第 5 章

PWM 输出与看门狗定时器应用实例

本章将介绍 PWM 输出与看门狗定时器应用实例，包括 STM32F103 定时器概述、STM32 通用定时器、STM32 PWM 输出应用实例和看门狗定时器。

5.1 STM32F103 定时器概述

从本质上讲，定时器就是“数字电路”课程中学过的计数器(Counter)，它像闹钟一样忠实地为处理器完成定时或计数任务，几乎是现代微处理器必备的一种片上外设。很多读者在初次接触定时器时都会提出这样一个问题：既然 Arm 内核每条指令的执行时间都是固定的，且大多数是相等的，那么我们可以用软件的方法实现定时吗？例如，在 72MHz 系统时钟下要实现 1μs 的定时，完全可以通过执行 72 条不影响状态的“无关指令”实现。既然这样，STM32 中为什么还要有定时/计数器这样一个完成定时工作的硬件结构呢？其实，读者的看法一点也没有错，确实可以通过插入若干条不产生影响的“无关指令”实现固定时间的定时。但这会带来两个问题：其一，在这段时间中，STM32 不能做其他任何事情，否则定时将不再准确；其二，这些“无关指令”会占据大量程序空间。而当嵌入式处理器中集成了硬件的定时结构以后，它就可以在内核运行执行其他任务的同时完成精确的定时，并在定时结束后通过中断、事件等方法通知内核或相关外设。简单地说，定时器最重要的作用就是将内核从简单、重复的延时工作中解放出来。

当然，定时器的核心电路结构是计数器。当它对 STM32 内部固定频率的信号进行计数时，只要指定计数器的计数值，也就相当于固定了从定时器启动到溢出之间的时间长度。这种对内部已知频率计数的工作方式称为“定时方式”。定时器还可以对外部管脚输入的未知频率信号进行计数，此时由于外部输入时钟频率可能改变，从定时器启动到溢出之间的时间长度是无法预测的，软件所能判断的仅仅是外部脉冲的个数。因此，这种计数时钟来自外部的工作方式只能称为“计数方式”。在这两种基本工作方式的基础上，STM32 的定时器又衍生出了输入捕获、输出比较、PWM、脉冲计数、编码器接口等多种工作模式。

定时与计数的应用十分广泛。在实际生产过程中，许多场合都需要定时或计数操作，如

产生精确的时间、对流水线上的产品进行计数等。因此，定时/计数器在嵌入式单片机应用系统中十分重要。定时和计数可以通过以下方式实现。

1．软件延时

单片机是在一定时钟下运行的，可以根据代码所需的时钟周期完成延时操作。软件延时会导致 CPU 利用率低，因此主要用于短时间延时，如高速 A/D 转换器。

2．可编程定时/计数器

微控制器中的可编程定时/计数器可以实现定时和计数操作，定时/计数器功能由程序灵活设置，重复利用。设置好后由硬件与 CPU 并行工作，不占用 CPU 时间，这样在软件的控制下，可以实现多个精密定时/计数。嵌入式处理器为了适应多种应用，通常集成多个高性能的定时/计数器。

微控制器中的定时器本质上是一个计数器，可以对内部脉冲或外部输入进行计数，不仅具有基本的延时/计数功能，还具有输入捕获、输出比较和 PWM 波形输出等高级功能。在嵌入式开发中，充分利用定时器的强大功能，可以显著提高外设驱动的编程效率和 CPU 利用率，增强系统的实时性。

STM32 内部集成了多个定时/计数器。根据型号不同，STM32 系列芯片最多包含 8 个定时/计数器。其中，TIM6 和 TIM7 为基本定时器；TIM2～TIM5 为通用定时器；TIM1 和 TIM8 为高级控制定时器，功能最强。3 种定时器的功能如表 5-1 所示。此外，在 STM32 中还有两个看门狗定时器和一个系统滴答定时器。

表 5-1　STM32 定时器的功能

主要功能	高级控制定时器	通用定时器	基本定时器
内部时钟源(8MHz)	√	√	√
带 16 位分频的计数单元	√	√	√
更新中断和 DMA	√	√	√
计数方向	向上、向下、双向	向上、向下、双向	向上
外部事件计数	√	√	×
其他定时器触发或级联	√	√	×
4 个独立输入捕获、输出比较通道	√	√	×
单脉冲输出方式	√	√	×
正交编码器输入	√	√	×
霍尔传感器输入	√	√	×
输出比较信号死区产生	√	×	×
制动信号输入	√	×	×

STM32F103 定时器相比于传统的 51 单片机要完善和复杂得多，它是专为工业控制应用量身定做的。定时器有很多用途，包括基本定时功能、生成输出波形(比较输出、PWM 和带死区插入的互补 PWM)和测量输入信号的脉冲宽度(输入捕获)等。

5.2 STM32 通用定时器

5.2.1 通用定时器简介

通用定时器(TIM2～TIM5)由一个通过可编程预分频器驱动的 16 位自动装载计数器构成。它适用于多种场合，包括测量输入信号的脉冲长度(输入捕获)或产生输出波形(输出比较和 PWM)。使用定时器预分频器和 RCC 时钟控制器预分频器，脉冲长度和波形周期可以在几微秒到几毫秒间调整。每个定时器都是完全独立的，没有互相共享任何资源，它们可以同步操作。

5.2.2 通用定时器的主要功能

通用定时器的主要功能如下。

(1) 16 位向上、向下、向上/向下自动装载计数器。

(2) 16 位可编程(可以实时修改)预分频器，计数器时钟频率的分频系数为 1～65536 的任意数值。

(3) 4 个独立通道：输入捕获、输出比较、PWM 生成(边缘或中间对齐模式)、单脉冲模式输出。

(4) 使用外部信号控制定时器和定时器互连的同步电路。

(5) 以下事件发生时产生中断/DMA：

① 更新、计数器向上溢出/向下溢出、计数器初始化(通过软件或内部/外部触发)；

② 触发事件(计数器启动、停止、初始化或由内部/外部触发计数)；

③ 输入捕获；

④ 输出比较。

(6) 支持针对定位的增量(正交)编码器和霍尔传感器电路。

(7) 触发输入作为外部时钟或按周期的电流管理。

5.2.3 通用定时器的功能描述

通用定时器内部结构如图 5-1 所示，相比于基本定时器，其内部结构要复杂得多，其中最显著的区别就是增加了 4 个捕获/比较寄存器 TIMx_CCR，这也是通用定时器拥有如此多强大功能的原因。

1. 时基单元

可编程通用定时器的主要部分是一个 16 位计数器和与其相关的自动装载寄存器。这个计数器可以向上计数、向下计数或向上/向下双向计数。计数器时钟由预分频器分频得到。计数器、自动装载寄存器和预分频器寄存器可以由软件读写，在计数器运行时仍可以读写。时基单元包含计数器寄存器(TIMx_CNT)、预分频器寄存器(TIMx_PSC)和自动装载寄存器(TIMx_ARR)。

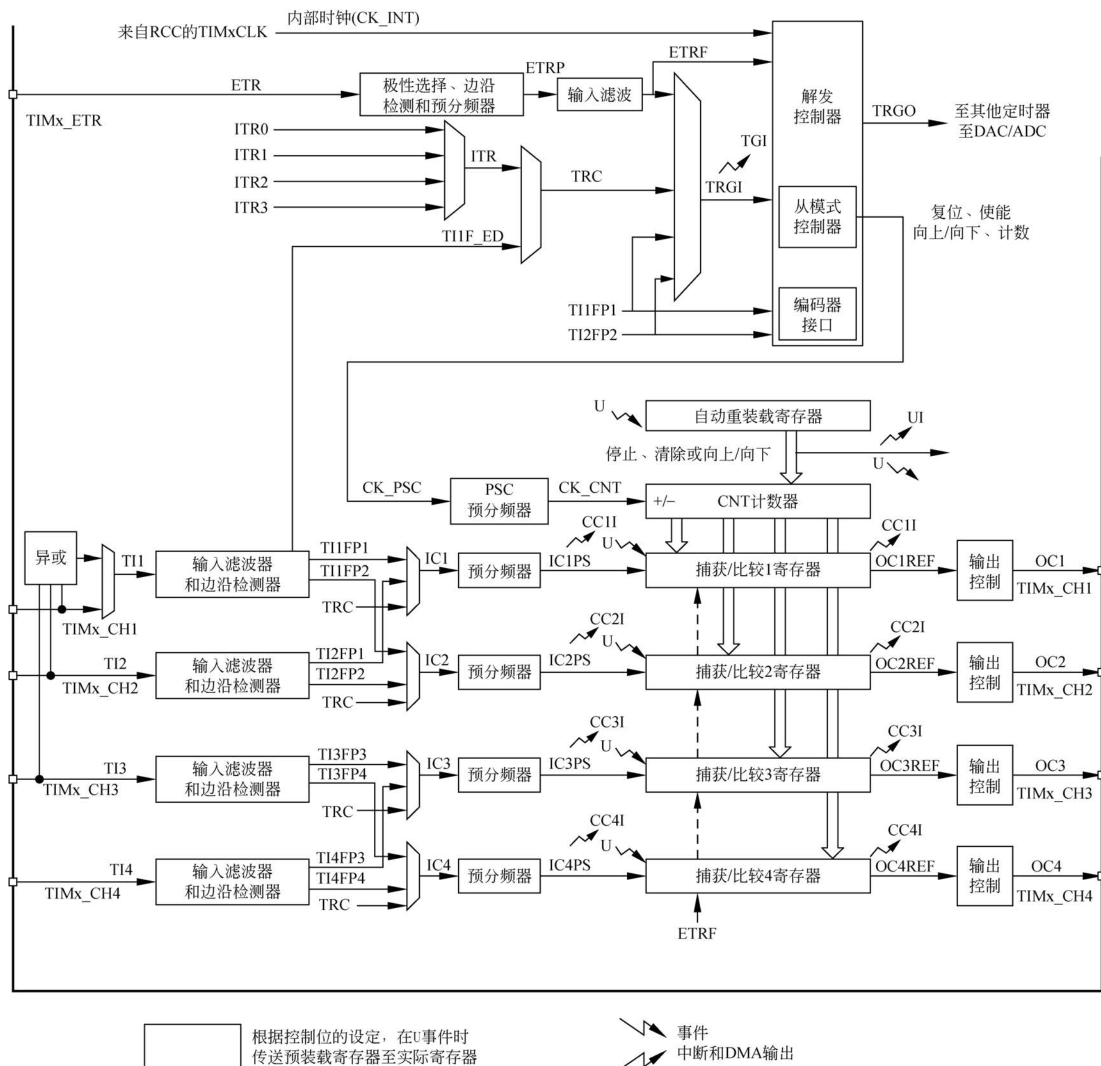

图 5-1 通用定时器内部结构

预分频器可以将计数器的时钟频率按 1～65536 的任意值分频。它是基于一个(在 TIMx_PSC 寄存器中的)16 位寄存器控制的 16 位计数器。这个控制寄存器带有缓冲器,它能够在工作时被改变。新的预分频器参数在下一次更新事件到来时被采用。

2. 计数模式

1）向上计数模式

向上计数模式的工作过程与基本定时器向上计数模式相同,工作过程如图 5-2 所示。在向上计数模式中,计数器在时钟 CK_CNT 的驱动下从 0 计数到自动重装载寄存器 TIMx_ARR 的预设值,然后重新从 0 开始计数,并产生一个计数器溢出事件,可触发中断或 DMA 请求。当发生一个更新事件时,所有寄存器都被更新,硬件同时设置更新标志位。

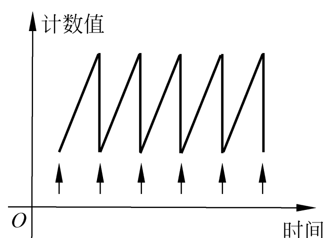

图 5-2 向上计数模式

对于一个工作在向上计数模式的通用定时器，自动重装载寄存器 T1Mx_ARR 的值为 0x0036，内部预分频系数为 4（预分频寄存器 TIMx_PSC 的值为 3），计数器时序图如图 5-3 所示。

2）向下计数模式

通用定时器向下计数模式工作过程如图 5-4 所示。在向下计数模式中，计数器在时钟 CK_CNT 的驱动下从自动重装载寄存器 TIMx_ARR 的预设值开始向下计数到 0，然后从自动重装载寄存器 TIMx_ARR 的预设值重新开始计数，并产生一个计数器溢出事件，可触发中断或 DMA 请求。当发生一个更新事件时，所有寄存器都被更新，硬件同时设置更新标志位。

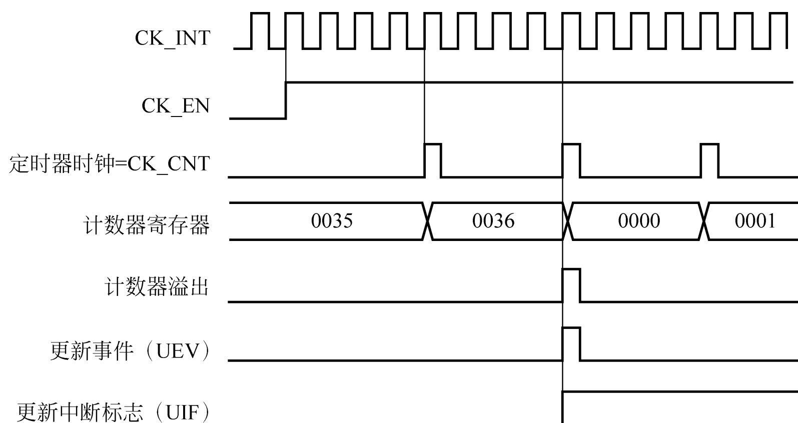

图 5-3 计数器时序图（内部预分频系数为 4）

对于一个工作在向下计数模式的通用定时器，自动重装载寄存器 TIMx_ARR 的值为 0x0036，内部预分频系数为 2（预分频寄存器 TIMx_PSC 的值为 1），计数器时序图如图 5-5 所示。

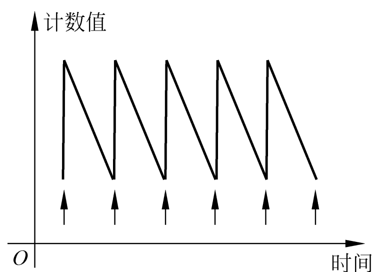

图 5-4 向下计数模式

3）向上/向下计数模式

向上/向下计数模式又称为中央对齐模式或双向计数模式，其工作过程如图 5-6 所示。计数器从 0 开始计数到自动重装载寄存器 TIMx_ARR 的值 −1，产生一个计数器溢出事件，向下计数到 1 并且产生一个计数器下溢事件；然后再从 0 开始重新计数。在这个模式下，不能写入 TIMx_CR1 中的 DIR 方向位，它由硬件更新并指示当前的计数方向。可以在每次计数上溢和每次计数下溢时产生更新事件，触发中断或 DMA 请求。

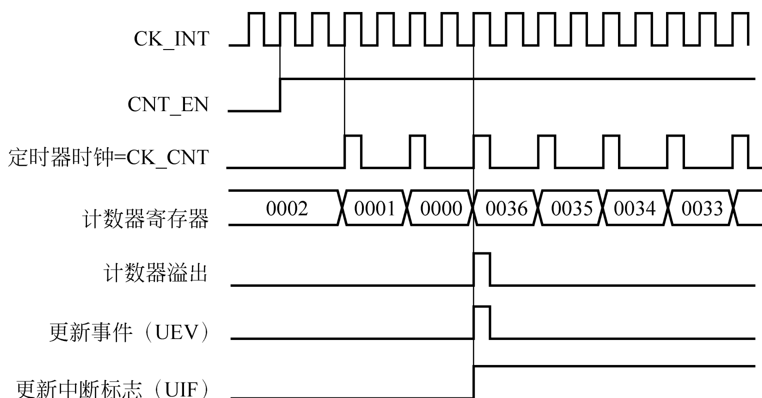

图 5-5 计数器时序图(内部预分频系数为 2)

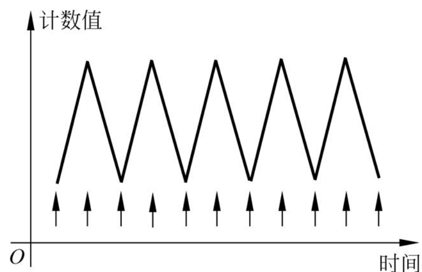

图 5-6 向上/向下计数模式

对于一个工作在向上/向下计数模式的通用定时器,自动重装载寄存器 TIMx_ARR 的值为 0x06,内部预分频系数为 1(预分频寄存器 TIMx_PSC 的值为 0),计数器时序图如图 5-7 所示。

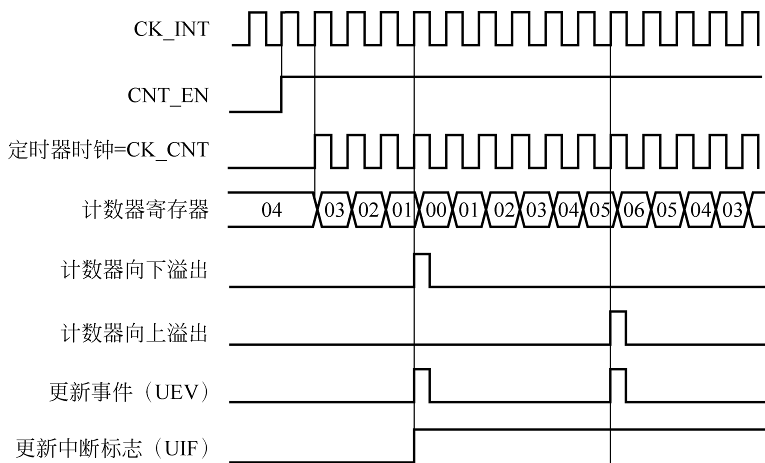

图 5-7 计数器时序图(内部预分频系数为 1)

3. 时钟选择

相比于基本定时器单一的内部时钟源，STM32F103 通用定时器的 16 位计数器的时钟源有多种选择，可由以下时钟源提供。

1）内部时钟(CK_INT)

内部时钟 CK_INT 来自 RCC 的 TIMxCLK，根据 STM32F103 时钟树，通用定时器 TIM2～TIM5 内部时钟 CK_INT 的来源 TIM_CLK 与基本定时器相同，都是来自 APB1 预分频器的输出。通常情况下，时钟频率为 72MHz。

2）外部输入捕获引脚 TIx(外部时钟模式 1)

外部输入捕获引脚 TIx(外部时钟模式 1)来自外部输入捕获引脚上的边沿信号。计数器可以在选定的输入端(引脚 1：TI1FP1 或 TI1F_ED，引脚 2：TI2FP2)的每个上升沿或下降沿计数。

3）外部触发输入引脚 ETR(外部时钟模式 2)

外部触发输入引脚 ETR(外部时钟模式 2)来自外部引脚 ETR。计数器能在外部触发输入 ETR 的每个上升沿或下降沿计数。

4）内部触发器输入 ITRx

内部触发输入 ITRx 来自芯片内部其他定时器的触发输入，使用一个定时器作为另一个定时器的预分频器。例如，可以配置 TIM1 作为 TIM2 的预分频器。

4. 捕获/比较通道

每个捕获/比较通道都围绕一个捕获/比较寄存器(包含影子寄存器)，包括捕获的输入部分(数字滤波、多路复用和预分频器)和输出部分(比较器和输出控制)。输入部分对相应的 TIx 输入信号采样，并产生一个滤波后的信号 TIxF。然后，一个带极性选择的边缘检测器产生一个信号(TIxFPx)，它可以作为从模式控制器的输入触发或作为捕获控制。该信号通过预分频器进入捕获寄存器(ICxPS)。输出部分产生一个中间波形 OCxRef(高有效)作为基准，链的末端决定最终输出信号的极性。

5.2.4 通用定时器的工作模式

1. 输入捕获模式

在输入捕获模式下，检测到 ICx 信号上相应的边沿后，计数器的当前值被锁存到捕获/比较寄存器(TIMx_CCRx)中。当捕获事件发生时，相应的 CCxIF 标志(TIMx_SR 寄存器)被置为 1，如果使能了中断或 DMA 操作，则将产生中断或 DMA 操作。如果捕获事件发生时 CCxIF 标志已经为高，那么重复捕获标志 CCxOF(TIMx_SR 寄存器)被置为 1。写 CCxIF＝0 可清除 CCxIF，或读取存储在 TIMx_CCRx 寄存器中的捕获数据也可清除 CCxIF；写 CCxOF＝0 可清除 CCxOF。

2. PWM 输入模式

PWM 输入模式是输入捕获模式的一个特例，除以下区别外，操作与输入捕获模式相同。

(1) 两个 ICx 信号被映射至同一个 TIx 输入。

(2) 这两个 ICx 信号为边沿有效,但是极性相反。

(3) 其中一个 TIxFP 信号被作为触发输入信号,而从模式控制器被配置成复位模式。例如,需要测量输入 TI1 的 PWM 信号的长度(TIMx_CCR1 寄存器)和占空比(TIMx_CCR2 寄存器),具体步骤如下(取决于 CK_INT 的频率和预分频器的值)。

① 选择 TIMx_CCR1 的有效输入:置 TIMx_CCMR1 寄存器的 CC1S=01(选择 TI1)。

② 选择 TI1FP1 的有效极性(捕获数据到 TIMx_CCR1 中,清除计数器):置 CC1P=0(上升沿有效)。

③ 选择 TIMx_CCR2 的有效输入:置 TIMx_CCMR1 寄存器的 CC2S=10(选择 14478)。

④ 选择 T11FP2 的有效极性(捕获数据到 TIMx_CCR2):置 CC2P=1(下降沿有效)。

⑤ 选择有效的触发输入信号:置 TIMx_SMCR 寄存器中的 TS=101(选择 TI1FP1)。

⑥ 配置从模式控制器为复位模式:置 TIMx_SMCR 中的 SMS=100。

⑦ 使能捕获:置 TIMx_CCER 寄存器中 CC1E=1 且 CC2E=1。

3. 强置输出模式

在输出模式(TIMx_CCMRx 寄存器中 CCxS=00)下,输出比较信号(OCxREF 和相应的 OCx)能够直接由软件强置为有效或无效状态,而不依赖于输出比较寄存器和计数器间的比较结果。置 TIMx_CCMRx 寄存器中相应的 OCxM=101,即可强置输出比较信号(OCxREF/OCx)为有效状态。这样 OCxREF 被强置为高电平(OCxREF 始终为高电平有效),同时 OCx 得到 CCxP 极性位相反的值。

例如,CCxP=0(OCx 高电平有效),则 OCx 被强置为高电平。置 TIMx_CCMRx 寄存器中的 OCxM=100,可强置 OCxREF 信号为低电平。该模式下,TIMx_CCRx 影子寄存器和计数器之间的比较仍然在进行,相应的标志也会被修改,因此仍然会产生相应的中断和 DMA 请求。

4. 输出比较模式

输出比较模式用于控制一个输出波形,或者指示一段给定的时间已经到时。

当计数器与捕获/比较寄存器的内容相同时,输出比较功能进行如下操作。

(1) 将输出比较模式(TIMx_CCMRx 寄存器中的 OCxM 位)和输出极性(TIMx_CCER 寄存器中的 CCxP 位)定义的值输出到对应的引脚上。在比较匹配时,输出引脚可以保持它的电平(OCxM=000)、被设置成有效电平(OCxM=001)、被设置成无效电平 OCxM=010)或进行翻转(OCxM=011)。

(2) 设置中断状态寄存器中的标志位(TIMx_SR 寄存器中的 CCxIF 位)。

(3) 若设置了相应的中断屏蔽(TIMx_DIER 寄存器中的 CCxIE 位),则产生一个中断。

(4) 若设置了相应的使能位(TIMx_DIER 寄存器中的 CCxDE 位,TIMx_CR2 寄存器中的 CCDS 位选择 DMA 请求功能),则产生一个 DMA 请求。

输出比较模式的配置步骤如下。

(1) 选择计数器时钟(内部、外部、预分频器)。

(2) 将相应的数据写入 TIMx_ARR 和 TIMx_CCRx 寄存器中。

(3) 如果要产生一个中断请求和/或一个 DMA 请求,设置 CCxIE 位和/或 CCxDE 位。

(4) 选择输出模式。例如,当计数器 CNT 与 CCRx 匹配时翻转 OCx 的输出引脚,CCRx 预装载未用,开启 OCx 输出且高电平有效,则必须设置 OCxM=011、OCxPE=0、CCxP=0 和 CCxE=1。

(5) 设置 TIMx_CR1 寄存器的 CEN 位启动计数器。

TIMx_CCRx 寄存器能够在任何时候通过软件进行更新以控制输出波形,条件是未使用预装载寄存器(OCxPE=0),否则 TIMx_CCRx 影子寄存器只能在发生下一次更新事件时被更新。

5. PWM 输出模式

PWM 输出模式是一种特殊的输出模式,在电力、电子和电机控制领域得到广泛应用。

1) PWM 简介

PWM 是 Pulse Width Modulation 的缩写,中文意思就是脉冲宽度调制,简称脉宽调制。它是利用微处理器的数字输出对模拟电路进行控制的一种非常有效的技术,因控制简单、灵活和动态响应好等优点而成为电力、电子技术中应用最广泛的控制方式。PWM 应用领域包括测量、通信、功率控制与变换、电动机控制、伺服控制、调光、开关电源,甚至某些音频放大器。因此,研究基于 PWM 技术的正负脉宽数控调制信号发生器具有十分重要的现实意义。PWM 是一种对模拟信号电平进行数字编码的方法,通过高分辨率计数器的使用,调制方波的占空比对一个具体模拟信号的电平进行编码。PWM 信号仍然是数字的,因为在给定的任何时刻,满幅值的直流供电要么完全有(ON),要么完全无(OFF),电压或电流源是以一种通(ON)或断(OFF)的重复脉冲序列被加载到模拟负载上的。通时即是直流供电被加到负载上,断时即是供电被断开。只要带宽足够,任何模拟值都可以使用 PWM 进行编码。

2) PWM 实现

目前在运动控制系统或电动机控制系统中实现 PWM 的方法主要有传统的数字电路、微控制器普通 I/O 模拟和微控制器的 PWM 直接输出等。

(1) 传统的数字电路方式:用传统的数字电路实现 PWM(如 555 定时器),电路设计较复杂,体积大,抗干扰能力差,系统的研发周期较长。

(2) 微控制器普通 I/O 模拟方式:对于微控制器中无 PWM 输出功能的情况(如 51 单片机),可以通过 CPU 操控普通 I/O 口实现 PWM 输出。但这样实现 PWM 将消耗大量的时间,大大降低 CPU 的效率,而且得到的 PWM 信号的精度不太高。

(3) 微控制器的 PWM 直接输出方式:对于具有 PWM 输出功能的微控制器,在进行简单的配置后即可在微控制器的指定引脚上输出 PWM 脉冲。这也是目前使用最多的 PWM 实现方式。

STM32F103 就是一款具有 PWM 输出功能的微控制器,除了基本定时器 TIM6 和

TIM7 外，其他的定时器都可以用来产生 PWM 输出。其中，高级控制定时器 TIM1 和 TIM8 可以同时产生多达 7 路的 PWM 输出；通用定时器也能同时产生多达 4 路的 PWM 输出；STM32 最多可以同时产生 30 路 PWM 输出。

3）PWM 输出模式的工作过程

STM32F103 微控制器脉冲宽度调制模式可以产生一个由 TIMx_ARR 寄存器确定频率、由 TIMx_CCRx 寄存器确定占空比的信号，其产生原理如图 5-8 所示。

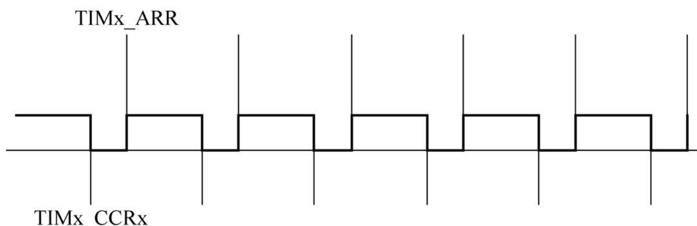

图 5-8 STM32F103 微控制器 PWM 产生原理

通用定时器 PWM 输出模式的工作过程如下。

（1）若配置脉冲计数器 TIMx_CNT 为向上计数模式，自动重装载寄存器 TIMx_ARR 的预设值为 N，则脉冲计数器 TIMx_CNT 的当前计数值 X 在时钟 CK_CNT 的驱动下从 0 开始不断累加计数。

（2）在脉冲计数器 TIMx_CNT 随着时钟 CK_CNT 触发进行累加计数的同时，脉冲计数 M_CNT 的当前计数值 X 与捕获/比较寄存器 TIMx_CCR 的预设值 A 进行比较。如果 $X<A$，输出高电平（或低电平）；如果 $X\geqslant A$，输出低电平（或高电平）。

（3）当脉冲计数器 TIMx_CNT 的计数值 X 大于自动重装载寄存器 TIMx_ARR 的预设值 N 时，脉冲计数器 TIMx_CNT 的计数值清零并重新开始计数。如此循环往复，得到的 PWM 输出信号周期为$(N+1)$TCK_CNT，其中 N 为自动重装载寄存器 TIMx_ARR 的预设值，TCK_CNT 为时钟 CK_CNT 的周期。PWM 输出信号脉冲宽度为 $A\times$TCK_CNT，其中 A 为捕获/比较寄存器 TIMx_CCR 的预设值，TCK_CNT 为时钟 CK_CNT 的周期。PWM 输出信号的占空比为 $A/(N+1)$。

下面举例具体说明，当通用定时器被设置为向上计数模式，自动重装载寄存器 TIMx_ARR 的预设值为 8，4 个捕获/比较寄存器 TIMx_CCRx 分别设为 0、4、8 和大于 8 时，通过用定时器的 4 个 PWM 通道的输出时序 OCxREF 和触发中断时序 CCxIF，如图 5-9 所示。例如，在 TIMx_CCR=4 的情况下，当 TIMx_CNT<4 时，OCxREF 输出高电平；当 TIMx_CNT≥4 时，OCxREF 输出低电平，并在比较结果改变时触发 CCxIF 中断标志。此 PWM 输出信号的占空比为 4/(8+1)。

需要注意的是，在 PWM 输出模式下，脉冲计数器 TIMx_CNT 的计数模式有向上计数、向下计数和向上/向下计数（中央对齐）3 种。以上仅介绍向上计数方式，读者在掌握了通用定时器向上计数模式的 PWM 输出原理后，其他两种计数模式的 PWM 输出也就容易推出了。

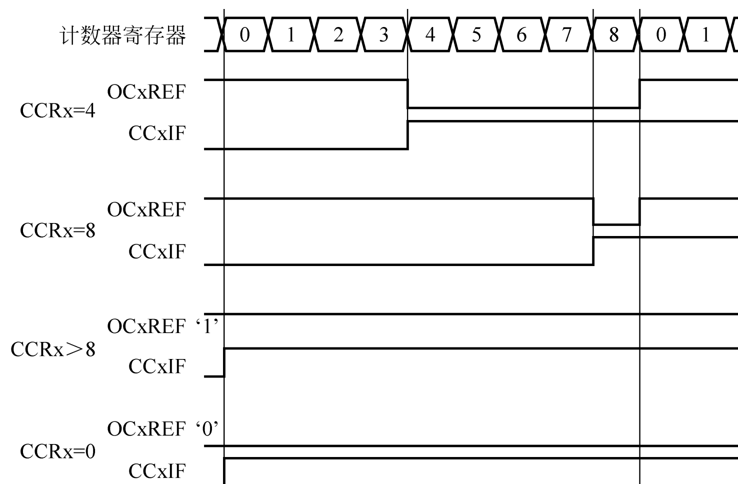

图 5-9 向上计数模式 PWM 输出时序图

5.3 STM32 PWM 输出应用实例

本节实现通过配置 STM32 的重映射功能，把定时器 TIM3 通道 2 重映射到引脚 PB5 上，由 TIM3_CH2 输出 PWM 控制 DS0 的亮度。下面介绍通过库函数配置该功能的步骤。

PWM 相关的函数设置在库函数文件 stm32f10x_tim.h 和 stm32f10x_tim.c 中。

(1) 开启 TIM3 时钟以及复用功能时钟，配置引脚 PB5 为复用输出。

要使用 TIM3，必须先开启 TIM3 的时钟。这里还要配置引脚 PB5 为复用输出，这是因为 TIM3 通道 2 将重映射到引脚 PB5 上，此时引脚 PB5 属于复用功能输出。库函数使能 TIM3 时钟的方法是

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3,ENABLE);        //使能 TIM3 时钟
```

库函数设置 AFIO 时钟的方法是

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO,ENABLE);        //复用时钟使能
```

这里简单列出 GPIO 初始化的一行代码：

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;             //复用推挽输出
```

(2) 设置 TIM3 通道 2 TIM3_CH2 重映射到引脚 PB5 上。

因为 TIM3_CH2 默认是接在引脚 PA7 上的，所以需要设置 TIM3_REMAP 为部分重映射(通过 AFIO_MAPR 配置)，让 TIM3_CH2 重映射到引脚 PB5 上。设置重映射的库函数是

```
void GPIO_PinRemapConfig(uint32_t GPIO_Remap,FunctionalState NewState);
```

STM32 只能重映射到特定的端口。第 1 个入口参数可以理解为设置重映射的类型，如

TIM3 部分重映射入口参数为 GPIO_PartialRemap_TIM3。所以，TIM3 部分重映射的库函数实现方法是

```
GPIO_PinRemapConf ig(GPIO_PartialRemap_TIM3,ENABLE);
```

(3) 初始化 TIM3，设置 TIM3 的 ARR 和 PSC。

开启了 TIM3 的时钟之后，就要设置 ARR 和 PSC 两个寄存器的值用于控制输出 PWM 的周期。当 PWM 周期太慢(低于 50Hz)时，我们就会明显感觉到闪烁了。

因此，PWM 周期在这里不宜设置得太小，通过 TIM_TimeBaseInit()函数实现，调用格式为

```
TIM_TimeBaseStructure.TIM_Period = arr;           //设置自动重装载值
TIM_TimeBaseStructure.TIM_Prescaler = psc;        //设置预分频值
TIM_TimeBaseStructure.TIM_ClockDivision = 0;      //设置时钟分割,TDTS = Tck_tim
TIM_TimeBaseStructure. TIM_CounterMode = TIM_CounterMode_Up;         //向上计数模式
TIM_TimeBaseInit(TIM3,&TIM_TimeBaseStructure);   //根据指定的参数初始化 TIMx
```

(4) 设置 TIM3_CH2 的 PWM 模式，使能 TIM3 的通道 2 输出。

接下来要设置 TIM3_CH2 为 PWM 模式(默认是冻结的)，因为 DS0 是低电平亮，而我们希望当 CCR2 值小时 DS0 就暗，CCR2 值大时 DS0 就亮，所以要通过配置 TIM3_CCMR1 的相关位控制 TIM3_CH2 的模式。PWM 通道是通过库函数 TIM_OC1Init～TIM_OC4Init 设置的，不同通道的设置函数不一样，这里使用的是通道 2，所以使用的函数是 TIM_OC2Init。

```
void TIM_OC2Init(TIM_TypeDef * TIMx, TIM_OCInitTypeDef * TIM_oCInitStruct);
```

TIM_OCInitTypeDef 结构体的定义如下。

```
typedef struct
{
    uint16_t TIM_OCMode;
    uint16_t TIM_OutputState;
    uint16_t TIM_OutputNState;
    uint16_t TIM_Pulse;
    uint16_t TIM_OCPolarity;
    uint16_t TIM_OCNPolarity;
    uint16_t TIM_OCIdleState;
    uint16_t TIM_OCNIdleState;
}TIM_OCInitTypeDef;
```

相关的几个成员变量介绍如下。

参数 TIM_OCMode 设置是 PWM 模式还是输出比较模式，这里是 PWM 模式。

参数 TIM OutputState 设置比较输出使能，也就是使能 PWM 输出到端口。

参数 TIM_OCPolarity 设置极性是高还是低。

其他参数(TIM_OutputNState、TIM_OCNPolarity、TIM_OCIdleState 和 TIM_OCIdleState)是高级控制定时器 TIM1 和 TIM8 才用到的。

要实现上面提到的场景，方法如下。

```
TIM_OCInitTypeDef TIM_OCInitStructure;
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2;            //选择 PWM 模式 2
TIM_OCInitStructure.OutputState = TIM_OutputState_Enable;     //比较输出使能
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;     //输出极性高
TIM_OC2Init(TIM3,&TIM_OCInitStructure);                       //初始化 TIM3 OC2
```

(5) 在完成以上设置之后,需要使能 TIM3。

```
TIM_Cmd(TIM3,ENABLE);                     //使能 TIM3
```

(6) 修改 TIM3_CCR2 控制占空比。

经过以上设置之后,PWM 其实已经开始输出了,只是其占空比和频率都是固定的,通过修改 TIM3_CCR2 可以控制 CH2 的输出占空比,继而控制 DSO 的亮度。

修改 TIM3_CCR2 占空比的库函数是

```
void TIM_SetCompare2(TIM_TypeDef * TIMx,uint16_t Compare2);
```

当然,其他通道分别有一个函数名称,为 TIM_SetCompare$x$(x=1,2,3,4)。

通过以上 6 个步骤,我们就可以控制 TIM3 的通道 2 输出 PWM 波形了。

5.3.1 PWM 输出硬件设计

本实例用到的硬件资源有指示灯 DS0 和定时器 TIM3。这里用到了 TIM3 的部分重映射功能,把 TIM3_CH2 直接映射到了引脚 PB5 上,引脚 PB5 和 DS0 是直接连接的。

5.3.2 PWM 输出软件设计

1. time.h 头文件

```
#ifndef __TIMER_H
#define __TIMER_H
#include "sys.h"
void TIM3_PWM_Init(u16 arr,u16 psc);
#endif
```

2. time.c 函数

```
#include "timer.h"
#include "led.h"
#include "usart.h"
//TIM3 PWM 部分初始化
//PWM 输出初始化
//arr:自动重装值
//psc:时钟预分频数
void TIM3_PWM_Init(u16 arr,u16 psc)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_OCInitTypeDef TIM_OCInitStructure;
```

```
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);  //使能 TIM3 时钟
    //使能 GPIO 外设和 AFIO 复用功能模块时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB | RCC_APB2Periph_AFIO, ENABLE);

    GPIO_PinRemapConfig(GPIO_PartialRemap_TIM3, ENABLE);      //TIM3 部分重映射 TIM3_CH2->PB5

    //设置该引脚为复用输出功能,输出 TIM3_CH2 的 PWM 脉冲波形
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;                 //TIM3_CH2
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;           //复用推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);                    //初始化 GPIO

    //初始化 TIM3
    TIM_TimeBaseStructure.TIM_Period = arr;        //设置在下一个更新事件装入活动的自动
                                                   //重装载寄存器周期的值
    TIM_TimeBaseStructure.TIM_Prescaler = psc;     //作为 TIMx 时钟频率除数的预分频值
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;   //设置时钟分割,TDTS = Tck_tim
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;    //TIM 向上计数模式
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);   //根据 TIM_TimeBaseInitStruct 中指定的
                                                   //参数初始化 TIMx 的时间基数单位

    //初始化 TIM3_CH2 的 PWM 模式
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2;          //选择定时器模式
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;   //比较输出使能
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;  //输出极性,TIM 输出比较极性高
    TIM_OC2Init(TIM3, &TIM_OCInitStructure);          //根据指定的参数初始化外设 TIM3 OC2

    TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Enable);   //使能 TIM3 在 CCR2 上的预装载寄存器

    TIM_Cmd(TIM3, ENABLE);                            //使能 TIM3
}
```

3. main.c 函数

```
#include "led.h"
#include "delay.h"
#include "key.h"
#include "sys.h"
#include "usart.h"
#include "timer.h"
int main(void)
{
    u16 led0pwmval = 0;
    u8 dir = 1;
    delay_init();             //延时函数初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);     //设置 NVIC 中断分组 2:两位抢占式优
                                                        //先级,两位响应优先级
    uart_init(115200);        //串口初始化为 115200
    LED_Init();               //LED 端口初始化
    TIM3_PWM_Init(899,0);     //不分频,PWM 频率 = 72000000/900 = 80kHz
```

```
        while(1)
        {
            delay_ms(10);
            if(dir) led0pwmval++;
            else led0pwmval--;

            if(led0pwmval > 300) dir = 0;
            if(led0pwmval == 0) dir = 1;
            TIM_SetCompare2(TIM3,led0pwmval);
        }
    }
```

从死循环函数可以看出，将 led0pwmval 的值设置为 PWM 比较值，也就是通过 led0pwmval 控制 PWM 的占空比，然后控制 led0pwmval 的值从 0 变到 300，再从 300 变到 0，如此循环。因此，DS0 的亮度也会跟着从暗变到亮，然后又从亮变到暗。这里取 300 是因为 PWM 的输出占空比达到这个值时，LED 亮度变化就不大了（虽然最大值可以设置到 899），因此设计过大的值是没有必要的。至此，软件设计就完成了。

在完成软件设计之后，将编译好的文件下载到战舰 STM32 开发板上，观看其运行结果是否与我们编写的一致。如果没有错误，则看到 DS0 不停地由暗变到亮，然后又从亮变到暗。每个过程持续时间大概为 3s。

PWM 输出的项目工程可参照本书数字资源中的程序代码。

5.4 看门狗定时器

5.4.1 看门狗应用介绍

微控制器系统的工作常常会受到来自外界的干扰（如电磁场），有时会出现程序跑飞的现象，甚至让整个系统陷入死循环。当出现这种现象时，微控制器系统中的看门狗模块或微控制器系统外的看门狗芯片就会强制对整个系统进行复位，使程序恢复到正常运行状态。看门狗实际上是一个定时器，因此也称为看门狗定时器，一般有一个输入操作，叫作“喂狗”。微控制器正常工作时，每隔一段时间输出一个信号到“喂狗端”，给看门狗定时器清零，如果超过规定的时间不喂狗（一般在程序跑飞时），看门狗定时器就会超时溢出，强制对微控制器进行复位，这样就可以防止微控制器死机。看门狗定时器的作用就是防止程序发生死循环，或者说在程序跑飞时能够进行复位操作。STM32 微控制器系统自带了两个看门狗，分别是独立看门狗（IWDG）和窗口看门狗（WWDG）。

STM32F10xxx 内置的两个看门狗提供了更高的安全性、时间的精确性和使用的灵活性。两个看门狗可用来检测和解决由软件错误引起的故障；当计数器达到给定的超时值时，触发一个中断（仅适用于窗口看门狗）或产生系统复位。

独立看门狗（IWDG）由专用的低速时钟（LSI）驱动，即使主时钟发生故障，它也仍然有效。

窗口看门狗(WWDG)由从 APB1 时钟分频后得到的时钟驱动,通过可配置的时间窗口检测应用程序非正常的过迟或过早的操作。

IWDG 适用于那些需要看门狗作为一个在主程序之外能够完全独立工作,并且对时间精度要求较低的场合。

WWDG 适用于那些要求看门狗在精确计时窗口起作用的应用程序。

5.4.2　独立看门狗

独立看门狗(IWDG)主要性能如下。

(1) 自由运行的递减计数器。

(2) 时钟由独立的 RC 振荡器提供(可在停止和待机模式下工作)。

(3) 看门狗被激活后,则在计数器计数至 0x000 时产生复位。

独立看门狗模块如图 5-10 所示。

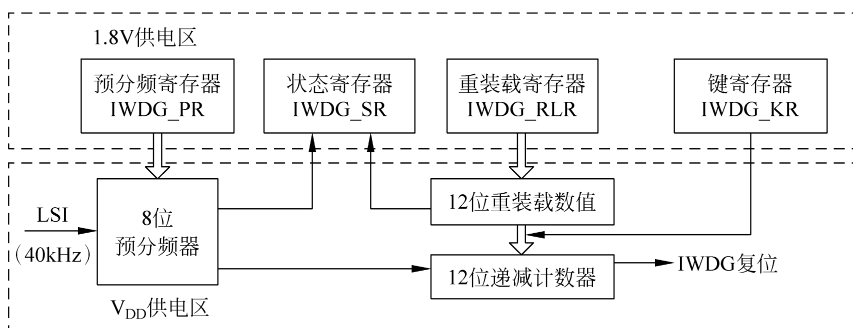

图 5-10　独立看门狗模块

1. 独立看门狗时钟

独立看门狗由专用的低速时钟(LSI 时钟)驱动,即使主时钟发生故障,它也仍然有效。LSI 时钟的标称频率为 40kHz,但是由于 LSI 时钟由内部 RC 电路产生,因此 LSI 时钟的频率约为 30～60kHz,所以 STM32 内部独立看门狗只适用于对时间精度要求比较低的场合,如果系统对时间精度要求高,建议使用外置独立看门狗芯片。

2. 独立看门狗预分频器

预分频器对 LSI 时钟进行分频之后,作为 12 位递减计数器的时钟输入。预分频系数由预分频寄存器(IWDG_PR)的 PR 决定,预分频系数可以取值 0、1、2、3、4、5、6 和 7,对应的预分频值分别为 4、8、16、32、64、128、256 和 512。

3. 12 位递减计数器

12 位递减计数器对预分频器的输出时钟进行计数,从复位值递减计算,当计数到 0 时,会产生一个复位信号。下面通过一个具体的例子对计数器的工作过程进行讲解。假如写入 IWDG_RLR 的值为 624,启动独立看门狗,即向键寄存器(IWDG_KR)中写入 0xCCCC,则计数器从复位值 624 开始递减计数,当计数到 0 时会产生一个复位信号。因此,为了避免产

生看门狗复位，即避免计数器递减计数到0，就需要向 IWDG_KR 的 KEY[15:0]写入0xAAAA，则 IWDG_RLR 的值会被加载到12位递减计数器，计数器就又从复位值624开始递减计数。

4. 状态寄存器

独立看门狗的状态寄存器(IWDG_SR)有两个状态位，分别是独立看门狗计数器重装载值更新状态位 RVU 和独立看门狗预分频值更新状态位 PVU。RVU 由硬件置为1，用来指示重装载值的更新正在进行中，当 V_{DD} 域中的重装载更新结束后，该位由硬件清零(最多需要5个40kHz 的时钟周期)，重装载值只有在 RVU 被清零后才可以更新。

5. 键寄存器

IWDG_PR 和 IWDG_RLR 都具有写保护功能，要修改这两个寄存器的值，必须先向 IWDG_KR 的 KEY[15:0]写入0x5555。以不同的值写入 KEY[15:0]将会打乱操作顺序，寄存器将会重新被保护。

除了可以向 KEY[15:0]写入0x5555允许访问 IWDG_PR 和 IWDG_RLR，也可以向 KEY[15:0]写入0xAAAA 使计数器从复位值开始重新递减计数；还可以向 KEY[15:0]写入0xCCCC，启动独立看门狗工作。

5.4.3 窗口看门狗

窗口看门狗(WWDG)通常被用来监测由外部干扰或不可预见的逻辑条件造成的应用程序背离正常的运行序列而产生的软件故障。除非递减计数器的值在 T6 位变为0前被刷新，看门狗电路在达到预置的时间周期时，会产生一个 MCU 复位。在递减计数器达到窗口寄存器数值之前，如果7位的递减计数器数值(在控制寄存器中)被刷新，那么也将产生一个 MCU 复位。这表明递减计数器需要在一个有限的时间窗口中被刷新。

1. WWDG 主要特性

WWDG 主要特性如下。

(1) 可编程的自由运行递减计数器。

(2) 条件复位：当递减计数器的值小于0x40时，则产生复位(若看门狗被启动)；当递减计数器在窗口外被重新装载时，则产生复位(若看门狗被启动)。

(3) 如果启动了看门狗并且允许中断，当递减计数器等于0x40时产生早期唤醒中断(EWI)，它可以被用于重装载计数器以避免 WWDG 复位。

2. WWDG 功能描述

如果看门狗被启动(WWDG_CR 寄存器中的 WDGA 位被置1)，并且当7位(T[6:0])递减计数器从0x40翻转到0x3F(T6 位清零)时，则产生一个复位。如果软件在计数器值大于窗口寄存器中的数值时重新装载计数器，将产生一个复位。窗口看门狗模块如图5-11所示。

应用程序在正常运行过程中必须定期地写入 WWDG_CR 寄存器以防止 MCU 发生复位。只有当计数器值小于窗口寄存器的值时，才能进行写操作。存储在 WWDG_CR 寄存

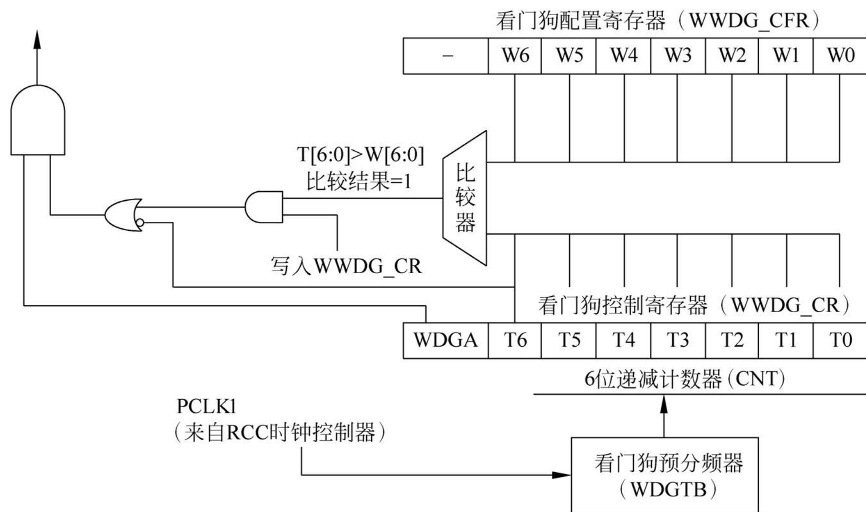

图 5-11　窗口看门狗模块

器中的数值必须在 0xC0 与 0xFF 之间。

(1) 启动看门狗。在系统复位后，看门狗总是处于关闭状态，设置 WWDG_CR 寄存器的 WDGA 位能够开启看门狗，随后它不能再被关闭，除非发生复位。

(2) 控制递减计数器处于自由运行状态，即使看门狗被禁止，递减计数器仍继续递减计数。当看门狗被启用时，T6 位必须被设置，以防止立即产生一个复位。

T[5:0]位包含了看门狗产生复位之前的计时数目；复位前的延时在一个最小值和一个最大值之间变化，这是因为写入 WWDG_CR 寄存器时，预分频值是未知的。

配置寄存器(WWDG_CFR)中包含窗口的上限值，要避免产生复位，递减计数器必须在其值小于窗口寄存器的数值并且大于 0x3F 时被重新装载。

另一个重装载计数器的方法是利用早期唤醒中断(EWI)。设置 WWDG_CFR 寄存器中的 EWI 位开启该中断。当递减计数器到达 0x40 时，则产生此中断，相应的中断服务程序(ISR)可以用来加载计数器以防止 WWDG 复位。在 WWDG_SR 寄存器中写 0 可以清除该中断。

注意：可以用 T6 位产生一个软件复位(设置 WDGA 位为 1，T6 位为 0)。

5.4.4　看门狗操作相关的库函数

1. 独立看门狗操作相关的库函数

```
void IWDG_Write AccessCmd(uintIWDG_WriteAccess);
```

功能描述：用默认参数初始化独立看门狗设置。

```
void IWDG_SetPrescal (wint8_t IWDG_Prescaler);
```

功能描述：设置独立看门狗的预置值。

```
void IWDG_SetReload(wint16_t Reload);
```

功能描述：设置 IWDG 的重新装载值。

```
void IWDG_ReloadCounter(void);
```

功能描述：重新装载设定的计数值。

```
void IWDG_Enable(void);
```

功能描述：使能 IWDG。

```
FlagStatus IWDG_GetFlagStatus(uint16_tIWDG_FLAG);
```

功能描述:检测独立看门狗电路的状态。

2. 窗口看门狗操作相关库函数

```
void WWDG_Delnit(void);
```

功能描述：用默认参数初始化窗口看门狗设置。

```
void WWDG_SetPrescaler(uint32_tWWDG_Prescaler);
```

功能描述：设置窗口看门狗的预置值。

```
void WWDG_Set Window Value(uint8_t WindowValue);
```

功能描述：设置窗口看门狗的值。

```
void WWDG_EnableIT(void);
```

功能描述：设置窗口看门狗的提前唤醒中断。

```
void WWDG_SetCounter(uint8_t Counter);
```

功能描述：设置窗口看门狗的计数值。

```
void WWDG_Enable(uint8_t Counter);
```

功能描述：使能窗口看门狗并装载计数值。

```
FlagStatus WWDG_GetFlagStatus(void);
```

功能描述：检测窗口看门狗提前唤醒中断的标志状态。

```
void WWDG_ClearFlag(void);
```

功能描述：清除 EWI 的中断标志。

5.4.5 独立看门狗程序设计

1. wdg.h 头文件

```
#ifndef __WDG_H
#define __WDG_H
#include "sys.h"
```

```
void IWDG_Init(u8 prer,u16 rlr);
void IWDG_Feed(void);

#endif
```

2. wdg.c 文件

```
#include "wdg.h"
//初始化独立看门狗
//prer:分频数,0~7(只有低 3 位有效)
//分频因子 = 4 * 2^prer,但最大值只能是 256
//rlr:重装载寄存器值,低 11 位有效
//时间计算(大概):Tout = ((4 * 2^prer) * rlr)/40ms
void IWDG_Init(u8 prer,u16 rlr)
{
    IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable); //使能对 IWDG_PR 和 IWDG_RLR 寄存器的写操作
    IWDG_SetPrescaler(prer);                      //设置 IWDG 预分频值
    IWDG_SetReload(rlr);                          //设置 IWDG 重装载值
    IWDG_ReloadCounter();                         //按照 IWDG 重装载的值重装载 IWDG 计数器

    IWDG_Enable();                                //使能 IWDG
}
//喂独立看门狗
void IWDG_Feed(void)
{
    IWDG_ReloadCounter();                         //重装载
}
```

代码中只有两个函数,IWDG_Init 是独立看门狗初始化函数,该函数有两个参数,分别用来设置预分频数与重装寄存器的值。通过这两个参数就可以大概知道看门狗复位的时间周期为多少了。IWDG_Feed 函数用来喂狗,因为 STM32 的喂狗只需要向键值寄存器写入 0xAAAA 即可,也就是调用 IWDG_ReloadCounter 函数,所以这个函数也很简单。

3. main.c 函数

接下来看看主函数的代码。在主程序中先初始化系统代码,然后启动按键输入和看门狗,在看门狗开启后马上点亮 LED0(DS0),并进入死循环等待按键的输入。一旦 WK_UP 有按键,则喂狗,否则等待 IWDG 复位的到来。

```
#include "led.h"
#include "delay.h"
#include "key.h"
#include "sys.h"
#include "usart.h"
#include "wdg.h"
int main(void)
{
    delay_init();                                    //延时函数初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);  //设置 NVIC 中断分组 2:两位抢占式优先级,
                                                     //两位响应优先级
```

```
 uart_init(115200);                          //串口初始化为 115200
 LED_Init();                                 //初始化与 LED 连接的硬件接口
 KEY_Init();                                 //按键初始化
 delay_ms(500);                              //视觉延时
 IWDG_Init(4,625);                           //预分频数为 4,重载值为 625,溢出时间为 1s
 LED0 = 0;                                   //点亮 LED0
 while(1)
 {
     if(KEY_Scan(0) == WKUP_PRES)
     {
      IWDG_Feed();                           //如果 WK_UP 按下,则喂狗
     }
     delay_ms(10);
 };
}
```

独立看门狗的项目工程请参照本书数字资源中的程序代码。

5.4.6 窗口看门狗程序设计

1. wdg.h 头文件

```
#ifndef __WDG_H
#define __WDG_H
#include "sys.h"
void IWDG_Init(u8 prer,u16 rlr);
void IWDG_Feed(void);

void WWDG_Init(u8 tr,u8 wr,u32 fprer);          //初始化 WWDG
void WWDG_Set_Counter(u8 cnt);                  //设置 WWDG 的计数器
void WWDG_NVIC_Init(void);
#endif
```

2. wdg.c 文件

```
#include "wdg.h"
#include "led.h"
//保存 WWDG 计数器的设置值,默认为最大
u8 WWDG_CNT = 0x7f;
//初始化窗口看门狗
//tr:T[6:0],计数器值
//wr:W[6:0],窗口值
//fprer:分频系数(WDGTB),仅最低两位有效
//Fwwdg = PCLK1/(4096 * 2^fprer)

void WWDG_Init(u8 tr,u8 wr,u32 fprer)
{
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE);   //WWDG 时钟使能
    WWDG_CNT = tr&WWDG_CNT;                                //初始化 WWDG_CNT
    WWDG_SetPrescaler(fprer);                              //设置 IWDG 预分频值
```

```
    WWDG_SetWindowValue(wr);                                   //设置窗口值
    WWDG_Enable(WWDG_CNT);                                     //使能看门狗 ,设置计数器
    WWDG_ClearFlag();                                          //清除提前唤醒中断标志位
    WWDG_NVIC_Init();                                          //初始化窗口看门狗 NVIC
    WWDG_EnableIT();                                           //开启窗口看门狗中断
}
//重设置 WWDG 计数器的值
void WWDG_Set_Counter(u8 cnt)
{
    WWDG_Enable(cnt);                                          //使能看门狗,设置计数器
}
//窗口看门狗中断服务程序
void WWDG_NVIC_Init()
{
    NVIC_InitTypeDef NVIC_InitStructure;
    NVIC_InitStructure.NVIC_IRQChannel = WWDG_IRQn;            //WWDG 中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;  //抢占式优先级 2
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;         //响应优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);                            //NVIC 初始化
}

void WWDG_IRQHandler(void)
{
    WWDG_SetCounter(WWDG_CNT);                                 //当注释此句后,窗口看门狗将
                                                               //产生复位
    WWDG_ClearFlag();                                          //清除提前唤醒中断标志位
    LED1 = !LED1;                                              //LED 状态翻转
}
```

新增的这 4 个函数都比较简单，WWDG_Init 函数用来设置 WWDG 的初始化值，包括看门狗计数器的值和看门狗比较值等。注意到这里有一个全局变量 WWDG_CNT，用来保存最初设置 WWDG_CR 计数器的值，在后续的中断服务函数里面，又把该数值放回到 WWDG_CR 上。

WWDG_Set_Counter 函数比较简单，用来重设窗口看门狗的计数器值，然后是中断分组函数。在中断服务函数中先重设窗口看门狗的计数器值，然后清除提前唤醒中断标志。最后对 LED1(DS1)取反，从而监测中断服务函数的执行状况。把这几个函数加入头文件，以方便其他文件调用。

3．main.c 函数

```
#include "led.h"
#include "delay.h"
#include "key.h"
#include "sys.h"
#include "usart.h"
#include "wdg.h"
int main(void)
```

```
{
    delay_init();                                   //延时函数初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);     //设置中断优先级分组为组 2:两位抢
                                                        //占式优先级,两位响应优先级
    uart_init(115200);                              //串口初始化为 115200
    LED_Init();
    KEY_Init();                                     //按键初始化
    LED0 = 0;
    delay_ms(300);
    WWDG_Init(0x7F,0x5F,WWDG_Prescaler_8);     //计数器值为 7F,窗口寄存器为 5F,分频数为 8
    while(1)
    {
    LED0 = 1;
    }
}
```

该函数通过 LED0(DS0)指示是否正在初始化,通过 LED1(DS1)指示是否发生了中断。先让 LED0 亮 300ms 然后关闭,从而判断是否有复位发生了。初始化 WWDG 之后回到死循环,关闭 LED1,并等待看门狗中断的触发/复位。

窗口看门狗的项目工程可参照本书数字资源中的程序代码。