第3章

C语言和GCC编译器

视频讲解

C语言是Paparazzi自动驾驶仪的主要编程语言，Paparazzi所用的C语言是基于GNU C标准的嵌入式C语言，默认的编译器是GCC编译器。

不同的C语言标准之间有些差异，在3.1节中，对常用的C语言标准以及各标准之间的差异做了简要说明，其中重点介绍C99标准相对于C89/C90标准的一些重要改进。

在C语言标准的基础上，大多数C语言编译器也对C语言做了不同程度的扩展，在3.2节中，结合Paparazzi中的部分C语言代码介绍GCC嵌入式C语言常用的扩展。

3.1 C语言标准

20世纪70—80年代，C语言被广泛应用，从大型主机到小型微机，也衍生了C语言的很多不同版本，其中以1978年出版的*The C Programming Language*一书中的C语言标准最重要，该版本称为K&R C语言标准。

1983年，美国国家标准协会(ANSI)成立了一个委员会X3J11，来制定C语言标准，1989年，该协会通过了被称为ANSI X3.159-1989的第一个官方C语言标准。因为这个标准是1989年通过的，所以一般简称为C89标准。因为这个标准是美国国家标准协会(ANSI)发布的，也简称为ANSI C。

1990年，国际标准化组织(ISO)和国际电工委员会(IEC)把C89标准定为C语言的国际标准，命名为ISO/IEC 9899:1990-Programming languages-C。因为此标准是在1990年发布的，所以也简称为C90标准，此标准与ANSI C89标准完全等同。

1994年，国际标准化组织和国际电工委员会发布了C89标准修订版，名叫ISO/IEC9899:1990/Cor 1:1994，简称为C94标准。1995年，国际标准化组织和国际电工委员会再次发布了C89标准修订版，名叫ISO/IEC 9899:1990/Amd 1:1995-C Integrity，有些人简称为C95标准。C89、C90、C94和C95这几个C语言标准基本是相同的，没有大的变化。

1999年1月，国际标准化组织和国际电工委员会发布了C语言的新标准，名叫ISO/

IEC 9899:1999-Programming languages-C,简称 C99 标准。

2011 年 12 月 8 日,国际标准化组织和国际电工委员会的 C 语言标准委员会(ISO/IEC JTC1/SC22/WG14)正式发布了 C11 标准,该标准是 C 语言当前的最新标准。

3.1.1　K&R C 语言标准

K&R C 语言标准并非正式的官方标准。1978 年由美国电话电报公司(AT&T)贝尔实验室正式发表了 C 语言,布莱恩·柯林汉(Brian Kernighan)和丹尼斯·里奇(Dennis Ritchie)出版了 *The C Programming Language* 一书,这本书被简称为 K&R,这本书中的 C 语言标准也被称为 K&R C 语言标准。目前,虽然很多 C 语言编译器仍旧支持 K&R C 语言标准,但是该标准在应用中几乎不再使用。

3.1.2　ANSI C 语言标准

由 ANSI 发布的 ANSI C 语言标准(即 ANSI X3.159-1989)是获得广泛应用的一个 C 语言标准,是 C 语言的第一个正式官方标准。在 1990 年,ANSI C 标准(带有一些小改动)被美国国家标准协会采纳为 ISO/IEC 9899:1990。因此,这一版本的 C 语言简称为 ANSI C、C89 或 C90。

GCC 中指定此版本所用参数时,有 3 种编译选项可以实现:

```
-ansi
-std=c90
-std=iso9899:1990
```

3.1.3　AMD1 C 语言标准

1995 年,针对之前 1990 年发布的 C90 标准,ISO 发布了一个修订版;添加了一个有向图(Digraphs)和宏,即 __STDC_VERSION__。此标准常被称为 AMD1,有时被叫作 C94 或 C95。GCC 中制定此版本时所用编译选项如下:

```
-std=iso9899:199409
```

3.1.4　C99 标准

在 ANSI 标准化发布了 C89 标准以后,C 语言标准在一段相当长的时间内都保持不变,C 语言标准在 20 世纪 90 年代才经历了改进,这就是 ISO/IEC 9899:1999(1999 年出版),这个版本的 C 语言标准通常简称为 C99 标准,在 C99 标准未完成之前的草案叫作 C9X 标准。

C99 是在 C89、C90 的基础上发展起来的,增加了基本数据类型、关键字和一些系统函数等。C99 增加了宽字符集,还加入了一些库函数,是继 C89、C90 标准之后的又一个重要的 C 语言官方标准。

GCC 中制定此版本时所用编译选项如下：

```
-std=c99
-std=iso9899:1999
```

Paparazzi 的 C 语言代码所用的标准就是和 C99 标准类似的 GNU99 标准，而 GNU99 标准则是 GCC 编译器对 C99 标准的扩展。

C99 标准对 C89/C90 标准的一些重要改进主要包括以下几个方面。

1. 增加 restrict 指针

C99 中增加了适用于指针的 restrict 类型修饰符，它是初始访问指针所指对象的唯一途径，因此只有借助 restrict 指针表达式才能访问对象。restrict 指针主要用作函数变元，或者指向由 malloc()函数所分配的内存变量。restrict 数据类型不改变程序的语义。

restrict 指针的主要作用是为编译器进行优化时提供重要的信息，帮助编译器进行更好的代码优化，生成更有效率的汇编代码。如果某个函数定义了两个 restrict 指针变元，编译器就假定它们指向两个不同的对象，memcpy()函数就是 restrict 指针的一个典型应用示例。

C89 中 memcpy()函数原型如下：

```
void * memcpy (void * s1, const void * s2, size_t size);
```

如果 s1 和 s2 所指向的对象重叠，其操作就是未定义的，memcpy()函数只能用于不重叠的对象。

C99 中 memcpy()函数原型如下：

```
void * memcpy(void * restrict s1, const void * restrict s2,size_t size);
```

通过使用 restrict 修饰 s1 和 s2 变元，可确保它们在该原型中指向不同的对象。

2. inline(内联)关键字

C99 增加了 inline(内联)关键字，被该关键字修饰的函数在编译时可以嵌入调用处，功能类似于宏的扩展。和普通函数相比，节省了函数调用和返回的环节，但增加了代码的体积，是一种用“空间换时间”手段。Paparazzi 的自动驾驶仪的 C 语言代码中大量使用了 inline(内联)关键字。

C99 的 inline 关键字和 GCC 的 inline 关键字在使用的细节上略有差异，但常用格式的含义是相同的，参考 3.2.7 小节。

3. 新增数据类型

C99 增加了一些新的数据类型，增强了对布尔类型、复数类型和长整数类型变量的支持。

(1) _Bool 类型。_Bool 类型值是 false 或 true。C99 中增加了用来定义 bool、true 及 false 宏的头文件 stdbool.h，以便程序员能够编写同时兼容于 C 与 C++的应用程序。在编写新的应用程序时，应该使用 stdbool.h 头文件中的 bool 宏。

(2) _Complex 和_Imaginary 类型。C99 标准中定义了复数类型，即 float_Complex、float_Imaginary、double_Complex、double_Imaginary、long double_Complex、long double_

Imaginary。在 complex.h 头文件中定义了 complex 和 imaginary 宏，并将它们扩展为 _Complex 和_Imaginary，在编写相应的应用程序时，应该使用 complex.h 头文件中的 complex 和 imaginary 宏。

(3) long long int 类型。C99 标准中引进了 long long int(范围为 $-2^{63}-1\sim2^{63}-1$)和 unsigned long long int(范围为 $0\sim2^{64}-1$)，能够支持 64 位的整型数据。

4. 可变长数组(Variable Length Array)

C99 中声明数组时，数组的长度可以由任一有效的整型表达式确定，包括只在运行时才能确定其值的表达式，这类数组就叫作可变长数组。在 C89 标准中数组的长度只能是确定值，一旦确定数组的长度将不能再变化。

在 C99 标准中只有局部数组才可以是变长的，并且可变长数组的长度在数组生存期内是不变的。也就是说，可变长数组不是动态的，不可以随时通过改变变量的值来动态修改数组的长度，仅仅是指可变长数组的长度可以到运行时才决定而已，代码如下：

```
void fun(int n)
{
    int vla[n];
    printf("vla len is %d\n",sizeof(vla));
}
```

另外，还可以使用 * 来定义不确定长的可变长数组。例如，定义可变长的整型数组如下：

```
void f(double a[ * ][ * ], int n)
{
  // ...
}
```

可变长数组不能用在全局变量、结构或联合里面，也就是说，具有 static 或 extern 修饰符的数组不能被定义为可变长数组。并且如果使用了可变长数组，goto 语句就受限制了。

5. 数组声明中的类型修饰符

在 C99 标准中，如果需要使用数组作为函数参量，可以在数组声明的方括号内使用 static 关键字，告诉编译器数组参量将至少包含指定的元素个数。例如，

```
void fadd(double a[static 10], const double b[static 10])
{
  int i;
  for (i = 0; i < 10; i++) {
    if (a[i] < 0.0)
    return;
    a[i] += b[i];
  }
  return;
}
```

编译器可以获知参量 a 和 b 至少各具有 10 个元素，在某些系统中能够根据这些信息实现程序的优化。static 关键字也能确保参量 a 和 b 并非指向 NULL，即 a 和 b 不是空指针。

若还需要指明 a 和 b 指向的内容不重叠，则需要再增加 restrict 修饰符，代码如下：

```
void fadd(double a[static restrict 10],
  const double b[static restrict 10])
{
  int i;
  for (i = 0; i < 10; i++) {
    if (a[i] < 0.0)
    return;
    a[i] += b[i];
  }
  return;
}
```

如果使用 const 关键字，则表示函数参量始终指向同一个数组。例如，下面的两个函数声明是等同的：

```
void f1(double x[const], const double y[const]);
void f2(double * const x, const double * const y);
```

另外，还可以使用 volatile 修饰函数参量，但是没有任何意义。

6. 柔性数组结构成员（Flexible Array Members）

在 C99 中，结构中的最后一个元素允许是未知大小的数组，称为柔性数组成员，但结构中的柔性数组成员前面必须至少有一个其他成员。例如，下面结构的 word 成员就是一个大小可变的数组：

```
typedef struct {
  int count;
  char word[];              // 只能放在末尾，等价于 GCC 的 char word[0]
} word_counter_t;
```

包含柔性数组成员的结构可以用 malloc()函数进行内存的动态分配，分配的内存应该大于结构的大小，以适应柔性数组的预期大小。sizeof 的返回值不包括结构柔性数组的内存，代码如下：

```
malloc(sizeof(word_counter_t) + strlen(str) + 1);
```

上面的代码得到了一个末尾包含 char word[strlen(str)+1]数组的结构。如果不采用柔性数组成员的方式，则需要为每个单词预留足够的空间，既会造成内存空间的浪费，也可能会造成单词缓冲区不够长的问题。

7. 单行注释

C99 新增了与 C++兼容的单行注释标记“//”。

8. 支持可变参数的宏（Variadic Macro）

C99 中宏可以带可变参数，在宏定义中用省略号(...)表示。内部预处理标识符 VA_ARGS 决定可变参数将在何处得到替换。可变参数的宏代码如下：

```
#define     dprintf(format, ...) \
dfprintf(stderr, format,__VA_ARGS__)
```

宏参数里面“...”的部分会展开到__VA_ARGS__处。如果在__VA_ARGS__前面加上##,就可以写出允许可变参数部分为空的变参宏：

```
#define debug(format, ...) printf(format, ##__VA_ARGS__)
```

##__VA_ARGS__表示变参“...”部分并且允许为空。当变参部分为空时,__VA_ARGS__会展开成空字符串,并且##前面的逗号也会在展开时去掉。

9. _pragma 运算符

在C90中引入了#pragma预处理指令,#pragma的作用是为特定的编译器提供特定的编译指示。在C语言标准中并没有规定具体的指示内容,不同的编译器对#pragma指示的含义理解也是不同的,也就是说,#pragma的实现是与具体平台相关的。当编译器不理解某个#pragma指示含义时,会自动将其忽略。

C99标准引入了在程序中定义编译指令的另一种方法,即_pragma运算符,相当于#pragma的高级版本。_pragma运算符如下：

```
_pragma ("directive")
```

上面的代码中directive表示编译指令,_pragma运算符允许编译指令参与宏替换。

例如,Paparazzi的sw/include/message_pragmas.h文件就使用了_pragma运算符,在编译过程中显示编译信息。message_pragmas.h文件代码如下：

```
#ifndef MESSAGE_PRAGMAS_H
#define MESSAGE_PRAGMAS_H

/* some helper macros */
#define DO_PRAGMA(x) _Pragma (#x)
#define VALUE_TO_STRING(x) #x
#define VALUE(x) VALUE_TO_STRING(x)

/* some convenience macros to print debug/config messages at compile time */
#define WARNING(x) DO_PRAGMA(GCC warning #x)
#define MESSAGE(x) DO_PRAGMA(message (x))
#define TODO(x) DO_PRAGMA(message ("TODO - " x))
#define INFO(x) DO_PRAGMA(message ("Info: " x))
#define INFO_VALUE(x,v) DO_PRAGMA(message ("Info: " x VALUE(v)))
#define INFO_VAR(var) DO_PRAGMA(message ("INFO: " #var " = " VALUE(var)))

/* only if PRINT_CONFIG is true */
#if PRINT_CONFIG
#define PRINT_CONFIG_MSG(x) DO_PRAGMA(message ("Config: " x))
#define PRINT_CONFIG_MSG_VALUE(x,v) DO_PRAGMA(message ("Config: " x VALUE(v)))
#define PRINT_CONFIG_VAR(var) DO_PRAGMA(message ("Config: " #var " = " VALUE(var)))
#else
#define PRINT_CONFIG_MSG(x)
#define PRINT_CONFIG_MSG_VALUE(x,v)
#define PRINT_CONFIG_VAR(var)
```

```
#endif

#endif
```

其中，MESSAGE("paparazzi config")宏会展开为：

```
_Pragma ("message (\"paparazzi config\")")
```

而_Pragma ("message ("paparazzi config")")相当于：

```
#pragma message ("paparazzi config")
```

在 GCC 编译器编译过程中，就会打印 paparazzi config 的字符串信息。

10. 混合声明（Mix Declarations and Code）

C99 以前的 C 标准要求在一个代码块（Block）的开始处声明变量。C99 的混合声明分散了代码和声明在程序中的位置，解除了原先必须在代码块的第一个语句之前声明变量的限制，可以在代码中随时声明变量，遵循混合声明的规则可以提高代码的可读性。

11. for 循环变量初始化（for Loop Intializers）

C99 中 for 循环的变量声明不必放在语句块的开头，可以在 for 语句的初始化部分定义一个或多个变量，这些变量的作用域仅在本 for 语句所控制的循环体内。因此，C99 中 for 语句可以写为：

```
for (int i = 0; i < n; i++) {
    ...;
}
```

除了写起来方便外，循环变量 int i 的生存周期也被最小化了（仅在 for 循环内有效）。

12. 复合字面值（Compound Literals）

C99 中引入了复合字面值，可以表示集合类型的匿名常量（未命名常量）。复合字面值为左值，所以它的元素是可以修改的。其语法形式如下：

```
compound-literal(符合字面值):
    (type-name){initializer-list}
```

复合字面值在结构、联合、数组和枚举中比较有用。下面的代码示例中 s1 利用复合字面值指向了一个可修改的字符数组，而 s2 则指向的是只读字符串：

```
char *s1 = (char[]){"Hello world!\n"};
char *s2 = "Hello world!\n";
```

13. 指定初始化（Designated Initializers）

C99 中允许使用指定初始化语句，可以命名要初始化的特定集合（数组、结构或联合）成分。指定初值与位置初值可以在同一初值列表中混合。

在数组初值列表中，指定符的形式为[index]，其中常量表达式 index 用于索引指定数组元素。如果数组的长度没有指定，则允许任何非负的索引，并且最大的索引值确定了数组

的长度。如果初值后面是位置索引，则从这个位置索引开始指定后面的元素，这种情况可能会使初始化列表后面的值覆盖前面的值。C99 中数组初始化的代码示例如下：

```
/* 数组的初值为: {0,1,3} */
int a1[3] = {[2] = 3,[1] = 1}

/* 数组的初值为: {0,1,12,13,0} */
int a2[5] = {0,1,2,3,[2] = 12,13}

/* 数组的初值为: {0,1,12,13} */
int a3[] = {0,1,2,3,[2] = 12,13}

/* 数组的初值为: {1,1,1,3} */
int a4[] = {[0 ... 2] = 1,3}
```

对于结构和联合而言，指定符的形式为.member＝name，初始化可以不必按顺序进行，也可以实现部分初始化，清晰地表明了成员名和初值之间的对应关系。C99 中结构初始化的代码示例如下：

```
struct member
  { char *name;
   char *course;
   int number;
   int score;
};
struct member a = {
  .name = "张三",
  .score = 90,
  .course = "数学"
}
/* a 的初值为: {"张三","数学",0,90} */
```

指定初始化提高了代码的可读性和可维护性，建议采用指定初始化的方式对结构或联合进行初始化。

14. printf()和 scanf()函数系列的增强

C99 标准中对 printf()和 scanf()函数引进了处理 long long int 和 unsigned long long int 数据类型的特性。long long int 类型的格式修饰符是 ll。在 printf()和 scanf()函数中，ll 适用于 d、i、o、u 和 x 格式说明符。

另外，C99 还引进了 hh 修饰符。当使用 d、i、o、u 和 x 格式说明符时，hh 用于指定 char 型变元。

15. C99 新增的库

在 C99 的标准中增加了一些函数库用于支持 C99 的新特性。在 C89 中常用的函数库见表 3-1。

在 C99 中新增的函数库见表 3-2。

表 3-1 C89 中标准库的头文件

头文件	功 能	头文件	功 能
assert. h	定义宏 assert()	signal. h	定义信号值
ctype. h	字符处理	stdarg. h	支持可变长度的变元列表
errno. h	错误报告	stddef. h	定义常用常数
float. h	定义与实现相关的浮点值	stdio. h	支持文件输入和输出
limits. h	定义与实现相关的各种极限值	stdlib. h	其他各种声明
locale. h	支持函数 setlocale()	string. h	支持串函数
math. h	数学函数库使用的各种定义	time. h	支持系统时间函数
setjmp. h	支持非局部跳转		

表 3-2 C99 新增的头文件和库

头文件	功 能
complex. h	支持复数算法
fenv. h	给出对浮点状态标记和浮点环境的其他方面的访问
inttypes. h	定义标准的、可移植的整型类型集合。也支持处理最大宽度整数的函数
iso646. h	首先在 1995 年第一次修订时引进，用于定义对应各种运算符的宏
stdbool. h	支持布尔数据类型。定义宏 bool，以便兼容于 C++
stdint. h	定义标准的、可移植的整型类型集合。该文件包含在 inttypes. h 中
tgmath. h	定义一般类型的浮点宏
wchar. h	首先在 1995 年第一次修订时引进，用于支持多字节和宽字节函数
wctype. h	首先在 1995 年第一次修订时引进，用于支持多字节和宽字节分类函数

16. __func__预定义标识符

除了已有的__line__和__file__以外，还增加了__func__得到当前的函数名。

17. 放宽的转换限制

和 C89/90 相比，C99 放宽了源代码的一些转换限制，见表 3-3。

表 3-3 C89 标准和 C99 标准的限制比较

限 制 项	C89 标准	C99 标准
每行程序字符数	254	4095
数据块的嵌套层数	15	127
条件语句的嵌套层数	8	63
内部标识符中的有效字符个数	31	63
外部标识符(extern)中的有效字符个数	6	31
结构或联合中的成员个数	127	1023
函数调用中的参数个数	31	127

18. 扩展的整数类型

C99 中新增加了部分整型类型，见表 3-4。

表 3-4　C99 中对整数类型的扩展

扩展类型	描　述	扩展类型	描　述
int16_t	整数长度为精确 16 位	intmax_t	最大整数类型
int_least16_t	整数长度为至少 16 位	uintmax_t	最大无符号整数类型
int_fast32_t	最稳固的整数类型，其长度为至少 32 位		

19. 对整数类型提升规则的改进

C89 中，表达式中类型为 char、short int 或 int 的值可以提升为 int 或 unsigned int 类型。

C99 中，每种整数类型都有一个级别。例如，long long int 的级别高于 int，int 的级别高于 char 等。在表达式中，其级别低于 int 或 unsigned int 的任何整数类型均可被替换成 int 或 unsigned int 类型。

20. 其他改动

C99 标准对 C89/C90 标准的改动还包括一些其他部分。例如，不再支持隐含式的 int 规则，即取消了函数返回类型默认为 int 的规定。C99 中，非空类型函数必须使用带返回值的 return 语句。

格式化字符串中，利用 u 支持 unicode 的字符。支持十六进制的浮点数的描述。浮点数的内部数据描述支持了新标准，可以使用 # pragma 编译器指令指定。

允许编译器化简非常数的表达式。修改了％运算符处理负数时的定义，这样可以给出明确的结果。例如，在 C89 中表达式－22/7 和－22％7 的计算结果可以是－3 和－1 ，也可以是－4 和 6 ，而 C99 中明确为只能是－3 和－1 这一种结果。

3.1.5　C11 标准

C11 标准是 ISO/IEC 9899：2011-Information technology-Programming languages-C 标准的简称。2011 年 12 月 8 日，国际标准化组织和国际电工委员会的 C 语言标准委员会（ISO/IEC JTC1/SC22/WG14）正式发布了 C11 标准，ANSI 采纳了 ISO/IEC 9899：2011 标准，这个标准是 C 程序语言的现行标准。GCC 中指定此版本所用参数时有以下两种写法：

```
-std=c11
-std=iso9899:2011
```

3.2　GCC 的 C 语言扩展

GCC 是一个功能强大的跨平台开源编译器。GCC 编译器事实上是 GNU 编译器的集合，包括了 C、C++、Objective-C、FORTRAN、ADA 和 Go 多种编程语言的编译器前端以及这些语言的库。对于 C 语言而言，不仅支持绝大多数 C 语言的 ISO 标准，而且还对标准 C 语言进行了一系列扩展，以增强标准 C 的功能，这些扩展对编译优化、目标代码布局、更安全的检查等方面提供了很强的支持，很多 GCC 的 C 语言扩展成为了后续的 C 语言 ISO 标

准。例如，C99 标准就采用了很多 GCC 的 C 语言扩展，以至于一些 GCC 的 C 语言扩展与 C99 标准的新增功能是重叠的。

GCC 的 C 语言方言主要有两个版本：一个是以 C89 为基础的 GCC C 语言方言；另一个是以 C99 为基础的 GCC C 语言方言。若使用以 C89 为基础的 GCC C 语言方言，可以在编译选项中增加下列代码中的任意一个：

```
-std=gnu89
-std=gnu90
```

若使用以 C99 为基础的 GCC C 语言方言，可以在编译选项中增加-std＝gnu99 选项。如果省略 －std 选项，不同的 GCC 版本采用的 C 语言方言是不同。例如，GCC 4.8.5 默认使用 gnu90 方言，而 GCC 5.4 默认使用 gnu11 方言。

下面简单介绍部分 GCC 的 C 语言扩展。

3.2.1 语句表达式

在标准 C 中表达式指的是运算符和操作数的组合，而复合语句指的是由一个或多个被括在花括号里的语句构成的代码块。GCC C 允许将语句和声明放在表达式中，在 GCC 中将 C 语言圆括号中出现的语句视为表达式，即在表达式中可以使用循环、switch 和局部变量等。

例如，对下面的语句表达式求值的结果为花括号中语句的最后求值结果，即变量 z 的值：

```
({int y = foo (); int z; if
    (y > 0) z = y; else z
    =- y;
    z; })
```

3.2.2 标号变量

可以使用一元运算符 && 获得函数内部定义的标号地址，返回值为 void * 类型，该地址可用于 goto 语句的跳转。需要注意的是，该方法不可用于获得其他函数内定义的标号地址。

3.2.3 case 范围

GNU C 允许在一个 case 标号中指定一个连续范围的值，示例代码如下：

```
case '0' ... '9': c -= '0'; break;
case 'a' ... 'f': c -= 'a' - 10; break;
case 'A' ... 'F': c -= 'A' - 10; break;
```

3.2.4　typeof 关键字

typeof 关键字可以用于获得表达式的数据类型。例如下面的代码中定义的变量 y0、y1、y2 的数据类型是相同的：

```
char *x[4];
char *y0[4];
typeof (typeof (char *)[4]) y1;
typeof (x) y2;
```

typeof 关键字一般应用在宏定义中，可以得到更通用的宏定义。下面的代码就利用了 typeof 关键字定义了一个两个变量交换的宏：

```
/*
 * swap - swap value of @a and @b
 */
#define swap(a, b) \
    do { typeof (a) tmp = (a); (a) = (b); (b) = tmp; } while (0)
```

3.2.5　条件表达式的省略

在 GNU C 的方言中条件表达式中间的操作数有时可以省略。如果当条件表达式的第一个操作数为非零值时，条件表达式的值为此非零值，那么条件表达式中间的操作数可以省略。下面的代码中的两个条件表达式是类似的：

```
x? : y;
x? x : y;
```

这两个条件表达式的不同点在于：前者在 x 为非零值时直接得到了表达式的值，而后者则会在 x 为非零值时对 x 再求值一次作为表达式的值，如果在对 x 求值过程中有副作用产生，后一个条件表达式的副作用也会作用两次，那么两者的效果可能是不同的，甚至条件表达式的值也可能是不同的。

3.2.6　内建函数

GNU C 提供了大量的内建函数，其中很多是标准 C 库函数的内建版本（如 memcpy() 函数等），这类函数与对应的 C 库函数功能相同，而标准 C 函数库之外的内建函数通常以 builtin 为函数名前缀。在此仅介绍部分 GCC 内建位运算函数。

(1) int builtin_ffs (unsigned int x) 函数。

返回二进制表示的 x 最后一位 1 的倒数位数，如 0x3118（二进制为 0011 0001 0001 1000）返回值为 4。

(2) int builtin_clz (unsigned int x) 函数。

返回二进制表示的 x 前导 0 的个数，如 0x3118（二进制为 0011 0001 0001 1000）返回值为 2。

（3）int __builtin_ctz（unsigned int x）函数。

返回二进制表示的 x 后面 0 的个数，如 0x3118（二进制为 0011 0001 0001 1000）返回值为 3。

（4）int __builtin_popcount（unsigned int x）函数。

返回二进制表示的 x 中 1 的个数，如 0x3118（二进制为 0011 0001 0001 1000）返回值为 5。

（5）int __builtin_parity（unsigned int x）函数。

返回 x 的奇偶校验位，也就是 x 的 1 的个数模 2 的结果，如 0x3118（二进制为 0011 0001 0001 1000）返回值为 1。

此外，以上这些函数都有相应的 usigned long 和 usigned long long 版本，只需要在函数名后加上 l 或 ll 后缀即可，如 int __builtin_clzll（ usigned long long）函数。

（6）uint32_t __builtin_bswap32（uint32_t x）函数。

按字节翻转 x，返回翻转后的结果，如 0x12345678 的返回值为 0x78563412 。该函数还有 16 位和 64 位的版本，即 uint16_t __builtin_bswap16（ uint16_t x）和 uint64_t __builtin_bswap64（uint64_t x）。

（7）long __builtin_expect（long exp，long c）函数

该函数可以为编译器提供分支预测信息，期望的执行是 exp == c 成立的情况。编译器可以根据这个信息适当地重排语句块的顺序，使程序在预期的情况下有更高的执行效率。示例代码如下：

```
if (__builtin_expect (x, 0))
          foo ();
```

函数 foo()被预测为低概率执行函数，在编译器进行编译优化时可能会将该函数放置得远些。如果条件为假，正常执行的速度可能会快些；而如果条件为真，则调用函数 foo()可能会慢一些。这是由于 CPU 的流水线可以实现下一条指令的预取指，而分支跳转可能会使预取的指令无效，针对低概率执行的语句，重排代码的指令顺序可以减少预取指无效的情况。

3.2.7 内联函数

GNU C 提供了 inline 关键字，被该关键字修饰的函数在编译时可以嵌入调用处，功能类似于宏的扩展。

inline 关键字仅仅是建议编译器做内联展开处理，而不是强制做内联展开，如果编译优化设置为-O0，则被 inline 修饰的函数不会被内联展开。如果函数设置了强制内联属性，即__attribute__((always_inline))属性，则该函数是一定会被内联展开的。

GNU C 中的 inline 关键字有以下 3 种使用方式。

1. static inline 方式

static inline 方式是比较容易理解，也是经常使用的方式，可以将 static inline 修饰的函数视为建议内联展开的 static 函数。相对于另外两种 inline 关键字的应用，这种方式对不同的 C 语言版本是兼容最好的，Paparazzi 的内联函数大多采用的是这种方式。

2. inline 方式

不同的 C 语言版本对这种方式的处理是不同的，在 gnu99 中，GNU C 的 inline 方式不生成独立的函数代码，仅用于内联展开，即使编译优化设置为-O0 也会对 inline 方式定义的函数进行内联展开。如果 inline 方式的函数有原型的声明和定义的声明，则两者的声明中都应是 inline 方式；否则不能保证之前的特性。

3. extern inline 方式

在 gnu99 中，GNU C 的 extern inline 方式可以认为是一个普通全局函数加上了 inline 的属性。在其定义所在文件内表现和 static inline 基本一致，在能展开时会被建议内联展开编译；但同时为了在该文件之外能够调用，inline 方式的函数还会生成一份独立的代码。从文件外部看来，它和一个普通的全局函数(extern 修饰的函数)没有差异。

3.2.8　内联汇编

使用 GCC 扩展的 asm 关键字，GCC 能够支持在 C/C++代码中嵌入汇编代码，这些汇编代码称为 GCC 内联汇编(GCC Inline ASM)。

GCC 内联汇编使用的是 AT&T 语法的汇编语言，主要有基本内联汇编和带有 C/C++表达式的内联汇编两种内联汇编的方式。在此仅介绍这两种内联汇编的基本格式，详细的使用方法可以查阅相关资料。

1. 基本内联汇编

基本内联汇编的格式如下：

```
asm volatile ("Instruction List");
```

其中的 volatile 关键字是可选的，该关键字要求编译器对"Instruction List"不要做任何优化，原封不动地保留每一条指令；否则在使用优化选项进行优化编译时，编译器可能会将内联汇编表达式中的指令优化掉。Instruction List 是合法的汇编指令序列，可以为空。

2. 带有 C/C++表达式的内联汇编

GCC 允许通过 C/C++表达式指定内联汇编中"Instruction List" 中指令的输入和输出，此时可以不必关心使用了哪个寄存器，而是完全由 GCC 来安排和指定。带有 C/C++表达式的内联汇编的格式如下：

```
asm volatile ("Instruction List" : Output : Input : Clobber/Modify);
```

其中，Output 为输出运算符列表；Input 为输入运算符列表；Clobber/Modify 为被更改资源列表。

在嵌入式应用中内联汇编通常用来实现 C 语言和底层硬件的接口、某些时间很敏感的代码、底层硬件的某些特殊指令等与底层硬件密切相关的部分。Paparazzi 的内联汇编的使用主要集中在硬件的底层函数库和代码的硬件移植等部分，大多已经比较成熟，如果不需要更改微控制器硬件体系，通常不需要改动这部分代码。

3.2.9 特殊属性声明

GNU C 允许声明函数、变量和数据类型的特殊属性，以便进一步进行代码的优化或满足一些特殊的需求，是 GNU C 对 C 语言的重要扩展，许多嵌入式 C 语言中的特殊需求需要使用该扩展实现。

要指定一个声明属性，只需要在声明后添加__attribute__ ((attribute-list))。其中 attribute-list 为属性说明，如果有多个属性，则以逗号分隔。

GNU C 支持 noused、noreturn、format、section、aligned、packed 等十几个属性，这里介绍最常用的一些。

1. noused 属性

noused 是可以修饰变量或函数的属性，表示该变量或函数可能不会被使用，在编译过程中不会因为该变量或函数没有被使用而生成警告。例如，摘自 Paparazzi 的一些代码如下：

```
/* 修饰函数的参数 */
void nav_follow(uint8_t  __attribute__ ((unused)) _ac_id,
                uint32_t __attribute__ ((unused)) distance,
                uint32_t __attribute__ ((unused)) height) {}
/* 修饰变量 */
uint8_t dummy __attribute__ ((unused)) = i2c_get_data(i2c);
/* 修饰函数 */
static void__attribute__ ((unused)) process_rx_dma_interrupt(struct spi_periph * periph);
```

2. noreturn 属性

noreturn 属性用于修饰函数，表示该函数从不返回。这可以让编译器生成稍微优化的代码，最重要的是可以消除不必要的警告信息，如未初始化的变量。

3. always_inline 属性

always_inline 属性用于修饰函数，声明为内嵌的函数如果具有该属性，即使没有指明进行优化处理，也总会被扩展为内嵌代码。

4. noinline 属性

noinline 属性用于修饰函数，具有该属性的函数永远不会被扩展为内嵌代码。

5. weak 属性

weak 属性用于修饰函数或变量，被该属性修饰的函数或变量具有弱定义的特性，允许用户定义同名的函数或变量替代弱定义的函数或变量，而用户如果没有定义同名的函数或变量，则编译器会使用弱定义的函数或变量。

6. section("section-name")属性

section("section-name") 属性用于函数和变量，通常编译器在链接时将函数放在 .text 区，变量放在 .data 区或 .bss 区，使用 section 属性可以将函数或变量放在指定的节中。

例如，以 STM32 微控制器为例，在嵌入式 C 语言中，可以由链接脚本（或链接选项）在 Flash 主存储块中指定某个页为可修改的非易失性存储区，利用 section 属性可以将某些可调变量放在该页中。例如，PID 控制器的参数、滤波器参数等，期望系统重新启动后能够保留上次设定结果的变量可以放置在该页中。

7．aligned（ALIGNMENT）属性

aligned（ALIGNMENT）属性用于修饰变量、数据类型或函数，被该属性修饰的变量会按照 ALIGNMENT 字节对齐的原则分配地址，被该属性修饰的数据类型声明的变量也会按照 ALIGNMENT 字节对齐的原则分配地址。但是需要注意的是，若 ALIGNMENT 值小于默认的对齐字节数，则会按照默认的对齐字节数分配地址，即 aligned（ALIGNMENT）属性只具有增大对齐字节数的效果。若省略（ALIGNMENT）则会按照允许的最大对齐字节数分配地址。

C 语言按照 ALIGNMENT 字节对齐方式对变量分配内存是指：变量的起始地址能够被 ALIGNMENT 整除，即按照 ALIGNMENT 位地址分配内存。

按照字节对齐原则给结构或联合分配地址内存时可能会在结构或联合所占用地址内存中存在空穴，在使用 sizeof 计算结构或联合的字节长度时会将这些空穴也计算在内，即使位于结构最后一个成员后面的空穴也会计算在内。

例如，假设编译器是按照默认 4B 对齐方式分配内存，最大允许 32B 对齐方式分配内存，且 char 类型的长度为 1，int 类型的长度为 4，那么在下面的代码中结构类型 struct A1、struct A2、struct A3、struct A4 和 struct A5 的长度分别为 12、12、16、32 和 16。

```
struct A1 {
  char n;   /* 地址: 0x0 */
  int i;    /* 地址: 0x4 */
  char p;   /* 地址: 0x8 */
};
struct A2 {
  char n;   /* 地址: 0x0 */
  int i;    /* 地址: 0x4 */
  char p;   /* 地址: 0x8 */
}__attribute__ ((aligned (1)));
struct A3 {
  char n;   /* 地址: 0x0 */
  int i;    /* 地址: 0x4 */
  char p;   /* 地址: 0x8 */
}__attribute__ ((aligned (8)));
struct A4 {
  char n;   /* 地址: 0x0 */
  int i;    /* 地址: 0x4 */
  char p;   /* 地址: 0x8 */
}__attribute__ ((aligned ));
struct A5 {
  char n;   /* 地址: 0x0 */
  int i __attribute__ ((aligned (8)));/* 地址: 0x8 */
  char p;   /* 地址: 0xC */
};
```

8. packed 属性

packed 属性用于修饰变量或数据类型，仅应用于结构类型、联合类型、枚举类型、结构类型的成员或联合类型的成员。

packed 属性修饰结构类型或联合类型时，指定结构类型或联合类型的每个成员（零宽度位字段除外）使用最少的内存，当修饰枚举类型时，表示枚举成员应该使用最小的整数类型。

为结构类型或联合类型附加此属性等同于在每个结构类型或联合类型成员上附加该属性，每个结构类型或联合类型的成员均按照最小地址对齐原则分配内存，但当结构类型或联合类型的成员是结构或联合时，不会影响作为成员的结构或联合的内部布局。

例如，假设编译器是按照默认 4B 对齐方式分配内存，最大允许 32B 对齐方式分配内存，且 char 类型的长度为 1，int 类型的长度为 4，那么在下面的代码中结构类型 foo1 中的成员 x 会紧跟着成员 a 分配内存，结构类型 foo2 中每个成员都会按照最小地址对齐原则分配内存，结构类型 foo1 和 foo2 所占内存均为 9B。结构类型 foo3 占内存 8B。因为 foo4 的成员 s 占 8B 内存，所以结构类型 foo4 占内存 13B，而不是 10B 内存。示例代码如下：

```
struct foo1 {
  char a;
  int x[2] __attribute__ ((packed ));
};
struct foo2 {
  char a;
  int x[2];
}__attribute__ ((packed ));

struct foo3 {
  char c;
  int i;
};
struct __attribute__ ((packed )) foo4 {
  char c;
  int i;
  struct foo3 s;
};
```

在嵌入式 C 语言中，packed 属性可以用于定义与硬件相关的结构，可以将硬件中描述某类功能的寄存器（如某外设的寄存器组）映射到结构中。另外，packed 属性还可用于描述通信协议的帧结构。

小结

Paparazzi 使用了 GCC 编译器作为嵌入式 C 语言的开发工具，其程序代码中使用了一些 GCC 的 C 语言扩展，本章在 C 语言标准的基础上重点介绍了 C99 标准和 GCC 的 C 语言扩展，在分析 Paparazzi 的程序代码时可以参考本章内容。本章所介绍的 GCC 的 C 语言扩展包括以下内容。

- 语句表达式。在 Paparazzi 的程序代码中有所应用。
- 标号变量。
- case 范围。
- typeof 关键字。
- 条件表达式的省略。
- 内建函数。
- 内联函数。在 Paparazzi 的程序代码经常被应用，已经成为 C99 标准的一部分。
- 内联汇编。常应用在与硬件紧密相关的嵌入式程序代码中。
- 特殊属性声明。这是 GCC 的 C 语言扩展中功能最为灵活和复杂的一部分，许多特殊的设计可以依靠该扩展实现。