AVR1000: Getting Started Writing C-code for XMEGA

8-bit AVR® Microcontrollers

Application Note

Features

- **Naming conventions**
  - **Register names**
  - **Bit names**
- **C-code names**
  - **Bit and group masks**
  - **Group configuration masks**
- **Methods for accessing registers**
- **Methods for writing reusable module functions**

1 Introduction

Short development times and high quality requirements on electronic products has made high-level programming languages a requirement. The main reason is that High level languages make it easier to maintain and reuse code due to better portability and readability.

The choice of programming language alone does not ensure high readability and reusability; good coding style does. Therefore the XMEGA™ peripherals, header files and drivers are designed with this in mind.

The most widely used high-level language for AVR® microcontrollers is C, and this application note therefore focuses on C programming. To support most of the AVR C compilers that are available the code examples are as far as possible written in ANSI C. A few examples are specific to IAR Embedded Workbench®, but the ideas and methods can be used for other compilers with minor changes. IAR specific examples are clearly marked.

Rev. 8075A-AVR-02/08

2 XMEGA Modules

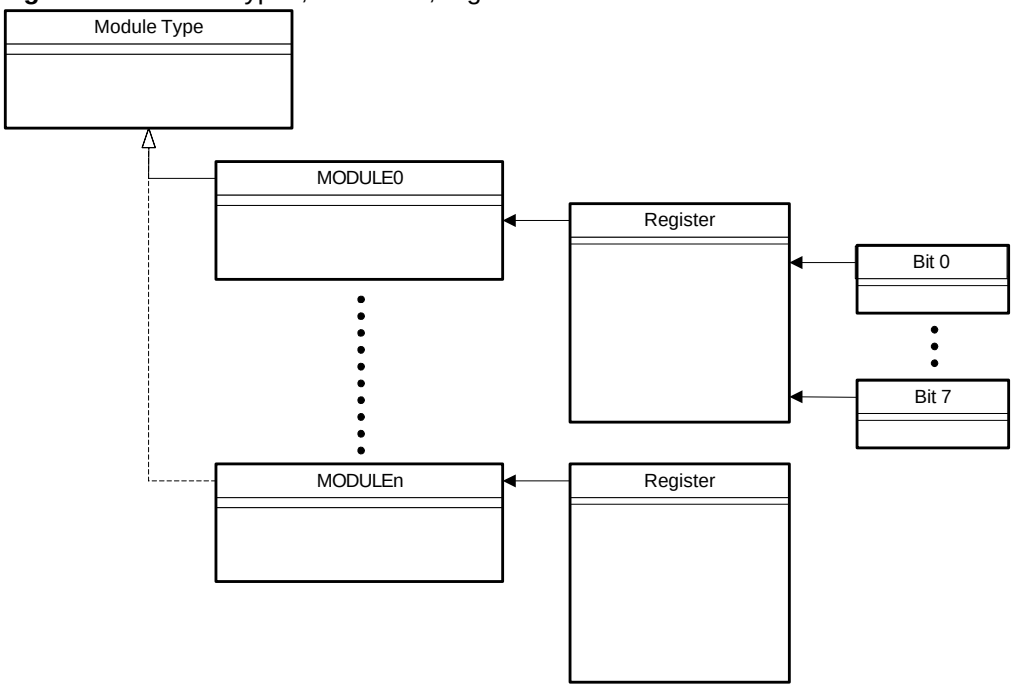
An AVR XMEGA is composed of several building blocks: An AVR CPU core, SRAM, Flash, EEPROM and a number of peripheral modules. These building blocks are called “module types”. An XMEGA can have one or more instances of a given module type. All instances of a module type have the same features and functions.

Some module types can be a subset of other module types. These inherit a subset of the features (and registers) of the super type, all inherited features are fully compatible. This applies to e.g. timers and IO ports. The subset of a module type can for a timer mean that it has fewer compare and capture channels than a full timer module. Similarly, an IO port may have less than eight pins.

A module type can be a “USART”, while the module instance is e.g. “USARTC0”, where the “C0” suffix indicates the instance is “USART number 0 on port C”. 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 number of 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.

Figure 2-1. Module types, instances, registers and bits.



Each module has a fixed base address in the IO 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 IO 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.

2.1 Register Naming Convention

Register are roughly speaking divided into control, status and data registers and the naming of registers reflect this. A general-purpose control register of the module is named CTRL. If multiple general-purpose control registers exists in a module they have a suffix character. In this case the control registers would be named CTRLA, CTRLB, CTRLC and so on. This also applies to STATUS registers.

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

Since the AVR 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 Timer/Counter count register is named CNT. The two bytes are named CNTL and CNTH.

For a register larger than 16 bit, the bytes are numbered from the least significant byte. For example, the 32-bit ADC calibration register is named CAL. The four bytes are named CAL0, CAL1, CAL2 and CAL3 (from least to most significant byte).

Most C compilers offer automatic handling of access to multi-byte registers. In that case the name CNT, without “H” or “L” suffix, could be used to perform a 16-bit access to the Timer/Counter count register. This is also the case for 32-bit registers.

2.2 Bit Naming Convention

Register bits can have an individual function or be part of a bit group that have a joint function: An individual bit could be a bit that enables a module, e.g. the USART ENABLE bit. A bit group can consist of two of more bits that jointly select a specific configuration of the module that they belong to. A bit group offers up to 2^n selections, where n is the number of bits in the bit group. The two bits that control the USART Receive Complete interrupt level, RXINTLVL[1:0], is an example of a bit group. These two bits offer the following selections:

Table 2-1. RXINTLVL bits and corresponding interrupt level selection.

RXINTLVL1	RXINTLVL0	Interrupt level selection
0	0	Interrupt Off
0	1	Low level interrupt
1	0	Medium level interrupt
1	1	High level interrupt

Bits that are part of a group will always have a number suffix. Bits that are not part of a bit group will never have a number suffix. A Timer/Counter control register D has two bit groups, EVACT and EVSEL. The bits in these groups have a number suffix, while the EVDLY bit, which is not part of a bit group has no number suffix.

Table 2-2. Bits groups and bit names for bits in Timer/Counter Control register D – CTRLD.

Bit Group	EVACT			-	EVSEL			
Bit name	**EVACT2**	**EVACT1**	**EVACT0**	**EVDLY**	**EVSEL3**	**EVSEL2**	**EVSEL1**	**EVSEL0**
Bit number	7	6	5	4	3	2	1	0

3 Writing C-code for XMEGA

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

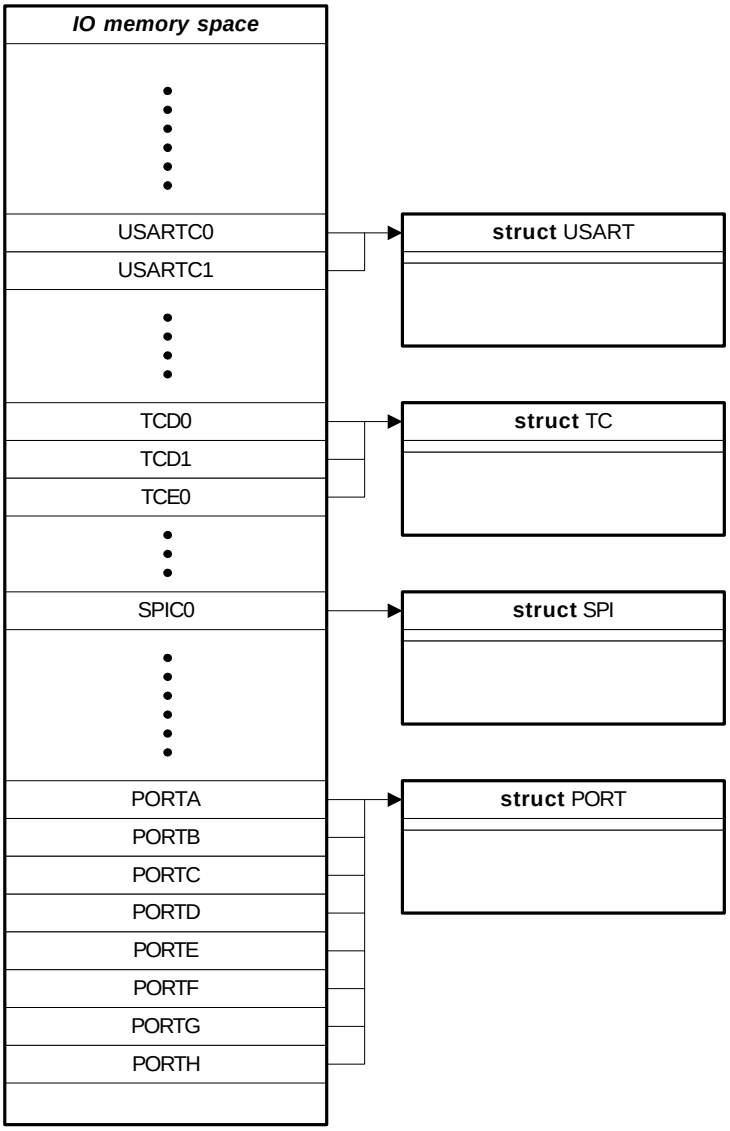
XMEGA modules are located in dedicated and continuous blocks in the memory space and can be seen as encapsulated units. This reflects on the way that the modules are accessed when coding C: modules are encapsulated using C structs, in which all module registers are contained. Figure 3-1 shows an illustration of this.

Note that some registers have no direct module association. These are not encapsulated in structs, as the struct is used to associate registers with a module.

For larger code projects the module structs provide advantages, not only to readability, but also because the compilers can reuse the module drivers and thereby make the code very compact. This is described in more details later.

This document introduces a naming convention and register access methods that are different from what AVR programming veterans are used to, but one should be aware that the “classic” way to access registers is still supported by the header files. This also applies on the bit level.

Figure 3-1. Modules placed in dedicated blocks in IO memory space.



3.1 XMEGA Header Files

A dedicated header file is available for each XMEGA device. If the target device is specified in the project settings (assuming that one uses the IDE for IAR EWAVR), the IAR compiler will automatically include the correct header file if the device file is included as shown in Code Listing 3-1.

Code Listing 3-1. IAR header file inclusion.

```
#include <ioavr.h>
```

The advantage is that if the target device changes, there is no need to change the source files, only the project settings.

3.2 Modules Registers

The IO map is laid out so that all registers for a given peripheral module are placed in one continuous memory block. Registers belonging to different modules are not mixed. This makes it possible to organize all peripheral modules in C structs, where the address of the struct defines the base address, of the module. All registers belonging to a module are elements in the module struct.

An example is the Programmable Multi-level Interrupt Controller (PMIC) module. The struct declaration for this module is shown in Code Listing 3-2 and an example of its use in Code Listing 3-3. Note that the example in Code Listing 3-3 assumes that there is an instance of the PMIC_t type named PMIC. This is covered later in this document.

Code Listing 3-2. Module struct declaration.

```
typedef struct PMIC_struct {
    unsigned char STATUS; // Status Register
    unsigned char INTPRI; // Interrupt Priority
    unsigned char CTRL; // Control Register
} PMIC_t;
```

Code Listing 3-3. Module struct usage.

```
unsigned char temp;
temp = PMIC.STATUS;          // Read status register into temp
PMIC.CTRL |= PMIC_PMRRPE_bm; // Set PMRRPE bit in control
                             // register
```

3.2.1 Multi-word Registers in Module Structs

Some registers are used in conjunction with other registers to represent 16 or 32 bit values. As example one could look at the ADC struct declaration shown in Code Listing 3-4.

Code Listing 3-4. ADC struct declaration.

```
typedef struct ADC_struct {
    unsigned char CH0MUXCTRL;     // Channel 0 MUX Control
    unsigned char CH1MUXCTRL;     // Channel 1 MUX Control
    unsigned char CH2MUXCTRL;     // Channel 2 MUX Control
    unsigned char CH3MUXCTRL;     // Channel 3 MUX Control
    unsigned char CTRLA;          // Control Register A
    unsigned char CTRLB;          // Control Register B
    unsigned char REFCTRL;        // Reference Control
    unsigned char EVCTRL;         // Event Control
    WORDREGISTER(CH0RES);         // Channel 0 Result
    WORDREGISTER(CH1RES);         // Channel 1 Result
    WORDREGISTER(CH2RES);         // Channel 2 Result
    WORDREGISTER(CH3RES);         // Channel 3 Result
    unsigned char reserved_0x10;
    unsigned char reserved_0x11;
    unsigned char CH0INTCTRL;     // Channel 0 Interrupt Control
    unsigned char CH1INTCTRL;     // Channel 1 Interrupt Control
    unsigned char CH2INTCTRL;     // Channel 2 Interrupt Control
    unsigned char CH3INTCTRL;     // Channel 3 Interrupt Control
    unsigned char INTFLAGS;       // Interrupt Flags
    WORDREGISTER(CMP);            // Compare Value
    unsigned char PRESCALER;      // Clock Prescaler
    unsigned char reserved_0x1A;
    unsigned char reserved_0x1B;
    unsigned char CALCTRL;        // Calibration Control
    DWORDREGISTER(CAL);           // Calibration Value
} ADC_t;
```

In Code Listing 3-4, the ADC channel result registers CH0RES, CH1RES, CH2RES, CH3RES and the compare register, CMP, are 16-bit values. These are declared using the `WORDREGISTER` macro shown in Code Listing 3-5. The calibration register, CAL, is a 32-bit value, declared using the `DWORDREGISTER` shown in Code Listing 3-6.

Code Listing 3-5. WORDREGISTER Macro.

```
#define WORDREGISTER(regname)  \
    union { \
           unsigned short regname; \
           struct { \
                  unsigned char regname ## L; \
                  unsigned char regname ## H; \
           }; \
    }
```

Code Listing 3-6. DWORDREGISTER Macro.

```
#define DWORDREGISTER(regname) \
    union { \
           unsigned long regname; \
           struct { \
                  unsigned char regname ## 0; \
                  unsigned char regname ## 1; \
                  unsigned char regname ## 2; \
                  unsigned char regname ## 3; \
           }; \
    }
```

As seen, the `WORDREGISTER` macro uses "`H`" and "`L`" suffix for the high and low bytes respectively. The `DWORDREGISTER` uses number suffix to indicate the byte order. Both the 16-bit and 32-bit registers can be accessed in 16-bit/32-bit mode, by using the register name without suffix as shown in Code Listing 3-7.

Code Listing 3-7. Accessing registers of varying size.

```
unsigned char tempByte;
unsigned int  tempWord;
unsigned long tempDword;

tempByte = ADCA0.CTRLA;  // Read control register A
tempWord = ADCA0.CH0RES; // Read Channel 0 16-bit result
tempDword = ADCA0.CAL;   // Read 32-bit Calibration value
```

Code Listing 3-7 shows how the single byte register `CTRLA` is read, how the two `CH0RES[H:L]` registers are read using a 16-bit operation, and how the four `CAL[3:0]` registers are read in a 32-bit operation. C compilers handle multi-byte registers automatically. Note however that in some cases it may be required to read and write

multi-byte registers in one atomic operation to avoid corruption. In this case, interrupts must be disabled during the multi-byte access to make sure that an interrupt service routine does not interfere with the multi-byte access. AVR1306 includes examples on how atomic access of registers is done for the XMEGA Timer/Counter modules.

3.3 Module Addresses

Definitions of all peripheral modules are found in the device header files available for the XMEGA. The address for the modules is specified in ANSI C to make it compatible with most available C compilers. Code Listing 3-8 shows how ADC 0 on port A is defined.

Code Listing 3-8. Peripheral module definition.

```
#define ADCA            (*(volatile ADC_t *) 0x0200)
```

Code Listing 3-8 shows how 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 pre-defined in the XMEGA header files, it is therefore not necessary to add these definitions in the source code

3.4 Bit Masks and Bit Group Masks

Register bits can be manipulated using pre-defined masks, or alternatively bit positions. Bit positions are not recommended for most tasks. The pre-defined bit masks are either related to individual bits, called a bit mask or a bit group, called a bit group mask, or group mask for short.

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 group. Setting multiple bit that are part of a bit group is covered in section 3.5.

3.4.1 Bit Mask

Consider a Timer Counter Control Register D, CTRLD. The bit groups, bit names, bit positions and bit masks of this register can be seen in Table 3-1.

Table 3-1. Bit groups, bit names, bit positions and bit masks for bits in Timer Counter Control register D – CTRLD.

Bit Group	**EVACT**			**-**	**EVSEL**			
Bit name	**EVACT2**	**EVACT1**	**EVACT0**	**EVDLY**	**EVSEL3**	**EVSEL2**	**EVSEL1**	**EVSEL0**
Bit position	7	6	5	4	3	2	1	0
Bit mask	0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01

Since the names of bits need to be unique for the compiler to handle them, all bits are prefixed with the module type it belongs to. In many cases, the module type name is abbreviated. For all bit defines related to the Timer/Counter modules, the bit names are prefixed by "`TC_`".

To differentiate between bit masks and bit positions, a suffix is also appended. For a bit mask, the suffix is "_bm". The name of the bit mask for the `EVDLY` bit is thus `TC_EVDLY_bm`. Code Listing 3-9 shows the typical usage of a bit mask. The EVDLY bit

in the CTRLD register of Timer/Counter D0 is set, leaving all the other bits in the register unchanged.

Code Listing 3-9. Bit mask usage.

```
TCD0.CTRLD |= TC_EVDLY_bm; // With bit mask specifier.
```

3.4.2 Bit Group Masks

Many functions are controlled by a group of bits. Timer Counter CTRLD register the EVACT[2:0] and the EVSEL[3:0] bits are grouped bits. The value of the bits in a group selects a specific configuration.

When changing bits in a bit group it is often required to clear the bit group before assigning a new value. To put it in another way: It is not enough to set the bits that should be set, it is also required to clear the bits that should be cleared. To make this easy a bit group mask is defined. The group mask uses same name as the bits in the bit group and is suffixed "_gm".

Code Listing 3-10 shows how the group mask relates to the bit masks. In reality, the group mask values are pre-calculated in the header files, so the compiler does not need to calculate the same constant over and over again.

Code Listing 3-10. Group mask and bit mask relation.

```
#define TC_EVACT_gm  (TC_EVACT2_bm | TC_EVACT1_bm | TC_EVACT0_bm)
```

The bit group mask is primarily intended to clear the old configuration of a bit group before writing a new value. Code Listing 3-11 shows how this can be done. The code will clear the EVACT bit group in the CTRLD register of Timer/Counter D0. This construct is not very useful in itself. The group mask will typically be used in conjunction with a group configuration mask, which is covered in section 3.5.

Code Listing 3-11. Group mask usage.

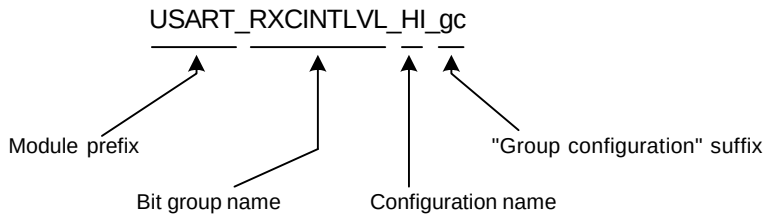
```
TCD0.CTRLD &= ~(TC_EVACT_gm); // Clear group bits with group mask.
```

3.5 Bit group Configuration Masks

It is often required to consult the datasheet to investigate what bit pattern needs to be used when setting a bit group to a desired configuration. This also applies when reading or debugging code. To increase the readability and to minimize the likeliness of setting bits in bit groups incorrectly, a number of group configuration masks are made available. 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 group name, a description of the configuration and a suffix, "_gc", indicating that this is a group configuration. See Figure 3-2 for an example.

Figure 3-2. Group configuration name composition (using USART receive complete interrupt level bits as example).



Inspecting the group configuration Figure 3-2, one can see that it is used to select a configuration for the RXCINTLVL bits in a USART module. This specific group configuration selects a high (HI) interrupt level.

The bit group for the Receive Complete Interrupt Level consists of two bits, RXINTLVL[1:0]. Table 3-2 shows the available group configurations for this bit group.

The configuration names are “OFF”, “LO”, “MED” and “HI”. These names make it very easy to write and maintain code, as it requires very little effort to understand what configuration the specific configuration mask selects.

Table 3-2. RXINTLVL bits and corresponding interrupt level of module interrupt.

RXINTLVL1	RXINTLVL0	Interrupt level	Group configuration mask
0	0	Interrupt off	USART_RXCINTLVL_OFF_gc
0	1	Low interrupt level	USART_RXCINTLVL_LO_gc
1	0	Medium interrupt level	USART_RXCINTLVL_MED_gc
1	1	High interrupt level	USART_RXCINTLVL_HIGH_gc

To change a bit group 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 erased first. Code Listing 3-12 shows how the group mask and a configuration mask can be used to reconfigure the USART C0 Receive Complete Interrupt level to medium level.

Code Listing 3-12. Changing a bit group configuration.

```
USARTC0.INTCTRL = (USARTC0.INTCTRL & ~USART_RXCINTLVL_gm ) | USART_RXCINTLVL_MED_gc;
```

Note that though it may be tempting to split the code in Code Listing 3-12 into two separate code lines, one that clear the bit group and one that sets the new value, this is not recommended. Since the INTCTRL register is defined as `volatile`, two separate code lines will trigger reading and writing the INTCTRL register twice instead of once. In addition to making the code inefficient, this can put the peripheral in an unintended state.

The use of group masks to clear bits is not always required. Code Listing 3-13 shows how all interrupt levels of USARTC0 can be configured at the same time. Receive Complete, Transmit Complete and USART Data Register Empty interrupt levels are set to MEDIUM, OFF and LOW interrupt levels respectively.

Code Listing 3-13. Setting all configurations in a register at the same time.

```
USARTC0.INTCTRL = USART_RXCINTLVL_MED_gc |
                  USART_TXCINTLVL_OFF_gc |
                  USART_DREINTLVL_LO_gc;
```

3.5.1 Enumeration of Group Configuration masks

Unlike bit masks and group masks, the bit group configuration masks are defined using C enumerators. One enumerator is defined for each bit group. The enumerator for the USART RXCINTLVL bit group is shown in Code Listing 3-14.

Code Listing 3-14. USART RXCINTLVL enumerator

```
typedef enum {
   USART_RXCINTLVL_OFF_gc = (0x00 << 4);
   USART_RXCINTLVL_LO_gc  = (0x01 << 4);
   USART_RXCINTLVL_MED_gc = (0x02 << 4);
   USART_RXCINTLVL_HI_gc  = (0x03 << 4);
} USART_RXCINTLVL_t;
```

As seen in Code Listing 3-14, the name of the enumerator is a concatenation of the module type (USART), the bit group (RXCINTLVL) and a suffix (_t) that indicates that this is a data type.

Each of the enumerator constants behaves much like a normal constant when used on its own. However, using an enumerator has the advantage that it creates a new data type. USART_RXCINTLVL_t can be used as a type name, just like an int or char. An enumerator constant can be used directly as integer, but assigning an integer to an enumerator type will trigger a compiler warning. This can be used to the programmers advantage. Imagine a function that sets the receive complete interrupt level for a USART module. If the function accepts the interrupt level as an integer type (e.g. unsigned char), any legal or illegal value can be passed to the function. If the function instead accepts a parameter of type USART_RXCINTLVL_t, only the four pre-defined constants in the USART_RXCINTLVL_t enumerator can be passed to the function. Passing anything else will result in a compiler warning.

Notice that the constants in Code Listing 3-14 are shifted to the actual bit position. This means that the enumerator contstants are the actual values that shall be written to the register. No additional shifting is needed.

3.6 Functions Calls and Module Drivers

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. When considering portability this represents a great advantage.

Consider a device with 8 Timers/Counters. The functions to initialize and access the Timer/Counter modules can be shared by all module instances. Even though there is a small overhead in passing the module pointer to the functions, the total code size will often be reduced because the code is reused for all instances of each module type. Even more important, development time, maintenance cost, and portability can be greatly improved by using this approach.

Code Listing 3-15 shows a function that uses a module-pointer to select a clock source for any Timer/Counter module.

Code Listing 3-15. Example function using module instance pointer

```
void TC_ConfigClockSource(volatile TC_t * tc,
                          TC_CLKSEL_t clockSelection)
{
  tc->CTRLA = tc->CTRLA & ~TC_CLKSEL_gm | clockSelection;
}
```

The function takes two arguments: a module pointer of the type, `TC_t`, and a group configuration type, `TC_CLKSEL_t`. The code in the function uses the Timer Counter module pointer to access the CTRLA register to set a new clock selection for the Timer/Counter module provided through the tc parameter.

Code Listing 3-16 shows how the function in Code Listing 3-15 can be used to configure different Timer/Counter modules with different clock selections.

Code Listing 3-16. Calling a function that takes a module pointer as parameter

```
TC_ConfigClockSource (&TCC0, TC_CLKSEL_OFF_gc);
TC_ConfigClockSource (&TCC1, TC_CLKSEL_DIV1_gc);
TC_ConfigClockSource (&TCD0, TC_CLKSEL_DIV2_gc);
TC_ConfigClockSource (&TCD1, TC_CLKSEL_DIV1024_gc);
```

4 Alternative Ways of Writing Code

For convenience and to avoid forcing AVR programming veterans to change their programming style it is still possible to use a programming style that does not involve structs. It is also possible to use the bit name style that is used for megaAVR®. 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 structs. To refer to a register directly, concatenate the module instance name, an underscore and the register name. The same naming convention is used when programming in assembly.

Example: To access the CTRLA register of Timer/Counter C0, use the name TCC0_CTRLA.

4.2 Bit Positions

It is possible to use bit masks to set or clear bits. A bits position within a register is defined using the same name as the bit mask, with an additional prefix, "_bp" for bit position. Code Listing 4-1 shows how the bit position can be used to configure a register.

Code Listing 4-1. Alternative register access code.

```
PORTB_OUT = (1 << PORTB_OUT0_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.

5 Summary

For reference, an overview of the different postfixes used when dealing with bit configuration is listed in Table 5-1.

Table 5-1. Overview of postfixes

Postfix	Meaning	Example
_gm	Group Mask	TC_CLKSEL_gm
_gc	Group Configuration	TC_CLKSEL_DIV1_gc
_bm	Bit Mask	TC_CCAEN_bm
_bp	Bit Position	TC_CCAEN_bp

Using the suggested methods to write C-code for XMEGA is by no means mandatory, but the advantages offered should be considered used to ensure robust, portable, reusable, and highly readable code. The larger the project, and the more features the device has the bigger the advantage. For small projects it can be argued that there is no benefit, but turning that argument upside down one can say that there is not disadvantage either.

One thing is for sure: good code style is always an advantage.

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
avr@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature

8075A-AVR-02/08