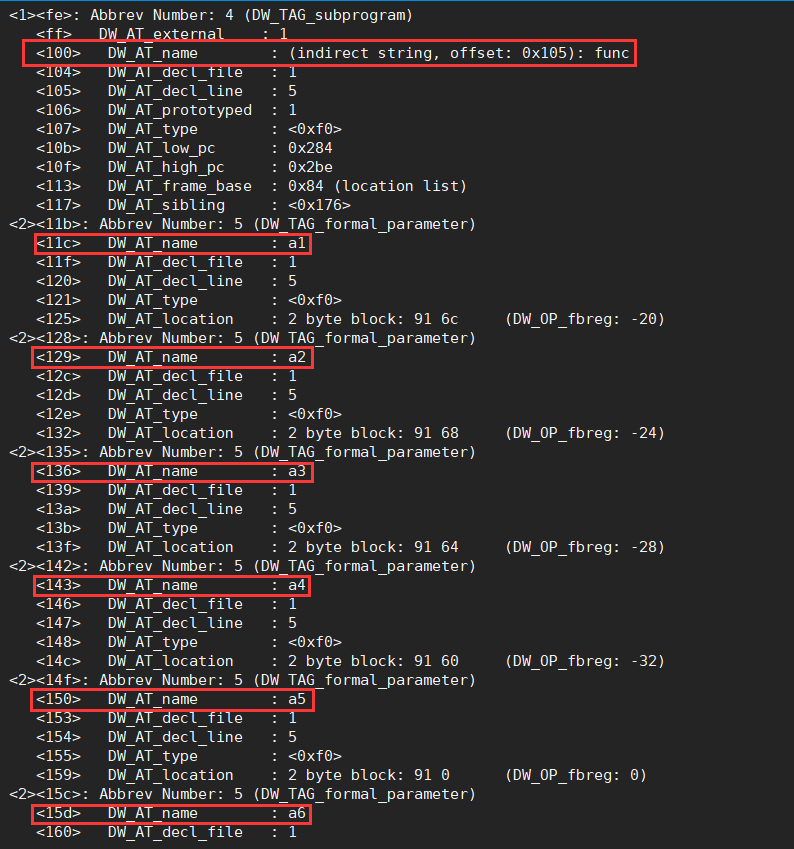
然后程序会跳转到func的入口处执行。

函数在被调用时参数如上图所示。

函数内部对参数的处理如上图所示。

在函数入口处，先将r7（栈帧指针寄存器）入栈，然后依次将寄存器中的值入栈。以后每次用到函数参数时会直接从栈中读取。

**方法二：**



使用“readelf –wi rtos.elf” 命令可以获取函数的详细信息。如左图所示，为func函数的详细信息，其中包括函数的参数。

## 返回值

可以清楚的看到函数从箭头指向位置开始准备返回，先将计算的结果保存在r0，然后是对sp栈指针的操作，最后通过bx指令返回函数被调用的位置。（根据AAPCS的规范，函数的返回值通常保存在R0中，若返回值为64位的，R1也用来保存返回值。）

# 函数调用

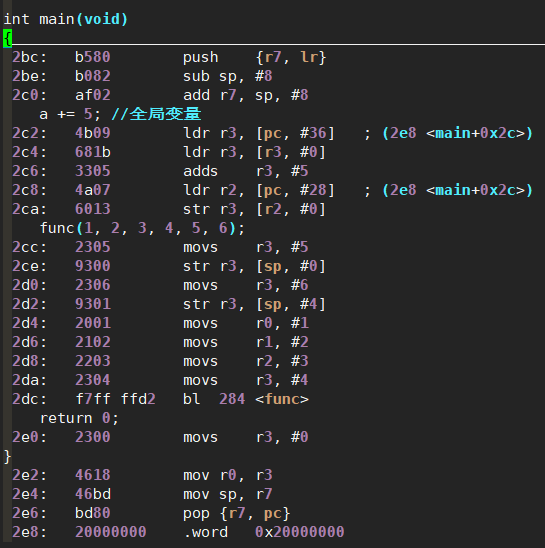
## 函数调用传参

函数调用过程中，当函数参数大于四个时，用栈和寄存器结合的方式传参。小于四个时用只用寄存器传参。（根据AAPCS的规范，R0 - R3寄存器用于传递参数，其他的参数通过系统栈来传递（除了VFP的参数）。）例：

参数超过四个时：流程分析如下。

1. main函数的汇编：

说明：fp叫做frame pointer寄存器，即栈帧指针寄存器；sp叫做stack pointer寄存器，即栈指针寄存器。在本例中因为加了编译选项，所以fp寄存器被r7寄存器替代。但效果相同

 关于寄存器的详细信息见[4.5](#_相关指令)

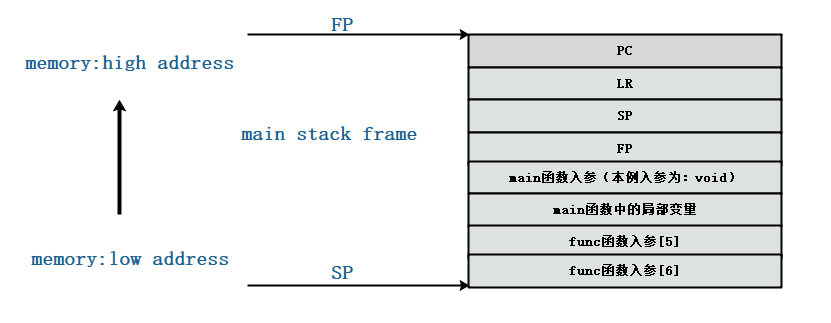
函数的参数如何传递在上面已经分析过，这里只对其栈的变化做分析。

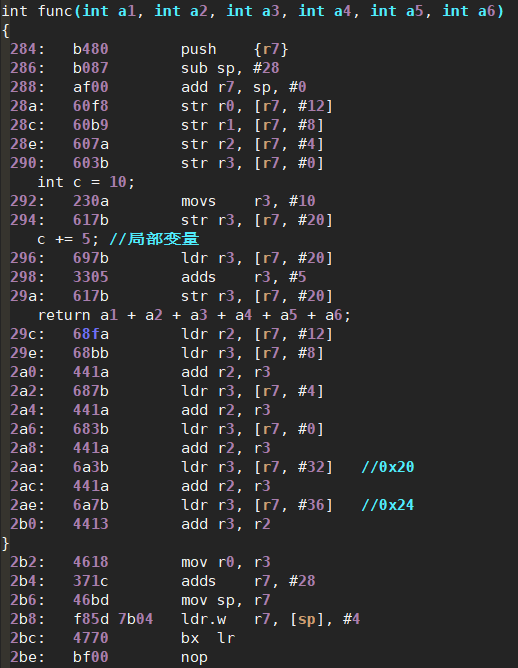
当main函数开始执行时，先将r7（栈帧指针寄存器）和lr（连接寄存器）入栈。然后对栈指针减8，为main函数开辟栈空间。减的值与所调用函数的参数个数、局部变量的个数有关。

本例因为此函数没有局部变量，所以减的值8，用于存放函数func最后两个值。

然后将r7指向的位置移动到sp+8，由此可见r7指向的是栈底，sp指向栈顶。开辟的空间大小为8字节。

当函数跳转之前main函数的栈帧中的状态如下图所示：



1. 函数 func的汇编如下：

func函数开始先将r7寄存器的值入栈，然后sp减28，开辟栈空间。

减20是因为有一个局部变量，还有在通过寄存器r0~r3传递过来的参数。

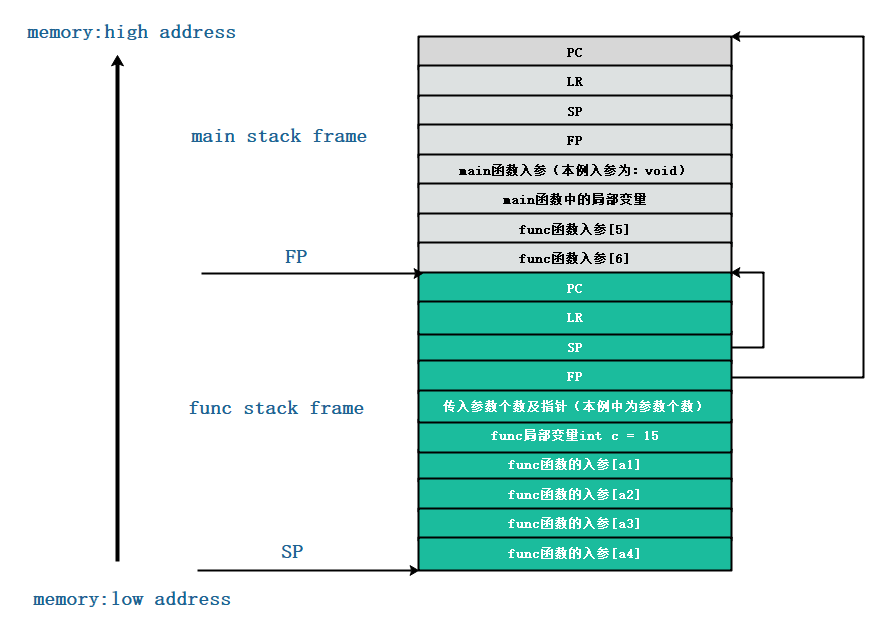
将r0~r3寄存器的值入栈。

将局部变量入栈

用栈中的数据计算结果，将最终结果保存在r3中。

函数返回时将运算的结果保存到r0寄存器然后修改sp，和r7寄存器的值，最后返回。

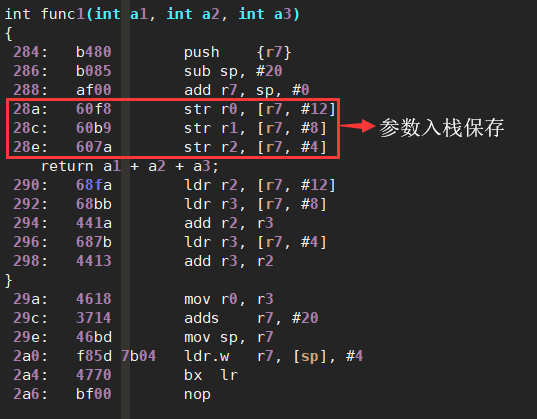
func函数在执行跳转指令前的函数栈帧下图所示。



对上图进行分析：

1. 在func函数的开始关键的时进行加法指令的一条语句，他将r7寄存器和sp寄存器指向同样的位置，之后通过r7寄存器的偏移来计算各个变量在栈中的位置。
2. 如上图当前函数中栈帧中存放的是，调用它的函数的栈帧位置信息。
3. 函数栈帧的概念，一个进程中每个函数所使用的栈空间是一个栈帧，所有的栈帧就组成了这个进程完整的栈，而fp就是栈基址寄存器，指向当前函数栈帧的栈底，sp则指向当前函数栈帧的栈顶。通过sp和fp所指出的栈帧可以恢复出母函数的栈帧，以此类推就可以backtrace出所有函数的调用顺序。

参数小于等于四个时：

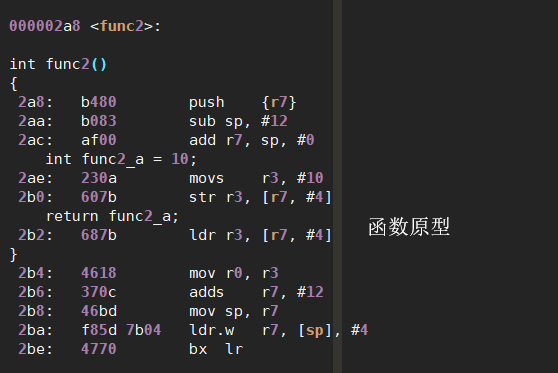
 

函数调用

函数原型

将要传递的参数从左往右，依次放入寄存器中然后跳转到子程序入口。其栈的分布与大于四个参数时栈的分布类似。

没有参数：

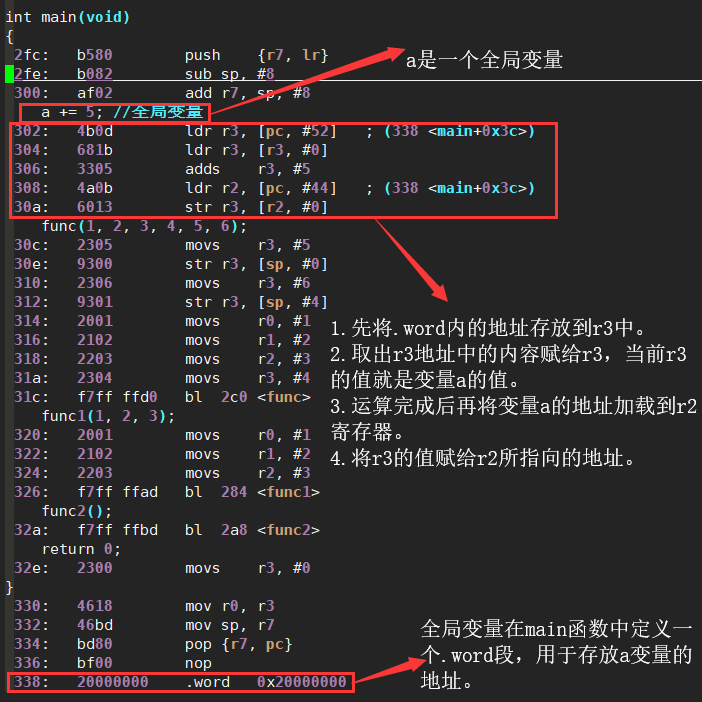


## 引用全局变量

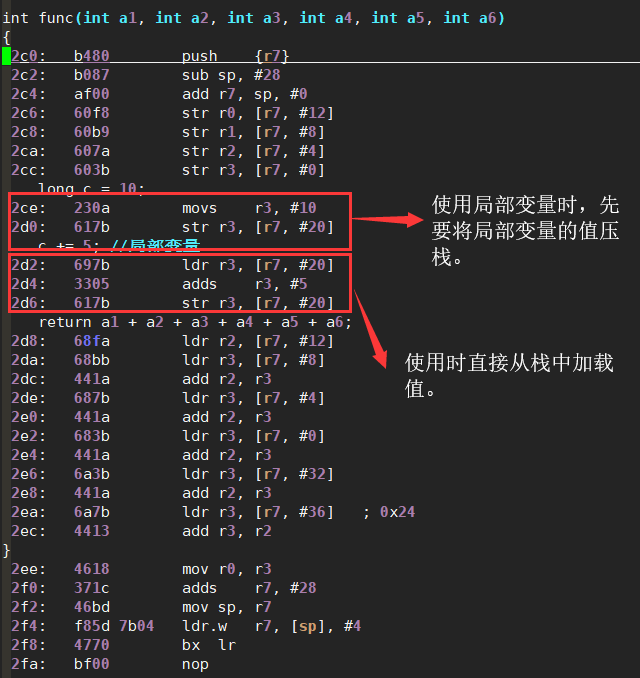
**全局变量和局部变量的区分：**

1. 通过符号表中的信息判断，只有全局变量会出现在符号表中。全局变量位于非代码段，非栈段，而位于data段（初始化）或者bss段（未初始化）。
2. 通过反汇编后的文本分析，局部变量在使用前要在栈中为它开辟空间，函数返回时要释放。
3. 局部变量会在函数内部定义使用，访问时通过寄存器直接访问。

**全局变量的使用：**



上图是全局变量的使用过程，可以看到在访问全局变量时，编译器会将全局变量映射到函数内部，然后通过函数的地址加偏移量来访问。

**局部变量的使用：**

## 函数调用

**函数调用常用的指令：**

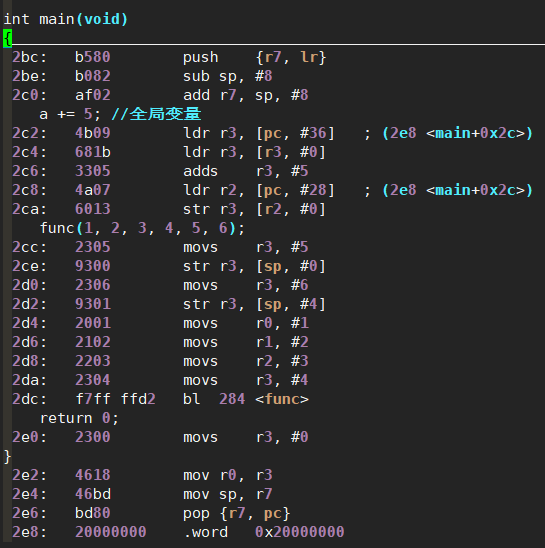
bl指令：bl指令是带链接的跳转。 首先将当前指令的下一条指令地址保存在LR寄存器，然后跳转的lable。通常用于调用子程序，可通过在子程序的尾部添加mov  pc, lr 返回。BL 是另一个跳转指令，但跳转之前，会在寄存器R14 中保存PC 的当前内容，因此，可以通过将R14 的内容重新加载到PC 中，来返回到跳转指令之后的那个指令处执行。该指令是实现子程序调用的一个基本但常用的手段。

**其他跳转指令：**

1. B：B 指令是最简单的跳转指令。一旦遇到一个B 指令，ARM 处理器将立即跳转到给定的目标地址，从那里继续执行。注意存储在跳转指令中的实际值是相对当前PC 值的一个偏移量，而不是一个绝对地址，它的值由汇编器来计算（参考寻址方式中的相对寻址）。它是 24 位有符号数，左移两位后有符号扩展为 32 位，表示的有效偏移为 26 位(前后32MB 的地址空间)。
2. BX： 带状态切换的跳转。最低位为1时，切换到Thumb指令执行，为0时，解释为ARM指令执行。
3. BLX:  带链接和状态切换的跳转。结合了BX与BL功能。
4. 还有其他的如BEQ、BNE等条件跳转指令其本质和B跳转指令一样，只是多了一个检测CPSR寄存器中Z标志位的功能。Z标志位等于零时, 跳转到BEQ后标签处，标志寄存器中Z标志位不等于零时, 跳转到BNE后标签处。

通过反汇编找出函数调用关系：

可以通过判断函数中的bl跳转指令来确定函数的调用关系，但要注意区分函数内部的分支判断和循环的跳转。如下图：



此处为函数调用。

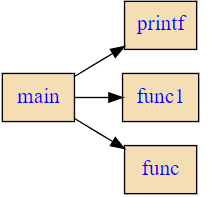
main函数中调用了func函数，这个很容易区分，objdump工具会在跳转指令后加上要跳转到的目标函数名。函数内部的跳转则是通过以当前函数的地址为基地址，加上当前的pc指针的值来确定要跳转的位置。

结论：可以通过反汇编代码来确定函数的调用关系。但要注意区分函数内部的跳转。

## 第三方工具

在Linux下可以通过如下方式方便的获取c代码的函数调用关系图：

1. 安装包：
   1. sudo apt-get install cflow graphviz
   2. wget -c <https://github.com/tinyclub/linux-0.11-lab/raw/master/tools/tree2dotx>
   3. wget -c https://github.com/tinyclub/linux-0.11-lab/raw/master/tools/callgraph
   4. sudo cp tree2dotx callgraph /usr/local/bin
   5. sudo chmod +x /usr/local/bin/{tree2dotx,callgraph}
   6. 使用tree2dotx能会出问题需要安装包：sudo apt-get install gawk
2. 使用：
   1. callgraph -f [函数名] 如下图：是一个简单的例子：



# 指令解析

## 跳转

跳转指令分析：

1. 以如下代码为例：

.text

.global \_start

\_start:

b step1

step1:

b step1

1. 将其链接到0x00000000地址生成反汇编代码如下：

00000000 <\_start>:

0: eaffffff b 4

4: e59ff000 ldr pc, [pc, #0] ; c <step2+0x4>

8: eafffffe b 8 <step2>

c: 00000008 andeq r0, r0, r8

1. 源代码中”b step1”的指令机器码为：

偏移量

1

L

0

1

条件码

1. [31:28] 位是条件码。
2. [27:24] 位是指令机器码，b为”1010”，bl为”1011”。（其24位的L，表示是否使用link（R14）寄存器拷贝pc的内容，这个是为了程序返回时的准备，该拷贝过程是由硬件自动完成的，该位置一，表示复制，为0表示不复制，该位一个典型应用是中断返回地址的copy，那么31-28的1110对应的意义为cond，不仅仅跳转指令有cond，数据操作类指令机器码中均存在该位，并且意义是相同的。）
3. [23:0] 表示偏移地址。
4. 使用b或bl跳转时，下一条指令的地址的计算

将指令中24位带符号的补码扩展为32位（扩展其符号位）；然后将此32位数左移2位；最后将得到的值加到pc寄存器中，即得到跳转的目标地址。

比如第一条指令“b step1”的机器码为eaffffff。

24位带符号的补码为0xffffff，将其扩展为32位得到0xffffffff。

将此32位左移2位得到0xfffffffc，其原码就是-4。

pc的值是当前指令的下两条指令的地址，加上上面得到的-4，这恰好是step1的地址。

BL和B跳转指令偏移值的计算：

1. 原理:

**假设跳转指令处的地址是A,跳转目标处的地址是B.**  
B,BL指令保存的是偏移地址,这个地址的计算方法是:

* 1. B-(A+8)。A+8是因为ARM的流水线使得指令执行到A处时，PC实际的值是A+8。
  2. 第一步得到的值是4的倍数，因为ARM的指令是4对齐的，即最低两位为00。于是将这个值右移两位。得到最终偏移。

1. 执行：

取出偏移。左移两位。加入PC,这时PC的值刚好为目标处的地址值，即目标地址指令进入取指，流水线前两级被清空。

## nm工具

**nm工具的使用：**

-a   按照man手册，仅列出调试信息，实际上却是调试信息+正常信息

-A   增加一列显示目标文件，没有实际意义

-C   将低级符号信息编码成便于查看的用户信息，最终效果是符号清单呈现出class::memberfunction形式的信息，最常用的选项

-D   显示动态符号，以目标文件为so为例，显示了so中增加的符号信息以及so引用的so的符号信息

-f   指定不同的输出格式，改变命令输出格式

-g   仅显示外部符号，实测和-D选项输出几乎一样

-l   显示符号的行号，附加一列显示符号在xx文件的xx行定义    较常用

-n   将符号按照地址进行排序

-p   不排序符号，使用该选项后的输出没有按照地址也没有按照符号名称排序

-r   逆序输出符号

-u   仅显示未定义符号，实测列出了GLIBC中的memset、memcpy等，不清楚具体作用

-h   显示帮助

-V   版本信息

**nm生成信息的解析：**

输出符号类型说明

　　A

　　该符号的值是绝对的，在以后的链接过程中，不允许进行改变。这样的符号值，常常出现在中断向量表中，例如用符号来表示各个中断向量函数在中断向量表中的位置。

　　B

　　该符号的值出现在非初始化数据段(bss)中。例如，在一个文件中定义全局static int test。则该符号test的类型为b，位于bss section中。其值表示该符号在bss段中的偏移。一般而言，bss段分配于RAM中

　　C

　　该符号为common。common symbol是未初始化数据段。该符号没有包含于一个普通section中。只有在链接过程中才进行分配。符号的值表示该符号需要的字节数。例如在一个c文件中，定义int test，并且该符号在别的地方会被引用，则该符号类型即为C。否则其类型为B。

　　D

　　该符号位于初始话数据段中。一般来说，分配到data section中。例如定义全局int baud\_table[5] = {9600, 19200, 38400, 57600, 115200}，则会分配于初始化数据段中。

　　G

　　该符号也位于初始化数据段中。主要用于small object提高访问small data object的一种方式。

　　I

　　该符号是对另一个符号的间接引用。

　　N

　　该符号是一个debugging符号。

　　R

　　该符号位于只读数据区。例如定义全局const int test[] = {123, 123};则test就是一个只读数据区的符号。注意在cygwin下如果使用gcc直接编译成MZ格式时，源文件中的test对应\_test，并且其符号类型为D，即初始化数据段中。但是如果使用m6812-elf-gcc这样的交叉编译工具，源文件中的test对应目标文件的test,即没有添加下划线，并且其符号类型为R。一般而言，位于rodata section。值得注意的是，如果在一个函数中定义const char \*test = “abc”, const char test\_int = 3。使用nm都不会得到符号信息，但是字符串“abc”分配于只读存储器中，test在rodata section中，大小为4。

　　S

　　符号位于非初始化数据区，用于small object。

　　T

　　该符号位于代码区text section。

　　U

　　该符号在当前文件中是未定义的，即该符号的定义在别的文件中。例如，当前文件调用另一个文件中定义的函数，在这个被调用的函数在当前就是未定义的；但是在定义它的文件中类型是T。但是对于全局变量来说，在定义它的文件中，其符号类型为C，在使用它的文件中，其类型为U。

　　V

　　该符号是一个weak object。

　　W

　　The symbol is a weak symbol that has not been specifically tagged as a weak object symbol.

　　-

　　该符号是a.out格式文件中的stabs symbol。

　　?

该符号类型没有定义

## objdump

**作用：**objdump命令是Linux下的反汇编目标文件或者可执行文件的命令，它以一种可阅读的格式让你更多地了解二进制文件可能带有的附加信息。

**使用：**

--archive-headers

-a

显示档案库的成员信息,类似ls -l将lib\*.a的信息列出。

-b bfdname

--target=bfdname

指定目标码格式。这不是必须的，objdump能自动识别许多格式，比如：

objdump -b oasys -m vax -h fu.o

显示fu.o的头部摘要信息，明确指出该文件是Vax系统下用Oasys编译器生成的目标文件。objdump -i将给出这里可以指定的目标码格式列表。

-C

--demangle

将底层的符号名解码成用户级名字，除了去掉所开头的下划线之外，还使得C++函数名以可理解的方式显示出来。

--debugging

-g

显示调试信息。企图解析保存在文件中的调试信息并以C语言的语法显示出来。仅仅支持某些类型的调试信息。有些其他的格式被readelf -w支持。

-e

--debugging-tags

类似-g选项，但是生成的信息是和ctags工具相兼容的格式。

--disassemble

-d

从objfile中反汇编那些特定指令机器码的section。

-D

--disassemble-all

与 -d 类似，但反汇编所有section.

--prefix-addresses

反汇编的时候，显示每一行的完整地址。这是一种比较老的反汇编格式。

-EB

-EL

--endian={big|little}

指定目标文件的小端。这个项将影响反汇编出来的指令。在反汇编的文件没描述小端信息的时候用。例如S-records.

-f

--file-headers

显示objfile中每个文件的整体头部摘要信息。

-h

--section-headers

--headers

显示目标文件各个section的头部摘要信息。

-H

--help

简短的帮助信息。

-i

--info

显示对于 -b 或者 -m 选项可用的架构和目标格式列表。

-j name

--section=name

仅仅显示指定名称为name的section的信息

-l

--line-numbers

用文件名和行号标注相应的目标代码，仅仅和-d、-D或者-r一起使用使用-ld和使用-d的区别不是很大，在源码级调试的时候有用，要求编译时使用了-g之类的调试编译选项。

-m machine

--architecture=machine

指定反汇编目标文件时使用的架构，当待反汇编文件本身没描述架构信息的时候(比如S-records)，这个选项很有用。可以用-i选项列出这里能够指定的架构.

--reloc

-r

显示文件的重定位入口。如果和-d或者-D一起使用，重定位部分以反汇编后的格式显示出来。

--dynamic-reloc

-R

显示文件的动态重定位入口，仅仅对于动态目标文件意义，比如某些共享库。

-s

--full-contents

显示指定section的完整内容。默认所有的非空section都会被显示。

-S

--source

尽可能反汇编出源代码，尤其当编译的时候指定了-g这种调试参数时，效果比较明显。隐含了-d参数。

--show-raw-insn

反汇编的时候，显示每条汇编指令对应的机器码，如不指定--prefix-addresses，这将是缺省选项。

--no-show-raw-insn

反汇编时，不显示汇编指令的机器码，如不指定--prefix-addresses，这将是缺省选项。

--start-address=address

从指定地址开始显示数据，该选项影响-d、-r和-s选项的输出。

--stop-address=address

显示数据直到指定地址为止，该项影响-d、-r和-s选项的输出。

-t

--syms

显示文件的符号表入口。类似于nm -s提供的信息

-T

--dynamic-syms

显示文件的动态符号表入口，仅仅对动态目标文件意义，比如某些共享库。它显示的信息类似于 nm -D|--dynamic 显示的信息。

-V

--version

版本信息

--all-headers

-x

显示所可用的头信息，包括符号表、重定位入口。-x 等价于-a -f -h -r -t 同时指定。

-z

--disassemble-zeroes

一般反汇编输出将省略大块的零，该选项使得这些零块也被反汇编。

## 符号表

**符号表中各段的含义：**

* **.text**：已编译程序的机器代码。
* **.rodata**：只读数据，比如printf语句中的格式串和开关（switch）语句的跳转表。
* **.data**：已初始化的全局C变量。局部C变量在运行时被保存在栈中，既不出现在.data中，也不出现在.bss节中。
* **.bss**：未初始化的全局C变量。在目标文件中这个节不占据实际的空间，它仅仅是一个占位符。目标文件格式区分**初始化**和**未初始化**变量是为了空间效率在：在目标文件中，未初始化变量不需要占据任何实际的磁盘空间。
* **.symtab**：一个符号表（symbol table），它存放在程序中被定义和引用的函数和全局变量的信息。一些程序员错误地认为必须通过-g选项来编译一个程序，得到符号表信息。实际上，每个可重定位目标文件在.symtab中都有一张符号表。然而，和编译器中的符号表不同，.symtab符号表不包含局部变量的表目。
* **.rel.text**：当链接噐把这个目标文件和其他文件结合时，.text节中的许多位置都需要修改。一般而言，任何调用外部函数或者引用全局变量的指令都需要修改。另一方面调用本地函数的指令则不需要修改。注意，可执行目标文件中并不需要重定位信息，因此通常省略，除非使用者显式地指示链接器包含这些信息。
* **.rel.data**：被模块定义或引用的任何全局变量的信息。一般而言，任何已初始化全局变量的初始值是全局变量或者外部定义函数的地址都需要被修改。
* **.debug**：一个调试符号表，其有些表目是程序中定义的局部变量和类型定义，有些表目是程序中定义和引用的全局变量，有些是原始的C源文件。只有以-g选项调用编译驱动程序时，才会得到这张表。
* **.line**：原始C源程序中的行号和.text节中机器指令之间的映射。只有以-g选项调用编译驱动程序时，才会得到这张表。
* **.strtab**：一个字符串表，其内容包括.symtab和.debug节中的符号表，以及节头部中的节名字。字符串表就是以null结尾的字符串序列。

## 关键的寄存器

1. 堆栈指针r13（SP）：每一种异常模式都有其自己独立的r13，它通常指向异常模式所专用的堆栈，也就是说五种异常模式、非异常模式（用户模式和系统模式），都有各自独立的堆栈，用不同的堆栈指针来索引。这样当ARM进入异常模式的时候，程序就可以把一般通用寄存器压入堆栈，返回时再出栈，保证了各种模式下程序的状态的完整性。
2. 连接寄存器r14（LR）：每种模式下r14都有自身版组，它有两个特殊功能。
   1. 保存子程序返回地址。使用BL或BLX时，跳转指令自动把返回地址放入r14中；子程序通过把r14复制到PC来实现返回，通常用下列指令之一：  
                              MOV PC, LR   
                              BX LR

     通常子程序这样写，保证了子程序中还可以调用子程序。  
                         stmfd sp!, {lr}  
                          ……  
                          ldmfd sp!, {pc}

* 1. 当异常发生时，异常模式的r14用来保存异常返回地址，将r14如栈可以处理嵌套中断。

1. 栈帧指针寄存器r11（FP）： stack frame就是一个函数所使用的stack的一部分，所有函数的stack frame串起来就组成了一个完整的栈。stack frame的两个边界分别由FP和SP来限定。在程序执行过程中（通常是发生了某种意外情况而需要进行调试），通过SP和FP所限定的stack frame，就可以得到母函数的SP和FP，从而得到母函数的stack frame（PC，LR，SP，FP会在函数调用的第一时间压栈），以此追溯，即可得到所有函数的调用顺序。

# 结论

|  |  |  |  |
| --- | --- | --- | --- |
|  | 难易程度 | 所用工具 | 编码实现 |
| 函数名 | 容易 | readelf | 输入：readelf命令的输出。 输出：函数名。 |
| 函数参数 | 容易 | readelf;objdump | 输入：反汇编后的文件和函数名。  输出：每个函数的入参个数。 |
| 返回值 |  |  |  |
| 函数调用关系 | 较难 | readelf;objdump | 输入：函数名、函数入口地址和反汇编文件信息。  输出：函数调用关系 |
| 全局变量 | 容易 | readelf -wi | 输入：readelf命令的输出。  输出：全局变量名。 |
| 局部变量 | 容易 | readelf -wi | 输入：readelf命令的输出。  输出：局部变量名。 |

根据本篇文章综合来看：

1. 函数名：有多种途径可以获得，其中获取符号表进行解析的方法比较简单。
2. 函数参数：函数的参数，通过“readelf -wi”命令可以获取函数的详细信息，其中包含函数的参数，可以通过解析得到。
3. 函数调用关系：先获取函数名后，通过函数名检索反汇编文本中的跳转指令，得到函数调用关系。
4. 全局变量和局部变量：同获取函数参数，通过解析命令生成的文本获得。

# 附录

编译的Makefile文件如下表所示：

|  |
| --- |
| start.bin:start.out  arm-none-eabi-objcopy start.out -O binary start.bin  arm-none-eabi-objdump -S start.out > start.dump  start.out:start.o start.ld  arm-none-eabi-ld -T start.ld -o start.out start.o  start.o:start.s  arm-none-eabi-as -mcpu=cortex-m3 -c -mthumb -g -mthumb-interwork start.s -o start.o  run:start.bin  qemu-system-arm -M lm3s6965evb --kernel start.bin -nographic  debug:start.bin  qemu-system-arm -M lm3s6965evb --kernel start.bin -nographic -s -S  clean:  rm -rf start.out start.o start.dump start.dump start.bin |

链接文件start.ld如下表所示：

|  |
| --- |
| MEMORY  {  FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 256K  SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 256K  }  SECTIONS  {  .text :  {  \_text = .;  KEEP(\*(.isr\_vector))  \*(.text\*)  \*(.rodata\*)  \_etext = .;  } > FLASH  /DISCARD/ :  {  \*(.ARM.exidx\*)  \*(.gnu.linkonce.armexidx.\*)  }  .data : AT(ADDR(.text) + SIZEOF(.text))  {  \_data = .;  \*(vtable)  \*(.data\*)  \_edata = .;  } > SRAM  .bss :  {  \_bss = .;  \*(.bss\*)  \*(COMMON)  \_ebss = .;  } > SRAM  . = ALIGN(32); /\*Not sure if this needs to be done, but why not.\*/  \_p\_stack\_bottom = .;  . = . + 0x4000;  \_p\_stack\_top = 0x20008000;  . = . + 0x4000; /\*Allocate 4K for the Stack.\*/  \_stack\_top = 0x2000c000; /\*Address of the top of the heap, also end of RAM.\*/  } |

int\_vector.c文件如下表所示：

|  |
| --- |
| void ResetISR(void);  static void NmiSR(void);  static void FaultISR(void);  static void IntDefaultHandler(void);  extern unsigned long \_etext;  extern unsigned long \_data;  extern unsigned long \_edata;  extern unsigned long \_stack\_bottom;  extern unsigned long \_stack\_top;  extern void reset\_handler(void);  extern void systick\_handler(void);  extern void main(void);  \_\_attribute\_\_ ((section(".isr\_vector")))void (\*g\_pfnVectors[])(void) =  {  0x2000c000, // StackPtr, set in RestetISR  main, // The reset handler  NmiSR, // The NMI handler  FaultISR, // The hard fault handler  IntDefaultHandler, // The MPU fault handler  IntDefaultHandler, // The bus fault handler  IntDefaultHandler, // The usage fault handler  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  IntDefaultHandler, // SVCall handler  IntDefaultHandler, // Debug monitor handler  0, // Reserved  IntDefaultHandler, // The PendSV handler  IntDefaultHandler, // The SysTick handler  IntDefaultHandler, // GPIO Port A  IntDefaultHandler, // GPIO Port B  IntDefaultHandler, // GPIO Port C  IntDefaultHandler, // GPIO Port D  IntDefaultHandler, // GPIO Port E  IntDefaultHandler, // UART0 Rx and Tx  IntDefaultHandler, // UART1 Rx and Tx  IntDefaultHandler, // SSI0 Rx and Tx  IntDefaultHandler, // I2C0 Master and Slave  IntDefaultHandler, // PWM Fault  IntDefaultHandler, // PWM Generator 0  IntDefaultHandler, // PWM Generator 1  IntDefaultHandler, // PWM Generator 2  IntDefaultHandler, // Quadrature Encoder 0  IntDefaultHandler, // ADC Sequence 0  IntDefaultHandler, // ADC Sequence 1  IntDefaultHandler, // ADC Sequence 2  IntDefaultHandler, // ADC Sequence 3  IntDefaultHandler, // Watchdog timer  IntDefaultHandler, // Timer 0 subtimer A  IntDefaultHandler, // Timer 0 subtimer B  IntDefaultHandler, // Timer 1 subtimer A  IntDefaultHandler, // Timer 1 subtimer B  IntDefaultHandler, // Timer 2 subtimer A  IntDefaultHandler, // Timer 2 subtimer B  IntDefaultHandler, // Analog Comparator 0  IntDefaultHandler, // Analog Comparator 1  IntDefaultHandler, // Analog Comparator 2  IntDefaultHandler, // System Control (PLL, OSC, BO)  IntDefaultHandler, // FLASH Control  IntDefaultHandler, // GPIO Port F  IntDefaultHandler, // GPIO Port G  IntDefaultHandler, // GPIO Port H  IntDefaultHandler, // UART2 Rx and Tx  IntDefaultHandler, // SSI1 Rx and Tx  IntDefaultHandler, // Timer 3 subtimer A  IntDefaultHandler, // Timer 3 subtimer B  IntDefaultHandler, // I2C1 Master and Slave  IntDefaultHandler, // Quadrature Encoder 1  IntDefaultHandler, // CAN0  IntDefaultHandler, // CAN1  IntDefaultHandler, // CAN2  IntDefaultHandler, // Ethernet  IntDefaultHandler, // Hibernate  IntDefaultHandler, // USB0  IntDefaultHandler, // PWM Generator 3  IntDefaultHandler, // uDMA Software Transfer  IntDefaultHandler, // uDMA Error  IntDefaultHandler, // ADC1 Sequence 0  IntDefaultHandler, // ADC1 Sequence 1  IntDefaultHandler, // ADC1 Sequence 2  IntDefaultHandler, // ADC1 Sequence 3  IntDefaultHandler, // I2S0  IntDefaultHandler, // External Bus Interface 0  IntDefaultHandler, // GPIO Port J  IntDefaultHandler, // GPIO Port K  IntDefaultHandler, // GPIO Port L  IntDefaultHandler, // SSI2 Rx and Tx  IntDefaultHandler, // SSI3 Rx and Tx  IntDefaultHandler, // UART3 Rx and Tx  IntDefaultHandler, // UART4 Rx and Tx  IntDefaultHandler, // UART5 Rx and Tx  IntDefaultHandler, // UART6 Rx and Tx  IntDefaultHandler, // UART7 Rx and Tx  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  IntDefaultHandler, // I2C2 Master and Slave  IntDefaultHandler, // I2C3 Master and Slave  IntDefaultHandler, // Timer 4 subtimer A  IntDefaultHandler, // Timer 4 subtimer B  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  0, // Reserved  IntDefaultHandler, // Timer 5 subtimer A  IntDefaultHandler, // Timer 5 subtimer B  IntDefaultHandler, // Wide Timer 0 subtimer A  IntDefaultHandler, // Wide Timer 0 subtimer B  IntDefaultHandler, // Wide Timer 1 subtimer A  IntDefaultHandler, // Wide Timer 1 subtimer B  IntDefaultHandler, // Wide Timer 2 subtimer A  IntDefaultHandler, // Wide Timer 2 subtimer B  IntDefaultHandler, // Wide Timer 3 subtimer A  IntDefaultHandler, // Wide Timer 3 subtimer B  IntDefaultHandler, // Wide Timer 4 subtimer A  IntDefaultHandler, // Wide Timer 4 subtimer B  IntDefaultHandler, // Wide Timer 5 subtimer A  IntDefaultHandler, // Wide Timer 5 subtimer B  IntDefaultHandler, // FPU  IntDefaultHandler, // PECI 0  IntDefaultHandler, // LPC 0  IntDefaultHandler, // I2C4 Master and Slave  IntDefaultHandler, // I2C5 Master and Slave  IntDefaultHandler, // GPIO Port M  IntDefaultHandler, // GPIO Port N  IntDefaultHandler, // Quadrature Encoder 2  IntDefaultHandler, // Fan 0  0, // Reserved  IntDefaultHandler, // GPIO Port P (Summary or P0)  IntDefaultHandler, // GPIO Port P1  IntDefaultHandler, // GPIO Port P2  IntDefaultHandler, // GPIO Port P3  IntDefaultHandler, // GPIO Port P4  IntDefaultHandler, // GPIO Port P5  IntDefaultHandler, // GPIO Port P6  IntDefaultHandler, // GPIO Port P7  IntDefaultHandler, // GPIO Port Q (Summary or Q0)  IntDefaultHandler, // GPIO Port Q1  IntDefaultHandler, // GPIO Port Q2  IntDefaultHandler, // GPIO Port Q3  IntDefaultHandler, // GPIO Port Q4  IntDefaultHandler, // GPIO Port Q5  IntDefaultHandler, // GPIO Port Q6  IntDefaultHandler, // GPIO Port Q7  IntDefaultHandler, // GPIO Port R  IntDefaultHandler, // GPIO Port S  IntDefaultHandler, // PWM 1 Generator 0  IntDefaultHandler, // PWM 1 Generator 1  IntDefaultHandler, // PWM 1 Generator 2  IntDefaultHandler, // PWM 1 Generator 3  IntDefaultHandler // PWM 1 Fault  };  static void NmiSR(void){  while(1);  }  static void FaultISR(void){  while(1);  }  static void IntDefaultHandler(void){  while(1);  } |

main.c文件如下表所示：

|  |
| --- |
| #include <stdint.h>  int a = 10;  int func(int a1, int a2, int a3, int a4, int a5, int a6)  {  long c = 10;  c += 5; //局部变量  return a1 + a2 + a3 + a4 + a5 + a6;  }  int main(void)  {  a += 5; //全局变量  func(1, 2, 3, 4, 5, 6);  return 0;  } |