AVR1000b: Getting Started with Writing C-Code for AVR® MCUs

Introduction

Authors: Cristina Ionescu, Cristian Săbiuţă, Microchip Technology Inc.

This technical brief provides the recommended steps to successfully program the AVR® microcontrollers (MCUs) and to define coding guidelines to help writing more readable and reusable code.

High-level programming languages have become a necessity due to the imposed short development time and high-quality requirements. They make it easier to maintain and reuse code due to better portability and readability than the low-level instructions specific for each microcontroller architecture.

Programming language alone does not ensure high readability and reusability, but good coding style does. Therefore, the AVR MCU peripherals, header files and drivers are designed according to this presumption.

The most widely used high-level language for AVR microcontrollers is C, so this document will focus on C programming. To ensure compatibility with most AVR C compilers, the code examples in this document are written using ANSI C coding standard.

This document contains code examples developed with the Atmel Studio Integrated Development Environment (IDE). Most code examples are compatible with other IDEs, presented in Section 5: Further Steps.

Table of Contents

1. Data Sheet Module Structure and Naming Conventions

The first step in writing C-code for a microcontroller is to know and understand what type of information can be found in the data sheet of the device used for programming. The data sheet contains information about the features, the memories, the core and the peripheral modules of the microcontroller, the functional description of the peripheral modules, the peripherals base addresses, the names and addresses of the registers, and other functional and electrical characteristics.

1.1 How to Find the Data Sheet

Any documentation related to Microchip products can be found at:

- Microchip Data Sheets

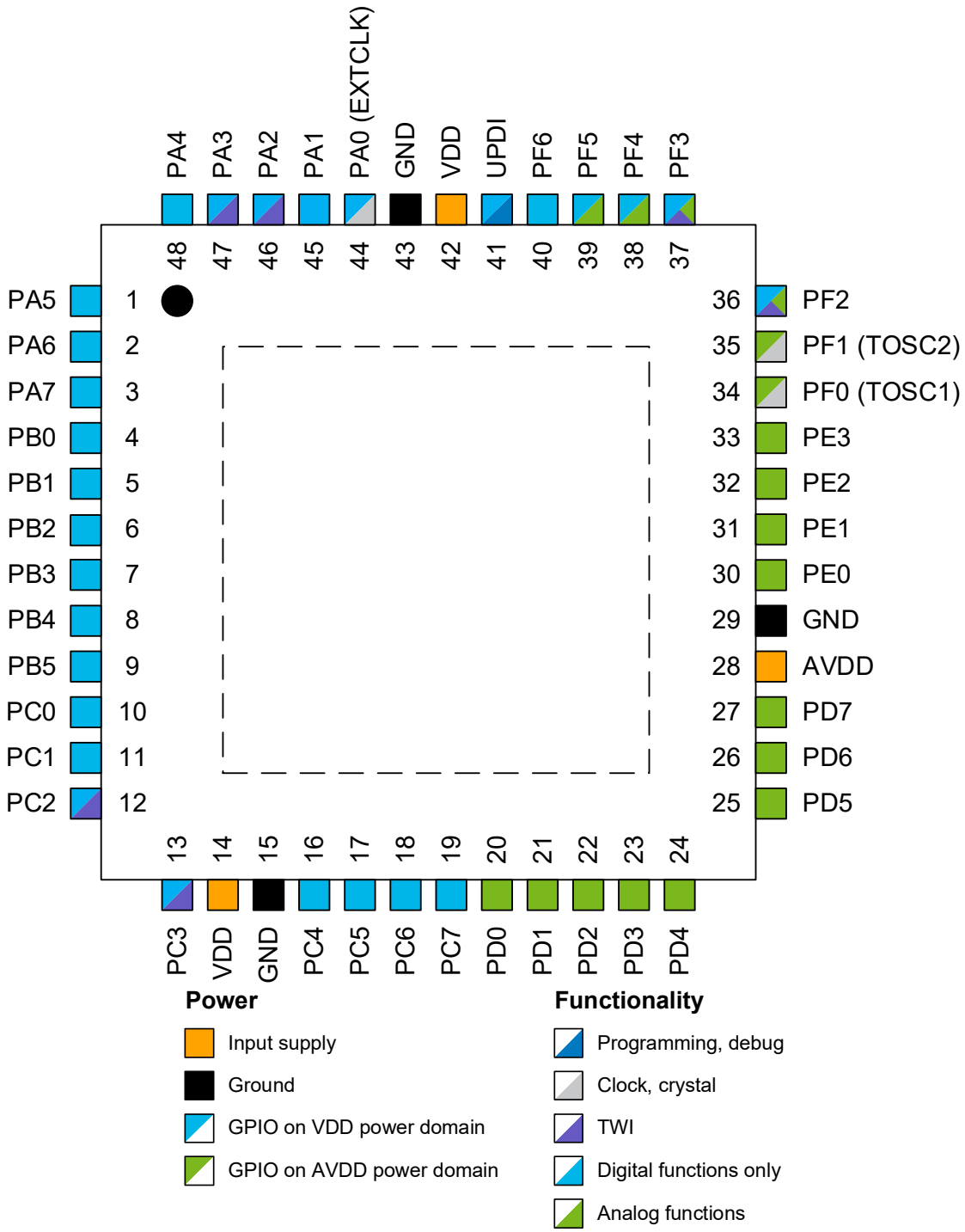
The device data sheets, for the device families of interest in this document, can be found at:

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

1.2 Pin Description

The pin description can be found in any device data sheet. The pinout is contained in the *Pinout*, *Pin Configurations* section, or any other name convention, depending on the device. The pinout of the ATmega809/1609/3209/4809 48-pin devices is presented in Figure 1-1.

Figure 1-1. ATmega809/1609/3209/4809 48-Pin Pinout(1)



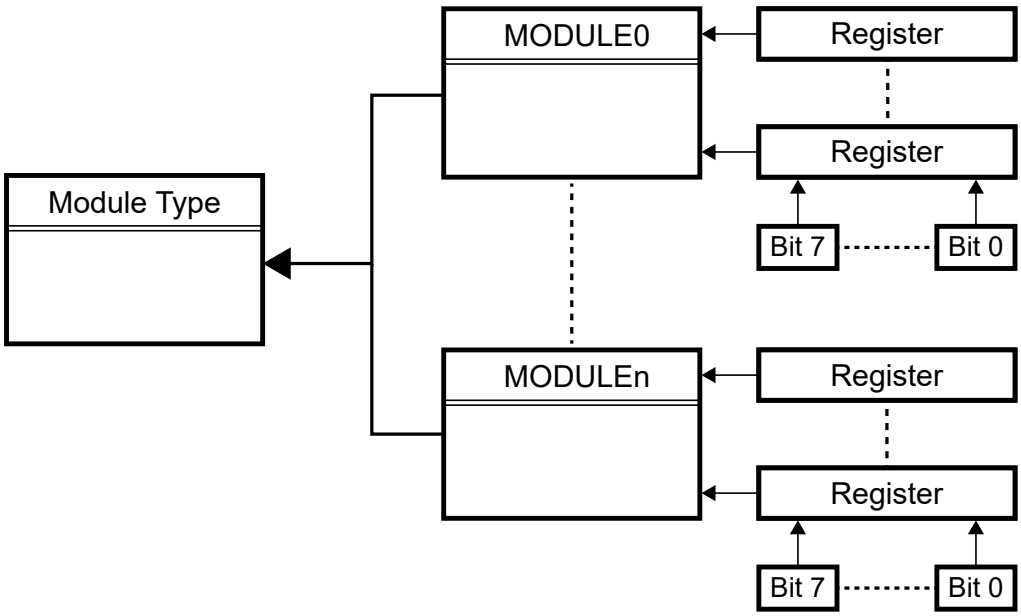
The configurable functionalities for each I/O pin are described in the *I/O Multiplexing and Considerations* section, or the *Alternate Port Functions* subsection of the *I/O Ports* section, depending on the device. If an evaluation board is used, such as AVR128DA48 Curiosity Nano, the user needs to know how the microcontroller's pins are allocated on the specific board. The information is available on the AVR128DA48 Curiosity Nano webpage, in the AVR128DA48 Curiosity Nano Schematics document. Other documents that describe the AVR128DA48 Curiosity Nano board and microcontroller characteristics are available on the same webpage.

1.3 Modules Description

An AVR microcontroller is comprised of several building blocks: AVR CPU, SRAM, Flash, EEPROM and several peripheral modules called module types. Throughout this document, all peripheral modules will be referred to as modules.

Newer AVR microcontroller families can have one or more instances of a given module type. All instances have the same features and functions. Some module types are a subset of others and inherit some of their features. The inherited features are fully compatible with the respective module type. For example, the subset module for a timer can have fewer compare and capture channels than a full timer module.

Figure 1-2. Modules Types, Instances, Registers and Bits



A module type can be the Universal Synchronous-Asynchronous Receiver Transmitter (USART), while the module instance is, e.g., USART0, where the 0 suffix indicates the instance is “USART number 0”. For simplicity, a module instance will be referred to as a module throughout this document, unless there is a need to differentiate.

Each module has a fixed base address in the I/O memory map, and all registers contained in the module have fixed offset addresses relative to the module base address. This way, each register will not only have an absolute address in the I/O memory space, but also a relative address defined by its offset. The register offset addresses are equal for all instances of a module type, simplifying the task of writing drivers that can be used for all modules of a specific type. The peripheral module address map can be found in the data sheet and shows the base address for each peripheral.

Each module has several registers that contain control or status bits. All modules of a given type contain the same set (or subset) of registers, and all these registers contain the same set (or subset) of control and status bits.

Table 1-1 presents the base address of some of the ATtiny804/1604 peripherals.

Table 1-1. Peripheral Module Address Map (section)(1)

Base Address	Name	Description
0x0000	VPORTA	Virtual Port A
0x0004	VPORTB	Virtual Port B
0x001C	GPIO	General Purpose I/O registers
0x0030	CPU	CPU
0x0040	RSTCTRL	Reset Controller

.....continued

Base Address	Name	Description
0x0050	SLPCTRL	Sleep Controller
0x0060	CLKCTRL	Clock Controller
...	...	...

Every module has a dedicated section presenting the features that the module has, a functional description of the module, and the specific signals and guidelines on how to configure a certain mode of operation with all the terminology explained. At the end of a module section, there is a register description subsection that contains the scope of every register, the initial value, and if it is readable or writable. It also provides the position of every configurable/accessible bit of a register.

All the registers, their addresses offsets, and the bit names and positions are described in the *Register Summary* section for each module. The register summary for the ADC module is presented in Figure 1-3.

Figure 1-3. ADC Register Summary(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	Reserved									
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	Reserved									
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 Naming Conventions

This section describes the register and bit naming conventions that can be found in the device data sheet and in the header file used to develop any application.

1.4.1 Register Naming Conventions

The registers are divided into control, status, and data registers, and the names of the registers reflect this. A general purpose control register of a module is named CTRL. If multiple general purpose control registers exist in the same module, they will be differentiated by a suffix character. In this case, the control registers will be named CTRLA, CTRLB, CTRLC, and so on. This also applies to status registers.

For registers that have a specific function, the name reflects their functionality. For example, a control register that controls the interrupts of a module is named INTCTRL.

Since, for the microcontrollers presented in this document, the data bus width is 8-bit, larger registers are implemented using several 8-bit registers. For a 16-bit register, the high and low bytes are accessed by appending H and L respectively to the register name. For example, the 16-bit Analog-to-Digital Result register is named RES. The

two bytes are named RESL (RES-Low, the Least Significant Byte of the register) and RESH (RES-High, the Most Significant Byte of the register). Another way to identify multiple registers with the same name is to add a number suffix; for example, the ADDR register in NVMCTRL is a 24-bit register, for the AVR DA family. The three bytes that can be accessed using the number suffix are ADDR0, ADDR1 and ADDR2.

Most C compilers offer automatic handling of access to multibyte registers. In that case, the name RES, without H or L suffix, can be used to perform a 16-bit access to the ADC Result register. This is also the case for 32-bit registers.

Additionally, the registers that contain the SET/CLR suffix allow the user to set and clear bits in those registers without doing a Read-Modify-Write operation since it is implemented in hardware. These registers come in pairs. Writing a logic '1' to a bit in the CLR register will clear the corresponding bit in both registers, while writing a logic '1' to a bit in the SET register will set the corresponding bits in both registers. For example, in the PORT module, writing a logic '1' to a bit in the Data Direction Set (DIRSET) register will clear the corresponding bit in the Data Direction (DIR) and Data Direction Clear (DIRCLR) registers. Both registers will return the same value when read. If both registers are written simultaneously, the write to the CLR register will take precedence.

1.4.2 Bit and Bit Field Naming Conventions

Most of the following examples are provided for the Analog-to-Digital Converter (ADC) module.

Register bits can have an individual function or be part of a bit field that has a joint function. An individual bit can be a bit that enables a module, e.g., the ENABLE bit of the ADC module in the CTRLA register. A bit field can consist of two or more bits that jointly select a specific configuration of the module they belong to. A bit field offers up to 2^n selections, where *n* is the number of bits in the bit field.

The position of the ENABLE bit is presented in Figure 1-4, and the PRESC bit field position is presented in Figure 1-5.

1.4.2.1 Bit Naming Conventions

The bit names found in the register diagrams are abbreviations of the full bit name, which describes the functionality that bit configures. For example, the RUNSTBY bit allows the user to enable the peripheral to Run in Standby sleep mode. The user can enable the ADC by configuring the ENABLE bit.

Figure 1-4. Bit Naming Conventions for the ADC Control A Register(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 Bit Field Naming Conventions

The bits from the same bit field can be referred to using a numerical suffix appended with the number of the bit position relative to the bit field. For example, the Most Significant bit (MSb) from the ADC Control C prescaler bit field will be PRESC2. This naming convention is not specified in the data sheet, but it will be described later in this document, and it is used in the header files to handle the individual register bits from a bit field.

Figure 1-5. The PRESC Configuration Bit Field in the Control C Register(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

The PRESC bits offer the selection presented in Figure 1-6, for each value of the bit field.

Figure 1-6. Prescaler Configuration for the ADC Clock Signal Frequency(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 Configuration Change Protection (CCP) Registers

CCP registers are used to protect system-critical I/O register settings from accidental modification, as well as Flash self-programming from accidental execution. Writing to a register under CCP is possible only after writing a specific signature/key to the CCP register, which is part of the CPU module. The values of the signatures can be found in the *Configuration Change Protection* subsection of the device data sheet.

There are two types of writes to a protected register, each with its key:

- protected I/O registers (the key/signature is IOREG)
- protected self-programming (the key/signature is SPM)

Some of the registers that are under the Configuration Change Protection are listed in the table below.

Register Name		Key	Functionality
CLKCTRL.MCLKCTRLA	Clock Controller – Main Clock Control A	IOREG	It allows the user to select the clock source for the main clock and to output the clock
CLKCTRL.MCLKLOCK	Clock Controller – Main Clock Lock	IOREG	It allows the user to lock the Main Clock Control registers
RSTCTRL.SWRR	Reset Controller – Software Reset Enable	IOREG	It allows the user to apply a software Reset to the device
IVSEL in CPUINT.CTRLA	CPU Interrupt Controller – the Interrupt Vector Select bit from the CTRLA register	IOREG	It allows the user to place the interrupt vector at the start of the application section of Flash or the start of the boot section of Flash
NVMCTRL.CTRLA	Nonvolatile Memory Controller – the CTRLA register	SPM	It allows the user to issue one of the next commands: • Write page buffer to memory • Erase page • Erase and write page • Page buffer clear, etc.

Changes to the protected I/O registers or bits, or execution of protected instructions are only possible after the CPU writes one of these signatures to the CCP register. The signatures are listed in the description of the CCP (CPU.CCP) register.

A code example is provided in 3.5 Writing to Configuration Change Protection (CCP) Registers.

1.6 Fuses

The fuses hold the device configuration and are part of the nonvolatile memory. They are available from device power-up. Their values are maintained through a chip erase. The fuses can be read by the CPU or the programming interface (e.g., UPDI), but can only be programmed or cleared through the program/debug interface. The configuration values stored in the fuses are copied to their respective target registers at the end of the start-up sequence so that the fuse values can be used by the CPU.

The Fuse Summary table can be found in the *Memories → Fuses (FUSE)* subsection from the device data sheet. An example extracted from the ATmega4808/4809 data sheet is presented below.

Figure 1-7. FUSE Register Summary

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]							

A code example is provided in 3.6 Configuring Fuses.

2. Module Representation in Header Files

A dedicated header file is available for each AVR device. The target device needs to be specified in the project settings (for any used IDE – MPLAB® X, Atmel Studio, or IAR EWAVR), and the header file will be automatically included in the main file of the created project. The header file is included as shown in the code below.

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

All the needed register and structure definitions can be found in the header file. The macros and structures definitions which are already defined in the device-specific header file can be used, instead of using a register's address.

This is useful in devices that contain the same module, and the header file definitions for that module are the same.

2.1 Module Location in Memory

All registers for a given peripheral module are placed in one continuous memory block. Registers that belong to different modules are not mixed up, which makes it possible to organize all peripheral modules in C structures, where the instance macro is defined as shown in the code below. The definitions of all peripheral modules are found in the device header files available for these AVR devices. The address for the modules is specified in ANSI C to make it compatible with most available C compilers.

```
#define PORTMUX           (*(PORTMUX_t *) 0x0200) /* Port Multiplexer */
#define PORTA                (*(PORT_t *) 0x0400) /* I/O Ports */
#define PORTB                (*(PORT_t *) 0x0420) /* I/O Ports */
#define PORTC                (*(PORT_t *) 0x0440) /* I/O Ports */
```

The module instance definition uses a dereferenced pointer to the absolute address in the memory, coinciding with the module instance base address. The module pointers are defined in the header files; therefore, it is not necessary to add these definitions in the source code.

For example, the base address of the PORTA module is 0x0400. The module with all its registers and reserved bytes has an available memory space from 0x0400 to 0x0420, which means 32 bytes in decimal. Therefore, the `PORT_t` contains 32 allocated bytes for all its registers (or reserved bytes).

2.2 Module Structures

2.2.1 Example - ADC

The `ADC_t` structure type is defined in the header file, as presented in the code below. It contains all the module's registers in the data sheet specified order, as they are organized in the memory.

```
/* Analog-to-Digital Converter */
typedef struct ADC_struct
{
    register8_t CTRLA;     /* Control A */
    register8_t CTRLB;     /* Control B */
    register8_t CTRLC;     /* Control C */
    register8_t CTRLD;     /* Control D */
    register8_t CTRLE;     /* Control E */
    register8_t SAMPCTRL;  /* Sample Control */
    register8_t MUXPOS;    /* Positive MUX input */
    register8_t reserved_1[1];
    register8_t COMMAND;   /* Command */
    register8_t EVCTRL;    /* Event Control */
    register8_t INTCTRL;   /* Interrupt Control */
    register8_t INTFLAGS;  /* Interrupt Flags */
    register8_t DBGCTRL;   /* Debug Control */
    register8_t TEMP;      /* Temporary Data */
    register8_t reserved_2[2];
    _WORDREGISTER(RES);    /* ADC Accumulator Result */
    _WORDREGISTER(WINLT);  /* Window comparator low threshold */
    _WORDREGISTER(WINHT);  /* Window comparator high threshold */
    register8_t CALIB;     /* Calibration */
```

```
    register8_t reserved_3[1];
} ADC_t;
```

A macro for an instance of a module is then defined in the header file using that structure type, as presented in the code below.

```
#define ADC0    (*(ADC_t *) 0x0600) /* Analog-to-Digital Converter */
```

Therefore, a particular module register, e.g., the CTRLA register, can be addressed as ADC0.CTRLA.

2.2.2 Example - PORT

The `PORT_t` structure type is defined in the header file, as presented in the code below.

```
/* I/O Ports */
typedef struct PORT_struct
{
    register8_t DIR;       /* Data Direction */
    register8_t DIRSET;    /* Data Direction Set */
    register8_t DIRCLR;    /* Data Direction Clear */
    register8_t DIRTGL;    /* Data Direction Toggle */
    register8_t OUT;       /* Output Value */
    register8_t OUTSET;    /* Output Value Set */
    register8_t OUTCLR;    /* Output Value Clear */
    register8_t OUTTGL;    /* Output Value Toggle */
    register8_t IN;        /* Input Value */
    register8_t INTFLAGS;  /* Interrupt Flags */
    register8_t reserved_1[6];
    register8_t PIN0CTRL;  /* Pin 0 Control */
    register8_t PIN1CTRL;  /* Pin 1 Control */
    register8_t PIN2CTRL;  /* Pin 2 Control */
    register8_t PIN3CTRL;  /* Pin 3 Control */
    register8_t PIN4CTRL;  /* Pin 4 Control */
    register8_t PIN5CTRL;  /* Pin 5 Control */
    register8_t PIN6CTRL;  /* Pin 6 Control */
    register8_t PIN7CTRL;  /* Pin 7 Control */
    register8_t reserved_2[8];
} PORT_t;
```

2.2.3 Multibyte Registers in Module Structures

Some registers are used in conjunction with other registers to represent 16- or 32-bit values. For example, the ADC result (RES), the Window Comparator Low Threshold WINLT, and the Window Comparator High Threshold WINTH are 16-bit registers for the ATmega4809 device, declared using the `_WORDREGISTER` macro. The macro is presented in the listing below, along with the `_DWORDREGISTER` macro used to declare 32-bit registers. These macros are already defined in the header file.

The `_WORDREGISTER` macro is used to extend the name of a register to access its lower byte and its higher byte, by adding the L or the H suffix. The `_DWORDREGISTER` macro is also providing access to all bytes of a multibyte register, by adding a number suffix. Both `_WORDREGISTER` and `_DWORDREGISTER` definitions are presented in the code below.

```
#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 Bit Masks, Bit Group Masks and Group Configuration Masks

The user can access any register using the structure configuration of the modules. The ADC can be fully configured using the steps found in the data sheet, in the module's description specific section. For example, the recommended steps to initialize the ADC are presented in *ADC - Analog-to-Digital Converter → Functional Description → Initialization*. Register bits can be manipulated using bit masks or bit position masks, which are defined in the header file. The predefined masks are either related to individual bits, in which case they are called bit masks, or to a group of bits (bit field), in which case they are called a bit group mask, or a group mask. For example, the ADC0 module can be enabled and configured to start a conversion cycle by using bit masks.

2.3.1 Bit Masks and Bit Position Masks

A bit mask is used both when setting and clearing individual bits. A bit group mask is mainly used when clearing multiple bits in a bit field. For example, the bit fields, bit names, bit positions, and bit masks of the CTRLD register of ADC0 are shown in Table 2-1.

Table 2-1. Bit Fields, Bit Names, Bit Positions and Bit Masks

Bit Field	INITDLY[2:0]			-	SAMPDLY[3:0]			
Bit Name	INITDLY2	INITDLY1	INITDLY0	ASDV	SAMPDLY3	SAMPDLY2	SAMPDLY1	SAMPDLY0
Bit Position	7	6	5	4	3	2	1	0
Bit Mask	0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01

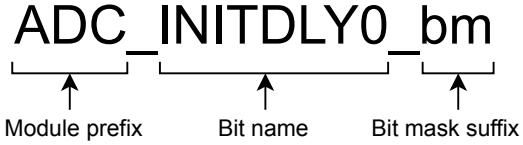
Since the bit names need to be unique for the compiler to handle them, all bits are prefixed with the module type they belong to. In many cases, the module type name is abbreviated. For all bit definitions related to the Timer/Counter Type A module, the bit names are prefixed by TCA_.

To differentiate between bit masks and bit positions, a suffix is also appended. For a bit mask, the suffix is _bm, and for a bit position it is _bp. The name of the bit mask for the INITDLY0 bit is thus ADC_INITDLY0_bm, and the name of the bit position is ADC_INITDLY0_bp. Additionally, the header file provides definitions for group positions. The suffix for a group position is _gp, and the name of the INITDLY group position mask, for example, is ADC_INITDLY_gp. The code below shows the definitions of the INITDLY bit masks, bit positions, and group positions, as they are available in the device header file.

```
#define ADC_INITDLY_gp  5        /* Initial Delay Selection group position */
#define ADC_INITDLY0_bm  (1<<5)  /* Initial Delay Selection bit 0 mask */
#define ADC_INITDLY0_bp  5       /* Initial Delay Selection bit 0 position */
#define ADC_INITDLY1_bm  (1<<6)  /* Initial Delay Selection bit 1 mask */
#define ADC_INITDLY1_bp  6       /* Initial Delay Selection bit 1 position */
#define ADC_INITDLY2_bm  (1<<7)  /* Initial Delay Selection bit 2 mask */
#define ADC_INITDLY2_bp  7       /* Initial Delay Selection bit 2 position */
```

A naming convention example for the INITDLY0 bit mask is presented in Figure 2-1.

Figure 2-1. Naming Convention of Bit Masks



2.3.2 Bit Field Masks (Group Masks)

Many functions such as clock selection for timers, prescaler selection for converters, or filter selection for the configurable custom logic are configured by a field of bits, referred to as a bit field or a bit group. The value of the bits in a group selects a specific configuration.

The masks for configuring a bit field are referred to as a bit group masks, or group configuration masks.

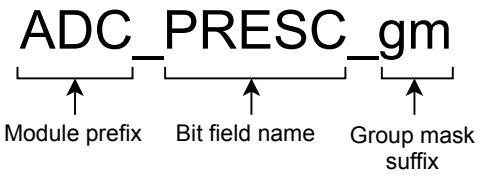
When changing bits in a bit field, it is not enough to set the bits for the desired configuration; it is also required to clear the bits from the old configuration before assigning a new value. To make this easy, a bit group mask is defined. The group mask uses the same name as the bits in the bit field and is suffixed _gm.

The code below shows how the ADC prescaler group mask is defined in the header file.

```
#define ADC_PRESC_gm  0x07  /* Clock the prescaler group mask */
```

The naming convention is presented in Figure 2-2.

Figure 2-2. Naming Convention of Group Masks



The bit group mask is primarily intended to clear the old configuration of a bit field before writing a new value. The code below shows how this can be done. The code will clear the PRESC bit field in the CTRLC register of the ADC0 module. This construction does not set a configuration. It only sets all the prescaler configuration bits. This is an advantage because there is no need to use all the bit masks to reset a specific configuration; they only need a group mask for this operation. The group mask will typically be used in conjunction with a group configuration mask to clear a particular configuration.

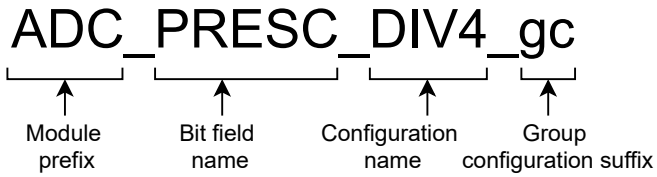
```
ADC0.CTRLC &= ~(ADC_PRESC_gm); /* Clearing the prescaler bit field using a group mask */
```

2.3.3 Group Configuration Masks and Enumerators

It is often required to consult the data sheet to check the bit pattern to be used when setting a bit field to the desired configuration. This also applies when reading or debugging code. To increase the readability and to minimize the possibility of setting bits in bit fields incorrectly, several group configuration masks are defined in the header file. Each group configuration mask selects a configuration for a specific group mask.

The name of a group configuration mask is a concatenation of the module type, the bit field name, a description of the configuration and a suffix, _gc, indicating that this is a group configuration. An example of the ADC prescaler configuration is presented in Figure 2-3.

Figure 2-3. Naming Convention of Group Configuration Masks



The group configuration presented in Figure 2-3 sets the prescaler from the peripheral clock (CLK_PER) to the ADC clock with the division factor 4.

The ADC prescaler bit field consists of three bits that define the division factor. The possible configuration names are DIV2, DIV4, DIV8, DIV16, DIV32, DIV64, DIV128, and DIV256. These names make writing and maintaining code very easy, as it requires very little effort to understand what configuration the specific mask selects. Table 2-2 shows the available configurations for this bit field.

Table 2-2. PRESC Bits and Corresponding Bit Group Configurations

PRESC2	PRESC1	PRESC0	Division Factor	Group Configuration Mask
0	0	0	CLK_PER divided by 2	ADC_PRESC_DIV2_gc
0	0	1	CLK_PER divided by 4	ADC_PRESC_DIV4_gc
0	1	0	CLK_PER divided by 8	ADC_PRESC_DIV8_gc
0	1	1	CLK_PER divided by 16	ADC_PRESC_DIV16_gc
1	0	0	CLK_PER divided by 32	ADC_PRESC_DIV32_gc
1	0	1	CLK_PER divided by 64	ADC_PRESC_DIV64_gc
1	1	0	CLK_PER divided by 128	ADC_PRESC_DIV128_gc
1	1	1	CLK_PER divided by 256	ADC_PRESC_DIV256_gc

To change a bit field to a new configuration, the bit group configuration is typically used in conjunction with the bit group mask, to ensure that the old configuration is cleared first.

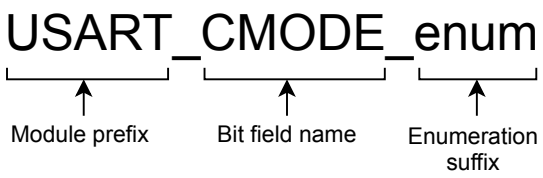
Unlike bit masks and group masks, the bit group configuration masks are defined using C enumerations. One enumeration is defined for each bit field. The enumeration for the USART CMODE bit field is shown in the code below.

```
typedef enum USART_CMODE_enum
{
    USART_CMODE_ASYNCHRONOUS_gc = (0x00<<6),  /* Asynchronous mode */
    USART_CMODE_SYNCHRONOUS_gc = (0x01<<6),   /* Synchronous mode */
    USART_CMODE_IRCOM_gc = (0x02<<6),         /* Infrared communication */
    USART_CMODE_MSPI_gc = (0x03<<6),          /* Master SPI mode */
} USART_CMODE_t;
```

The name of the enumeration is a concatenation of the module type (USART), the bit field (CMODE), and a suffix (_enum).

The naming convention is presented in Figure 2-4.

Figure 2-4. Naming Convention of Enumerations



Each of the enumeration constants behaves much like a normal constant when used on its own. However, using an enumeration type definition has the advantage of creating a new data type. The name of the data type, for this example, is `USART_CMODE_t`. A `USART_CMODE_t` variable can be used directly as an integer, but assigning an integer to an enumeration type will trigger a compiler warning. This can be used to the programmer's advantage.

For example, if a function that sets the communication mode for a USART module accepts the communication mode as an integer type (for example, `unsigned char`), any legal or illegal value can be passed to the function. If the function instead accepts a parameter of type `USART_CMODE_t`, only the four predefined constants in the `USART_CMODE_t` enumeration type can be passed to the function. Passing anything else will result in a compiler warning.

Note: It is important to notice that the constants in the code listing are already shifted to their bit position. The enumeration constants are the actual values to be written in the register, and no additional shifting is needed.

3. Writing Bare Metal C-Code for AVR®

The following subsections focus on how to write C-code for AVR. The examples describe how to make the code readable and portable between different AVR devices. The examples can also be used as a guideline on how to write code that is easy to verify and maintain.

The microcontroller modules are located in dedicated and continuous blocks in the memory space and can be seen as encapsulated units. The modules are encapsulated in C structures, in which all module registers are contained.

This document introduces a naming convention and register access methods that are compliant with the AVR header files, providing readability and portability to the codes written in the C-code.

3.1 Set, Clear and Read Register Bits

Setting and clearing register bits are fundamental operations used in embedded programming and represent a technique any application is based on.

The Read-Modify-Write operations are a class of atomic operations. They read a memory location and simultaneously write a new value to it, either with a completely new value or a part of the previous value.

As it has wide applicability, reading the value of a bit is mainly used in conditional expressions (e.g., `if` statement) and as a condition in loop expressions (e.g., `while` statement). A common use case of this technique is polling on an interrupt flag, which means reading the value of the bit and executing a set of instructions if the bit is set/clear.

Note: For further details on binary arithmetic, refer to the Bitwise Operators.

3.1.1 Set, Clear and Read Register Bits Using Bit Masks

The register bits can be set, cleared and read using the bit masks provided by the header file.

Access the registers using the structure declaration

To set a specific bit in a register, the recommended coding style is to use the structure instance declared in the header file and bit masks macro definitions. Using the binary OR operator with the bit mask will ensure that the other bit settings made inside that register will remain the same, unaffected by the operation.

```
ADC0.CTRLA |= ADC_ENABLE_bm; /* Enable the ADC peripheral */
```

To clear a bit from a register, the binary AND operator will be applied between the register and the negated value of the bit mask. This operation also keeps the other bit settings unchanged.

```
ADC0.CTRLA &= ~ADC_ENABLE_bm; /* Disable the ADC peripheral */
```

The `ADC_ENABLE_bm` bit mask is defined in the header file, as presented below.

```
#define ADC_ENABLE_bm  0x01  /* ADC Enable bit mask */
```

Reading a bit value is necessary for various applications. For example, the ADC RESRDY flag is set by hardware when an ADC result is ready. The code listing below shows how to test if this bit is set and execute some instructions in this case.

```
if(ADC0.INTFLAGS & ADC_RESRDY_bm) /* Check if the ADC result is ready */
{
    /* Insert some instructions here */
}
```

To test if a bit is clear, and execute instructions or wait while it remains clear, the following code can be used. In this case, the instructions inside this loop will be executed until the ADC result is ready.

```
while(!(ADC0.INTFLAGS & ADC_RESRDY_bm))
{
    /* Insert some instructions here */
}
```

Access the registers using the macro definition
Alternatively, the registers can be accessed also using the macro definitions from the header file. The example below shows how to set a bit using these definitions.

```
ADC0_CTRLA |= ADC_ENABLE_bm; /* Enable the ADC peripheral */
```

To clear a bit using macro definitions, the following code example can be used.

```
ADC0_CTRLA &= ~ADC_ENABLE_bm; /* Disable the ADC peripheral */
```

The code listing below shows how to test if this bit is set and to execute some instructions in this case.

```
if(ADC0_INTFLAGS & ADC_RESRDY_bm) /* Check if the ADC result is ready */
{
    /* Insert some instructions here */
}
```

To test if a bit is clear, and execute instructions while it remains clear, the following code can be used. In this case, the instructions inside this loop will be executed until the ADC result is ready.

```
while(!(ADC0_INTFLAGS & ADC_RESRDY_bm))
{
    /* Insert some instructions here */
}
```

3.1.2 Set, Clear and Read Register Bits Using Bit Positions

An alternative way to set, clear and read register bits is by using bit positions, also provided by the device header file.

Access the registers using the structure declaration
To set a bit using a bit position macro and access the registers using the structure declaration, the following code example can be used.

```
ADC0.CTRLA |= (1 << ADC_ENABLE_bp); /* Enable the ADC peripheral */
```

To clear the bit using a bit position macro, the following example is provided.

```
ADC0.CTRLA &= ~(1 << ADC_ENABLE_bp); /* Disable the ADC peripheral */
```

To test if a bit is set, the following code can be used.

```
if(ADC0.INTFLAGS & (1 << ADC_RESRDY_bp)) /* Check if the ADC result is ready */
{
    /* Insert some instructions here */
}
```

To test if a bit is clear, and execute instructions while it remains clear, the following code can be used. In this case, the instructions inside this loop will be executed until the ADC result is ready.

```
while(!(ADC0.INTFLAGS & (1 << ADC_RESRDY_bp)))
{
    /* Insert some instructions here */
}
```

Access the registers using the macro definition
To set a bit using a bit position macro and access the registers using the macro definitions, the following code example can be used.

```
ADC0_CTRLA |= (1 << ADC_ENABLE_bp); /* Enable the ADC peripheral */
```

To clear the bit using a bit position macro, the following example is provided.

```
ADC0_CTRLA &= ~(1 << ADC_ENABLE_bp); /* Disable the ADC peripheral */
```

To test if a bit is set, the following code can be used.

```
if(ADC0_INTFLAGS & (1 << ADC_RESRDY_bp)) /* Check if the ADC result is ready */
{
    /* Insert some instructions here */
}
```

To test if a bit is clear, and execute instructions while it remains clear, the following code can be used. In this case, the instructions inside this loop will be executed until the ADC result is ready.

```
while(!(ADC0_INTFLAGS & (1 << ADC_RESRDY_bp)))
{
    /* Insert some instructions here */
}
```

3.2 Register Initialization

To initialize a register, the user has to find the desired configuration by consulting the device data sheet. Then, the register bits must be set or cleared so that the value in the register matches the desired configuration.

Register initialization is most often performed as part of device initialization after Reset, when the register is in a known Reset state (default).

For most AVR registers, the Reset value for all bits and bit fields is '0'. The Reset value of a register can be found, as shown in Figure 3-1, along with all the register's bits configurations desired for this example.

Figure 3-1. ADC Control A Register - Reset Value and Bit Settings

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

Read-Modify-Write operations are not needed when working with bit masks or bit positions if the Reset value of the register is 0x00, and the register can be configured in a single line.

The desired configuration can be, for example:

- ADC is enabled – the ENABLE bit will be '1'
- ADC is operating in Differential mode – the CONVMODE bit will be '1'
- ADC will run with 10-bit resolution – the RESSEL bit field value will be 0x00 (default)

This configuration can be implemented by writing the resulting value directly to the register, using either the binary, hexadecimal, or decimal value, as presented below.

```
ADC0.CTRLA = 0b00100001;   /* binary */
ADC0.CTRLA = 0x21;         /* hexadecimal */
ADC0.CTRLA = 33;           /* decimal */
```

However, to improve the readability (and potentially the portability) of the code, it is recommended to use the device defines, which are shown in the upcoming sections.

Note: For AVR registers, the Reset value for most bits and bit fields is '0', but there are exceptions. For example, the USART0 Control C register has a couple of bits with the Reset value '1'. In this case, the user must explicitly set the desired configuration without relying on the fact that the Reset values of the bits are usually '0'.

Figure 3-2. USART Control C Register – Reset Value

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

The USART0 Control C register has a Reset value of 0x03. In this case, a Read-Modify-Write operation is needed to initialize one of the other bits or bit fields without changing the value of the CHSIZE bit field. The register is presented in Figure 3-2.

3.2.1 Register Initialization Using Bit Masks and Group Configuration Masks

This subsection provides the recommended way to configure the ADC CTRLA register using bit masks and group configuration masks.

```
ADC0.CTRLA = ADC_ENABLE_bm          /* Enable the ADC */
           | ADC_CONVMODE_bm        /* Select Differential Conversion mode */
           | ADC_RESSEL_10BIT_gc;   /* 10-bit conversion */
```

Note the absence of the bitwise OR ('|') on the register side of the assignment. In most cases, device and peripheral routines are written in this way.

The ADC_RESSEL_enum enumeration contains the group configuration mask presented below.

```
ADC_RESSEL_10BIT_gc = (0x01<<2),  /* 10-bit mode */
```

⚠ CAUTION The above initialization of the register must be done in a single line of C-code. Writing as follows, the group configuration in the second line will clear the bit set in the first line.

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

The correct way of writing this code using three code lines is presented below.

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

Note: Bit masks can only set bits in a single line of code, so any configurations which require bits to be set to '0' can be left out since they are correctly configured by their Reset value.

3.2.2 Register Initialization Using Bit Positions

The same configuration, as presented above, can be done by using the bit position macros, as follows.

```
ADC0.CTRLA = (1 << ADC_ENABLE_bp)     /* Enable the ADC */
           | (1 << ADC_CONVMODE_bp)   /* Select Differential Conversion mode */
           | (1 << ADC_RESSEL0_bp)    /* 10-bit conversion */
           | (0 << ADC_RESSEL1_bp);
```

Note: The `(0 << ADC_RESSEL0_bp)` line is added simply for readability, but it can be removed.

```
ADC0.CTRLA = (1 << ADC_ENABLE_bp)     /* Enable the ADC */
           | (1 << ADC_CONVMODE_bp)   /* Select Differential Conversion mode */
           | (1 << ADC_RESSEL0_bp);   /* 10-bit conversion */
```

Other position masks that can be used to configure bit fields are group position masks. They can be used as presented below. The desired configuration value must be shifted with the bit field position (group position).

```
ADC0.CTRLA = (1 << ADC_ENABLE_bp)      /* Enable the ADC */
           | (1 << ADC_CONVMODE_bp)    /* Select Differential Conversion mode */
           | (0x01 << ADC_RESSEL_gp);  /* 10-bit conversion */
```

3.3 Change Register Bit Field Configurations

This section covers considerations when updating a register bit field, using various header file defines. The RXMODE bit field will be used as an example, where an update is made compared to the initialization presented below.

```
USART0.CTRLB = USART_RXEN_bm            /* Receiver Enable */
             | USART_TXEN_bm            /* Transmitter Enable */
             | USART_RXMODE_LINAUTO_gc; /* LIN Constrained Auto-Baud mode */
```

The following subsections will provide code examples to change the Receiver Mode (RXMODE) configuration to Generic Auto-Baud (GENAUTO) mode. The available configurations for this bit field, as they are in the device data sheet, are presented in the table below.

Figure 3-3. USART0 Receiver Mode Configurations

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 Change Register Bit Field Configurations Using Group and Group Configuration Masks

When updating only a bit field in a register, a Read-Modify-Write operation must be used. Therefore, to change the configuration of a register bit field, it is recommended to first clear the old configuration and then set a new one.

It is recommended using a bit group configuration mask to set a configuration in a bit field.

One way to change a bit field to a new configuration is presented in the code listing below. To be sure that the desired configuration is obtained, the user must clear the old configuration first, using the group mask.

```
/* Changing a bit group configuration */
USART0.CTRLB &= (~USART_RXMODE_gm); /* Clear the old configuration */
/* Set the new configuration using the group configuration mask */
USART0.CTRLB |= USART_RXMODE_GENAUTO_gc;
```

Note: The first line will put the USART0 RXMODE, for a short time, into a specific state: `0x00`.

A group mask macro is defined in the header file, as presented below.

```
#define USART_RXMODE_gm 0x06 /* Receiver mode group mask */
```

The `USART_RXMODE_enum` enumeration contains the group configuration mask presented below.

```
USART_RXMODE_GENAUTO_gc = (0x02<<1), /* Generic Auto-Baud mode */
```

⚠ CAUTION Even though it may seem easier to split the code into two separate code lines, one to clear the register and another one to set the desired configuration, it is recommended to use a single line to do this, as presented in the code listing below.

```
/* Changing a bit group configuration */
USART0.CTRLB = (USART0.CTRLB & ~USART_RXMODE_gm) | USART_RXMODE_GENAUTO_gc;
```

These steps must be implemented in a single line to avoid putting the microcontroller in an unintended state.

The CTRLB register is declared this way:

```
register8_t CTRLB;
```

The `register8_t` data type is a volatile defined data type.

```
typedef volatile uint8_t register8_t;
```

Since the register is defined as volatile, two different code lines will trigger the reading and writing in the CTRLB register twice instead of once. In addition to making the code inefficient, this can also put the peripheral in an unintended state. The reason is that when an interrupt is triggered between the two lines of code, the context will be changed, and the register can remain in an undefined state.

3.3.2 Change Register Bit Field Configurations Using Bit Masks

The example below shows how to update a register bit field using bit masks to set the new configuration. The current configuration is cleared, and the new configuration is set, in a single line of code.

```
USART0.CTRLB = (USART0.CTRLB & ~(USART_RXMODE0_bm | USART_RXMODE1_bm))
             | USART_RXMODE1_bm; /* Generic Auto-Baud mode */
```

3.3.3 Change Register Bit Field Configurations Using Bit Positions

The example below shows how to update a register bit field using bit positions to set the new configuration. The current configuration is cleared, and the new configuration is set, in a single line of code.

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

Note: The `(0 << USART_RXMODE0_bp)` line is added simply for readability, but it can be removed.

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

3.4 Advantages of Using Bit Masks and Group Configuration Masks

The advantages of using the recommended coding style are:

- Obtaining a more readable code. By using the group configuration mask, the user will know for sure what configuration is being set – the configuration name being contained in the mask name.
- Obtaining a more compact code. The group mask will help the user clear all the bits in a bit field, without the need of using a bit mask for each bit. The configuration mask can also be used to configure the entire bit field.
- The code is easier to maintain. For example, to change a bit field configuration, the user will only have to change the group configuration mask name instead of changing each bit value using bit masks or bit positions.

3.5 Writing to Configuration Change Protection (CCP) Registers

Attempting to write to a protected register without following the appropriate CCP unlock sequence leaves the protected register unchanged. The sequence, specified in the device data sheet, typically involves writing a signature to the CPU.CCP register and then, within four instructions, writing the desired value to the protected register.

The CCP signatures are provided by the device header file, as presented below.

```
/* CCP signature select */
typedef enum CCP_enum
{
    CCP_SPM_gc = (0x9D<<0),    /* SPM instruction protection */
    CCP_IOREG_gc = (0xD8<<0),  /* I/O register protection */
} CCP_t;
```

The following example shows how to write to a register under CCP using an IOREG signature. The main clock prescaler division will be set to 16.

```
CCP = CCP_IOREG_gc; /* Write the needed signature to CCP*/
CLKCTRL.MCLKCTRLB = CLKCTRL_PDIV_16X_gc; /* Set the prescaler division to 16 */
```

The following example shows how to write to a register under CCP using the SPM signature.

```
CPU.CCP = CCP_SPM_gc; /* Write the needed signature to CCP*/
NVMCTRL.CTRLA = NVMCTRL_CMD_FLPER_gc; /* Flash Page Erase Enable*/
```

⚠ CAUTION These are not the recommended ways of writing to registers under CCP. To write to registers protected by CCP, after writing the signature to the CCP register, the software must write the desired data to the protected register within four instructions. To meet the timing requirements, writes to CCP registers are often handled by assembly code.

Thus, the recommended way of writing to a protected I/O register is by using the `ccp_write_io` function. To use this function, the following header file must be included.

```
#include <avr/cpufunc.h> /* Required header file */
```

The following code example provides the same functionality as the example presented above, but using the `ccp_write_io` function.

```
/* Set the prescaler division to 16 */
ccp_write_io((void *) & (CLKCTRL.MCLKCTRLB), (CLKCTRL_PDIV_16X_gc));
```

Alternatively, a macro is defined to write a value to a CCP register. This is protected through the XMEGA® CCP mechanism, which implements the timed sequence that is required for CCP.

Note: Since the CCP registers were introduced with the XMEGA family of AVR devices, the header file that contains `_PROTECTED_WRITE` and `_PROTECTED_WRITE_SPM` macros is `xmega.h`, which must be included as shown below.

```
#include <avr/xmega.h> /* Required header file */
```

These macros must be used to write to the desired registers, as presented in the code examples below.

```
/* Select the 32 kHz internal Ultra-Low Power oscillator */
_PROTECTED_WRITE(CLKCTRL.MCLKCTRLA, CLKCTRL.MCLKCTRLA | CLKCTRL_CLKSEL_OSCULP32K_gc);
/* Write page command */
_PROTECTED_WRITE_SPM(NVMCTRL.CTRLA, NVMCTRL_CMD_PAGEWRITE_gc);
```

Note: Both the XC8 Compiler (MPLAB X IDE) and GCC (Atmel Studio 7) support the usage of these macros.

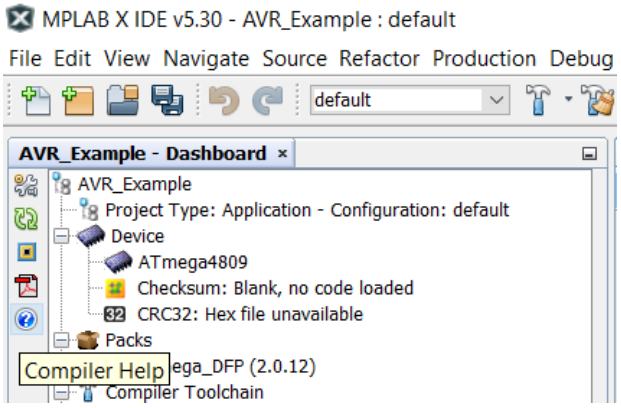
3.6 Configuring Fuses

The fuses are preprogrammed but the fuse registers can be altered by the user. A fuse can be changed only by programming it. However, some fuse values are loaded into registers and these register values can be changed during run-time. The fuses can be configured by writing C-code, as described in the following subsections.

3.6.1 Configuring Fuses Using XC8 Configuration Bits

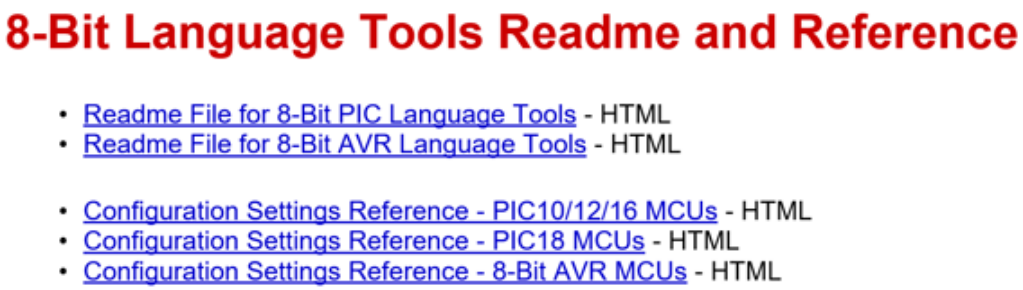
To configure the fuses with the MPLAB X IDE, configuration pragmas can be used. More information about the compiler and the configuration bits of the desired device can be found by accessing the Compiler Help – the blue question mark from the MPLAB X IDE project dashboard, as presented in Figure 3-4.

Figure 3-4. Accessing Compiler Help



The configuration settings for all supported devices are presented at *Configuration Settings Reference → 8-bit AVR MCUs*, as shown in Figure 3-5.

Figure 3-5. Compiler Help → 8-bit Language Tools Readme and Reference



The configuration pragmas can be used as presented below.

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

The following example shows how to disable the Watchdog Timer and the CRC and to configure the start-up time to be 8 ms, using configuration pragmas.

```
/* Disable Watchdog Timer */
#pragma config PERIOD = PERIOD_OFF
/* Disable CRC and set the Reset Pin Configuration to GPIO mode */
#pragma config CRCSRC = CRCSRC_NOCRC, RSTPINCFG = RSTPINCFG_GPIO
/* Start-up Time select: 8 ms */
#pragma config SUT = SUT_8MS
```

3.6.2 Configuring Fuses Using AVR® LibC

To configure the fuses using Atmel Studio, the `fuse.h` header file must be included, as presented below.

```
#include <avr/fuse.h> /* Required header file */
```

The following example shows how to disable the Watchdog Timer using the Watchdog Configuration (WDTCFG) fuse register, disable the CRC and set the Reset Pin Configuration to GPIO mode using the System Configuration 0 (SYSCFG0) register, and configure the start-up time to 8 ms using the System Configuration 1 (SYSCFG1) register.

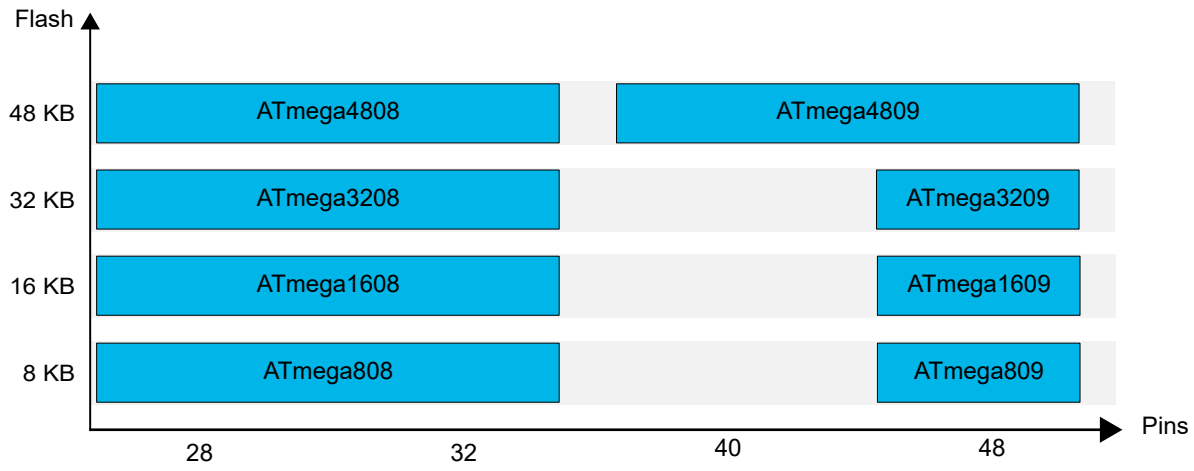
```
FUSES = {
/* Disable Watchdog Timer */
.WDTCFG = PERIOD_OFF_gc,
/* Disable CRC and set the Reset Pin Configuration to GPIO mode */
.SYSCFG0 = CRCSRC_NOCRC_gc | RSTPINCFG_GPIO_gc,
/* Start-up Time select: 8 ms */
.SYSCFG1 = SUT_8MS_gc
};
```

⚠ CAUTION If not initialized by the user, the fuses will be initialized with default values ('0') when using AVR LibC. If there are fuses not initialized which must be different from '0', the device may not work as expected.

3.7 Function Calls Using Module Pointers

When writing drivers for module types that have multiple instances, the fact that all instances have the same register memory map can be utilized to make the driver reusable for all instances of the module type. If the driver takes a pointer argument pointing to the relevant module instance, the driver can be used for all modules of this type. This represents a great advantage when considering portability. Moreover, the written code may be portable between devices of the same family. Details on the compatibility between devices from the same family are provided in the data sheet's series *Overview* section, and some differences are shown in Figure 3-6.

Figure 3-6. megaAVR® 0-series Overview(1)



In a device with multiple timers/counters, the functions to initialize and access these modules can be shared by all module instances, instead of replicating the same lines of code for every instance. Even though there is a small overhead in passing the module pointer to the functions, the total code size will be reduced because the code is reused for all instances of each module type. Moreover, development time, maintenance cost, and portability can be greatly improved by using this approach.

The code below shows a function that uses a module pointer to select a clock source for any timer/counter module.

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

The function takes two arguments: A module pointer of type `TCB_t` and a group configuration type `TCB_CLKSEL_t`. The code in the function uses the timer/counter module pointer to access the CTRLA register and set a new clock selection for the timer/counter module with the address provided through the `tc` parameter. The code below shows how the function described above is used to set different configurations for different timer/counter instances.

```
/* Configure the TCB0 clock selection as CLKDIV2*/
TCB_ConfigClockSource (&TCB0, TCB_CLKSEL_CLKDIV2_gc);
/* Configure the TCB0 clock selection as CLKDIV1*/
TCB_ConfigClockSource (&TCB0, TCB_CLKSEL_CLKDIV1_gc);
/* Configure the TCB1 clock selection as CLKDIV2*/
TCB_ConfigClockSource (&TCB1, TCB_CLKSEL_CLKDIV2_gc);
/* Configure the TCB2 clock selection as CLKDIV2*/
TCB_ConfigClockSource (&TCB2, TCB_CLKSEL_CLKDIV2_gc);
```

4. Application Example Showing Alternative Ways of Writing Code

For convenience and for maintaining compatibility with older versions of AVR coding styles, it is still possible to use a programming style that does not involve structures. This section briefly describes alternative ways of accessing registers and using bit names.

4.1 Register Names

It is possible to access any register without using the module structures. To refer to a register directly, concatenate the module instance name, then add an underscore and the register name. The same naming convention is used when programming in assembly.

For example, to access the CTRLA register of Timer/Counter type B, instance 0, the name `TCB0_CTRLA` can be used, instead of the access through the structures.

4.2 Bit Positions

It is possible to use bit masks to set or clear bits. A bit's position within a register is defined using the same name as the bit mask, but a different suffix. The code below shows how the bit position can be used to configure a register.

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

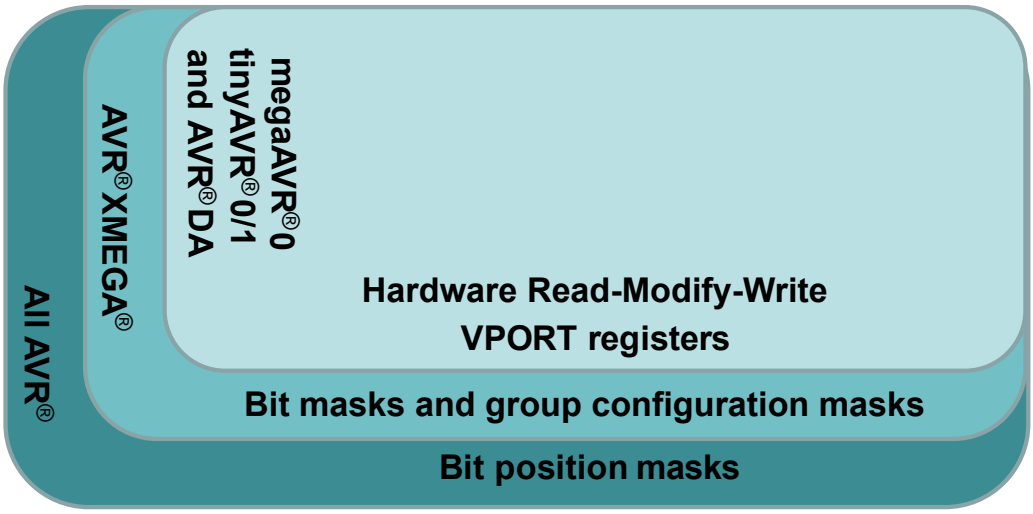
The bit position definitions are included for compatibility reasons. They are also needed when programming in assembly for instructions that use a bit number.

4.3 Virtual Ports

Virtual port (VPORT) registers allow some PORT registers to be mapped virtually in the bit accessible I/O memory space. When this is done, writing to the VPORT register will be the same as writing to the real PORT register. This enables the use of I/O memory specific instructions, such as bit manipulation instructions (`SBI`/`CBI`), on a PORT register that resides in the extended I/O memory space. The number of available virtual ports is, in most cases, limited to four or six, resulting in a smaller number of available virtual ports than the number of PORT modules. The advantages of using virtual ports are a smaller and less complex program and less Flash memory used.

A compatibility diagram is presented in Figure 4-1, which explains what coding styles are available for specific AVR product families.

Figure 4-1. AVR® Family Register Configuration Options



4.4 PORT Example

This subsection contains an example on how to configure a PORT to turn on an LED when pressing a user button. To identify which pins of the microcontroller are routed to the user LED and to the user button, the used board schematic is necessary. For the AVR128DA48 Curiosity Nano board used for this example, the LED is connected to the sixth pin of PORTC, PC6, and the button is connected to the seventh pin of PORTC, PC7. To make sure this is the correct configuration of the pins, the user must check the board schematic.

The code below shows how to write code for turning on and off an LED, using bit position macros.

Example 4-1. Turn LED On when Button Pressed Using Bit Positions

```
/* Direction configuration of the pins */
/* Read-Modify-Write: Software */
PORTC.DIR |= (1 << PIN6_bp);               /* Output for the user LED */
PORTC.DIR &= ~(1 << PIN7_bp);              /* Input for the user button */
PORTC.PIN7CTRL |= (1 << PORT_PULLUPEN_bp); /* The pull-up configuration */
while (1)
{
    if(PORTC.IN & (1 << PIN7_bp))          /* Check the button state */
    {
        /* The button is released */
        PORTC.OUT |= (1 << PIN6_bp);       /* Turn off the LED */
    }
    else
    {
        /* The button is pressed */
        PORTC.OUT &= ~(1 << PIN6_bp);      /* Turn on the LED */
    }
}
```

Note: This is the coding style used by Atmel START for peripheral configuration.

The code below provides the same functionality but using the bit masks.

Example 4-2. Turn LED On when Button Pressed Using Bit Masks

```
/* Direction configuration of the pins */
/* Read-Modify-Write: Software */
PORTC.DIR |= PIN6_bm;               /* Output for the user LED */
PORTC.DIR &= ~ PIN7_bm;             /* Input for the user button */
PORTC.PIN7CTRL |= PORT_PULLUPEN_bm; /* The pull-up configuration */
while (1)
{
    if(PORTC.IN & PIN7_bm)          /* Check the button state */
    {
        * The button is released */
        PORTC.OUT |= PIN6_bm;       /* Turn off the LED */
    }
    else
    {
        /* The button is pressed */
        PORTC.OUT &= ~(PIN6_bm);    /* Turn on the LED */
    }
}
```

Note: This is the coding style used by MCC when generating code for AVR.

Also, the SET and CLR registers can be used to set and clear the pin value without Read-Modify-Write operations, as shown in the code below. This allows performing the setting and the clearing of the desired value using an atomic instruction, instead of reading the register value first. The main advantage is that this action cannot be interrupted. Only one instruction is executed instead of three (Read-Modify-Write).

Example 4-3. Turn LED On when Button Pressed Using SET and CLR Registers

```
PORTC.DIRSET = PIN6_bm;              /* Output for the user LED */
PORTC.DIRCLR = PIN7_bm;              /* Input for the user button */
PORTC.PIN7CTRL |= PORT_PULLUPEN_bm; /* The pull-up configuration */

while (1)
{
    /* Read-Modify-Write: Hardware */
    if(PORTC.IN & PIN7_bm)           /* Check the button state */
    {
        /* The button is released */
        PORTC.OUTSET = PIN6_bm;      /* Turn off the LED */
    }
    else
    {
        /* The button is pressed */
        PORTC.OUTCLR = PIN6_bm;      /* Turn on the LED */
    }
}
```

Another way to configure the direction and set/clear the value of the output pin is by using virtual ports, as presented in the code below.

Example 4-4. Turn LED On when Button Pressed Using Virtual Ports

```
/* Direction configuration of the pins */
/* Read-Modify-Write: VPORT registers */
VPORTC.DIR |= PIN6_bm;               /* Output for the user LED */
VPORTC.DIR &= ~PIN7_bm;              /* Input for the user button */
PORTC.PIN7CTRL |= PORT_PULLUPEN_bm; /* The pull-up configuration */
while (1)
{
    if(VPORTC.IN & PIN7_bm)          /* Check the button state */
    {
        /* The button is released */
        VPORTC.OUT |= PIN6_bm;       /* Turn off the LED */
    }
    else
    {
        /* The button is pressed */
        VPORTC.OUT &= ~PIN6_bm;      /* Turn on the LED */
    }
}
```

4.5 ADC Example

This section presents a simple configuration example for the ADC0 module on an AVR128DA48 device from the AVR DA series of devices. The information needed to program this controller can be found in the device data sheet, as mentioned in Section 1: Data Sheet Structure and Naming Conventions.

To fully configure the ADC, the user can follow the recommended initialization steps that can be found in the *Functional Description* subsection from the *ADC – Analog-to-Digital Converter* section of the data sheet.

After the ADC prescaler is configured, the ADC module is enabled, and a conversion is started. In the code below, the configurations are made using the bit position macros.

Example 4-5. Configure ADC Using Bit Positions

```
    /* ADC register configuration using bit position macros */
    ADC0.CTRLC |= (1 << ADC_PRESC0_bp) | (1 << ADC_PRESC1_bp); /* Configuring
the prescaler bits. Configuration: DIV8 */
    ADC0.CTRLA |= (1<< ADC_ENABLE_bp);          /* Enabling the ADC */
    ADC0.COMMAND |= (1<< ADC_STCONV_bp);        /* Starting a conversion */
```

Alternatively, the ADC can be configured using the bit masks, as in the code below.

Example 4-6. Configure ADC Using Bit Masks

```
    /* ADC register configuration using bit masks macros */
    ADC0.CTRLC |= ADC_PRESC0_bm | ADC_PRESC1_bm;              /* Configuring
the prescaler bits. Configuration: DIV8 */
    ADC0.CTRLA |= ADC_ENABLE_bm;               /* Enabling the ADC */
    ADC0.COMMAND = ADC_STCONV_bm;              /* Starting a conversion */
```

To change the prescaler configuration, a group configuration mask can be used. The original configuration must be cleared first, as shown in the code below.

Example 4-7. Change the ADC Prescaler Configuration Using a Group Configuration Mask

```
    /* Changing an ADC prescaler configuration, ensuring to clear the original
configuration */
    ADC0.CTRLC = (ADC0.CTRLC & ~ADC_PRESC_gm) | ADC_PRESC_DIV4_gc;
```

5. Further Steps

This section has the purpose to direct the user to the IDE installation guides and instructions, and the available application notes.

5.1 Application Notes and Technical Briefs Description

This series of technical briefs utilizes the same principles recommended in this document, covering all peripherals for the megaAVR® 0-series family of AVR microcontrollers. Each technical brief starts with a summary of the use cases covered. Each use case is then developed, showing how to use the data sheet to configure the peripheral in the required configuration.

For example, the Getting Started with ADC technical brief provides an overview of the peripheral and a description on how to use the peripheral in several use cases in different operation modes: single conversion, free-running, sample accumulator, etc.

- TB3209 - Getting Started with ADC
- TB3211 - Getting Started with AC
- TB3213 - Getting Started with RTC
- TB3214 - Getting Started with TCB
- TB3215 - Getting Started with SPI
- TB3216 - Getting Started with USART
- TB3217 - Getting Started with TCA
- TB3218 - Getting Started with CCL
- TB3229 - Getting Started with GPIO

5.2 Relevant Videos for Bare Metal AVR® Development

The following videos are particularly relevant for this technical brief.

- Getting Started with Atmel Studio 7 - Episode 10 - I/O View & Bare Metal Programming References
- Getting Started - AVR® in MPLAB® X - Context Data Sheet Help & AVR Interrupts

The following is a 28-part video series, which builds up functionality using the data sheet and device header files as primary programming references.

- Getting Started with AVR®

5.3 MPLAB® XC8 Compiler

The XC compilers are comprehensive solutions for the software development of any suitable project. The MPLAB XC8 compiler supports all 8-bit PIC® and AVR microcontrollers[(2)], and it is available as free, unrestricted-use download. A pro license is also available. By using a pro license, the user will obtain a more efficient code. Additionally, a certified XC8 Functional Safety license is now available.

When combined with the MPLAB X IDE, the front-end provides editing errors and breakpoints, that match corresponding lines in the source code, and single-stepping through C source code to inspect variables and structures at critical points.

More information on Microchip's MPLAB X IDE can be found on the user guides page, by searching for "MPLAB X IDE User's Guide".

5.4 IDE (MPLAB® X, Atmel Studio, IAR) – Getting Started

To program AVR microcontrollers, either the MPLAB X, Atmel Studio, or IAR Embedded Workbench IDEs can be used.

MPLAB X Integrated Development Environment (IDE) is an expandable and highly configurable software program. It incorporates powerful tools to help the user discover, configure, develop, debug and qualify embedded designs for most of Microchip's microcontrollers and digital signal controllers. MPLAB X IDE offers support for AVR MCUs.

All the information needed to be familiar with Atmel Studio, including hands-on and video tutorials, is provided in this user's guide: Getting Started with Atmel Studio 7. Additionally, information on all the project configurations needed to develop a project (fuse programming, oscillator calibration, interface setting, etc) can be found in the Atmel Studio 7 user's guide. To find and develop more examples for any board, configure drivers, find example projects, and easily configure system clock settings, the online code configuration tool Atmel START can be used. For more details, refer to the Atmel START User Guide.

The IAR Embedded Workbench for AVR with all the features is described in the AN3419 - Getting Started with IAR Embedded Workbench for AVR application note.

The microcontroller families tinyAVR® 0/1-series, megaAVR 0-series, and AVR DA have also a series of *Getting Started with* dedicated guides available online, with information on how to create a project and what starter kit to use. Examples of these guides are listed below:

- Getting Started with megaAVR® 0-series
- Getting Started with tinyAVR® 1-series Microcontrollers
- Getting Started with tinyAVR® 0-series
- Getting Started with the AVR DA Family

6. Conclusion

This document introduces the user to a preferred coding style for programming the AVR microcontrollers. After going through this document, the user knows what type of information the data sheet is providing, and also the macro definitions, the variable declarations, and the data type definitions provided by the header files. The goals are to use an easily maintainable, portable and readable coding style, to get familiar with the naming conventions for the AVR registers and bits, and to get ready for the further steps in developing a project using these microcontrollers.

This document covers information on specific data sheets, information description, naming conventions, how to write C-code for AVR microcontrollers, alternative ways of writing the code, and, finally, the next steps in developing a project.

Using the suggested methods to write C-code is not mandatory, but the advantages presented here can be considered. The larger the project and the more features the device has, the bigger the advantage.

7. References

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. Revision History

Document Revision	Date	Comments
A	05/2020	Initial document release

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip design partner program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Product Change Notification Service

Microchip's product change notification service helps keep customers current on Microchip products. Subscribers will receive email notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with

your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PackeTime, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TempTrackr, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, FlashTec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, Vite, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, INICnet, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, SAM-ICE, Serial Quad I/O, SMART-I.S., SQI, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, and Symmcom are registered trademarks of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

ISBN: 978-1-5224-6054-1

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Australia - Sydney** Tel: 61-2-9868-6733 **China - Beijing** Tel: 86-10-8569-7000 **China - Chengdu** Tel: 86-28-8665-5511 **China - Chongqing** Tel: 86-23-8980-9588 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 **China - Hong Kong SAR** Tel: 852-2943-5100 **China - Nanjing** Tel: 86-25-8473-2460 **China - Qingdao** Tel: 86-532-8502-7355 **China - Shanghai** Tel: 86-21-3326-8000 **China - Shenyang** Tel: 86-24-2334-2829 **China - Shenzhen** Tel: 86-755-8864-2200 **China - Suzhou** Tel: 86-186-6233-1526 **China - Wuhan** Tel: 86-27-5980-5300 **China - Xian** Tel: 86-29-8833-7252 **China - Xiamen** Tel: 86-592-2388138 **China - Zhuhai** Tel: 86-756-3210040	**India - Bangalore** Tel: 91-80-3090-4444 **India - New Delhi** Tel: 91-11-4160-8631 **India - Pune** Tel: 91-20-4121-0141 **Japan - Osaka** Tel: 81-6-6152-7160 **Japan - Tokyo** Tel: 81-3-6880- 3770 **Korea - Daegu** Tel: 82-53-744-4301 **Korea - Seoul** Tel: 82-2-554-7200 **Malaysia - Kuala Lumpur** Tel: 60-3-7651-7906 **Malaysia - Penang** Tel: 60-4-227-8870 **Philippines - Manila** Tel: 63-2-634-9065 **Singapore** Tel: 65-6334-8870 **Taiwan - Hsin Chu** Tel: 886-3-577-8366 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 **Thailand - Bangkok** Tel: 66-2-694-1351 **Vietnam - Ho Chi Minh** Tel: 84-28-5448-2100	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4485-5910 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-72400 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-72884388 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820