第19章

# 使用GNU工具链开始Cortex-M3开发

* 背景
* 获取GNU工具链
* 开发流程
* 示例程序
* 访问特殊功能寄存器
* 使用未支持的指令
* GNU C编译器的内联汇编

## 19.1 背景

GNU工具链在ARM产品开发中使用得很广泛，并且有些为ARM打造的开发工具也是基于GNU工具链的。在目前，支持CM3的GNU工具链可以由CodeSourcery处免费下载到(www.codesourcery.com)。而GNU的主打C编译器则在以后支持CM3（在2008年3月31日以后，主流的GNU工具链已经支持Cortex-M3，对应的开发工具为WinARM——译者注）。

本章只介绍使用GNU工具链的基础知识，更详细的信息还需要参阅联机帮助文档。值得一提的是，GNU的汇编语法（GNU工具链中的AS程序）与ARM的汇编语法是有些不同的。这些不同点包括变量定义、编译指示字、以及the like。因此，使用ARM RealView工具的汇编代码在使用GNU工具前，还需要一些（很枯燥的）修改工作。

## 19.2 获取GNU工具链

编译好的GNU工具链可以从[www.codesourcery.com/gnu\_toolchains/arm/](http://www.codesourcery.com/gnu_toolchains/arm/)处下载。有一系列的二进制构建版本。对于最简单的使用，可以使用EABI[注]，并且不带嵌入式OS支持的版本。这个工具链既有在Windows上使用的版本，也有在Linux上使用的版本。本章给出的示例程序可以用于任何一个版本上。

注：（EABI表示嵌入式应用程序二进制接口。可执行目标文件必须符合该规格，从而可以跨开发工具集使用）

### 19.2.1 开发流程

和ARM开发工具的相似，GNU工具链也包含了编译器、汇编器和连接器，从而使得源代码既可以使用C，也可以使用汇编写成，如图19.1所示。

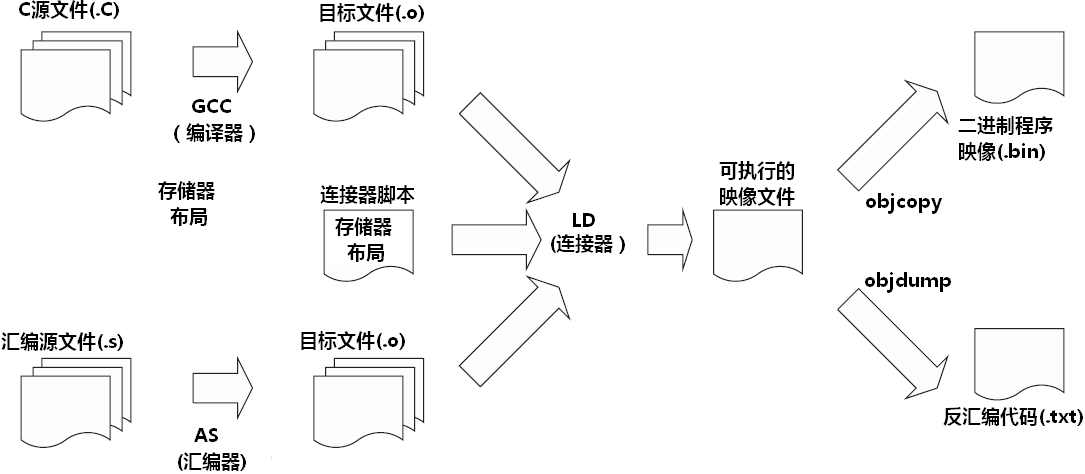


图19.1 基于GNU工具链的开发流程模式图

不同的应用程序环境中也有不同版本的工具链（Symbian, Linux, EABI等）。取决于工具链的目标平台，相应的可执行文件通常有一个前缀。例如，如果使用了EABI环境，则GCC命令为arm-xxxx-eabi-gcc。本章的目标代码使用CodeSourcery的GNU ARM工具链，如表19.1所示。

表19.1 winARM20080331 GNU工具链的命令名称

|  |  |
| --- | --- |
| 功能 | 命令 |
| 汇编器 | arm-none-eabi-as |
| 编译器 | arm-none-eabi-gcc |
| 连接器 | arm-none-eabi-ld |
| 二进制映像产生器 | arm-none-eabi-objcopy |
| 反汇编器 | arm-none-eabi-objdump |

在开发流程图中，连接脚本是可选的。但是当存储器映射比较复杂时，常常是必需的。

## 19.3 示例程序

让我们开开眼，看一看GNU工具链下的源代码的众生相。

### 19.3.1 例1：第一个程序

作为启蒙，让我们把在第10章引入的简单程序使用GCC重写一遍。这个程序计算10+9+8+…+1的值，如下所示：

========== example1.s ==========

/\* 定义常数 \*/

.equ STACK\_TOP, 0x20000800

.text

.global \_start

.code 16

.syntax unified

/\* .thumbfunc \*/

/ \* .thumbfunc仅仅在2006Q3-26之前的CodeSourcery工具中需要\*/

\_start:

.word STACK\_TOP, start

.type start, function

/\* 主程序入口点 \*/

start:

movs r0, #10

movs r1, #0

/\* 计算 10+9+8...+1 \*/

loop:

adds r1, r0

subs r0, #1

bne loop

/\* Result is now in R1 \*/

deadloop:

b deadloop

.end

========== end of file ==========

* **.word**指示字定义MSP起始值为0x2000\_0800，并且把”start”作为复位向量。
* **.text**也是一个预定义的指示字，表示从这以后是一个代码区，需要予以汇编。
* **.global**使\_start标号可以由其它目标文件使用。
* **.code 16**指示程序代码使用thumb写成。
* **.syntax unified**指示使用了统一汇编语言语法。
* **\_start**是一个标号，指示出程序区的入口点
* **start**是另一个标号，它指示复位向量。
* **.type start, function**宣告了**start**是一个函数。对于所有处于向量表中的异常向量，这种宣告都是必要的，否则汇编器会把向量的LSB清零——这在thumb中是不允许的。
* **.end**指示程序文件的结束。

与ARM汇编器不同的是，GNU汇编器中的标号要以“:”结尾；注释可以使用/\*和\*/，并且指示字要以一个”.”作为前缀。

要注意：在thumb代码（.code 16）里面，复位向量（start）被定义成了一个函数（.type start, function）。这是为了使复位向量的LSB被强制为1，从而表示这是以Thumb状态开始执行。否则，处理器就会尝试以ARM态开始，从而引起一个硬fault。

程序写好后，使用as来汇编这个源程序，命令格式为：

$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o

执行了这个命令，就产生了目标文件example1.o。命令行中的-mcpu和-mthumb决定使用的指令集。接下来执行连接，命令如下

$> arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o

然后，使用目标拷贝命令（objcopy）来产生二进制文件：

$> arm-none-eabi-objcopy -Obinary example1.out example1.bin

我们还可以使用目标倾倒(dump)命令（objdump）来创建一个反汇编代码来检查生成的目标文件：

$> arm-none-eabi-objdump -S example1.out > example1.list

生成的反汇编应如下所示：

example1.out: file format elf32-littlearm

Disassembly of section .text:

00000000 <\_start>:

0: 0800 lsrs r0, r0, #32

2: 2000 movs r0, #0

4: 0009 lsls r1, r1, #0

...

00000008 <start>:

8: 200a movs r0, #10

a: 2100 movs r1, #0

0000000c <loop>:

c: 1809 adds r1, r1, r0

e: 3801 subs r0, #1

10: d1fc bne.n c <loop>

00000012 <deadloop>:

12: e7fe b.n 12 <deadloop>

### 19.3.2 例2：连接多个文件

如前所述，我们可以创建多个目标文件，并且把它们连接到一起。在这个例子里，我们有两个汇编程序文件，分别是example2a.s和example2b.s。前者只包含向量表，而后者包含了正常的程序代码。这里，.global指示字就派上用场了——在文件之前传递全局符号。

========== example2a.s ==========

/\* 定义常数\*/

.equ STACK\_TOP, 0x20000800

.global vectors\_table

.global start

.global nmi\_handler

.code 16

.syntax unified

vectors\_table:

.word STACK\_TOP, start, nmi\_handler, 0x00000000

.end

========== end of file ==========

========== example2b.s ==========

/\* 主程序 \*/

.text

.global \_start

.global start

.global nmi\_handler

.code 16

.syntax unified

.type start, function

.type nmi\_handler, function

\_start:

/\* 主程序入口点\*/

start:

movs r0, #10

movs r1, #0

/\* 计算 10+9+8...+1 \*/

loop:

adds r1, r0

subs r0, #1

bne loop

/\* 结果存储在R1中 \*/

deadloop:

b deadloop

/\* 为演示而设置的空NMI服务例程 \*/

nmi\_handler:

bx lr

.end

========== end of file ==========

创建可执行映像的步骤为：

1. 汇编example2a.s

$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example2a.s -o example2a.o

1. 汇编example2b.s

$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example2b.s -o example2b.o

1. 把2个目标文件连接成单一的映像。要注意的是，目标文件在命令行中的顺序是重要的，它会影响在最终的目标文件中，把这两个目标文件的代码编排的顺序。

$> arm-none-eabi-ld -Ttext 0x0 -o example2.out example2a.o example2b.o

1. 产生二进制文件

$> arm-none-eabi-objcopy -Obinary example2.out example2.bin

1. 如上例，可以创建一个反汇编文件来检查所产生目标文件的内容。

$> arm-none-eabi-objdump -S example2.out > example2.list

当目标文件增多时，为简化处理过程，我们可以使用make来管理工程。另外，开发套件也常常有各自内建的功能来简化编译过程。

### 19.3.3 例3：一个简单的”Hello World”程序

前两个例子算是热身，现在该动真格的了。让我们试一个“hello world”程序。但是在这里为了突出主题，我们省去了UART初始化代码。第20章给出了一个C语言写成的UART示例代码。

========== example3a.s ==========

/\* 定义常数 \*/

.equ STACK\_TOP, 0x20000800

.global vectors\_table

.global \_start

.code 16

.syntax unifi ed

vectors\_table:

.word STACK\_TOP, \_start

.end

========== end of file ==========

========== example3b.s ==========

.text

.global \_start

.code 16

.syntax unifi ed

.type \_start, function

\_start:

/\* 主程序入口点 \*/

movs r0, #0

movs r1, #0

movs r2, #0

movs r3, #0

movs r4, #0

movs r5, #0

ldr r0, =hello

bl puts

movs r0, #0x4

bl putc

deadloop:

b deadloop

hello:

.ascii "Hello\n"

.byte 0

.align

puts:

/\* 该子程序向UART发送字符串 \*/

/\* 入口条件： r0 = 字符串的起始地址 \*/

/\* 字符串要以零结尾 \*/

push {r0, r1, lr} /\* 保存寄存器 \*/

mov r1, r0 /\* 把地址拷贝到R1，因为 \*/

/\* R0 还要用于作putc的参数 \*/

putsloop:

ldrb.w r0, [r1], #1 /\* 读取一个字符并且自增地址 \*/

cbz r0, putsloopexit /\* 如果字符为NULL，则跳转到结束 \*/

bl putc

b putsloop

putsloopexit:

pop {r0, r1, pc} /\* 返回 \*/

.equ UART0\_DATA, 0x4000C000

.equ UART0\_FLAG, 0x4000C018

putc:

/\* 该子程序通过UART发送一个字符 \*/

/\* 入口条件： R0 = 要发送的字符 \*/

push {r1, r2, r3, lr} /\* 保存寄存器 \*/

ldr r1, =UART0\_FLAG

putcwaitloop:

ldr r2, [r1] /\* 获取状态位 \*/

tst.w r2, #0x20 /\* 检查发送缓冲区满标志 \*/

bne putcwaitloop /\* 如果已满则循环等待 \*/

ldr r1, =UART0\_DATA /\* 否则继续往发送缓冲区里送数据 \*/

str r0, [r1]

pop {r1, r2, r3, pc} /\* 返回 \*/

.end

========== end of file ==========

在这个例子里，我们使用了.ascii和.byte指示字来创建一个零结尾的字符串。在定义了字符串之后，我们又使用了.align来确保下一条指令会以正确的位置开始。如果不使用.align，汇编器则可能把下一条指令放到未对齐的地址。

创建目标代码的步骤如下所示，读者应理解下述命令的含义和作用。

$> arm-none-eabi-as -mcpu\_cortex-m3 -mthumb example3a.s -o example3a.o

$> arm-none-eabi-as -mcpu\_cortex-m3 -mthumb example3b.s -o example3b.o

$> arm-none-eabi-ld -Ttext 0x0 -o example3.out example3a.o example3b.o

$> arm-none-eabi-objcopy -Obinary example3.out example3.bin

$> arm-none-eabi-objdump -S example3.out > example3.list

### 19.3.4 例4：把数据放到RAM中

RW数据需要放到RAM中，本例就演示在RAM中定义变量的方法。

========== example4.s ==========

.equ STACK\_TOP, 0x20000800

.text

.global \_start

.code 16

.syntax unified

\_start:

.word STACK\_TOP, start

.type start, function

start:

movs r0, #10

movs r1, #0

/\* 计算10+9…+1 \*/

loop:

adds r1, r0

subs r0, #1

bne loop

/\* 结果现在存储到R1中了 \*/

ldr r0, =result

str r1, [r0]

deadloop:

b deadloop

/\* 数据区 \*/

**.data**

result:

**.word 0**

.end

========== end of fi le ==========

本例的核心就是粗体的.data指示字。使用它创建一个数据区。在该区中，使用一个.word指示字来保留一个4字节的空间，并且取名为Result（其实result就相当于C中的变量名）。欲连接本程序，需要告诉连接器RAM在何处，这可以使用-Tdata选项来实现，它把数据段设置到所需的位置上：

$> arm-none-eabi-as -mcpu\_cortex-m3 -mthumb example4.s -o example4.o

$> arm-none-eabi-ld -Ttext 0x0 **-Tdata 0x20000000** -o example4.out

example4.o

$> arm-none-eabi-objcopy -Obinary **–R .data** example4.out example4.bin

$> arm-none-eabi-objdump -S example4.out > example4.list

还要注意的是，在objcopy中对-R .data选项的使用。它避免在二进制输出文件中把数据存储区也包含进去。

### 19.3.5 例5：纯C程序

想必大家已经受够了在汇编下过日子了吧！在GNU工具链中的一个主要组件就是C编译器。在本例中，整个可执行程序——甚至是复位向量和MSP初值都由C写成。此外，还添加了一个连接器脚本，用来把各段放到正确的位置。那么，先让我们看一看C程序文件。

========== example5.c ==========

#define STACK\_TOP 0x20000800

#define NVIC\_CCR ((volatile unsigned long \*)(0xE000ED14))

// 声明函数原型

void myputs(char \*string1);

void myputc(char mychar);

int main(void);

void nmi\_handler(void);

void hardfault\_handler(void);

// 定义向量表

**\_\_attribute\_\_ (( section(“vectors”) ))** void (\* const VectorArray[])(void) =

{

STACK\_TOP,

main,

nmi\_handler,

hardfault\_handler

};

// 主程序入口点

int main(void)

{

const char \*helloworld[]="Hello world\n";

\*NVIC\_CCR = \*NVIC\_CCR | 0x200; /\* 设置NVIC的STKALIGN \*/

myputs(\*helloworld);

while(1);

return(0);

}

// 函数

void myputs(char \*string1)

{

char mychar;

int j;

j=0;

do

{

mychar = string1[j];

if (mychar!=0)

{

myputc(mychar);

j++;

}

} while (mychar != 0);

return;

}

void myputc(char mychar)

{

#define UART0\_DATA ((volatile unsigned long \*)(0x4000C000))

#define UART0\_FLAG ((volatile unsigned long \*)(0x4000C018))

// Wait until busy fl ag is clear

while ((\*UART0\_FLAG & 0x20) != 0);

// Output character to UART

\*UART0\_DATA = mychar;

return;

}

//空的服务例程

void nmi\_handler(void)

{

return;

}

void hardfault\_handler(void)

{

return;

}

========== end of file ==========

注意粗体字显示的部分，它使用 \_\_attribute(( ))（注意，是双小括号）来指定特殊的属性。在这里则指出那个函数指针数组是放到vectors段中的。然而，这个C程序并没有指定vectors段在何处。那么在哪里指定vectors段的位置呢？现在该请出我们的连接器脚本文件了，工作就在这里完成。本例的连接器脚本文件为simple.ld，内容如下：

========== simple.ld ==========

/\* MEMORY命令：定义允许的存储器区域 \*/

/\* 本部分定义了连接器允许放入数据的各存储器区域，这是 \*/

/\* 一个可选的功能，但是对于开发很有益，它使连接器在在 \*/

/\* 程序太大时能给你警告 \*/

MEMORY

{

/\* ROM是可读的(r)和可执行的(x) \*/

rom (rx) : ORIGIN = 0, LENGTH = 2M

/\* RAM是可读的(r)，可写的(w)，可执行的(x) \*/

ram (rwx) : ORIGIN = 0x20000000, LENGTH = 4M

}

/\* SECTIONS 命令 : 定义各输入段到输出段的映射 \*/

SECTIONS

{

. = 0x0; /\* 从0x00000000开始 \*/

.text : {

\*(vectors) /\* 向量表 \*/

\*(.text) /\* 程序代码 \*/

\*(.rodata) /\* 只读数据 \*/

}

. = 0x20000000; /\* 从0x20000000开始 \*/

.data : {

\*(.data) /\* 数据存储器 \*/

}

.bss : {

\*(.bss) /\* 预留的数据存储器，必须初始化为零 \*/

}

}

========== end of file ==========

为使用连接脚本，需要在编译阶段把simple.ld传给编译器。

$> arm-none-eabi-gcc -mcpu\_cortex-m3 -mthumb example5.c -nostartfiles

**-T simple.ld** -o example5.o

然后在连接时，需要再次使用simple.ld。

$> arm-none-eabi-ld **-T simple.ld** -o example5.out example5.o

本例中我们只有一个源文件，因此连接过程其实是可以省略的。最后再创建二进制目标文件和反汇编文件。

$> arm-none-eabi-objcopy -Obinary example5.out example5.bin

$> arm-none-eabi-objdump -S example5.out > example5.list

读者可能还注意到了，在本例中我们使用了另一个称为-nostartfiles的编译器开关。使用它，就可以让编译器不再往可执行映像中插入启动代码(crt)，这样做的目的之一就是减少程序映像的尺寸。不过，使用该选项的主要原因，其实是在于GNU工具链的启动代码是与发布包的提供者相关的，而有些人提供的启动代码不适合CM3——它们往往是用于传统的ARM处理器的——如ARM7（典型地这些启动代码使用了ARM代码，而没有使用Thumb代码）。

但是，在许多情况下，取决于应用程序和使用的库，都必须使用启动代码来执行初始化的过程，最主要的就是对数据的初始化（例如，把bss区的存储单元全部清零）。在最后一个例子中，我们将演示这个过程。

### 19.3.6 例6：纯C程序，带有标准C启动代码

在正常情况下，当编译C程序时，会自动地把标准C库的启动代码包含在目标文件中，它保证运行时库得以正确地初始化。标准C运行时库的启动代码由GNU工具链提供，但是不同提供者提供的工具链可能有不同的启动代码。下例是基于CodeSourcery GNU ARM工具链2006q3-26版本的。因此，最好检查一下从工具链中的启动代码，或者从供应者处获取最新的启动代码。对于这个版本的CodeSourcery提供的工具链，其启动代码目标文件为armv7m-crt0.o。但是这个版本提供的启动代码是错误的——使用了ARM代码来编写。到了2006q3-27及更晚的版本中才修正了这个bug。不同提供者的GNU工具链会有不同的启动代码，而且文件名也常常不同。此时，就需要检查你所使用的GNU工具链之帮助文档来获取准确信息了。

在编译C源代码之前，例5中的C程序需要一些小改动。缺省情况下，armv7m-crt0已经包含了一张向量表，并且在它里面，NMI服务例程和硬fault服务例程分别取名为\_nmi\_isr和\_fault\_isr。因此，需要移除例5中的向量表，并且重命名NMI和硬Fault的服务例程，如下所示：

// 声明函数原型

void myputs(char \*string1);

void myputc(char mychar);

int main(void);

void \_nmi\_isr(void);

void \_fault\_isr(void);

// 主程序入口点

int main(void)

{

const char \*helloworld[]="Hello world\n";

myputs(\*helloworld);

while(1);

return(0);

}

// 函数

void myputs(char \*string1)

{

char mychar;

int j;

j=0;

do

{

mychar = string1[j];

if (mychar!=0)

{

myputc(mychar);

j++;

}

} while (mychar != 0);

return;

}

void myputc(char mychar)

{

#define UART0\_DATA ((volatile unsigned long \*)(0x4000C000))

#define UART0\_FLAG ((volatile unsigned long \*)(0x4000C018))

// Wait until busy fl ag is clear

while ((\*UART0\_FLAG & 0x20) != 0);

// Output character to UART

\*UART0\_DATA = mychar;

return;

}

//空的服务例程

void \_nmi\_isr(void)

{

return;

}

void \_fault\_isr(void)

{

return;

}

在安装了CodeSourcery后，已经包含了一系列的连接脚本，可以从codesourcery/sourcery g++/arm-none-eabi/lib目录下找到。在下例中，我们就使用了lm3s8xx-rom.ld文件。这个连接器脚本顾名思义，是用于LM3S8XX系列芯片的。

在当前目录之外，当C程序代码定位后，一个名为“lib”的库子目录也在在当前目录下创建，（Aside from the current directory, when the C program code is located, a library subdirectory called *lib* is also created in the current directory）这使得库搜索路径的设置更加简单——所需的目标文件armvrm-crt0.o以及连接器脚本都被拷贝到这个“lib”目录下。在下一个例子中，我们就使用-L lib选项来把“lib”添加到库的搜索路径中。

现在我们可以编译这个C程序了：

$> arm-none-eabi-gcc –mcpu=cortex-m3 -mthumb example6.c **-L lib** –T

lm3s8xx-rom.ld -o example6.out

执行了上条命令后，就创建并且连接了目标文件example6.out。因为只有一个目标文件，二进制文件可以直接由它来生成：

$> arm-none-eabi-objcopy -Obinary example6.out example6.bin

产生反汇编的方式则与上例相同：

$> arm-none-eabi-objdump -S example6.out > example6.list

## 19.4 访问特殊功能寄存器

在CodeSourcery的GNU ARM工具链中，可以直接使用小写的名字来访问特殊功能寄存器（注意，必须是小写的），如下所示：

msr control, r1

mrs r1, control

msr apsr, r1

mrs r0, psr

## 19.5 使用未支持的指令

如果使用了另外的GNU工具链，有可能那个GNU汇编器不支持一些指令。在这种情况下，则可以直接使用.word来插入不支持指令的二进制机器码，如下所示：

.equ DW\_MSR\_CONTROL\_R0, 0x8814F380

...

MOV R0, #0x1

.word DW\_MSR\_CONTROL\_R0 /\* 相当于执行 MSR CONTROL, R0 指令 \*/

...

## 19.6 GNU C编译器的内联汇编

GNU的ARM C编译器是支持内联汇编的，但此时的汇编语法看起来有点怪：

**\_\_asm (" inst1 op1, op2... \n"**

**" inst2 op1, op2... \n"**

**...**

**" inst op1, op2... \n"**

**: 输出操作数s /\* 可选 \*/**

**: 输入操作数s /\* 可选 \*/**

先举一个简单的例子，进入睡眠模式的代码如下所示：

void Sleep(void)

{

// 使用Wait-For-Interrupt进入睡眠模式

\_\_asm (

“WFI\n”

);

}

如果汇编代码需要一个输入变量和一个输出变量，例如，把一个变量除以5，则格式如下：

\_\_asm ( "mov r0, %0\n"

"mov r3, #5\n"

"udiv r0, r0, r3\n"

"mov %1, r0\n"

: "=r" (DataOut) : "r" (DataIn) : "cc", "r3" );

在这个代码中，输输入参数是一个C变量，名为DataIn（%0代表第一个参数），该代码把结果返回到另外一个C变量DataOut中（%1表示第2个参数）。内联汇编的代码还手工修改了寄存器r3，并且修改了条件标志cc，因此它们被列在被破坏的（clobbered）寄存器列表中。

更详细的内联汇编信息在GCC-Inline-Assembly-HOWTO文档中。