TB3262

AVR1000b：编写用于 AVR® MCU 的 C 代码入门

简介

作者：Cristina Ionescu 和 Cristian Săbiuţă，Microchip Technology Inc.

本技术简介就如何成功编程 AVR®单片机（MCU）和如何定义代码编写指南提供了建议步骤，旨在帮助您编写更具可读性和可重用性的代码。

为了在缩短开发时间的同时满足高质量的要求，使用高级编程语言已成为必然选择。与特定于单片机架构的低级指令不同，高级编程语言具有更高的可移植性和可读性，因此有助于轻松维护和重用代码。

编程语言本身并不能确保高可读性和可重用性，但良好的编码风格可以。因此，AVR MCU 外设、头文件和驱动程序都是基于该假设而设计。

对于 AVR 单片机而言，使用最广泛的高级语言是 C 语言，因此本文档将重点介绍 C 语言编程。为了确保与大多数 AVR C 编译器都兼容，本文档中的代码示例采用 ANSI C 编码标准编写。

本文档包含使用 Atmel Studio 集成开发环境（Integrated Development Environment，IDE）开发的代码示例。大多数代码示例都兼容其他 IDE（见第 5 章：更多步骤）。

目录

1. 数据手册模块结构和命名约定

为单片机编写 C 代码的第一步是认识和了解器件数据手册中提供的哪类信息可用于编程。数据手册中包含单片机的特性、存储器、内核与外设模块，外设模块的功能说明，外设基址，寄存器的名称与地址以及其他功能和电气特性的相关信息。

1.1 如何查找数据手册

可在以下位置获取与 Microchip 产品相关的任何文档：

- Microchip 数据手册

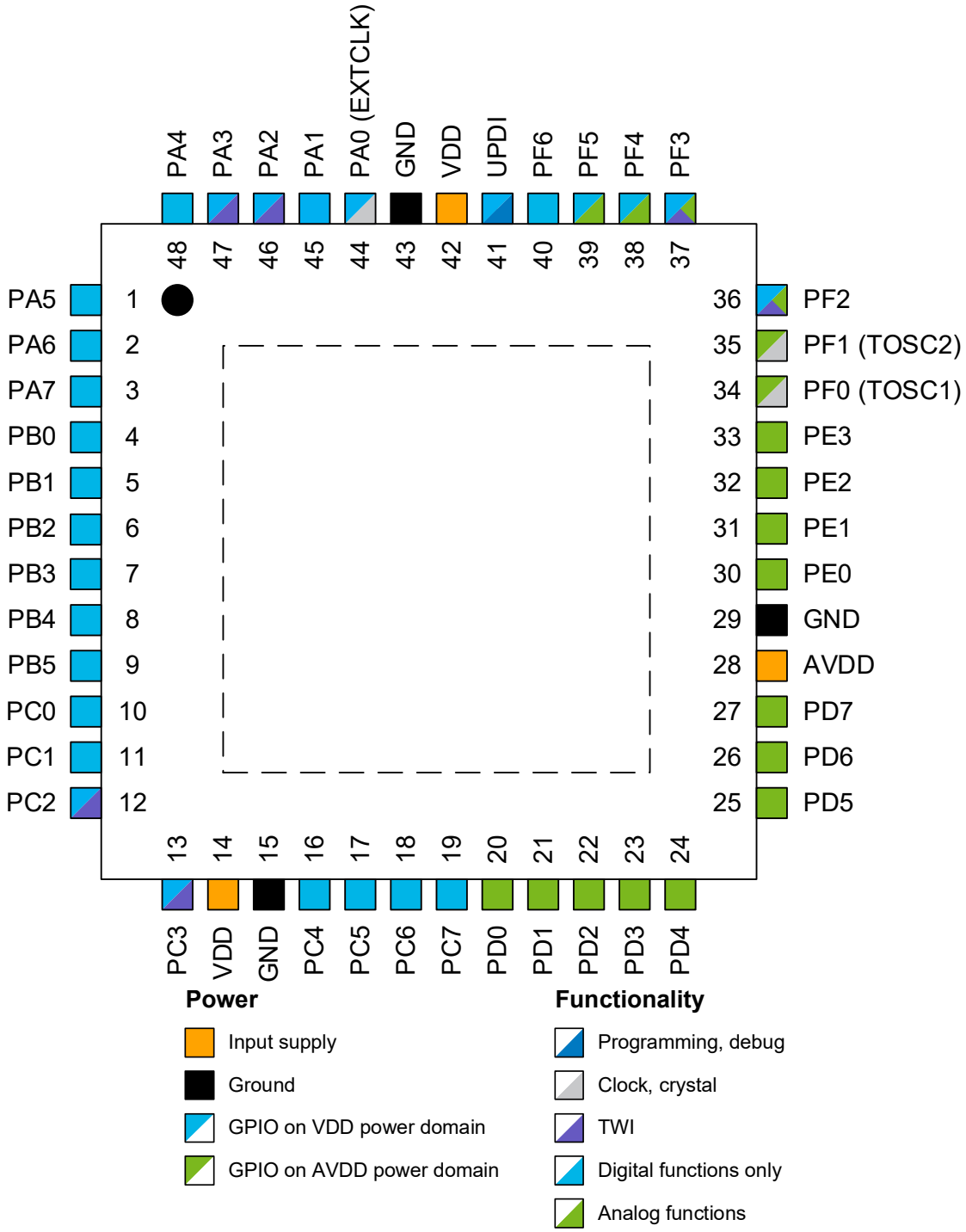
可在以下位置获取本文档中所述器件系列的器件数据手册：

- ATtiny416 概述
- ATtiny212 概述
- ATtiny3217 概述
- ATtiny1634 概述
- ATmega4809 概述
- ATmega808 概述
- AVR128DA28 概述

1.2 引脚说明

任何器件数据手册都会提供引脚说明。引脚分配位于*引脚分配*、*引脚配置*部分或任何其他名称约定中，具体取决于器件。ATmega809/1609/3209/4809 48 引脚器件的引脚分配如图 1-1 所示。

图 1-1. ATmega809/1609/3209/4809 48 引脚器件的引脚分配[(1)]



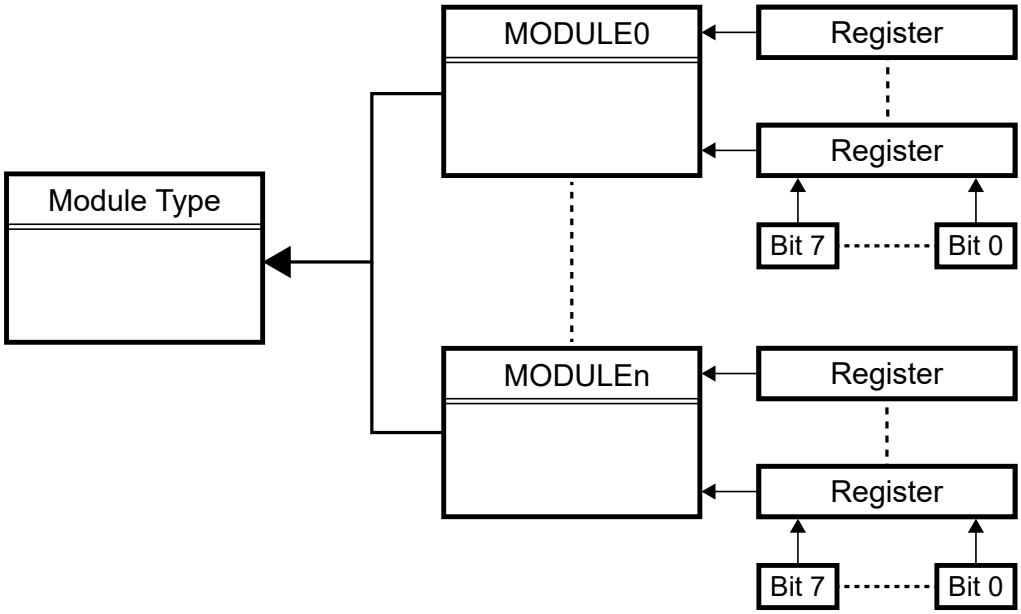
每个 I/O 引脚可配置的功能在 *I/O 复用和注意事项*章节或 *I/O 端口*章节的*备用端口功能*小节中进行说明，具体取决于器件。如果使用评估板（例如 AVR128DA48 Curiosity Nano），则用户需要了解如何在该评估板上分配单片机的引脚。相关信息可参见 AVR128DA48 Curiosity Nano 网页上提供的 AVR128DA48 Curiosity Nano 原理图文档。该网页上还提供有介绍 AVR128DA48 Curiosity Nano 板和单片机特性的其他文档。

1.3 模块说明

AVR 单片机由多个构件组成：AVR CPU、SRAM、闪存、EEPROM 和几个被称为模块类型的外设模块。在本文档中，所有外设模块统称为模块。

在新款 AVR 单片机系列中，一个给定模块类型可以有一个或多个实例。所有实例都具有相同的特性和功能。其中一些模块类型是其他模块类型的子集，会继承后者的部分特性。继承的特性与父项模块类型完全兼容。例如，与完整的定时器模块相比，定时器的子集模块会具有较少的比较和捕捉通道。

图 1-2. 模块类型、实例、寄存器和位



假设模块类型是通用同步异步收发器（Universal Synchronous Asynchronous Receiver-Transmitter，USART），则对应的模块实例可以是 USART0 等，其中后缀 0 表示实例为“USART 编号 0”。为简单起见，除非需要特别区分，否则本文档中都将使用模块来代表模块实例。

每个模块在 I/O 存储器映射中都具有固定的基址，并且模块中包含的所有寄存器都具有固定的偏移地址（相对于模块基址）。这样，每个寄存器不仅在 I/O 存储空间中有一个绝对地址，而且还有一个由其偏移量定义的相对地址。同一模块类型的所有实例的寄存器偏移地址均相等，这样可以轻松为该类型的所有模块编写通用的驱动程序。外设模块地址映射会给出每个外设的基址，具体可参见数据手册。

每个模块都有几个包含控制或状态位的寄存器。给定类型的所有模块使用同一组寄存器（或其子集），而且所有这些寄存器都包含同一组控制位和状态位（或其子集）。

表 1-1 列出了一些 ATtiny804/1604 外设的基址。

表 1-1. 外设模块地址映射（章节）(1)

基址	名称	说明
0x0000	VPORTA	虚拟端口 A
0x0004	VPORTB	虚拟端口 B
0x001C	GPIO	通用 I/O 寄存器
0x0030	CPU	CPU
0x0040	RSTCTRL	复位控制器
0x0050	SLPCTRL	休眠控制器

.....（续）		
基址	名称	说明
0x0060	CLKCTRL	时钟控制器
...	...	...

每个模块都有一个专门的章节介绍模块具有的特性、模块的功能说明以及具体的信号和关于如何配置特定工作模式的方法指南，并解释所有术语。在模块章节的末尾有一个寄存器说明小节，其中包含每个寄存器的范围、初始值以及可读/写性。此外，还提供寄存器中每个可配置/可访问位的位置。

每个模块的*寄存器汇总*章节都会介绍所有寄存器及其地址偏移量以及位名称和位置。图 1-3 给出了 ADC 模块的寄存器汇总。

图 1-3. ADC 寄存器汇总(1)

Offset	Name	Bit Pos.								
0x00	CTRLA	7:0	RUNSTBY					RESSEL	FREERUN	ENABLE
0x01	CTRLB	7:0						SAMPNUM[2:0]		
0x02	CTRLC	7:0		SAMPCAP	REFSEL[1:0]			PRESC[2:0]		
0x03	CTRLD	7:0	INITDLY[2:0]			ASDV	SAMPDLY[3:0]			
0x04	CTRLE	7:0						WINCM[2:0]		
0x05	SAMPCTRL	7:0				SAMPLEN[4:0]				
0x06	MUXPOS	7:0				MUXPOS[4:0]				
0x07	保留									
0x08	COMMAND	7:0								STCONV
0x09	EVCTRL	7:0								STARTEI
0x0A	INTCTRL	7:0							WCOMP	RESRDY
0x0B	INTFLAGS	7:0							WCOMP	RESRDY
0x0C	DBGCTRL	7:0								DBGRUN
0x0D	TEMP	7:0	TEMP[7:0]							
0x0E ... 0x0F	保留									
0x10	RES	7:0	RES[7:0]							
		15:8	RES[15:8]							
0x12	WINLT	7:0	WINLT[7:0]							
		15:8	WINLT[15:8]							
0x14	WINHT	7:0	WINHT[7:0]							
		15:8	WINHT[15:8]							
0x16	CALIB	7:0								DUTYCYC

1.4 命名约定

本节介绍器件数据手册和用于开发应用程序的头文件中所提供的寄存器和位命名约定。

1.4.1 寄存器命名约定

寄存器分为控制寄存器、状态寄存器和数据寄存器，具体可通过寄存器的名称来区分。例如，模块的通用控制寄存器被命名为 CTRL。如果同一模块中存在多个通用控制寄存器，则可通过添加后缀字符进行区分。因此，这些控制寄存器将分别被命名为 CTRLA、CTRLB、CTRLC，依此类推。这对于状态寄存器同样适用。

对于具有特定功能的寄存器，其名称会反映具体的功能。例如，控制模块中断的控制寄存器被命名为 INTCTRL。

由于本文档中所述单片机的数据总线宽度为 8 位，因此较大的寄存器是通过组合使用多个 8 位寄存器来实现。对于 16 位寄存器，可通过在寄存器名称末尾附加 H 和 L 来分别访问高字节和低字节。例如，16 位模数结果寄存器被命名为 RES。其低字节和高字节分别被命名为 RESL（RES-Low，寄存器的最低有效字节）和 RESH（RES-High，寄存器的最高有效字节）。另一种标识多个同名寄存器的方法是添加编号后缀；例如，对于 AVR DA 系列，NVMCTRL 中的 ADDR 寄存器是一个 24 位寄存器。因此，可以通过在寄存器名称末尾附加编号后缀（ADDR0、ADDR1 和 ADDR2）来分别访问其中的三个字节。

大多数 C 编译器都可以自动处理对多字节寄存器的访问。因此，使用名称 RES（不带 H 或 L 后缀）即可对 ADC 结果寄存器执行 16 位访问。这对于 32 位寄存器同样适用。

此外，包含 SET/CLR 后缀的寄存器允许用户在不执行读-修改-写操作的情况下将寄存器中的位置 1 和清零，因为这是在硬件中实现。这类寄存器都是成对出现。向 CLR 寄存器中的某位写入逻辑 1 会同时将两个寄存器中的相应位清零，而向 SET 寄存器中的某位写入逻辑 1 则会同时将两个寄存器中的相应位置 1。例如，在 PORT 模块中，向数据方向置 1（DIRSET）寄存器中的位写入逻辑 1 会同时将数据方向（DIR）寄存器和数据方向清零（DIRCLR）寄存器中的相应位清零。读取时两个寄存器将返回相同的值。如果同时写入两个寄存器，则将优先写入 CLR 寄存器。

1.4.2 位和位域命名约定

以下命名约定大多以模数转换器（Analog-to-Digital Converter，ADC）模块为例。

寄存器中的每个位既可以单独用于一项功能，也可以与其他位组合成位域用于一项功能。单个位可以是模块使能位，例如，CTRLA 寄存器中 ADC 模块的 ENABLE 位。位域可以由两个或多个位组成，这些位共同选择其所属模块的具体配置。一个位域最多可提供 2^n 种选择，其中 *n* 为位域中的位数。

ENABLE 位的位置如图 1-4 所示，PRESC 位域的位置如图 1-5 所示。

1.4.2.1 位命名约定

寄存器框图中的位名称是完整位名称的缩写，完整位名称描述位配置的功能。例如，RUNSTBY 位允许用户使能外设在待机休眠模式下运行。用户可以通过配置 ENABLE 位来使能 ADC。

图 1-4. ADC 控制 A 寄存器的位命名约定(1)

Name: CTRLA
Offset: 0x00
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	RUNSTBY					RESSEL	FREERUN	ENABLE
Access	R/W					R/W	R/W	R/W
Reset	0					0	0	0

1.4.2.2 位域命名约定

同一位域中的各个位可通过附加数字后缀（即，各个位在位域中的位置编号）来引用。例如，ADC 控制 C 预分频比位域的最高有效位（Most Significant bit，MSb）将为 PRESC2。数据手册中并未详细说明该命名约定，但本文档的后面部分将对其进行介绍，并在头文件中使用该约定来处理寄存器位域中的各个位。

图 1-5. 控制 C 寄存器中的 PRESC 配置位域(1)

Name: CTRLC
Offset: 0x02
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
		SAMPCAP	REFSEL[1:0]			PRESC[2:0]		
Access	R	R/W	R/W	R/W	R	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

图 1-6 列出了 PRESC 位域的各个值及其对应的选择。

图 1-6. ADC 时钟信号频率的预分频比配置(1)

Bits 2:0 – PRESC[2:0] Prescaler
These bits define the division factor from the peripheral clock (CLK_PER) to the ADC clock (CLK_ADC).

Value	Name	Description
0x0	DIV2	CLK_PER divided by 2
0x1	DIV4	CLK_PER divided by 4
0x2	DIV8	CLK_PER divided by 8
0x3	DIV16	CLK_PER divided by 16
0x4	DIV32	CLK_PER divided by 32
0x5	DIV64	CLK_PER divided by 64
0x6	DIV128	CLK_PER divided by 128
0x7	DIV256	CLK_PER divided by 256

1.5 配置更改保护（CCP）寄存器

CCP 寄存器用于保护系统关键 I/O 寄存器设置不受意外修改的影响，以及保护闪存自编程不受意外执行的影响。只有将指定签名/密钥写入 CCP 寄存器（CPU 模块操作的一部分）后，才能写入具有 CCP 的寄存器。签名值可参见器件数据手册的*配置更改保护*小节。

可对受保护的寄存器执行两种类型的写操作，每种类型都有各自的密钥：

- 受保护的 I/O 寄存器（密钥/签名为 IOREG）
- 受保护的自编程（密钥/签名为 SPM）

下表列出了一些受配置更改保护的寄存器。

寄存器名称		密钥	功能
CLKCTRL.MCLKCTRLA	时钟控制器——主时钟控制 A	IOREG	允许用户选择主时钟的时钟源并输出时钟
CLKCTRL.MCLKLOCK	时钟控制器——主时钟锁定	IOREG	允许用户锁定主时钟控制寄存器
RSTCTRL.SWRR	复位控制器——软件复位使能	IOREG	允许用户对器件执行软件复位
CPUINT.CTRLA 中的 IVSEL	CPU 中断控制器——CTRLA 寄存器中的中断向量选择位	IOREG	允许用户将中断向量放置在闪存应用程序段的开头或闪存引导段的开头
NVMCTRL.CTRLA	非易失性存储器控制器——CTRLA 寄存器	SPM	允许用户发出以下命令之一： • 将页缓冲区写入存储器 • 擦除页 • 擦除和写入页 • 页缓冲区清除等

只有在 CPU 将上述签名之一写入 CCP 寄存器后，才能对受保护的 I/O 寄存器或位进行更改或执行受保护的指令。CCP（CPU.CCP）寄存器的说明中会列出这些签名。

3.5. 写入配置更改保护（CCP）寄存器提供了代码示例。

1.6 熔丝

熔丝是非易失性存储器的一部分，用于保存器件配置。熔丝在器件上电后即可使用，即使执行全片擦除也会保留熔丝值。熔丝可通过 CPU 或编程接口（如 UPDI）读取，但只能通过编程/调试接口进行编程或清零。存储在熔丝中的配置值会在启动序列结束时复制到各自的目标寄存器中，以便 CPU 可以使用这些熔丝值。

熔丝汇总表位于器件数据手册的*存储器→熔丝（FUSE）*小节。以下示例摘录自 ATmega4808/4809 数据手册。

图 1-7. FUSE 寄存器汇总

Offset	Name	Bit Pos.								
0x00	WDTCFG	7:0	WINDOW[3:0]				PERIOD[3:0]			
0x01	BODCFG	7:0	LVL[2:0]			SAMPFREQ	ACTIVE[1:0]		SLEEP[1:0]	
0x02	OSCCFG	7:0	OSCLOCK						FREQSEL[1:0]	
0x03 ... 0x04	Reserved									
0x05	SYSCFG0	7:0	CRCSRC[1:0]				RSTPINCFG			EESAVE
0x06	SYSCFG1	7:0						SUT[2:0]		
0x07	APPEND	7:0	APPEND[7:0]							
0x08	BOOTEND	7:0	BOOTEND[7:0]							
0x09	Reserved									
0x0A	LOCKBIT	7:0	LOCKBIT[7:0]							

3.6. 配置熔丝提供了代码示例。

2. 头文件中的模块表示

每个 AVR 器件都有一个专用的头文件。需要在项目设置中指定目标器件（无论使用哪种 IDE——MPLAB® X、Atmel Studio 或 IAR EWAVR），头文件将自动包含在所创建项目的主文件中。包含的头文件如以下代码所示。

```
#include <avr/io.h>
```

所有需要的寄存器和结构定义均可在头文件中找到。已在器件特定的头文件中定义的宏和结构定义可代替寄存器地址使用。

这对于包含相同模块且模块的头文件定义也相同的多个器件来说非常有用。

2.1 存储器中的模块位置

给定外设模块的所有寄存器位于一个连续的存储块中。属于不同模块的寄存器不会混放，因此可以将所有外设模块都组织在 C 结构中，其中实例宏的定义如以下代码所示。所有外设模块的定义均位于这些 AVR 器件可用的器件头文件中。模块的地址采用 ANSI C 指定，以便与大多数可用的 C 编译器兼容。

```
#define PORTMUX           (*(PORTMUX_t *) 0x0200) /* 端口多路开关 */
#define PORTA                (*(PORT_t *) 0x0400) /* I/O 端口 */
#define PORTB                (*(PORT_t *) 0x0420) /* I/O 端口 */
#define PORTC                (*(PORT_t *) 0x0440) /* I/O 端口 */
```

模块实例定义使用解引用指针，该指针指向存储器中的绝对地址，与模块实例基址一致。模块指针在头文件中进行定义；因此，无需在源代码中添加这些定义。

例如，PORTA 模块的基址为 0x0400。该模块及其所有寄存器和保留字节的可用存储空间范围为 0x0400 至 0x0420，换算成十进制即 32 个字节。因此，`PORT_t` 包含为其所有寄存器（或保留字节）分配的 32 个字节。

2.2 模块结构

2.2.1 示例——ADC

在头文件中定义 `ADC_t` 结构类型，如以下代码所示。其中包含模块的所有寄存器，按数据手册中指定的顺序排列，与它们在存储器中的组织顺序相同。

```
/* 模数转换器 */
typedef struct ADC_struct
{
    register8_t CTRLA;     /* 控制 A */
    register8_t CTRLB;     /* 控制 B */
    register8_t CTRLC;     /* 控制 C */
    register8_t CTRLD;     /* 控制 D */
    register8_t CTRLE;     /* 控制 E */
    register8_t SAMPCTRL;  /* 采样控制 */
    register8_t MUXPOS;    /* 多路开关同相输入 */
    register8_t reserved_1[1];
    register8_t COMMAND;   /* 命令 */
    register8_t EVCTRL;    /* 事件控制 */
    register8_t INTCTRL;   /* 中断控制 */
    register8_t INTFLAGS;  /* 中断标志 */
    register8_t DBGCTRL;   /* 调试控制 */
    register8_t TEMP;      /* 临时数据 */
    register8_t reserved_2[2];
    _WORDREGISTER(RES);    /* ADC 累加器结果 */
    _WORDREGISTER(WINLT);  /* 窗口比较器下限阈值 */
    _WORDREGISTER(WINHT);  /* 窗口比较器上限阈值 */
    register8_t CALIB;     /* 校准 */
    register8_t reserved_3[1];
} ADC_t;
```

然后，使用该结构类型在头文件中定义模块实例的宏，如以下代码所示。

```
#define ADC0    (*(ADC_t *) 0x0600) /* 模数转换器 */
```

因此，特定的模块寄存器，例如 CTRLA 寄存器，可寻址为 ADC0.CTRLA。

2.2.2 示例——PORT

在头文件中定义 PORT_t 结构类型，如以下代码所示。

```
/* I/O端口 */
typedef struct PORT_struct
{
    register8_t DIR;       /* 数据方向 */
    register8_t DIRSET;    /* 数据方向置1 */
    register8_t DIRCLR;    /* 数据方向清零 */
    register8_t DIRTGL;    /* 数据方向翻转 */
    register8_t OUT;       /* 输出值 */
    register8_t OUTSET;    /* 输出值置1 */
    register8_t OUTCLR;    /* 输出值清零 */
    register8_t OUTTGL;    /* 输出值翻转 */
    register8_t IN;        /* 输入值 */
    register8_t INTFLAGS;  /* 中断标志 */
    register8_t reserved_1[6];
    register8_t PIN0CTRL;  /* 引脚0控制 */
    register8_t PIN1CTRL;  /* 引脚1控制 */
    register8_t PIN2CTRL;  /* 引脚2控制 */
    register8_t PIN3CTRL;  /* 引脚3控制 */
    register8_t PIN4CTRL;  /* 引脚4控制 */
    register8_t PIN5CTRL;  /* 引脚5控制 */
    register8_t PIN6CTRL;  /* 引脚6控制 */
    register8_t PIN7CTRL;  /* 引脚7控制 */
    register8_t reserved_2[8];
} PORT_t;
```

2.2.3 模块结构中的多字节寄存器

一些寄存器会与其他寄存器组合使用来表示 16 位或 32 位值。例如，ATmega4809 器件中的 ADC 结果寄存器（RES）、窗口比较器下限阈值寄存器（WINLT）和窗口比较器上限阈值寄存器（WINTH）都是 16 位寄存器（使用 _WORDREGISTER 宏进行声明）。该宏（如下所示）与_DWORDREGISTER 宏组合使用来声明 32 位寄存器。这两个宏已在头文件中定义。

_WORDREGISTER 宏通过添加 L 或 H 后缀来扩展寄存器的名称，以此访问其低字节和高字节。_DWORDREGISTER 宏通过添加编号后缀来提供对多字节寄存器中所有字节的访问。_WORDREGISTER 和_DWORDREGISTER 定义如以下代码所示。

```
#ifdef _WORDREGISTER
#undef _WORDREGISTER
#endif
#define _WORDREGISTER(regname) \
    __extension__ union \
    { \
        register16_t regname; \
        struct \
        { \
            register8_t regname ## L; \
            register8_t regname ## H; \
        }; \
    }
#ifdef _DWORDREGISTER
#undef _DWORDREGISTER
#endif
#define _DWORDREGISTER(regname)  \
    __extension__ union \
    { \
        register32_t regname; \
        struct \
        { \
```

```
            register8_t regname ## 0; \
            register8_t regname ## 1; \
            register8_t regname ## 2; \
            register8_t regname ## 3; \
        }; \
    }
```

2.3 位掩码、位组掩码和组配置掩码

用户可以使用模块的结构配置来访问任何寄存器。有关 ADC 的完全配置步骤，可参见数据手册中该模块的具体说明部分。例如，有关初始化 ADC 的建议步骤位于 *ADC——模数转换器→功能说明→初始化*。寄存器位可使用在头文件中定义的位掩码或位位置掩码来操作。当预定义的掩码与单个位相关时，称为位掩码；当预定义的掩码与一组位（位域）相关时，称为位组掩码或组掩码。例如，可使用位掩码使能并配置 ADC0 模块，以启动转换周期。

2.3.1 位掩码和位位置掩码

置 1 和清零单个位时均使用位掩码。位组掩码主要用于同时清零位域中的多个位。例如，ADC0 的 CTRLD 寄存器的位域、位名称、位位置和位掩码如表 2-1 所示。

表 2-1. 位域、位名称、位位置和位掩码

位域	INITDLY[2:0]			-	SAMPDLY[3:0]			
位名称	INITDLY2	INITDLY1	INITDLY0	ASDV	SAMPDLY3	SAMPDLY2	SAMPDLY1	SAMPDLY0
位位置	7	6	5	4	3	2	1	0
位掩码	0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01

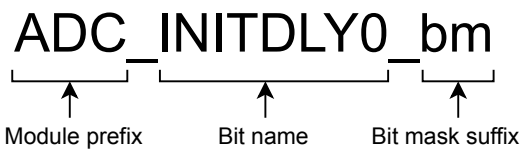
位名称必须是惟一的，这样编译器才能处理它们，因此所有位都以其所属的模块类型为前缀。在许多情况下，模块类型名称都是缩写形式。对于与 A 型定时器/计数器模块相关的所有位定义，位名称均以 TCA_为前缀。

为了区分位掩码和位位置，还附加了一个后缀。位掩码的后缀为_bm，位位置的后缀为_bp。因此，INITDLY0 位的位掩码名称为 ADC_INITDLY0_bm，位位置的名称为 ADC_INITDLY0_bp。此外，头文件还提供了组位置的定义。组位置的后缀为_gp，例如 INITDLY 组位置掩码的名称为 ADC_INITDLY_gp。以下代码给出了 INITDLY 位掩码、位位置和组位置的定义，这些定义均位于器件的头文件中。

```
#define ADC_INITDLY_gp  5        /* 初始延时选择组位置 */
#define ADC_INITDLY0_bm  (1<<5)  /* 初始延时选择 bit 0 掩码 */
#define ADC_INITDLY0_bp  5       /* 初始延时选择 bit 0 位置 */
#define ADC_INITDLY1_bm  (1<<6)  /* 初始延时选择 bit 1 掩码 */
#define ADC_INITDLY1_bp  6       /* 初始延时选择 bit 1 位置 */
#define ADC_INITDLY2_bm  (1<<7)  /* 初始延时选择 bit 2 掩码 */
#define ADC_INITDLY2_bp  7       /* 初始延时选择 bit 2 位置 */
```

图 2-1 给出了 INITDLY0 位掩码的命名约定示例。

图 2-1. 位掩码的命名约定



2.3.2 位域掩码（组掩码）

许多函数（例如定时器的时钟选择、转换器的预分频比选择或可配置定制逻辑的滤波器选择）都是由一个区域的位（称为位域或位组）来配置。这些位组合使用来选择特定配置。

用于配置位域的掩码称为位组掩码或组配置掩码。

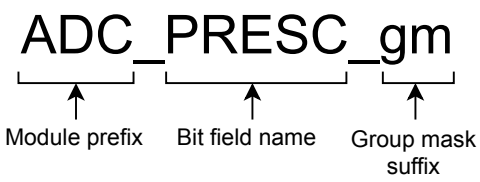
更改位域中的位（即，分配新值）时，不能直接根据所需配置对位进行设置，在此之前还需要先清除位的旧配置。为方便起见，定义了位组掩码。组掩码使用的名称与位域中的位基本相同，只是带有后缀_gm。

以下代码展示了如何在头文件中定义 ADC 预分频比组掩码。

```
#define ADC_PRESC_gm  0x07  /* 时钟预分频比组掩码 */
```

图 2-2 给出了命名约定。

图 2-2. 组掩码的命名约定



位组掩码主要用于在写入新值前清除位域的旧配置。以下代码展示了如何实现该操作。这段代码会将 ADC0 模块的 CTRLC 寄存器中的 PRESC 位域清零。此结构并非设置配置，而只是设置所有的预分频比配置位。这样的优势在于复位特定配置时无需使用所有的位掩码，只需使用组掩码即可。组掩码通常与组配置掩码组合使用来清除特定配置。

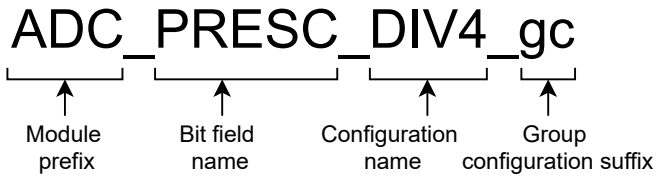
```
ADC0.CTRLC &= ~(ADC_PRESC_gm); /* 使用组掩码清零预分频比位域 */
```

2.3.3 组配置掩码和枚举器

在将位域设置为所需的配置时，通常需要查阅数据手册来确定要使用的位模式。读取或调试代码时也同样如此。为了提高可读性并最大程度地降低误将位域中的位置 1 的可能性，在头文件中定义了几个组配置掩码。每个组配置掩码为特定的组掩码选择配置。

组配置掩码的名称由模块类型、位域名称、配置说明和后缀_gc（指示组配置）组成。图 2-3 给出了 ADC 预分频比配置的示例。

图 2-3. 组配置掩码的命名约定



组配置（如图 2-3 所示）将外设时钟（CLK_PER）到 ADC 时钟的预分频比设置为分频系数 4。

ADC 预分频比位域包含 3 个位，用于定义分频系数。可能的配置名称包括 DIV2、DIV4、DIV8、DIV16、DIV32、DIV64、DIV128 和 DIV256。这些名称大大简化了代码的编写和维护，因为非常容易理解特定掩码所选的配置。 表 2-2 列出了该位域的可用配置。

表 2-2. PRESC 位及相应的位组配置

PRESC2	PRESC1	PRESC0	分频系数	组配置掩码
0	0	0	CLK_PER 进行 2 分频	ADC_PRESC_DIV2_gc
0	0	1	CLK_PER 进行 4 分频	ADC_PRESC_DIV4_gc
0	1	0	CLK_PER 进行 8 分频	ADC_PRESC_DIV8_gc
0	1	1	CLK_PER 进行 16 分频	ADC_PRESC_DIV16_gc
1	0	0	CLK_PER 进行 32 分频	ADC_PRESC_DIV32_gc
1	0	1	CLK_PER 进行 64 分频	ADC_PRESC_DIV64_gc
1	1	0	CLK_PER 进行 128 分频	ADC_PRESC_DIV128_gc
1	1	1	CLK_PER 进行 256 分频	ADC_PRESC_DIV256_gc

要将位域更改为新配置，通常需要搭配使用位组配置和位组掩码以确保先清除旧配置。

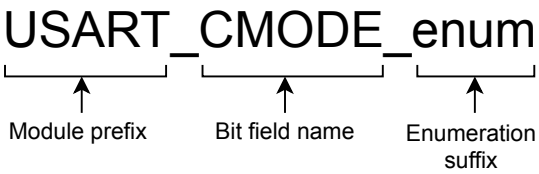
与位掩码和组掩码不同，位组配置掩码使用 C 枚举定义。每个位域都会对应定义一个枚举类型。USART CMODE 位域的枚举如以下代码所示。

```
typedef enum USART_CMODE_enum
{
    USART_CMODE_ASYNCHRONOUS_gc = (0x00<<6),  /* 异步模式 */
    USART_CMODE_SYNCHRONOUS_gc = (0x01<<6),   /* 同步模式 */
    USART_CMODE_IRCOM_gc = (0x02<<6),         /* 红外通信 */
    USART_CMODE_MSPI_gc = (0x03<<6),          /* 主 SPI 模式 */
} USART_CMODE_t;
```

枚举名称由模块类型（USART）、位域（CMODE）和后缀（_enum）组成。

图 2-4 给出了命名约定。

图 2-4. 枚举命名约定



单独使用时，每个枚举常量的行为都与普通常量十分相似。但是，使用枚举类型定义的优势在于可以创建新的数据类型。在本例中，数据类型的名称是 USART_CMODE_t。USART_CMODE_t 变量可直接用作整数，但如果为枚举类型分配整数，则会触发编译器警告。这可以为编程人员带来一些优势。

例如，如果为 USART 模块设置通信模式的函数接受整型值（如 unsigned char）作为通信模式，则任何合法或非法的值都可以传递给函数。如果函数改为接受 USART_CMODE_t 类型的参数，则只有 USART_CMODE_t 枚举类型中的四个预定义常量可以传递给函数。传递任何其他值都将触发编译器警告。

注： 使用普通常量时，需要注意代码列表中的常量是否已移到各自的位位置。而使用枚举常量时，枚举常量即是实际要写入寄存器的值，因此无需额外的移动操作。

3. 为 AVR® 编写裸机 C 代码

以下各小节将重点介绍如何为 AVR 编写 C 代码，举例说明如何在不同的 AVR 器件之间实现代码的可读性和可移植性。这些示例还可用作代码编写指南，以介绍如何编写易于验证和维护的代码。

单片机模块位于存储空间内的专用连续块中，可视为封装单元。各模块均封装在 C 结构中，其中包含所有的模块寄存器。

本文档介绍了符合 AVR 头文件的命名约定和寄存器访问方法，可为采用 C 语言编写的代码提高可读性和可移植性。

3.1 置 1、清零和读取寄存器位

置 1 和清零寄存器位是嵌入式编程中使用的基本操作，代表着应用的方法基础。

读-修改-写操作是一类原子操作。该操作会读取存储单元，同时向其中写入一个新值（写入一个全新的值或者写入上一个值的一部分）。

尽管适用范围很广，但单个位值的读操作主要用于条件表达式（如 `if` 语句），也可用作循环表达式（如 `while` 语句）中的条件。该方法的一个常见用例是轮询中断标志，即读取某个位的值并在该位置 1/清零时执行一组指令。

注： 有关二进制算术运算的更多详细信息，请参见按位运算符。

3.1.1 使用位掩码置 1、清零和读取寄存器位

使用头文件提供的位掩码，可以置 1、清零和读取寄存器位。

使用结构声明访问寄存器

要将寄存器中的特定位置 1，建议的编码风格是使用头文件中声明的结构实例和位掩码宏定义。搭配使用二进制或运算符和位掩码，将确保寄存器内的其他位设置保持不变，不受运算的影响。

```
ADC0.CTRLA |= ADC_ENABLE_bm; /* 使能 ADC 外设 */
```

要将寄存器中的某个位清零，应在寄存器和位掩码的反值之间应用二进制与运算符。该操作也会使其他位设置保持不变。

```
ADC0.CTRLA &= ~ADC_ENABLE_bm; /* 禁止 ADC 外设 */
```

`ADC_ENABLE_bm` 位掩码在头文件中定义，如下所示。

```
#define ADC_ENABLE_bm  0x01  /* ADC 使能位掩码 */
```

无论何种应用都需要读取位值。例如，当 ADC 结果就绪时，硬件会将 ADC RESRDY 标志置 1。下面的代码列表展示了如何测试该位是否置 1，以及如何在该位置 1 时执行一些指令。

```
if(ADC0.INTFLAGS & ADC_RESRDY_bm) /* 检查 ADC 结果是否就绪 */
{
    /* 在此处插入一些指令 */
}
```

要测试某个位是否清零，并在该位保持清零状态时执行指令或等待，可使用以下代码。当该位保持清零状态时，将执行此循环内的指令，直到 ADC 结果就绪。

```
while(!(ADC0.INTFLAGS & ADC_RESRDY_bm))
{
    /* 在此处插入一些指令 */
}
```

使用宏定义访问寄存器

此外，也可以使用头文件中的宏定义来访问寄存器。以下示例展示了如何使用这些定义将位置 1。

```
ADC0_CTRLA |= ADC_ENABLE_bm; /* 使能 ADC 外设 */
```

要使用宏定义将位清零，可使用以下代码示例。

```
ADC0_CTRLA &= ~ADC_ENABLE_bm; /* 禁止 ADC 外设 */
```

下面的代码列表展示了如何测试该位是否置 1，以及如何在该位置 1 时执行一些指令。

```
if(ADC0_INTFLAGS & ADC_RESRDY_bm) /* 检查 ADC 结果是否就绪 */
{
    /* 在此处插入一些指令 */
}
```

要测试某个位是否清零，并在该位保持清零状态时执行指令，可使用以下代码。当该位保持清零状态时，将执行此循环内的指令，直到 ADC 结果就绪。

```
while(!(ADC0_INTFLAGS & ADC_RESRDY_bm))
{
    /* 在此处插入一些指令 */
}
```

3.1.2 使用位位置置 1、清零和读取寄存器位

另一种置 1、清零和读取寄存器位的方法是使用位位置，位位置同样是由器件头文件提供。

使用结构声明访问寄存器

要使用位位置宏将某个位置 1 并使用结构声明来访问寄存器，可使用以下代码示例。

```
ADC0.CTRLA |= (1 << ADC_ENABLE_bp); /* 使能 ADC 外设 */
```

要使用位位置宏将某个位清零，可使用以下代码示例。

```
ADC0.CTRLA &= ~(1 << ADC_ENABLE_bp); /* 禁止 ADC 外设 */
```

要测试某个位是否置 1，可使用以下代码。

```
if(ADC0.INTFLAGS & (1 << ADC_RESRDY_bp)) /* 检查 ADC 结果是否就绪 */
{
    /* 在此处插入一些指令 */
}
```

要测试某个位是否清零，并在该位保持清零状态时执行指令，可使用以下代码。当该位保持清零状态时，将执行此循环内的指令，直到 ADC 结果就绪。

```
while(!(ADC0.INTFLAGS & (1 << ADC_RESRDY_bp)))
{
    /* 在此处插入一些指令 */
}
```

使用宏定义访问寄存器

要使用位位置宏将某个位置 1 并使用宏定义来访问寄存器，可使用以下代码示例。

```
ADC0_CTRLA |= (1 << ADC_ENABLE_bp); /* 使能 ADC 外设 */
```

要使用位位置宏将某个位清零，可使用以下代码示例。

```
ADC0_CTRLA &= ~(1 << ADC_ENABLE_bp); /* 禁止 ADC 外设 */
```

要测试某个位是否置 1，可使用以下代码。

```
if(ADC0_INTFLAGS & (1 << ADC_RESRDY_bp)) /* 检查 ADC 结果是否就绪 */
{
    /* 在此处插入一些指令 */
}
```

要测试某个位是否清零，并在该位保持清零状态时执行指令，可使用以下代码。当该位保持清零状态时，将执行此循环内的指令，直到 ADC 结果就绪。

```
while(!(ADC0_INTFLAGS & (1 << ADC_RESRDY_bp)))
{
    /* 在此处插入一些指令 */
}
```

3.2 寄存器初始化

要初始化寄存器，用户必须查阅器件数据手册来确定所需配置。然后，必须将寄存器位置 1 或清零，以便寄存器中的值与所需配置相匹配。

当寄存器处于已知的复位状态（默认）时，寄存器初始化通常作为复位后器件初始化的一部分执行。

对于大多数 AVR 寄存器，所有位和位域的复位值都是 0。图 3-1 给出了寄存器的复位值以及本示例所需的全部寄存器的位配置。

图 3-1. ADC 控制 A 寄存器——复位值和位设置

Name: CTRLA
Offset: 0x00
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	RUNSTDBY		CONVMODE	LEFTADJ	RESSEL[1:0]		FREERUN	ENABLE
Access	R/W		R/W	R/W	R/W	R/W	R/W	R/W
Reset	0		0	0	0	0	0	0

Bit 7 – RUNSTDBY Run in Standby
This bit determines whether the ADC still runs during Standby.

Value	Description
0	ADC will not run in Standby sleep mode. An ongoing conversion will finish before the ADC enters sleep mode.
1	ADC will run in Standby sleep mode.

Bit 5 – CONVMODE Conversion Mode
This bit defines if the ADC is working in Single-Ended or Differential mode.

Value	Name	Description
0x0	SINGLEENDED	The ADC is operating in Single-Ended mode where only the positive input is used. The ADC result is presented as an unsigned value.
0x1	DIFF	The ADC is operating in Differential mode where both positive and negative inputs are used. The ADC result is presented as a signed value.

Bit 4 – LEFTADJ Left Adjust Result
Writing a '1' to this bit will enable left adjustment of the ADC result.

Bits 3:2 – RESSEL[1:0] Resolution Selection
This bit field selects the ADC resolution. When changing the resolution from 12-bit to 10-bit, the conversion time is reduced from 13.5 CLK_ADC cycles to 11.5 CLK_ADC cycles.

Value	Description
0x00	12-bit resolution
0x01	10-bit resolution
Other	Reserved

Bit 1 – FREERUN Free-Running
Writing a '1' to this bit will enable the Free-Running mode for the ADC. The first conversion is started by writing a '1' to the Start Conversion (STCONV) bit in the Command (ADCn.COMMAND) register.

Bit 0 – ENABLE ADC Enable

Value	Description
0	ADC is disabled
1	ADC is enabled

如果寄存器的复位值为 0x00，则使用位掩码或位位置时无需执行读-修改-写操作，并且使用单行代码即可配置寄存器。

所需的配置可能如下：

- 使能 ADC——ENABLE 位将为 1
- ADC 在差分模式下工作——CONVMODE 位将为 1
- ADC 将在 10 位分辨率下运行——RESSEL 位域值将为 0x00（默认）

该配置可以通过将结果值直接写入寄存器来实现（使用二进制、十六进制或十进制值），如下所示。

```
ADC0.CTRLA = 0b00100001;   /* 二进制 */
ADC0.CTRLA = 0x21;         /* 十六进制 */
ADC0.CTRLA = 33;           /* 十进制 */
```

但是，为了提高代码的可读性（和潜在的可移植性），建议使用器件定义，这些定义将在后续章节中进行介绍。

注： 对于 AVR 寄存器，大多数位和位域的复位值都是 0，但也有例外。例如，USART0 控制 C 寄存器有几个位的复位值是 1。在这种情况下，用户必须明确地设置所需的配置，而不能依赖于位的复位值通常是 0 这一事实。

图 3-2. USART 控制 C 寄存器——复位值

Name: CTRLC
Offset: 0x07
Reset: 0x03
Property: -

This register description is valid for all modes except the Master SPI mode. When the USART Communication Mode (CMODE) bits in this register are written to 'MSPI', see CTRLC - Master SPI mode for the correct description.

Bit	7	6	5	4	3	2	1	0
	CMODE[1:0]		PMODE[1:0]		SBMODE	CHSIZE[2:0]		
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	1	1

USART0 控制 C 寄存器的复位值为 0x03。这种情况下，需要在不更改 CHSIZE 位域值的前提下执行读-修改-写操作来初始化其他位或位域之一。该寄存器如图 3-2 所示。

3.2.1 使用位掩码和组配置掩码的寄存器初始化

本小节介绍使用位掩码和组配置掩码配置 ADC CTRLA 寄存器的建议方法。

```
ADC0.CTRLA = ADC_ENABLE_bm          /* 使能 ADC */
           | ADC_CONVMODE_bm        /* 选择差分转换模式 */
           | ADC_RESSEL_10BIT_gc;   /* 10 位转换 */
```

请注意，赋值的寄存器一侧没有进行按位或（|）运算。在大多数情况下，器件和外设程序都是以这种方式编写。

ADC_RESSEL_enum 枚举包含如下组配置掩码。

```
ADC_RESSEL_10BIT_gc = (0x01<<2),  /* 10 位模式 */
```

⚠ CAUTION

上述寄存器初始化必须在一行 C 代码中完成。编写方式如下，第二行中的组配置将清零在第一行中置 1 的位。

```
ADC0.CTRLA = ADC_ENABLE_bm;
ADC0.CTRLA = ADC_CONVMODE_bm;
ADC0.CTRLA = ADC_RESSEL_10BIT_gc;
```

下面给出了使用三行代码编写上述代码的正确方法。

```
ADC0.CTRLA = ADC_ENABLE_bm;
ADC0.CTRLA |= ADC_CONVMODE_bm;
ADC0.CTRLA |= ADC_RESSEL_10BIT_gc;
```

注： 位掩码只能在一行代码中将位置 1，因此任何需要将位设置为 0 的配置都会被忽略，因为这些位会正确配置为各自的复位值。

3.2.2 使用位位置的寄存器初始化

上一小节中的配置也可以使用位位置宏来完成，具体如下所示。

```
ADC0.CTRLA = (1 << ADC_ENABLE_bp)      /* 使能 ADC */
           | (1 << ADC_CONVMODE_bp)    /* 选择差分转换模式 */
           | (1 << ADC_RESSEL0_bp)     /* 10 位转换 */
           | (0 << ADC_RESSEL1_bp);
```

注： 添加(0 << ADC_RESSEL0_bp)行的目的仅为提高可读性，可将其删除。

```
ADC0.CTRLA = (1 << ADC_ENABLE_bp)      /* 使能 ADC */
           | (1 << ADC_CONVMODE_bp)    /* 选择差分转换模式 */
           | (1 << ADC_RESSEL0_bp);    /* 10 位转换 */
```

此外，还可以使用组位置掩码来配置位域， 具体使用方法如下所示。所需的配置值必须随位域位置（组位置）移位。

```
ADC0.CTRLA = (1 << ADC_ENABLE_bp)       /* 使能 ADC */
           | (1 << ADC_CONVMODE_bp)     /* 选择差分转换模式 */
           | (0x01 << ADC_RESSEL_gp);   /* 10 位转换 */
```

3.3 更改寄存器位域配置

本节介绍使用各种头文件定义更新寄存器位域时的注意事项。具体将以 RXMODE 位域为例，在初始化的基础之上进行更新，如下所示。

```
USART0.CTRLB = USART_RXEN_bm             /* 接收器使能 */
             | USART_TXEN_bm             /* 发送器使能 */
             | USART_RXMODE_LINAUTO_gc; /* LIN 约束自动波特率模式 */
```

以下小节提供了将接收器模式（RXMODE）配置更改为通用自动波特率（GENAUTO）模式的代码示例。下表列出了该位域的可用配置（如器件数据手册中所述）。

图 3-3. USART0 接收器模式配置

Value	Name	Description
0x00	NORMAL	Normal USART mode, standard transmission speed
0x01	CLK2X	Normal USART mode, double transmission speed
0x02	GENAUTO	Generic Auto-Baud mode
0x03	LINAUTO	LIN Constrained Auto-Baud mode

3.3.1 使用组和组配置掩码更改寄存器位域配置

只更新寄存器中的一个位域时，必须使用读-修改-写操作。因此，要更改寄存器位域的配置，建议首先清除旧配置，然后再设置新配置。

建议使用位组配置掩码来设置位域的配置。

以下代码给出了一种将位域更改为新配置的方法。为了确保获得所需的配置，用户必须先使用组掩码将旧配置清除。

```
/* 更改位组配置 */
USART0.CTRLB &= (~USART_RXMODE_gm); /* 清除旧配置 */
/* 使用组配置掩码设置新配置 */
USART0.CTRLB |= USART_RXMODE_GENAUTO_gc;
```

注： 第一行代码将使 USART0 RXMODE 在短时间内进入特定状态：0x00。

组掩码宏在头文件中定义，如下所示。

```
#define USART_RXMODE_gm 0x06 /* 接收器模式组掩码 */
```

USART_RXMODE_enum 枚举包含如下组配置掩码。

```
USART_RXMODE_GENAUTO_gc = (0x02<<1), /* 通用自动波特率模式 */
```

⚠ CAUTION 尽管将代码拆成两行独立的代码似乎更加简单，一行用于清零寄存器，另一行用于设置所需的配置，但仍建议使用一行代码来执行该操作，具体如以下代码所示。

```
/* 更改位组配置 */
USART0.CTRLB = (USART0.CTRLB & ~USART_RXMODE_gm) | USART_RXMODE_GENAUTO_gc;
```

上述步骤必须在一行中实现，以避免单片机进入意外状态。

CTRLB 寄存器以如下方式声明：

```
register8_t CTRLB;
```

register8_t 数据类型为 volatile 数据类型。

```
typedef volatile uint8_t register8_t;
```

由于寄存器被定义为 volatile，因此两行不同的代码将分两次触发对 CTRLB 寄存器的读写操作，而非一次。这除了会降低代码效率之外，还会使外设进入意外状态。 原因在于如果在两行代码之间触发中断，现场将发生变化，而寄存器会保持在未定义状态。

3.3.2 使用位掩码更改寄存器位域配置

以下示例展示了如何使用位掩码来更新寄存器位域，以此设置新的配置。在一行代码中清除当前配置并设置新的配置。

```
USART0.CTRLB = (USART0.CTRLB & ~(USART_RXMODE0_bm | USART_RXMODE1_bm))
             | USART_RXMODE1_bm; /* 通用自动波特率模式 */
```

3.3.3 使用位位置更改寄存器位域配置

以下示例展示了如何使用位位置来更新寄存器位域，以此设置新的配置。在一行代码中清除当前配置并设置新的配置。

```
USART0.CTRLB = (USART0.CTRLB & ~((1 << USART_RXMODE0_bp) | (1 << USART_RXMODE1_bp)))
             | (0 << USART_RXMODE0_bp)
             | (1 << USART_RXMODE1_bp);
```

注： 添加(0 << USART_RXMODE0_bp)行的目的仅为提高可读性，可将其删除。

```
USART0.CTRLB = (USART0.CTRLB & ~((1 << USART_RXMODE0_bp) | (1 << USART_RXMODE1_bp)))
             | (1 << USART_RXMODE1_bp);
```

3.4 使用位掩码和组配置掩码的优势

使用建议的编码风格的优势如下：

- 获得可读性更高的代码。通过使用组配置掩码，用户将能够确定正在设置的配置（掩码名称中包含配置名称）。
- 获得更加紧凑的代码。组掩码将帮助用户清零位域中的所有位，而无需对每个位使用位掩码。配置掩码还可用于配置整个位域。
- 可以更加轻松地维护代码。例如，要更改位域配置，用户只需更改组配置掩码名称，而不是使用位掩码或位位置更改每个位的值。

3.5 写入配置更改保护（CCP）寄存器

必须遵循相应的 CCP 解锁序列，否则无法更改受保护的寄存器。该序列（见器件数据手册）通常包括向 CPU.CCP 寄存器写入签名，然后在四条指令内将所需值写入受保护的寄存器。

CCP 签名由器件头文件提供，如下所示。

```
/* CCP 签名选择 */
typedef enum CCP_enum
{
    CCP_SPM_gc = (0x9D<<0),    /* SPM 指令保护 */
    CCP_IOREG_gc = (0xD8<<0),  /* I/O 寄存器保护 */
} CCP_t;
```

以下示例展示了如何使用 IOREG 签名写入具有 CCP 的寄存器。主时钟预分频系数将设置为 16。

```
CCP = CCP_IOREG_gc; /* 将需要的签名写入 CCP*/
CLKCTRL.MCLKCTRLB = CLKCTRL_PDIV_16X_gc; /* 将预分频系数设置为 16 */
```

以下示例展示了如何使用 SPM 签名写入具有 CCP 的寄存器。

```
CPU.CCP = CCP_SPM_gc; /* 将需要的签名写入 CCP*/
NVMCTRL.CTRLA = NVMCTRL_CMD_FLPER_gc; /* 闪存页擦除使能*/
```

⚠ CAUTION 不建议采用上述方法写入具有 CCP 的寄存器。要写入受 CCP 保护的寄存器，在将签名写入 CCP 寄存器后，软件必须在四条指令内将所需数据写入受保护的寄存器。为满足时序要求，对 CCP 寄存器的写操作通常由汇编代码来处理。

因此，建议使用 `ccp_write_io` 函数来写入受保护的 I/O 寄存器。要使用此函数，必须包含以下头文件。

```
#include <avr/cpufunc.h> /* 需要的头文件 */
```

以下代码示例提供的功能与上述示例相同，但使用的是 `ccp_write_io` 函数。

```
/* 将预分频系数设置为 16 */
ccp_write_io((void *) & (CLKCTRL.MCLKCTRLB), (CLKCTRL_PDIV_16X_gc));
```

或者，也可以定义一个宏，用于将值写入 CCP 寄存器。这种情况下通过 XMEGA® CCP 机制获得保护，该机制可实现 CCP 所需的定时序列。

注： 由于 CCP 寄存器是在 XMEGA 系列 AVR 器件中引入，因此包含 `_PROTECTED_WRITE` 和 `_PROTECTED_WRITE_SPM` 宏的头文件是 `xmega.h`，必须按以下方式包含该头文件。

```
#include <avr/xmega.h> /* 需要的头文件 */
```

必须使用这些宏来写入所需的寄存器，如以下代码示例所示。

```
/* 选择 32 kHz 内部超低功耗振荡器 */
_PROTECTED_WRITE(CLKCTRL.MCLKCTRLA, CLKCTRL.MCLKCTRLA | CLKCTRL_CLKSEL_OSCULP32K_gc);
/* 写入页命令 */
_PROTECTED_WRITE_SPM(NVMCTRL.CTRLA, NVMCTRL_CMD_PAGEWRITE_gc);
```

注： XC8 编译器（MPLAB X IDE）和 GCC（Atmel Studio 7）均支持使用这些宏。

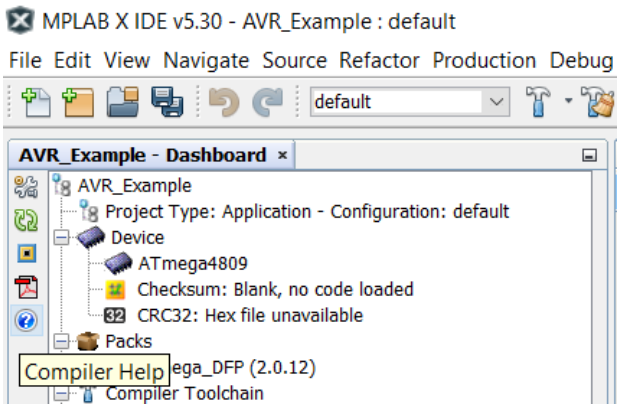
3.6 配置熔丝

熔丝通常已经过预编程，但用户可以更改熔丝寄存器。熔丝只能通过编程来更改。但是，一些熔丝值会装入寄存器，而这些寄存器值可以在运行时更改。熔丝可通过编写 C 代码来配置，以下各小节将对此进行介绍。

3.6.1 使用 XC8 配置位配置熔丝

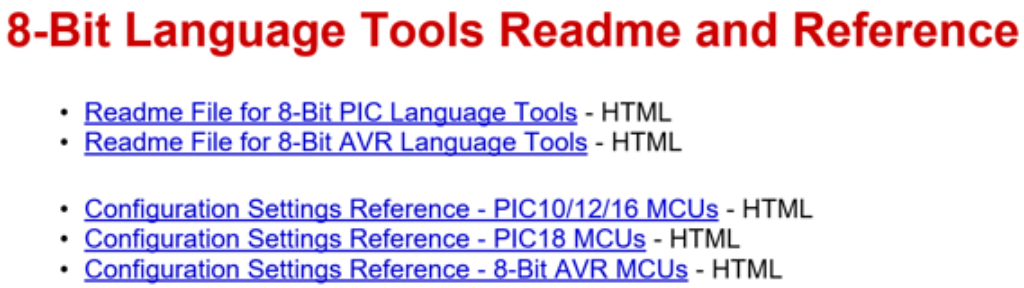
要通过 MPLAB X IDE 配置熔丝，可以使用配置 pragma 伪指令。有关所需器件的编译器和配置位的更多信息，可访问 Compiler Help（编译器帮助）（MPLAB X IDE 项目仪表板中的蓝色问号），如图 3-4 所示。

图 3-4. 访问 Compiler Help



所有支持的器件的配置设置均位于 *Configuration Settings Reference → 8-bit AVR MCUs（配置设置参考 → 8 位 AVR MCU）*，如图 3-5 所示。

图 3-5. Compiler Help → 8 位语言工具自述文件和参考



可按如下方式使用配置 pragma 伪指令。

```
#pragma config <setting> = <named value | literal constant>
```

以下示例展示了如何使用配置 pragma 伪指令来禁止看门狗定时器和 CRC 并将启动时间配置为 8 ms。

```
/* 禁止看门狗定时器 */
#pragma config PERIOD = PERIOD_OFF
/* 禁止 CRC 并将复位引脚配置设置为 GPIO 模式 */
#pragma config CRCSRC = CRCSRC_NOCRC, RSTPINCFG = RSTPINCFG_GPIO
/* 启动时间选择：8 ms */
#pragma config SUT = SUT_8MS
```

3.6.2 使用 AVR® LibC 配置熔丝

要使用 Atmel Studio 配置熔丝，必须包含 `fuse.h` 头文件，如下所示。

```
#include <avr/fuse.h> /* 需要的头文件 */
```

以下示例展示了如何使用看门狗配置（WDTCFG）熔丝寄存器禁止看门狗定时器，如何使用系统配置 0（SYSCFG0）寄存器禁止 CRC 并将复位引脚配置设置为 GPIO 模式，以及如何使用系统配置 1（SYSCFG1）寄存器将启动时间配置为 8 ms。

```
FUSES = {
/* 禁止看门狗定时器 */
.WDTCFG = PERIOD_OFF_gc,
/* 禁止 CRC 并将复位引脚配置设置为 GPIO 模式 */
.SYSCFG0 = CRCSRC_NOCRC_gc | RSTPINCFG_GPIO_gc,
/* 启动时间选择：8 ms */
```

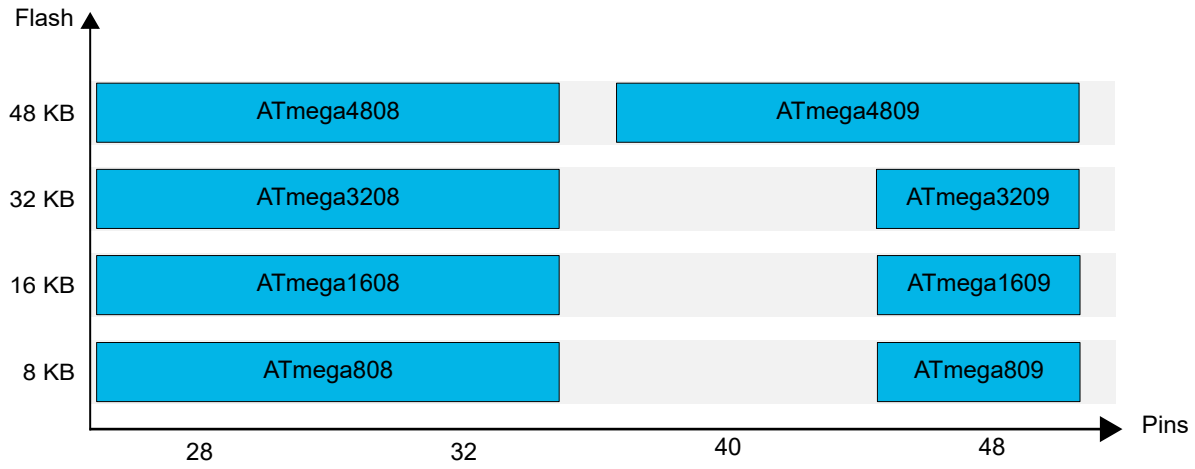
```
.SYSCFG1 = SUT_8MS_gc
};
```

⚠ CAUTION 如果用户未初始化熔丝，则在使用 AVR LibC 时，熔丝将初始化为默认值（0）。如果存在不得为 0 的熔丝且用户没有为其初始化，则器件可能无法按预期工作。

3.7 使用模块指针执行函数调用

在为具有多个实例的模块类型编写驱动程序时，由于所有实例都具有相同的寄存器存储器映射，因此可以使驱动程序供模块类型的所有实例重用。如果驱动程序使用指向相关模块实例的指针参数，则驱动程序可用于此类型的所有模块。从可移植性的角度来看，这是一项很大的优势。此外，编写的代码可以在同一系列的器件之间移植。有关同一系列器件之间的兼容性的详细信息，请参见数据手册的系列*概览*部分，其中一些差异如图 3-6 所示。

图 3-6. megaAVR® 0 系列概览(1)



在具有多个定时器/计数器的器件中，用于初始化和访问这些模块的函数可由所有模块实例共用，而不是为每个实例复制相同的代码行。尽管将模块指针传递给函数的开销很小，但总代码大小将会减小，因为每个模块类型的所有实例都可以重用代码。此外，使用这种方法还可以大幅缩短开发时间、降低维护成本并提高可移植性。

以下代码给出了使用模块指针为任意定时器/计数器模块选择时钟源的函数。

```
void TC_ConfigClockSource (volatile TCB_t *tc, TCB_CLKSEL_t clockSelection)
{
    tc->CTRLA = (tc->CTRLA & ~TCB_CLKSEL_gm) | clockSelection;
}
```

该函数带有两个参数：TCB_t 类型的模块指针和 TCB_CLKSEL_t 类型的组配置。 函数中的代码使用定时器/计数器模块指针访问 CTRLA 寄存器，并使用由 tc 参数提供的地址为定时器/计数器模块设置新的时钟选择。以下代码展示了如何使用上述函数为不同的定时器/计数器实例设置不同的配置。

```
/* 将 TCB0 时钟选择配置为 CLKDIV2 */
TCB_ConfigClockSource (&TCB0, TCB_CLKSEL_CLKDIV2_gc);
/* 将 TCB0 时钟选择配置为 CLKDIV1 */
TCB_ConfigClockSource (&TCB0, TCB_CLKSEL_CLKDIV1_gc);
/* 将 TCB1 时钟选择配置为 CLKDIV2 */
TCB_ConfigClockSource (&TCB1, TCB_CLKSEL_CLKDIV2_gc);
/* 将 TCB2 时钟选择配置为 CLKDIV2 */
TCB_ConfigClockSource (&TCB2, TCB_CLKSEL_CLKDIV2_gc);
```

4. 展示其他代码编写方法的应用示例

为了方便，同时保持与旧版本 AVR 编码风格的兼容性，仍然可以使用不涉及结构的编程风格。本章将简要介绍其他访问寄存器和使用位名称的方法。

4.1 寄存器名称

在不使用模块结构的情况下，仍然可以访问任何寄存器。若要直接引用寄存器，应连接模块实例名称，然后添加下划线和寄存器名称。使用汇编语言编程时的命名约定同样如此。

例如，要访问 B 型定时器/计数器实例 0 的 CTRLA 寄存器，可以使用名称 `TCB0_CTRLA` 代替结构进行访问。

4.2 位位置

可以使用位掩码将位置 1 或清零。寄存器中位的位置是使用与位掩码相同的名称来定义，只是后缀不同。以下代码展示了如何使用位位置来配置寄存器。

```
PORTB_OUT |= (1 << PIN0_bp); /* 设置 PORTB_OUT，bit0 */
```

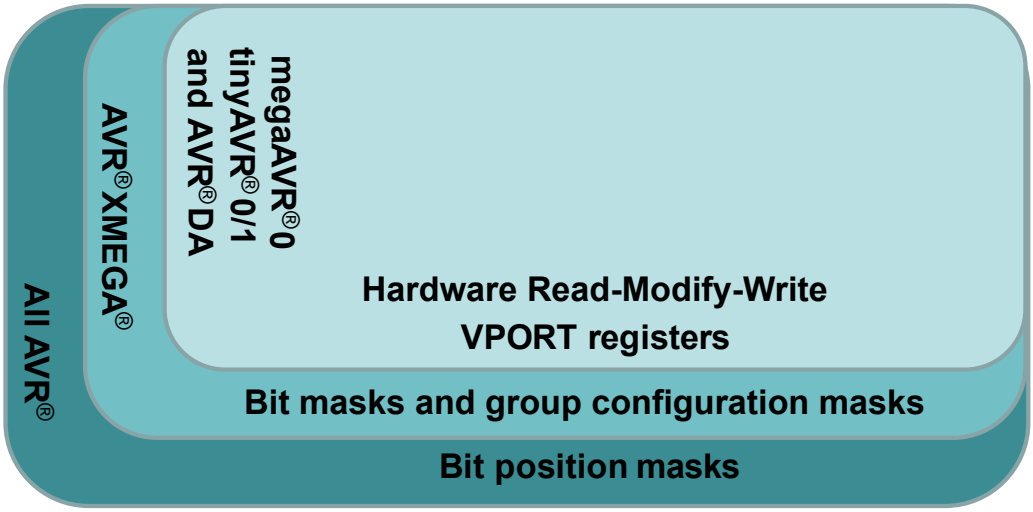
出于兼容性原因，包含了位位置定义。采用汇编语言对使用位编号的指令进行编程时，也需使用这些定义。

4.3 虚拟端口

虚拟端口（VPORT）寄存器允许将一些 PORT 寄存器以虚拟方式映射到位可访问的 I/O 存储空间。完成映射后，写入 VPORT 寄存器即相当于写入实际的 PORT 寄存器。这样，便可在位于扩展 I/O 存储空间中的 PORT 寄存器上使用 I/O 存储器特定的指令，例如位操作指令（SBI/CBI）。在大多数情况下，可用虚拟端口的数量被限制为四个或六个，从而导致可用虚拟端口的数量少于 PORT 模块的数量。使用虚拟端口的优势在于程序更小、更简单，使用的闪存也更小。

图 4-1 给出了兼容性图，其中说明了不同 AVR 产品系列可采用的编码风格。

图 4-1. AVR®系列寄存器配置选项



4.4 PORT 示例

本小节举例说明了如何配置 PORT 以在按下用户按钮时点亮 LED。为了确定单片机的哪些引脚连接到用户 LED 和用户按钮，需要查看所用电路板的原理图。本例使用的是 AVR128DA48 Curiosity Nano 板，LED 与 PORTC 的第 6 个引脚（PC6）相连，按钮与 PORTC 的第 7 个引脚（PC7）相连。为确保引脚的配置正确，用户必须查看电路板原理图。

以下代码展示了如何使用位位置宏来编写用于点亮和熄灭 LED 的代码。

例 4-1. 使用位位置在按下按钮时点亮 LED

```
/* 引脚方向配置 */
/* 读-修改-写：软件 */
PORTC.DIR |= (1 << PIN6_bp);                /* 用户 LED 的输出 */
PORTC.DIR &= ~(1 << PIN7_bp);               /* 用户按钮的输入 */
PORTC.PIN7CTRL |= (1 << PORT_PULLUPEN_bp); /* 上拉配置 */
while (1)
{
    if(PORTC.IN & (1 << PIN7_bp))           /* 检查按钮状态 */
    {
        /* 按钮被释放 */
        PORTC.OUT |= (1 << PIN6_bp);        /* 熄灭 LED */
    }
    else
    {
        /* 按钮被按下 */
        PORTC.OUT &= ~(1 << PIN6_bp);       /* 点亮 LED */
    }
}
```

注： 这是通过 Atmel START 进行外设配置时所采用的编码风格。

以下代码展示了如何使用位掩码实现相同的功能。

例 4-2. 使用位掩码在按下按钮时点亮 LED

```
/* 引脚方向配置 */
/* 读-修改-写：软件 */
PORTC.DIR |= PIN6_bm;                /* 用户 LED 的输出 */
PORTC.DIR &= ~ PIN7_bm;              /* 用户按钮的输入 */
PORTC.PIN7CTRL |= PORT_PULLUPEN_bm; /* 上拉配置 */
while (1)
{
    if(PORTC.IN & PIN7_bm)           /* 检查按钮状态 */
    {
        * 按钮被释放 */
        PORTC.OUT |= PIN6_bm;        /* 熄灭 LED */
    }
    else
    {
        /* 按钮被按下 */
        PORTC.OUT &= ~(PIN6_bm);     /* 点亮 LED */
    }
}
```

注： 这是通过 MCC 为 AVR 生成代码时所采用的编码风格。

此外，使用 SET 和 CLR 寄存器可以在不进行读-修改-写操作的情况下将引脚值置 1 和清零，如以下代码所示。这样便可使用原子指令将所需值置 1 和清零，而无需先读取寄存器值。该操作的主要优势在于不会被中断。只执行一条指令，而不是三条指令（读-修改-写）。

例 4-3. 使用 SET 和 CLR 寄存器在按下按钮时点亮 LED

```
PORTC.DIRSET = PIN6_bm;              /* 用户 LED 的输出 */
PORTC.DIRCLR = PIN7_bm;              /* 用户按钮的输入 */
PORTC.PIN7CTRL |= PORT_PULLUPEN_bm; /* 上拉配置 */

while (1)
{
    /* 读-修改-写：硬件 */
    if(PORTC.IN & PIN7_bm)           /* 检查按钮状态 */
    {
        /* 按钮被释放 */
        PORTC.OUTSET = PIN6_bm;      /* 熄灭 LED */
    }
    else
    {
        /* 按钮被按下 */
        PORTC.OUTCLR = PIN6_bm;      /* 点亮 LED */
    }
}
```

另一种配置方向和置 1/清零输出引脚值的方法是使用虚拟端口，如以下代码所示。

例 4-4. 使用虚拟端口在按下按钮时点亮 LED

```
/* 引脚方向配置 */
/* 读-修改-写：VPORT 寄存器 */
VPORTC.DIR |= PIN6_bm;               /* 用户 LED 的输出 */
VPORTC.DIR &= ~PIN7_bm;              /* 用户按钮的输入 */
PORTC.PIN7CTRL |= PORT_PULLUPEN_bm; /* 上拉配置 */
while (1)
{
    if(VPORTC.IN & PIN7_bm)          /* 检查按钮状态 */
    {
        /* 按钮被释放 */
        VPORTC.OUT |= PIN6_bm;       /* 熄灭 LED */
    }
    else
    {
        /* 按钮被按下 */
        VPORTC.OUT &= ~PIN6_bm;      /* 点亮 LED */
    }
}
```

4.5 ADC 示例

本节以 AVR DA 系列 AVR128DA48 器件上的 ADC0 模块为例，简单说明了如何配置 ADC。有关对该控制器进行编程时所需的信息，可参见器件数据手册，如第 1 章：数据手册结构和命名约定中所述。

要完全配置 ADC，用户可以按照数据手册 *ADC——模数转换器*一节的*功能说明*小节中建议的步骤来执行初始化。

配置完 ADC 预分频比后，使能 ADC 模块，并启动转换。在以下代码中，配置是使用位位置宏来完成。

例 4-5. 使用位位置配置 ADC

```
    /* 使用位位置宏的 ADC 寄存器配置 */
    ADC0.CTRLC |= (1 << ADC_PRESC0_bp) | (1 << ADC_PRESC1_bp); /* 配置预分频比位。
配置：DIV8 */
    ADC0.CTRLA |= (1<< ADC_ENABLE_bp);          /* 使能 ADC */
    ADC0.COMMAND |= (1<< ADC_STCONV_bp);        /* 启动转换 */
```

或者，也可以使用位掩码来配置 ADC，如以下代码所示。

例 4-6. 使用位掩码配置 ADC

```
    /* 使用位掩码宏的 ADC 寄存器配置 */
    ADC0.CTRLC |= ADC_PRESC0_bm | ADC_PRESC1_bm;                /* 配置预分频比位。
配置：DIV8 */
    ADC0.CTRLA |= ADC_ENABLE_bm;                /* 使能 ADC */
    ADC0.COMMAND = ADC_STCONV_bm;               /* 启动转换 */
```

要更改预分频比配置，可以使用组配置掩码。必须首先清除原有配置，如以下代码所示。

例 4-7. 使用组配置掩码更改 ADC 预分频比配置

```
    /* 更改 ADC 预分频比配置，确保清除原有配置 */
    ADC0.CTRLC = (ADC0.CTRLC & ~ADC_PRESC_gm) | ADC_PRESC_DIV4_gc;
```

5. 更多步骤

本章旨在引导用户了解 IDE 安装指南和说明，以及可用的应用笔记。

5.1 应用笔记和技术简介说明

本系列技术简介采用本文档中推荐的原则，涵盖了 megaAVR® 0 系列 AVR 单片机的所有外设。每本技术简介均首先概述涵盖的用例。然后开发各个用例，展示如何参照数据手册将外设配置为所需的配置。

例如，ADC 入门技术简介概述了该外设，并说明了如何在多个用例中以不同的工作模式（单次转换、自由运行和采样累加器等）使用该外设。

- TB3209——Getting Started with ADC
- TB3211——Getting Started with AC
- TB3213——RTC 入门
- TB3214——Getting Started with TCB
- TB3215——Getting Started with SPI
- TB3216——通用同步异步收发器（USART）入门
- TB3217——Getting Started with TCA
- TB3218——Getting Started with CCL
- TB3229——Getting Started with GPIO

5.2 裸机 AVR 开发的相关视频

以下视频非常适用于本技术简介。

- Atmel Studio 7 入门——第 10 集——I/O 视图和裸机编程参考
- MPLAB® X 中的 AVR® 入门——上下文数据手册帮助和 AVR 中断

以下是一个包含了 28 段视频的视频系列，其中使用数据手册和器件头文件作为主要编程参考来构建功能。

- AVR® 入门

5.3 MPLAB XC8 编译器

XC 编译器是一套全面的解决方案，适用于任何合适项目的软件开发。MPLAB XC8 编译器支持所有 8 位 PIC®和 AVR 单片机[(2)]，可以免费下载，且使用不受限制。还提供专业版许可证。通过使用专业版许可证，用户将获得更高效的代码。此外，现已提供经过认证的 XC8 功能安全许可证。

与 MPLAB X IDE 配合使用时，前端将提供编辑错误和断点（与相应的源代码行匹配），并单步执行 C 源代码以检查关键点处的变量和结构。

有关 Microchip 的 MPLAB X IDE 的更多信息，请在用户指南页面中搜索“MPLAB X IDE 用户指南”。

5.4 IDE（MPLAB X、Atmel Studio 和 IAR）——入门

要对 AVR 单片机进行编程，可以使用 MPLAB X、Atmel Studio 或 IAR Embedded Workbench IDE。

MPLAB X 集成开发环境（IDE）是一款可扩展且高度可配置的软件程序，包含多种功能强大的工具，可帮助用户发现、配置、开发、调试和验证大多数 Microchip 单片机和数字信号控制器的嵌入式设计。MPLAB X IDE 支持 AVR MCU。

以下用户指南提供了全面的动手实验和视频教程来帮助您熟悉 Atmel Studio：Getting Started with Atmel Studio 7。此外，有关开发项目所需的全部项目配置（熔丝编程、振荡器校准和接口设置等）的信息，可参见 Atmel Studio 7 用户指南。要发现和开发电路板的更多示例、配置驱动程序、查找示例项目以及轻松配置系统时钟设置，可以使用在线代码配置工具 Atmel START。更多详细信息，请参见 Atmel START User Guide。

AN3419——《适用于 AVR®单片机的 IAR Embedded Workbench®入门指南》应用笔记中介绍了适用于 AVR 的 IAR Embedded Workbench 及所有功能。

tinyAVR® 0/1 系列、megaAVR 0 系列和 AVR DA 系列单片机也有一系列的在线入门专用指南，其中包含有关如何创建项目以及使用哪款入门工具包的信息。下面列出了这些指南的示例：

- megaAVR® 0 系列入门
- tinyAVR® 1 系列单片机入门
- tinyAVR® 0 系列入门
- AVR DA 系列入门

6. 结论

本文档向用户介绍了 AVR 单片机编程的首选编码风格。通过本文档，用户可以了解数据手册提供的信息类型，以及头文件提供的宏定义、变量声明和数据类型定义。本文档旨在使用易于维护且具有可移植性和可读性的编码风格，熟悉 AVR 寄存器和位的命名约定，并为使用该系列单片机进行进一步的项目开发做好准备。

本文档提供的信息范围涵盖特定数据手册，信息说明，命名约定，如何为 AVR 单片机编写 C 代码，其他代码编写方法以及项目开发的更多步骤。

本文档建议的 C 代码编写方法并不强制使用，但介绍的诸多优势值得考虑。项目规模越大，器件功能越多，优势就越显著。

7. 参考资料

1. ATmega4808/4809 Data Sheet
2. MPLAB® XC Compilers
3. AVR Libc Library Reference
4. AVR® Devices in MPLAB® XC8
5. MPLAB® XC8 C Compiler User's Guide for AVR® MCU
6. Fundamentals of the C Programming Language
7. Fundamentals of C Programming - Enumerations

8. 版本历史

文档版本	日期	备注
B	2020 年 6 月	删除了首页中的汇总标签
A	2020 年 5 月	文档初始版本

Microchip 网站

Microchip 网站（www.microchip.com/）为客户提供在线支持。客户可通过该网站方便地获取文件和信息。我们的网站提供以下内容：

- **产品支持**——数据手册和勘误表、应用笔记和示例程序、设计资源、用户指南以及硬件支持文档、最新的软件版本以及归档软件
- **一般技术支持**——常见问题解答（FAQ）、技术支持请求、在线讨论组以及 Microchip 设计伙伴计划成员名单
- **Microchip 业务**——产品选型和订购指南、最新 Microchip 新闻稿、研讨会和活动安排表、Microchip 销售办事处、代理商以及工厂代表列表

产品变更通知服务

Microchip 的产品变更通知服务有助于客户了解 Microchip 产品的最新信息。注册客户可在他们感兴趣的某个产品系列或开发工具发生变更、更新、发布新版本或勘误表时，收到电子邮件通知。

欲注册，请访问 www.microchip.com/pcn，然后按照注册说明进行操作。

客户支持

Microchip 产品的用户可通过以下渠道获得帮助：

- 代理商或代表
- 当地销售办事处
- 应用工程师（ESE）
- 技术支持

客户应联系其代理商、代表或 ESE 寻求支持。当地销售办事处也可为客户提供帮助。本文档后附有销售办事处的联系方式。

也可通过 www.microchip.com/support 获得网上技术支持。

Microchip 器件代码保护功能

请注意以下有关 Microchip 产品代码保护功能的要点：

- Microchip 的产品均达到 Microchip 数据手册中所述的技术规范。
- Microchip 确信：在按照操作规范正常使用的情况下，Microchip 系列产品非常安全。
- Microchip 重视并积极保护其知识产权。任何试图破坏 Microchip 产品代码保护功能的行为均可视为违反了《数字器件千年版权法案（Digital Millennium Copyright Act）》并予以严禁。
- Microchip 或任何其他半导体厂商均无法保证其代码的安全性。代码保护并不意味着我们保证产品是“牢不可破”的。代码保护功能处于持续发展中。Microchip 承诺将不断改进产品的代码保护功能。

法律声明

本出版物中提供的信息仅仅是为方便您使用 Microchip 产品或使用这些产品来进行设计、测试以及与应用相集成。以任何其他方式使用这些信息，都将违反相关条款。器件应用信息仅为您提供便利，它们可能由更新之信息所替代。确保应用符合技术规范，是您自身应负的责任。如需更多支持，请联系您当地的 Microchip 销售办事处，或访问 www.microchip.com/en-us/support/design-help/client-support-services。

Microchip“按原样”提供这些信息。Microchip 对这些信息不作任何明示或暗示、书面或口头、法定或其他形式的声明或担保，包括但不限于针对非侵权性、适销性和特定用途的适用性的暗示担保，或针对其使用情况、质量或性能的担保。

在任何情况下，对于因这些信息或使用这些信息而产生的任何间接的、特殊的、惩罚性的、偶然的或间接的损失、损害或任何类型的开销，Microchip 概不承担任何责任，即使 Microchip 已被告知可能发生损害或损害可以预见。在法律

商标

ISBN： 978-1-5224-9685-4

质量管理体系

有关 Microchip 的质量管理体系的信息，请访问 www.microchip.com/quality。

全球销售及服务网点

美洲	亚太地区	亚太地区	欧洲
公司总部 2355 West Chandler Blvd. Chandler, AZ 85224-6199 电话：480-792-7200 传真：480-792-7277 技术支持： www.microchip.com/support 网址： www.microchip.com **亚特兰大** 德卢斯，佐治亚州 电话：678-957-9614 传真：678-957-1455 **奥斯汀，德克萨斯州** 电话：512-257-3370 **波士顿** 韦斯特伯鲁，马萨诸塞州 电话：774-760-0087 传真：774-760-0088 **芝加哥** 艾塔斯卡，伊利诺伊州 电话：630-285-0071 传真：630-285-0075 **达拉斯** 阿迪森，德克萨斯州 电话：972-818-7423 传真：972-818-2924 **底特律** 诺维，密歇根州 电话：248-848-4000 **休斯顿，德克萨斯州** 电话：281-894-5983 **印第安纳波利斯** 诺布尔斯维尔，印第安纳州 电话：317-773-8323 传真：317-773-5453 电话：317-536-2380 **洛杉矶** 米慎维荷，加利福尼亚州 电话：949-462-9523 传真：949-462-9608 电话：951-273-7800 **罗利，北卡罗来纳州** 电话：919-844-7510 **纽约，纽约州** 电话：631-435-6000 **圣何塞，加利福尼亚州** 电话：408-735-9110 电话：408-436-4270 **加拿大 - 多伦多** 电话：905-695-1980 传真：905-695-2078	**澳大利亚 - 悉尼** 电话：61-2-9868-6733 **中国 - 北京** 电话：86-10-8569-7000 **中国 - 成都** 电话：86-28-8665-5511 **中国 - 重庆** 电话：86-23-8980-9588 **中国 - 东莞** 电话：86-769-8702-9880 **中国 - 广州** 电话：86-20-8755-8029 **中国 - 杭州** 电话：86-571-8792-8115 **中国 - 香港特别行政区** 电话：852-2943-5100 **中国 - 南京** 电话：86-25-8473-2460 **中国 - 青岛** 电话：86-532-8502-7355 **中国 - 上海** 电话：86-21-3326-8000 **中国 - 沈阳** 电话：86-24-2334-2829 **中国 - 深圳** 电话：86-755-8864-2200 **中国 - 苏州** 电话：86-186-6233-1526 **中国 - 武汉** 电话：86-27-5980-5300 **中国 - 西安** 电话：86-29-8833-7252 **中国 - 厦门** 电话：86-592-2388138 **中国 - 珠海** 电话：86-756-3210040	**印度 - 班加罗尔** 电话：91-80-3090-4444 **印度 - 新德里** 电话：91-11-4160-8631 **印度 - 浦那** 电话：91-20-4121-0141 **日本 - 大阪** 电话：81-6-6152-7160 **日本 - 东京** 电话：81-3-6880-3770 **韩国 - 大邱** 电话：82-53-744-4301 **韩国 - 首尔** 电话：82-2-554-7200 **马来西亚 - 吉隆坡** 电话：60-3-7651-7906 **马来西亚 - 槟榔屿** 电话：60-4-227-8870 **菲律宾 - 马尼拉** 电话：63-2-634-9065 **新加坡** 电话：65-6334-8870 **台湾地区 - 新竹** 电话：886-3-577-8366 **台湾地区 - 高雄** 电话：886-7-213-7830 **台湾地区 - 台北** 电话：886-2-2508-8600 **泰国 - 曼谷** 电话：66-2-694-1351 **越南 - 胡志明市** 电话：84-28-5448-2100	**奥地利 - 韦尔斯** 电话：43-7242-2244-39 传真：43-7242-2244-393 **丹麦 - 哥本哈根** 电话：45-4485-5910 传真：45-4485-2829 **芬兰 - 埃斯波** 电话：358-9-4520-820 **法国 - 巴黎** 电话：33-1-69-53-63-20 传真：33-1-69-30-90-79 **德国 - 加兴** 电话：49-8931-9700 **德国 - 哈恩** 电话：49-2129-3766400 **德国 - 海尔布隆** 电话：49-7131-72400 **德国 - 卡尔斯鲁厄** 电话：49-721-625370 **德国 - 慕尼黑** 电话：49-89-627-144-0 传真：49-89-627-144-44 **德国 - 罗森海姆** 电话：49-8031-354-560 **以色列 - 若那那市** 电话：972-9-744-7705 **意大利 - 米兰** 电话：39-0331-742611 传真：39-0331-466781 **意大利 - 帕多瓦** 电话：39-049-7625286 **荷兰 - 德卢内市** 电话：31-416-690399 传真：31-416-690340 **挪威 - 特隆赫姆** 电话：47-72884388 **波兰 - 华沙** 电话：48-22-3325737 **罗马尼亚 - 布加勒斯特** 电话：40-21-407-87-50 **西班牙 - 马德里** 电话：34-91-708-08-90 传真：34-91-708-08-91 **瑞典 - 哥德堡** 电话：46-31-704-60-40 **瑞典 - 斯德哥尔摩** 电话：46-8-5090-4654 **英国 - 沃金厄姆** 电话：44-118-921-5800 传真：44-118-921-5820