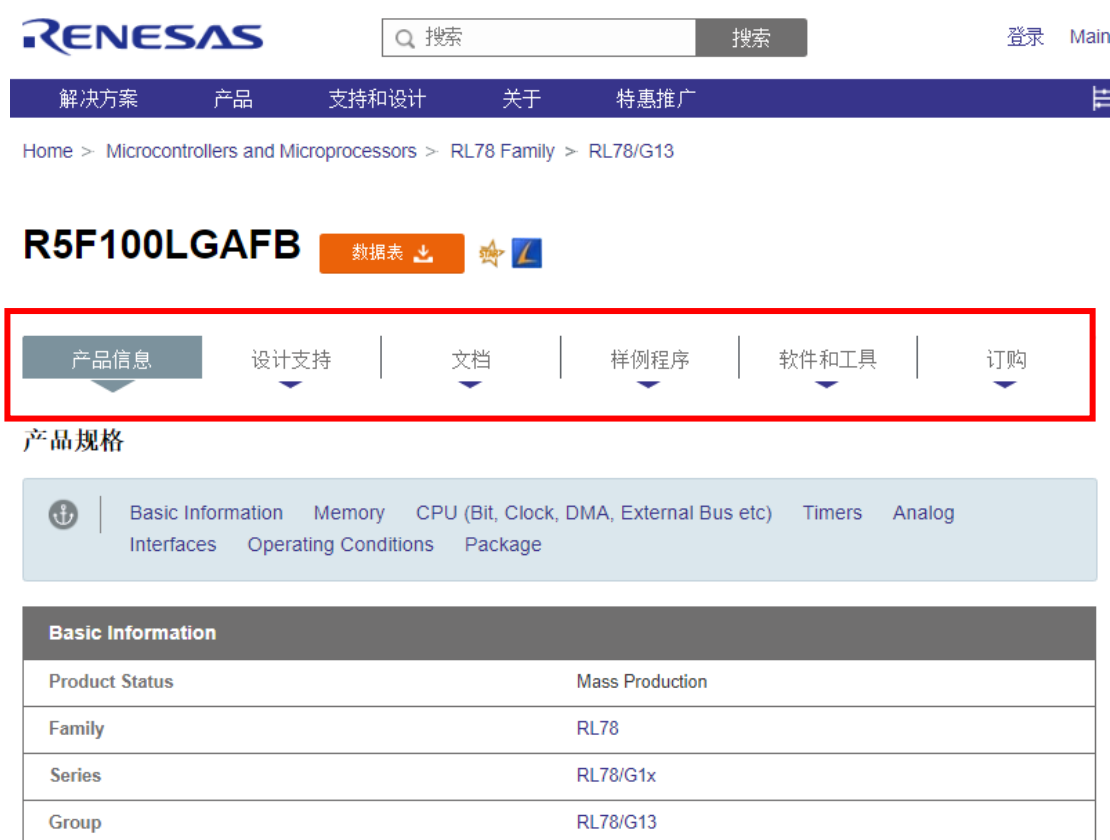


# 报告

## 1 写在前面

### 1.1 如何找到相关开发资料

基础理论部分不必多说，一般都是到相应的半导体公司官网找到该型号处理器，大都会提供丰富的设计资源（如下图），足以满足开发需求。下面列举几个比较重要的文档/资源。



RENEASAS

搜索

登录 Main

解决方案 产品 支持和设计 关于 特惠推广

Home > Microcontrollers and Microprocessors > RL78 Family > RL78/G13

**R5F100LGAFB** 数据表 下载

产品信息 设计支持 文档 样例程序 软件和工具 订购

产品规格

Basic Information Memory CPU (Bit, Clock, DMA, External Bus etc) Timers Analog  
Interfaces Operating Conditions Package

Basic Information	
Product Status	Mass Production
Family	RL78
Series	RL78/G1x
Group	RL78/G13

**数据手册：**这个文档主要描述器件的电气参数，画电路板时可能需要用到。

**编程参考手册/用户手册：**这个文档主要描述单片机上的外设（定时器，ADC，），其寄存器的具体功能。对于开发者而言，开发单片机实际上就是在**配置外设寄存器**，再加上自己的程序逻辑。所以该文档最重要。

**开发环境：**一些公司有自家的开发环境和编译器，入门该 IDE 只需要参考软件菜单中的 help 文档即可。其中也有对编译器的介绍。

**中间件/驱动：**为开发方便，公司一般也会有提供通用的兼容该单片机的中间件，如文件系统，实时操作系统，`fft` 库，闪存编程驱动等，以加快开发进度，如下图：

## 1.2 处理器架构

<b>Memory</b> Program Flash up to 512KB SRAM up to 32KB Data Flash up to 8KB	<b>RL78 16-bit CPU</b> 32MHz 41DMIPS CISC Harvard Architecture 3-stage Pipeline Four Register Banks 16-bit Barrel Shifter	
<b>System</b> DMA 4ch Interrupt Controller 4 Levels, 20 pins Clock Generation Internal, External POR, LVD MUL / DIV / MAC Debug Single-Wire	<b>Safety</b> RAM Parity Check ADC Self-diagnostic Clock Monitoring Memory CRC	<b>Analog</b> ADC 10-bit, 26ch Internal Vref. Temp. Sensor
<b>Power Management</b> HALT RTC, DMA Enabled SNOOZE Serial, ADC Enabled STOP SRAM On	<b>Timers</b> Timer Array 16-bit, 16ch Interval Timer 12-bit, 1ch WDT 17-bit, 1ch RTC Calendar	<b>Communication</b> 8 x I <sup>2</sup> C Master 2 x I <sup>2</sup> C Multi-Master 8 x CSI / SPI 7-, 8-bit 4 x UART 7-, 8-, 9-bit 1 x LIN 1ch

上图是 RL78 系列 16 位单片机的系统框图，所谓单片机即是把 CPU 和外设集成在一个芯片上。CPU 采用复杂指令集，哈佛架构，三级流水线。这里哈佛架构指的是程序跟数据是分开放置的（不一定要放在不同储存器，只要分开就算哈佛架构）。对于该单片机，粗略地说，程序放在 ROM，数据/变量放在 RAM。

### 1.3 ROM 和 RAM

RAM 随机存取；ROM 只读，这里的 ROM 指代码闪存，只读的意思是单片机对它是只读的，但是可以通过工具对它进行烧写，即下载程序。

在单片机这种嵌入式环境下进行编程，与在 PC 环境下进行编程最大的不同就是，单片机的资源是有限的，因此需要考虑程序在这些资源中是如何分配的，以便合理利用资源，为编程提供指导。

瑞萨单片机 R5F100LG 的 ROM 是 128KB，RAM 是 12288B，即只有 12KB。

### 1.4 为什么 ROM 远比 RAM 大？

因为 RAM 成本比 ROM 高，自然就比较少。另外，ROM 放体积相对大的程序，RAM 只需放计算的中间结果，空间需求不一样，所以单片机中，一般 ROM 远比 RAM 大。

另外，RAM 的读取速度比 ROM 快，在 RAM 上的程序运行的速度同样远比 ROM 上的程序运行速度快。（程序执行包括取指，译指，执行，取指即需要读取 ROM/RAM 中的程序指令，读取越快，执行越快）所以一些高级的单片机在启动时会把 ROM 上的程序直接复制到 RAM 上去运行，但是对于处理速度相对低，资源又捉襟见肘的低级单片机而言，直接在 ROM 中运行程序已经足矣，RAM 则只负责储存程序运行中间结果。

### 1.5 程序的什么部分放在 ROM，什么部分放在 RAM？

程序一般可分为代码段，常量区/只读数据段，数据段，bss 段（其实就是把程序和数据分开放置）。介绍如下：

**代码段：**程序代码。

**常量区：**字符串常量，以及 `const` 修饰的变量等。

**数据段：**已初始化的全局变量和 `static` 修饰的局部变量。

**bss 段：**未初始化或初始化为 0 的全局变量和 `static` 修饰的局部变量

其中代码段，常量区，数据段对应一个程序文件，存储在 ROM 中。这部分对应程序的大小，下载也即是把这个文件烧写到 ROM 中。

程序文件：

数据段
常量区
代码段

ROM 空间：

……（空闲）
……（空闲）
数据段
常量区
代码段

对于 RAM 而言，在程序运行时，开辟一个固定的空间（一般在低地址处）给程序中的全局变量，在剩下的空间中设置合适的堆栈（一般栈从高地址向下生长）。局部变量需要用到时就临时存放在栈中，不需要用到时就释放。另外，程序调用子函数需要“保存现场”时，同样把当前的参数存放在栈中。所以 RAM 的空闲空间是跟据栈的大小动态变化的。

程序运行时的 RAM 空间

栈底（向下生长）
局部变量，参数临时放在栈中
……（空闲）
……（空闲）
BSS 段（初始化为 0）
数据段（把 ROM 中数据段的初值复制过来）

对于高级的单片机，RAM 空间除上所述外，还把程序复制到 RAM 中，以加快执行速度。

高级单片机程序运行时的 RAM 空间

栈底（向下生长）
局部变量，参数临时放在栈中
……（空闲）

.....（空闲）
<b>BSS 段（初始化为 0）</b>
<b>数据段（把 ROM 中数据段的初值复制过来）</b>
<b>常量区</b>
<b>代码段</b>

上面的数据段和 bss 段都是针对有全局生命周期的变量而言的，之所以单独把它拎出来作为独立的段，是因为与局部变量不同，它在程序的整个生命周期中都是可见的，需要在内存 RAM 中给它单独开辟空间，一直存在。而局部变量是在栈中临时分配空间，用完后立即释放。

又之所以把 bss 段，既未初始化的部分单独拎出来，是因为该段中的变量初值为 0，没必要专门在 ROM 中专门开辟一块空间存储全 0 的变量。因此 BSS 段不占用程序大小，也不占用 ROM 空间的大小。

在初始化 RAM 并为全局变量开辟空间时，ROM 中的数据段直接复制到 RAM 中，作为全局变量的初值，而 BSS 段只需直接赋为 0。

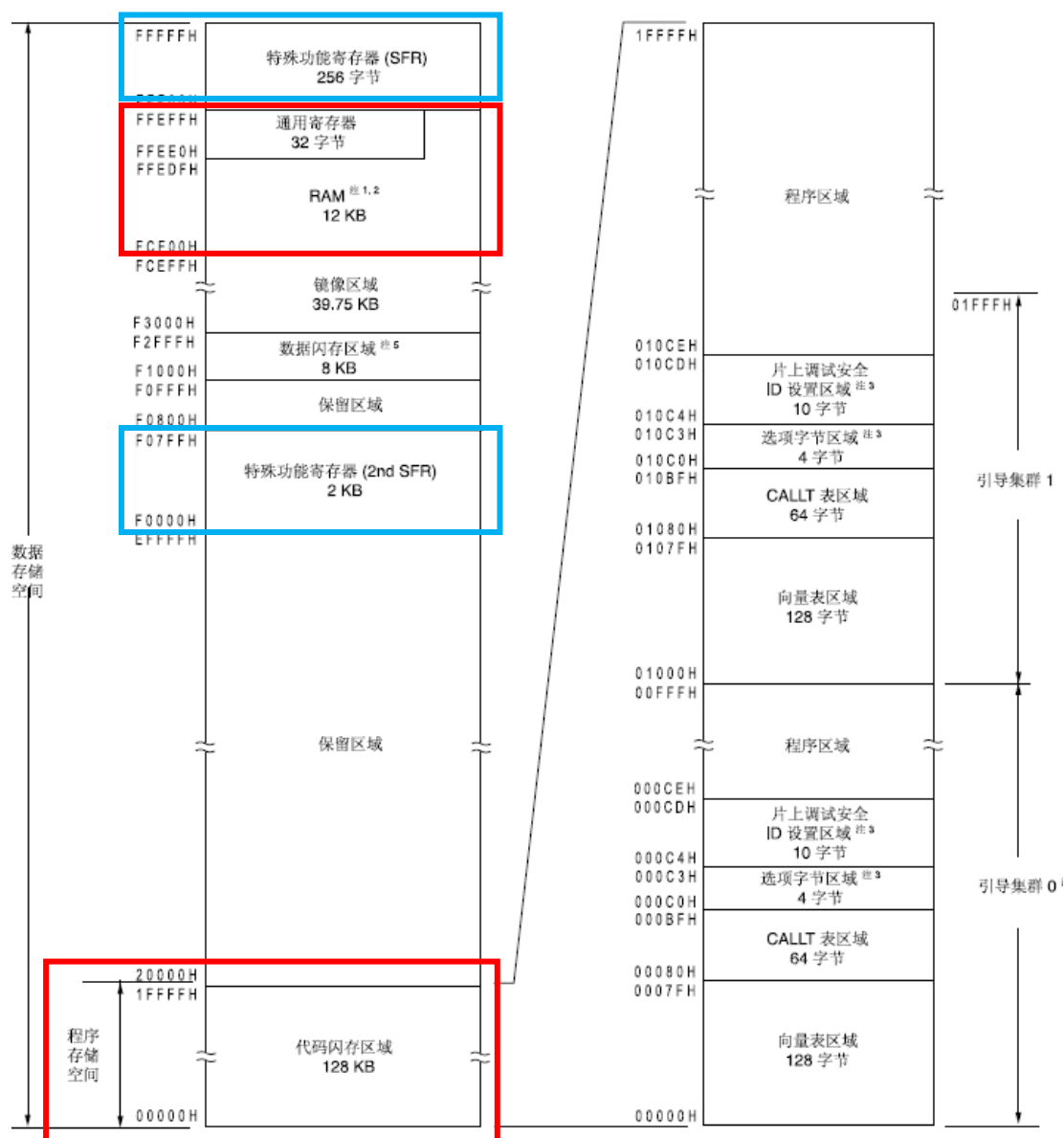
## 1.6 const 修饰符有什么作用，什么时候需要用到？

const 影响数据的存放位置。const 修饰的变量放在 ROM 的常量区，只占用 ROM 空间，而一般的变量，若为全局变量，则占用 RAM 空间，若为局部变量，变量有效时，占用 RAM 中的栈空间。

字库，图片数据等这些占用空间比较大的常量数组一般都要用 const 修饰，使之只放在 ROM 中，否则大量占用 RAM 空间，可能导致空间不够。

## 1.7 R5F100LG 的存储空间

R5F100LG 的存储空间如下图所示，其中 ROM，即程序闪存区地址为 00000H~1FFFFH，本身包含了 8KB 的引导程序空间；RAM 地址为 FCF00H~FFEFFH，本身包含 32B 的通用寄存器空间。



下面提一下特殊功能寄存器（SFR）。前面提到开发单片机实际上就是在配置各种寄存器，从而操作单片机的硬件外设。这些寄存器与硬件直接相关。每个寄存器都有独立的地址，并且有特殊的名称和它对应。如下图：

表 3-5. SFR 列表 (1/5)

地址	特殊功能寄存器(SFR)名称	符号	R/W	可操作位范围			复位后
				1 位	8 位	16 位	
FFF00H	端口寄存器 0	P0	R/W	√	√	—	00H
FFF01H	端口寄存器 1	P1	R/W	√	√	—	00H
FFF02H	端口寄存器 2	P2	R/W	√	√	—	00H

例如，P0 是端口寄存器 0，地址为 FFF00H。各个寄存器的具体含义需要参考芯片的编程手册/用户手册。

但是，编译器如何识别这些名称呢？

根据数据手册,可以在程序中添加`#pragma sfr`指令,使编译器识别这些名称,即把这些名称和对应的地址关联起来,提高代码可读性。

如:

```
#pragma sfr
```

```
P0 = 0x00;
```

等价于:

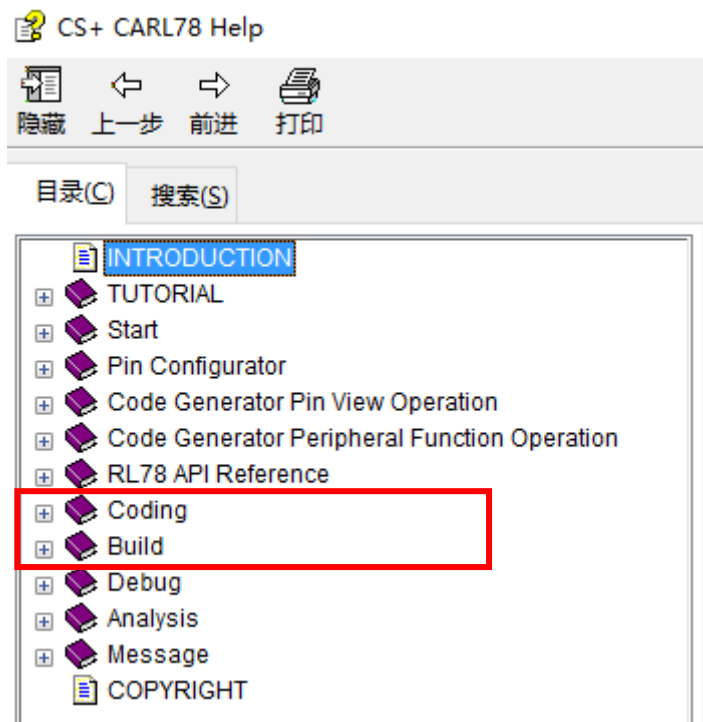
```
*FFF00H = 0x00;
```

## 1.8 开发该单片机跟普通 C 语言有什么不同?

这个问题需要查看编译器的帮助文档。该编译器叫"CA78K0R",即 RL78,78K0R C compiler package。

在 D:\Program Files (x86)\Renesas Electronics\CS+\CACX\CA78K0R\V1.72\Hlp 目录下可以找到该编译器的帮助文档。

另外在软件的帮助文档的 coding, build 子目录下同样是编译器的帮助文档。



下面主要介绍几个该编译器特有的`#pragma`指令,也是主要的不同之处。

`#pragma`指令的作用是使编译器用特殊的行为进行编译。具体如何特殊要看`#pragma`的具体指令,以及编译器是否支持。

```
#pragma sfr
```

前面已经提到,使编译器能识别特殊寄存器的名称,程序中可以直接使用该

名称。

`#pragma vect`

`#pragma interrupt`

这两个指令使我们能在 C 语言中编写中断函数。

在 C 语言中写中断服务函数，有特定的格式，如下图，这里是 INTP0 的中断服务函数。（使用代码生成器会自动生成中断函数的特殊声明和定义）。

#### Example Processing for input to INTP0 pin

```
#pragma interrupt INTP0 inter rb1

void    inter ( void ) {
        /* Processing for input to INTP0 pin*/
}
```

`#pragma di`

`#pragma ei`

这两个分别为开中断和关中断的指令。使能在 C 程序中开和关中断。

C 程序中的

`EI();`//开中断

`DI();`//关中断

对应汇编指令的

`EI`

`DI`

`#pragma halt`

`#pragma stop`

`#pragma brk`

`#pragma nop`

这几个是 CPU 的控制指令，使能在 C 程序中使用 CPU 的控制指令。

如：

`NOP();`

表示一条空指令，但占用一个指令周期的时间。



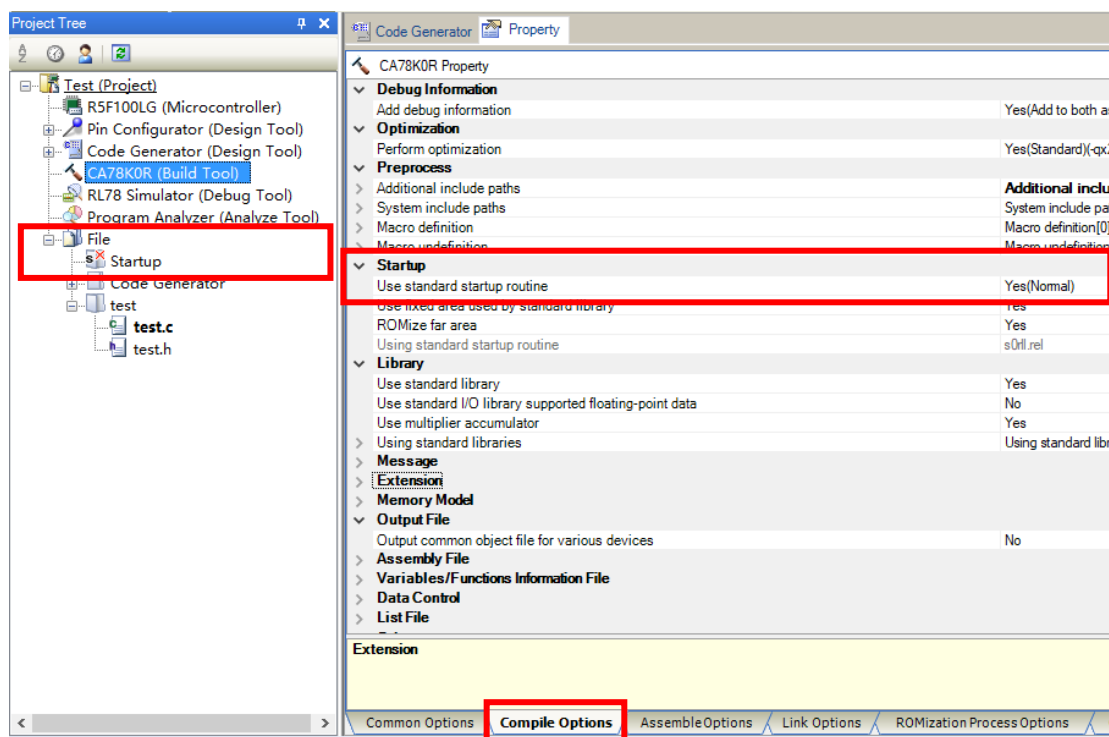
## 1.9 单片机执行的第一条执行语句是 main 函数吗？

不是。

前面说过，程序执行前，要在 RAM 开辟一个空间并初始化全局变量，还要设置好栈堆，这肯定不是硬件能自动完成的，那这又是怎么做到的呢？

又比如我们在计算机上写的最简单的一个 hello world 程序，执行它的时候也不是立即执行 main 函数里面的内容，而是由操作系统要先准备好程序执行的环境（把程序加载到内存上，设置栈堆等），最后才跳到 main 函数里面执行。而这里生成的执行文件，可不是只有我们写的函数那么简单，而是在链接的时候链接了别的文件/启动文件，这些文件就是为程序准备执行环境的。

但是单片机没有操作系统，又怎么给程序设置好执行环境呢？这是启动文件做的事。工程里都有默认的启动文件，在编译工具的编译属性页里设置，而工程里的 startup 目录下则为空，如下图：



默认的启动文件 cstart.asm 在 D:\Program Files (x86)\Renesas Electronics\CS+\CACX\CA78K0R\V1.72\Src\cc78k0r\src 里面，文件为汇编语言形式，做的工作主要有：

1. 设置栈指针
2. 调用 hdwinit() 函数进行硬件初始化
3. 初始化 RAM
4. 调用 main() 函数

注意启动文件会先调用 hdwinit() 进行硬件初始化，最后调用 main() 函数，所

以不必自己再重复调用 `hdwinit()` 函数。

## 1.10 代码生成器都做了什么？

代码生成器生成的代码可以分为两部分，一部分与外设无关，一部分与外设有关。

第一部分与外设无关，有如下文件：

1. `r_cg_macrodriver.h`
2. `r_cg_userdefine.h`
3. `r_systeminit.c`
4. `r_main.c`

其中 `r_cg_macrodriver.h` 有一些比较重要的 `#pragma` 指令（前面已介绍过），如下：

```
#pragma sfr
#pragma DI
#pragma EI
#pragma NOP
#pragma HALT
#pragma STOP
```

还有一些宏定义，基本不需要用。如果自己写的程序需要直接操作寄存器/sfr，需要在文件最前面包含这个头文件，或者直接在文件最前面加上 `#pragma sfr`。注意是文件最前面，因为 `#pragma` 语句必须在文件的最前面，否则会报错。

`r_cg_userdefine.h` 顾名思义，用户自己的一些宏定义可以放在这个文件里面，可以不用。

`r_systeminit.c` 是这里介绍到的第一个 `.c` 文件，代码生成器生成的 `.c` 文件有统一的格式，这里简要说一下，主要有如下几部分：

1. 免责声明  
忽略。
2. 文件说明  
忽略。
3. Pragma 指令  
把 Pragma 指令放在 start 和 end 之间。

```

3/*****
Pragma directive
*****
/* Start user code for pragma. Do not edit comment generated here */
/* End user code. Do not edit comment generated here */

```

#### 4. 包含头文件

把包含头文件放在 start 和 end 之间。

```

3/*****
Includes
*****
#include "r_cg_macrodriver.h"
#include "r_cg_cgc.h"
#include "r_cg_port.h"
#include "r_cg_rtc.h"
/* Start user code for include. Do not edit comment generated here */
/* End user code. Do not edit comment generated here */

```

#### 5. 全局变量和函数申明

把全局变量和函数申明放在 start 和 end 之间。

```

3/*****
Global variables and functions
*****
/* Start user code for global. Do not edit comment generated here */
/* End user code. Do not edit comment generated here */

```

#### 6. 系统函数定义

由系统生成，若需添加用户代码，务必放在 start 和 end 之间。

#### 7. 用户函数定义

把用户函数定义放在 start 和 end 之间。

```

/* Start user code for adding. Do not edit comment generated here */
/* End user code. Do not edit comment generated here */

```

这里是针对代码生成器生成的代码而言，之所以强调用户代码放置的位置，是因为代码生成器生成新代码的时候会覆盖原文件，但是不会改变/\* Start……和/\*End……这两行注释之间的内容。当然自己写的文件就无所谓。

r\_systeminit.c 文件里面由一个 hdwinit()函数，如下：

```

void hdwinit(void)
{
    DI();
    R_Systeminit();
}

```

```

void R_Systeminit(void)
{
    PIOR = 0x00U;
    R_CGC_Get_ResetSource();
    R_CGC_Create();
    R_XXX_Create();//XXX 为某一个外设

    IAWCTL = 0x00U;
}

```

可见，`hdwinit()`函数先关中断 `DI()`，然后执行一些硬件外设初始化的工作，如时钟初始化，看门狗初始化等，并且只要在代码生成器中使用了某一个外设 `XXX`，相应的外设初始化函数 `R_XXX_Create()`会被自动添加到这个函数中。所以外设初始化函数 `R_XXX_Create()`也不需要自己再重复调用。

前面提到，`hdwinit()`会被启动文件调用，所以不需要自己再去调用一次。

`r_main.c` 的主要内容如下：

```

void main(void)
{
    R_MAIN_UserInit();
    /* Start user code. Do not edit comment generated here */
    while (1U)
    {
        ;
    }
    /* End user code. Do not edit comment generated here */
}

void R_MAIN_UserInit(void)
{
    /* Start user code. Do not edit comment generated here */
    EI();
    /* End user code. Do not edit comment generated here */
}

```

可见，`main` 函数先调用初始化函数，再进入死循环。（`1U` 指无符号整数 1）。初始化函数中，添加必要的初始化代码，然后开中断 `EI()`，以便响应中断。

第二部分与外设有关，有如下文件：

假设外设名称为 xxx，则生成如下相应的文件：

1. `r_cg_xxx.h`
2. `r_cg_xxx.c`
3. `r_cg_xxx_users.c`

其中，`r_cg_xxx.h` 包含该外设相关宏和函数等定义。

`r_cg_xxx.c` 包含改外设的初始化函数，另提供一定的该外设相关的操作接口，比如关闭/开启定时器，设置实时时钟等，方便开发。

`r_cg_xxx_users.c` 主要是中断函数的申明和定义，用户代码主要写在这里。

## 1.11 用与不用代码生成器的区别

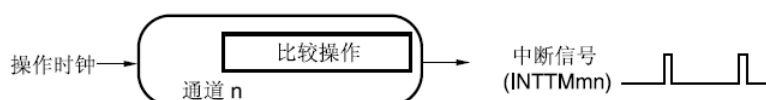
下面以定时器为例，说明代码生成器如何加速开发，以及和不用代码生成器直接开发的区别。

下面用定时器实现每隔 1ms 执行一件事的功能。

查看《用户手册》第六章 定时器阵列单元部分，定时器有诸多功能，这里用到间隔定时器功能，如下：

### (1) 间隔定时器

单元中的各个定时器都可用作按照固定间隔产生中断(INTTMmn)的基准定时器。



具体如何设置定时时间，下面有说明：

### (1) 间隔定时器

定时器阵列单元可用作以固定间隔产生 INTTMmn（定时器中断）的基准定时器。

中断产生周期可以用下述表达式计算。

$$\text{INTTMmn (定时器中断) 的产生周期} = \text{计数时钟的周期} \times (\text{TDRmn 的设置值} + 1)$$

再看两幅图：

图 6-41. 作为间隔定时器/方波输出的操作框图

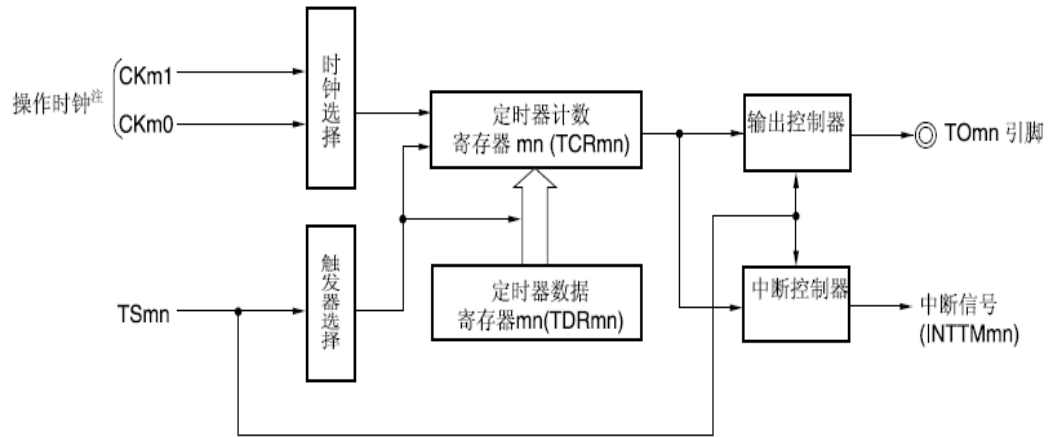
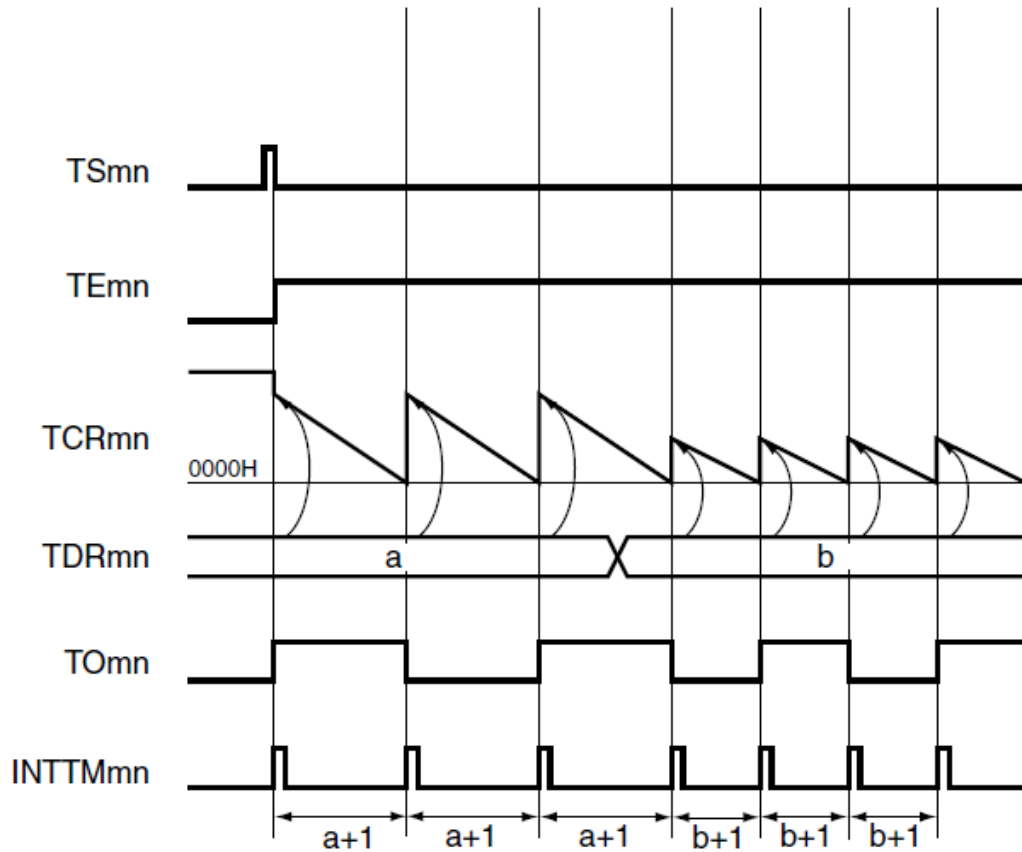


图 6-42. 作为间隔定时器/方波输出的操作基本时序示例 (MDmn0 = 1)



TSmn: 定时器通道开始寄存器 m (TSm) 的位 n  
 TEmn: 定时器通道允许状态寄存器 m (TEm) 的位 n  
 TCRmn: 定时器计数寄存器 mn (TCRmn)  
 TDRmn: 定时器数据寄存器 mn (TDRmn)  
 T0mn: T0mn 引脚输出信号

在  $TE_{mn}$  为高的前提下， $TS_{mn}$  下降沿触发计数， $TDR_{mn}$  中的数保存到  $TCR_{mn}$  中，随后，每来一个时钟， $TCR_{mn}$  计数减 1，直到减为 0，触发一个中断  $INTTM_{mn}$ ，程序执行相应的中断处理函数，并且，定时器硬件重新把  $TDR_{mn}$  中的数保存到  $TCR_{mn}$  中，开始下一轮计数。若  $TDR_{mn}$  改变，相应的计数周期改变。

这里有两个问题，一是前面说过定时器有诸多功能，怎么选择当前的间隔定时器模式，这需要配置相关寄存器；二是定时器的时钟源多快，这也需要配置相关寄存器。

设置定时器的模式和时钟源，需要设置定时器模式寄存器  $TMR_{mn}$ ，描述如下：

符号	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$TMR_{mn}$ (n = 2, 4, 6)	CKS mn1	CKS mn0	0	CCS mn	MAST ERmn	STS mn2	STS mn1	STS mn0	CIS mn1	CIS mn0	0	0	MD mn3	MD mn2	MD mn1	MD mn0

其中第 1，2，3 位，描述如下：

MD mn3	MD mn2	MD mn1	通道n的操作模式	对应功能	TCR的计数操作
0	0	0	间隔定时器模式	间隔定时器/ 方波输出/ 分频器功能 / PWM输出（主）	递减
0	1	0	捕捉模式	输入脉冲间隔测量	递增
0	1	1	事件计数器模式	外部事件计数器	递减
1	0	0	单计数模式	延迟计数器/ 单触发脉冲输出/ PWM输出（从属）	递减
1	1	0	捕捉&单计数模式	输入信号的高/低电平宽度的测量	递增
其他			禁止设置		
各模式操作根据MDmn0位的不同而有所差异（参阅下表）。					

所以把这三位设置为 0 即可。

第 14，15 位的描述如下：

CKS mn1	CKS mn0	通道n操作时钟( $f_{MCK}$ )的选择
0	0	定时器时钟选择寄存器m (TPSm)设置的操作时钟CKm0
0	1	定时器时钟选择寄存器m (TPSm)设置的操作时钟CKm2
1	0	定时器时钟选择寄存器m (TPSm)设置的操作时钟CKm1
1	1	定时器时钟选择寄存器m (TPSm)设置的操作时钟CKm3
操作时钟( $f_{MCK}$ )用于边沿检测电路。通过设置CCSmn位来产生计数时钟( $f_{CLK}$ )和采样时钟。 仅限通道1和通道3可以选择操作时钟CKm2和CKm3。		

设置不同的值，即可选择不同的操作时钟  $CK_{mn}$ ，那么，操作时钟  $CK_{mn}$  的频率又是多少呢？这在另一个寄存器  $TPS_m$  设置，如下：

符号	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TPSm	0	0	PRS m31	PRS m30	0	0	PRS m21	PRS m20	PRS m13	PRS m12	PRS m11	PRS m10	PRS m03	PRS m02	PRS m01	PRS m00

PRS mk3	PRS mk2	PRS mk1	PRS mk0	操作时钟(CKmk)的选择 <sup>注</sup> (k = 0, 1)					
					f <sub>CLK</sub> = 2 MHz	f <sub>CLK</sub> = 5 MHz	f <sub>CLK</sub> = 10 MHz	f <sub>CLK</sub> = 20 MHz	f <sub>CLK</sub> = 32 MHz
0	0	0	0	f <sub>CLK</sub>	2 MHz	5 MHz	10 MHz	20 MHz	32 MHz
0	0	0	1	f <sub>CLK</sub> /2	1 MHz	2.5 MHz	5 MHz	10 MHz	16 MHz
0	0	1	0	f <sub>CLK</sub> /2 <sup>2</sup>	500 kHz	1.25 MHz	2.5 MHz	5 MHz	8 MHz
0	0	1	1	f <sub>CLK</sub> /2 <sup>3</sup>	250 kHz	625 kHz	1.25 MHz	2.5 MHz	4 MHz
0	1	0	0	f <sub>CLK</sub> /2 <sup>4</sup>	125 kHz	312.5 kHz	625 kHz	1.25 MHz	2 MHz

各个位设置不同的值，即可设置始终分频数，确定始终频率。假设不分频。这里还有一个 f<sub>clk</sub>，这是系统时钟，是多少，又是怎么设置的呢，系统时钟设置在另一个地方，过程比定时器的设置复杂多了，也是配置相关寄存器，这里不再赘述，默认系统时钟为 16MHz。

回顾一下，为了产生周期为 1ms 的周期中断，TDRmn 设置的值为：

$$16000000/1000-1=15999=0x3E7F。$$

可见，用定时器实现一个周期事件功能需要查看用户手册，配置相应的寄存器，过程稍微繁琐，代码可读性也不强。

再看看代码生成器生成了什么。

首先是 r\_cg\_timer.h，里面主要有两部分，一是用宏去代表各个寄存器各个位不同值的含义，如下：

```
/* Operating mode and clear mode selection (PRSm03 - PRSm00) */
#define _0000_TAU_CKM0_FCLK_0 (0x0000U) /* ckm0 - fCLK */
#define _0001_TAU_CKM0_FCLK_1 (0x0001U) /* ckm0 - fCLK/2^1 */
#define _0002_TAU_CKM0_FCLK_2 (0x0002U) /* ckm0 - fCLK/2^2 */
#define _0003_TAU_CKM0_FCLK_3 (0x0003U) /* ckm0 - fCLK/2^3 */
```

对比如下数据手册内容，就会发现，它把寄存器值的不同组合代表的不同含义用宏的形式表现出来，使代码更具可读性。

PRS mk3	PRS mk2	PRS mk1	PRS mk0	操作时钟(CKmk)的选择 <sup>注</sup> (k = 0, 1)					
					f <sub>CLK</sub> = 2 MHz	f <sub>CLK</sub> = 5 MHz	f <sub>CLK</sub> = 10 MHz	f <sub>CLK</sub> = 20 MHz	f <sub>CLK</sub> = 32 MHz
0	0	0	0	f <sub>CLK</sub>	2 MHz	5 MHz	10 MHz	20 MHz	32 MHz
0	0	0	1	f <sub>CLK</sub> /2	1 MHz	2.5 MHz	5 MHz	10 MHz	16 MHz
0	0	1	0	f <sub>CLK</sub> /2 <sup>2</sup>	500 kHz	1.25 MHz	2.5 MHz	5 MHz	8 MHz
0	0	1	1	f <sub>CLK</sub> /2 <sup>3</sup>	250 kHz	625 kHz	1.25 MHz	2.5 MHz	4 MHz

二是代码生成器提供的接口函数的申明。



```

3/*****
Global functions
*****
void R_TAU0_Create(void);
void R_TAU0_Channel0_Start(void);
void R_TAU0_Channel0_Stop(void);

```

这里提供了三个函数，一是时钟的初始化函数，前面说过该函数会在 `hdwinit()` 中被调用，所以不需要自己再重复调用，另外另一个是定时器的开启和关闭函数。

接着是 `r_cg_timer.c` 文件，里面主要是接口函数的实现，下面看看定时器是如何初始化的，即函数 `R_TAU0_Create()`。

首先是使能定时器，接着配置 CKMn 的分频系数。

```

TAU0EN = 1U; /* supplies input clock */
TPS0 = _0000_TAU_CKM0_FCLK_0 | _0000_TAU_CKM1_FCLK_0 | _0000_TAU_CKM2_FCLK_1 | _0000_TAU_CKM3_FCLK_8;

```

然后关闭所有通道，清除所有中断。

```

/* Stop all channels */
TT0 = _0001_TAU_CH0_STOP_TRG_ON | _0002_TAU_CH1_STOP_TRG_ON | _0004_TAU_CH2_STOP_TRG_ON |
      _0008_TAU_CH3_STOP_TRG_ON | _0010_TAU_CH4_STOP_TRG_ON | _0020_TAU_CH5_STOP_TRG_ON |
      _0040_TAU_CH6_STOP_TRG_ON | _0080_TAU_CH7_STOP_TRG_ON | _0200_TAU_CH1_H8_STOP_TRG_ON |
      _0800_TAU_CH3_H8_STOP_TRG_ON;
/* Mask channel 0 interrupt */
TMMK00 = 1U; /* disable INTTM00 interrupt */
TMIF00 = 0U; /* clear INTTM00 interrupt flag */

```

最后设置通道 0 的中断优先级，设置定时器 0 的模式，装载值，使能。

```

/* Set INTTM00 low priority */
TMPR100 = 1U;
TMPR000 = 1U;
/* Channel 0 used as interval timer */
TMR00 = _0000_TAU_CLOCK_SELECT_CKM0 | _0000_TAU_CLOCK_MODE_CKS | _0000_TAU_COMBINATION_SLAVE |
      _0000_TAU_TRIGGER_SOFTWARE | _0000_TAU_MODE_INTERVAL_TIMER | _0000_TAU_START_INT_UNUSED;
TDR00 = _3E7F_TAU_TDR00_VALUE;
TO0 &= ~_0001_TAU_CH0_OUTPUT_VALUE_1;
TOE0 &= ~_0001_TAU_CH0_OUTPUT_ENABLE;

```

这里的装载值设置为 `_3E7F_TAU_TDR00_VALUE`（查看宏定义，也即 `0x3E7F`，跟之前计算的一致）。

```

/* 16-bit timer data register 00 (TDR00) */
#define _3E7F_TAU_TDR00_VALUE (0x3E7FU)

```

再看看开始定时器的函数 `R_TAU0_Channel0_Start()`

如图，先清除中断标志位，使能中断，再触发定时器，直接调用很方便。

```

void R_TAU0_Channel0_Start(void)
{
    TMIF00 = 0U;    /* clear INTTM00 interrupt flag */
    TMMK00 = 0U;    /* enable INTTM00 interrupt */
    TS0 |= _0001_TAU_CH0_START_TRG_ON;
}

```

最后看看停止定时器的函数 `R_TAU0_Channel0_Stop()`

如图，先停止定时器，禁止中断，最后清除中断标志位，很规范，直接调用也很方便。

```

void R_TAU0_Channel0_Stop(void)
{
    TT0 |= _0001_TAU_CH0_STOP_TRG_ON;
    /* Mask channel 0 interrupt */
    TMMK00 = 1U;    /* disable INTTM00 interrupt */
    TMIF00 = 0U;    /* clear INTTM00 interrupt flag */
}

```

另外是 `r_cg_timer_user.c` 文件，里面主要是中断函数的申明和实现。主要的用户代码，即中断里面要做的事，也写在这里。

中断函数用 `#pragma` 指令申明，表示这是一个响应 `INTTM00` 的函数，不是普通的函数。

```

#pragma interrupt INTTM00 r_tau0_channel0_interrupt

__interrupt static void r_tau0_channel0_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    /* End user code. Do not edit comment generated here */
}

```

## 1.12 总结

以上是对单片机编程与直接在 PC 操作系统环境下编程区别的介绍，了解了这些，学会代码生成器操作单片机硬件的原理，剩下的就是纯逻辑算法的编程了，跟在 PC 操作系统环境下编程差异不大。所以就先写到这里。