AVR1512: XMEGA-A1 Xplained training - XMEGA Basics

Prerequisites

- **Required knowledge**
  - **Basic knowledge of microcontrollers and the C programming language**
- **Software prerequisites**
    - **Atmel® AVR Studio® 5**
- **Hardware prerequisites**
  - **XMEGA-A1 Xplained evaluation board**
  - **JTAGICE 3 (or JTAGICE mkII or AVR ONE!)**
- **Estimated completion time:**
  - **2 hours**

1 Introduction

This application note covers Atmel AVR® XMEGA® basics, using IO ports to show various concepts in four tasks. The goal of this training is to get you started with small code fragments, utilizing the XMEGA header files and some of the XMEGA features for more efficient and compact code.
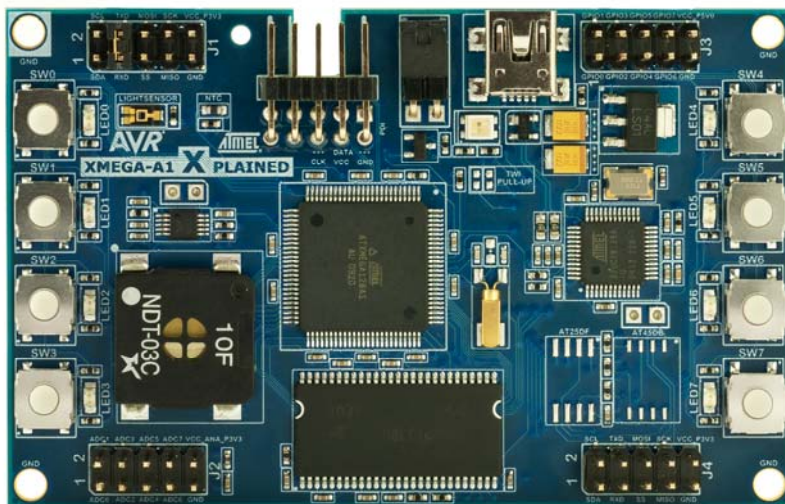
2 Writing C-code for XMEGA

The pressure to reduce development time and at the same time ensure high quality of electronic products has made high-level programming languages a requirement. It is easier to maintain and reuse and gives better portability and readability.

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

The following sub-sections will give a brief overview of some of the programming style that is new to XMEGA. A more detailed description is given in the application note "AVR1000: Getting Started Writing C-code for XMEGA".

2.1 Bit Masks and Bit Group Masks

Register bits can be manipulated using pre-defined masks, or alternatively bit positions (which are not recommended). The pre-defined bit masks are either related to individual bits, called a bit mask, or a bit group. The pre-defined bit group masks are called 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.

If you are used to the Atmel megaAVR® or Atmel tinyAVR® microcontrollers you would typically set the Event delay (EVDLY) bit similar to Example 1.

Example 1. Bit position usage in standard Atmel AVR microcontroller

```
TCD0.CTRLD |= (1<< EVDLY );
```

This is because the header files for those microcontrollers specify the *bit position*, and with the shift operation (<<) you create a bit mask. With the XMEGA header files this is more readable because both the *bit positions* and the *bit masks* are already defined.

Example 2. Bit mask usage in XMEGA

```
TCD0.CTRLD |= TC0_EVDLY_bm; // with bit mask specifier
```

Example 3. Bit position usage in XMEGA

```
TCD0.CTRLD |= (1 << TC0_EVDLY_bp); // with bit position specifier
```

Using the format in example 2 is recommended, but both examples achieve setting bit 4 to the value 1, that is, register value will be ORed with the binary value 0001 0000.

Many configurations are controlled by a group of bits. For example; in Timer/Counter CTRLD register (see Figure 1) 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. The group mask uses the same name as the bits in the bit group and is suffixed "_gm", while the position of the bit group is suffixed "_gp".

Figure 1. Timer Control D register as depicted in the Atmel XMEGA A Manual

7	6	5	4	3	2	1	0
EVACT[2:0]			EVDLY	EVSEL[3:0]			
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
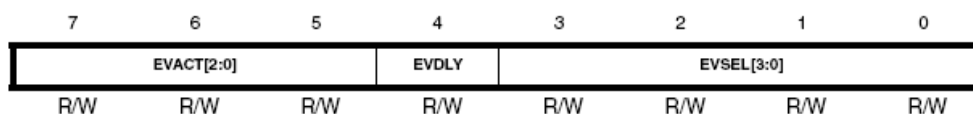
The bit group mask is primarily intended for clearing old configuration of a bit group before writing a new value. The bit group position is useful when setting numerical factors, for example, multiplication factors for PLL.

Example 4. Group mask usage

```
TCD0.CTLD &= ~(TC0_EVACT_gm); // Clear group bits with group mask
```

By looking at the mask and bit relation as they are in the XMEGA header files, we see what the above does:

```
#define TC0_EVACT_gm  0xE0  /* Event Action group mask. */
#define TC0_EVACT_gp  5  /* Event Action group position. */

#define TC0_EVACT0_bm  (1<<5)  /* Event Action bit 0 mask. */
#define TC0_EVACT0_bp  5  /* Event Action bit 0 position. */
#define TC0_EVACT1_bm  (1<<6)  /* Event Action bit 1 mask. */
#define TC0_EVACT1_bp  6  /* Event Action bit 1 position. */
#define TC0_EVACT2_bm  (1<<7)  /* Event Action bit 2 mask. */
#define TC0_EVACT2_bp  7  /* Event Action bit 2 position. */
```

TC0_EVACT_gm will have a binary value of 1110 0000, which is very useful for clearing a group of bits.

2.2 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 a 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. The name of a group configuration has the suffix "_gc".

Figure 2. Group configuration name composition
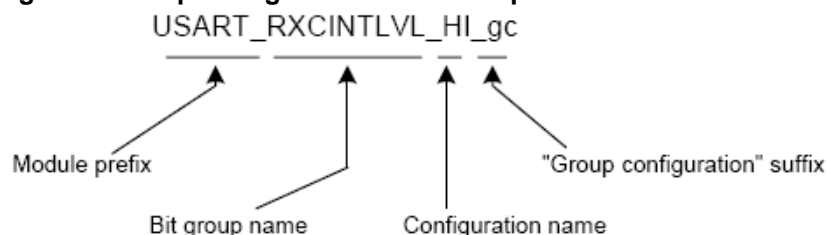
From Figure 2 one can see that the group configuration is used for the Receive complete interrupt level (RXCINTLVL) bits in a USART module. This specific group configuration selects a high (HI) interrupt level.

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.

Example 5.

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

The example above shows group mask and configuration mask used together. The first part of the example (`USARTC0.CTRLA & ~USART_RXCINTLVL_gm`) clears the RCXINTLVL bits in the USARTC0 register in a similar way to what is shown in example 4. The last part (| `USART_RXCINTLVL_MED_gc`) sets the new value to get a medium interrupt level.

This code is used to reconfigure USARTC0 Receive Complete Interrupt level to medium, without affecting the other bits in the register. You will see code similar to this in a lot of the drivers that comes with the different application notes.

2.3 Module registers and addresses

The IO map in the Atmel XMEGA is organized so that all registers for a given peripheral module are placed in one continuous memory block. 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 that belongs to a module are elements in the module struct.

The following code shows how the different registers for the Programmable Multilevel Interrupt Controller (PMIC) is defined in the Atmel XMEGA header files.

Example. Definition from XMEGA header file (iox128a1.h):

```
typedef struct PMIC_struct
{
    register8_t STATUS; /* Status Register */
    register8_t INTPRI; /* Interrupt Priority */
    register8_t CTRL; /* Control Register */
} PMIC_t
```

This code defines the registers available in the PMIC module. The struct is used to define the PMIC at a specific memory address.

Example. Peripheral module definition:

```
#define PMIC        (*(PMIC_t *) 0x00A0)
```

The example above shows how the module instance definition uses a de-referenced 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 your source code.

If the examples above do not make sense to you, don't worry.

What you need to know is how to use these definitions. With the above definitions, which are part of the Atmel AVR XMEGA header files, you can access any registers within a module with the "." (dot) syntax, as shown in the following example:

Example usage

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

The main advantage of using the Module registers and the Module addresses shown above is the ability to create drivers that are independent of the actual peripheral (for example works for both USART 1 and USART 4) and between different parts of the Atmel XMEGA family.

3 Overview

Here is a short overview of the tasks in this training:

This training covers XMEGA basics, using IO ports to show various concepts in four tasks. The goal of this training is to get you started with small code fragments, utilizing the XMEGA header files and some of the XMEGA features for more efficient and compact code.

Task 1. Basic LED Control

This task shows how to use #defines and module names from the XMEGA header files to create portable code and how to manipulate IO ports.

Task 2. Generic Drivers

This task shows how to use pointers to peripheral module to make generic driver code and how to read switches and output to LEDs.

Task 3. Output and Pull Configuration

This task shows how to use the XMEGA header files with its group mask and group configuration values to efficiently modify bit fields within registers.

Task 4. Multi Configuration

This task shows how to use the multi configuration register to configure more than one pin at a time.

Note: "board.h" in the base folder defines the evaluation board used. This file contains all custom definitions used in the training code.

Good luck!

4 Task 1: Basic LED Control

Nothing is more fun than blinking LEDs on your development board! This task does just that, and nothing else. It shows how to utilize the Atmel XMEGA header file format to make code that is easy to customize. It also shows some of the features of the XMEGA IO ports.

The goals for this task are that you:

- Know how to change which port to use for LEDs with only one `#define`
- Know how to configure IO pin directions and set output values
- Know how to change existing port output values with only one write access
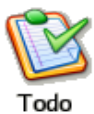
1. Locate the Atmel XMEGA-Basics folder. Open the `xmega_basics.avrsln` solution file and set Task 1 active by selecting it as StartUp project. Then take a look at task1.c, see Figure 3.

Figure 3. Open the project in AVR Studio 5
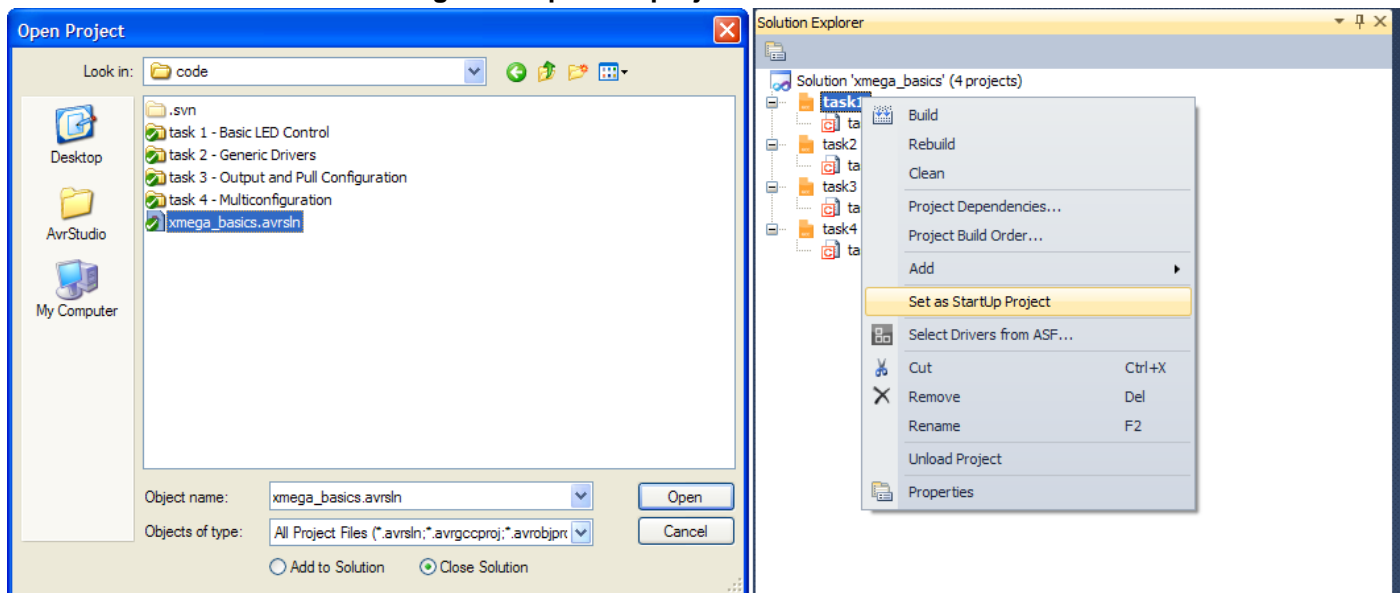
2. Figure 4 shows how easy it is to change LEDPORT definition if you want another LED port. The LEDPORT define is found in the file "board.h".

Figure 4. Changing the LED port


```
// file, it is enough to define t
// #defines for all registers for
// IO port to use for LED output
#define LEDPORT PORTE

volatile unsigned int counter;
```

3. Connect the JTAGICE3 to the XMEGA JTAG header on the evaluation board, see Figure 5. Then connect the board to the computer with the corresponding USB cable. Connect the JTAGICE3 to the computer with the USB cable and switch it on.

Figure 5 - XMEGA-A1 Xplained
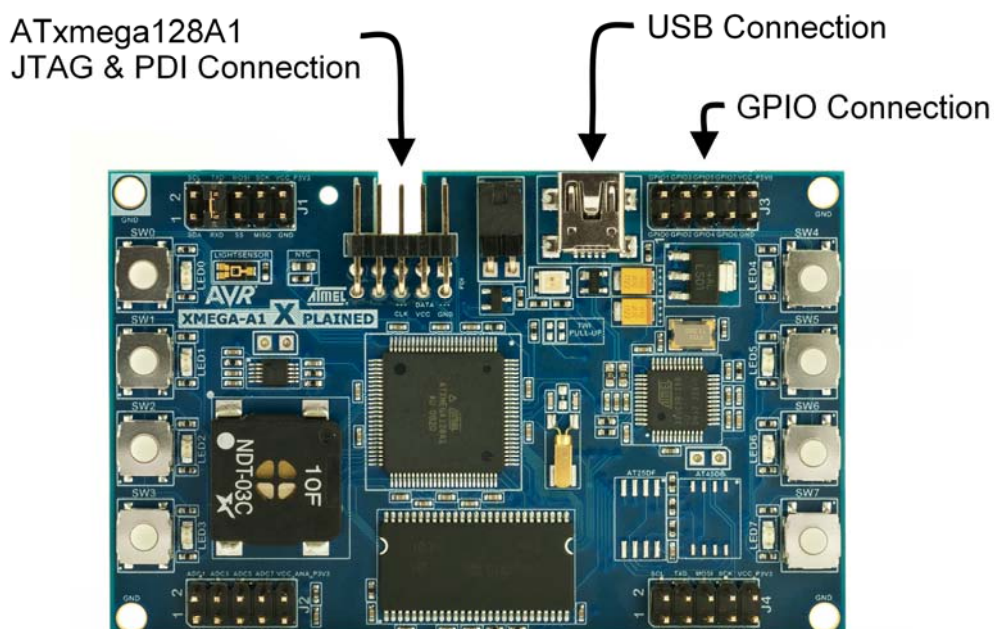
4. Build the project (shortcut Ctrl-Shift-B), see Figure 6

5. Start debugging by pressing the Play button, see Figure 7

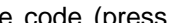**Figure 6. Build project (Ctrl-Shift-B)**
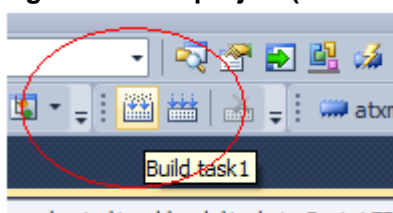
Figure 7. Start Debugging

7
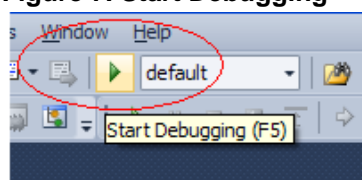
6. Open the I/O view for the LED-port, see Figure 8. Note that there are quite many registers that deal with the IO port pins. These registers make the code more compact and they offer more options to the developer.

Figure 8. Registers in ledPort

7. Single-step through the code (press F11) and observe the LEDs in the I/O view and on the target board, see Figure 9

8. Try to understand the different port registers, SET, CLR, TGL as you single step (F11) and look in the I/O view

9. Run the code (press F5) to see the final part of the code blink the LEDs

Figure 9. Start single stepping (F10)


```
volatile unsigned int counter;

int main( void )
{
    LEDPORT.DIR = 0xff;    // Set all pin
    LEDPORT.OUT = 0xf0;    // Set upper f
    LEDPORT.OUTSET = 0x0f; // Set lower f
    LEDPORT.OUTCLR = 0xf0; // Clr upper f
    LEDPORT.OUTTGL = 0xff; // Toggle all
    LEDPORT.OUTTGL = 0xff; // Toggle all
    LEDPORT.OUTTGL = 0xAA; // Toggle some
    LEDPORT.OUTTGL = 0xAA; // Toggle some
    LEDPORT.OUTTGL = 0xAA; // Toggle some
    LEDPORT.OUTTGL = 0xAA; // Toggle some

    // Now, just for fun, we toggle all L
```

5 Task 2: Generic Drivers

Ever wanted one function that can access any IO port (or any ADC, or any DAC, or any Timer/Counter…) without having to use `switch/case` or `if-else` statements to select the correct registers? This task shows how to utilize the layout of the Atmel XMEGA IO map to make code that takes a pointer to an IO port module and uses generic code to access the correct registers. This method can be used to create generic driver code for IO ports, ADCs, DACs, Timer/Counters etc.

The goals for this task are that you:

- Know how to create a pointer variable and make it point to any IO port module
- Know how to use module pointers as function parameters
- Know how to access module registers through module pointers
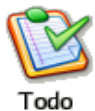
1. Locate the Atmel XMEGA-Basics folder. Open the `xmega_basics.avrsln` solution file and set Task 2 active by selecting it as StartUp project.

2. Observe that the ledPort and switchPort pointers assignments can be changed easily if you want to use other ports for LEDs and switches

3. Build the project (F7/ ctrl-shift-b)

4. Start debugging

5. Single-step into the code (press F11) and observe the LEDs. Try pressing different switches when inside the `GetSwitches` function

6. Run code (press F5) to see the code copy switch states to LEDs real-time

Figure 10. Single step the functions

```
// Prepare pointers to the peripheral port modules we want t
PORT_t * ledPort = &LEDPORT;
PORT_t * switchPortL = &SWITCHPORTL;
PORT_t * switchPortH = &SWITCHPORTH;

// Enable pullups for switches. MPCMASK will be explained in
PORTCFG.MPCMASK = SWITCHPORTL_MASK_gc;
switchPortL->PIN0CTRL = PORT_OPC_PULLUP_gc;

PORTCFG.MPCMASK = SWITCHPORTH_MASK_gc;
switchPortH->PIN6CTRL = PORT_OPC_PULLUP_gc;

// Now copy switch state to LEDs again and again and again..
// Because the switches are split between two ports on the X
// the value statement combines bits from each port and aran
while(1)
{
    value = GetSwitches( switchPortL, SWITCHPORTL_MASK_gc )
    SetLEDs( ledPort, value );
}
}
```

6 Task 3: Output and Pull Configuration

With Atmel AVR XMEGA, all configuration options for all bit fields of all modules are available as named constants in one header file (for example the iox128a1.h). This task shows how to use *group mask* and *group configuration* values from header files, how to find them in datasheets and how to use them in an efficient and compact way in your code. Also, this task gives insight into some of the features of the XMEGA IO ports.

The goals for this task are that you:

- Know how to write efficient and compact code to modify bit fields within a register
- Know how to find and use group mask and group configuration values from datasheets and header files
- Know about different output and pull configurations of the XMEGA IO ports

1. Locate the Atmel XMEGA-Basics folder. Open the `xmega_basics.avrsln` solution file and set Task 3 active by selecting it as StartUp project.

2. Build the project, press F7/ ctrl-shift-b, and start a debug session

3. Open the XMEGA A manual and locate the PINnCTRL register configuration. Look at the Output and Pull Configuration (OPC) values in the Register description section

4. In the build-tree, expand the Dependencies and open `iox128a1.h`, see Figure 11. Can you find the PORT_OPC_WIREDAND_gc OPC configuration? (Hint: search for "wired")

5. In task3.c, locate and single step to this line:

   ```
   SWITCHPORTL.PIN0CTRL = (SWITCHPORT.PIN0CTRL & ~PORT_OPC_gm) |
   PORT_OPC_WIREDANDPULL_gc;
   ```

   Refer to Chapter 2.2 and try to understand the bit configuration mask above. (The wired-and configuration will be detailed in the next task)

6. Single-step further while looking at the I/O view. Compare with the register description configurations in the manual, and verify the OPC configurations
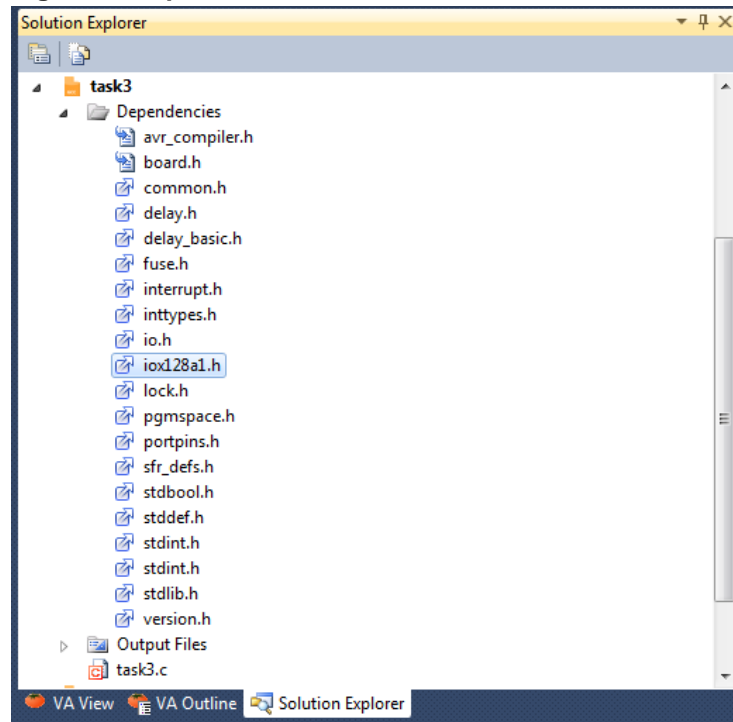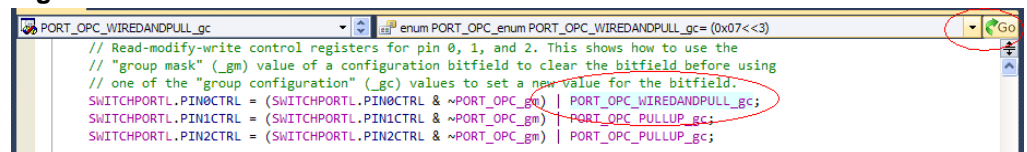
Figure 11. Open iox128a1.h

Figure 12 Go to definition in AVR Studio 5

7 Task 4: Multi Configuration

In the previous task you learned that there is one configuration register for each I/O pin. What if you want to configure several pins at once? With Multi Configuration, several pins in a port can be configured at the same time.

Having configuration registers for each pin means that the number of operations necessary for configuring a single port increases. The number of write operations is reduced by the introduction of a global Multi-pin Configuration Mask (MPCMASK) register that is common for all ports. MPCMASK can be used to set a bit mask for the pin configuration registers. When setting bit n in MPCMASK, PINnCTRL is added to the pin configuration mask. During the next write to any of the port's pin configuration registers, the same value will be written to all the ports pin configuration registers set by the mask. The MPCMASK register is cleared automatically after the write operation to the pin configuration registers is finished.

This task will also demonstrate one of the Output Pull Configurations (OPC), the Wired And Pull. The figure below illustrates two microcontrollers (MCUs) connected together and an external pull-up. In the table below the figure you see how the I/O pin drivers respond depending on the value of the OUT register.

In this task we have only one microcontroller so we are going to connect two pins together with a jumper, and use the internal pull-ups.
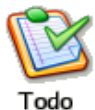
Figure 13. Typical Wired-And usage
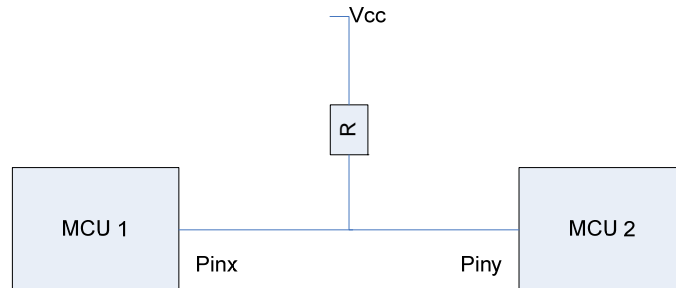
I/O port behavior when Wired-AND is configured:

OUT-register	IO pin output
0	Pulled low (0)
1	Tri-state

The goals for this task are that you:

- Know how to use the Multi-pin Configuration Mask register
- Understand more about the output and pull configurations of the Atmel XMEGA IO ports
- Understand the specific Wired-And example

Locate the Atmel XMEGA-Basics folder. Open the `xmega_basics.avrsln` solution file and set Task 4 active by selecting it as StartUp project.

1. Use a **jumper** to connect Pin0 and Pin1 on the TESTPORT header located at the bottom right of the evaluation board.

2. Build the project and start debugging in Atmel AVR Studio

3. Add a break-point before the while-loop as shown in Figure 14 and press F5 to run. To place a break-point, place the cursor at the code-line you want the break-point and press F9

Figure 14. Add a break-point before the while(1)-loop

```
// same value will be written to all the port's pin confi
PORTCFG.MPCMASK = 0xFF;

// The WIREDANDPULL enables the internal pull-up.
// Also, when the port is configured as output (dir=1),
// setting a pin low will drive low,
// but setting a pin high will set the pin in tri-state
TESTPORT.PIN0CTRL = (TESTPORT.PIN0CTRL & ~PORT_OPC_gm) |

// The MPCMASK register is cleared automatically after th
// configuration registers is finished.

// Connect TESTPORT pin 0 and 1 with a jumper on the boar

while(1)
{
    // Set pins 0 and 1 high
    TESTPORT.OUTSET = PIN0_bm;
    TESTPORT.OUTSET = PIN1_bm;
    // Now, the pins are in tri-state
```

4. Open the I/O view and look at the TESTPORT register view. What happens to the PINnCTRL pins when you now single step (F10)?

5. Let's look at what the Wired-AND setting does. The code sets both Pin0 and Pin1 high and check what the result is in the IN register. Afterwards it sets Pin0 low and then checks the result in the IN register for Pin1

6. Single step the while-loop and observe the TESTPORT.IN register in the I/O view. What happens with Pin1 IN value when Pin0 is set low?

7. Please note that if you connect two pins together (that is short-circuit) and configure one pin high and the other low, both pins with the *normal configuration* the pin output drivers will drive in different directions and you will get a strong current flowing between the pins. When using the Wired-AND configuration as the example shows this is avoided

Figure 15. Driving Pin0 low drives Pin1 low
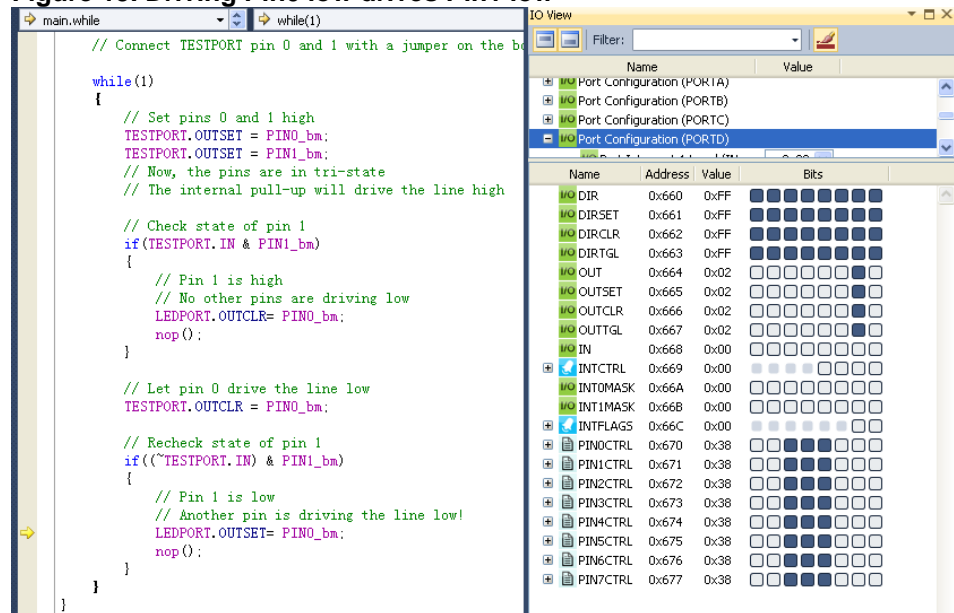
8. Remove the jumper from the TESTPORT

8 Summary

Here are some of the main features/functionalities you have learned during this session:

- Writing C-code for Atmel XMEGA
- Basic Port configuration
- Making generic drivers
- Output And Pull Configuration
- Multi-configuration of pins
- Wired-And

9 Resources

- XMEGA Manual and Datasheets
  - o http://www.atmel.com/xmega
- Atmel AVR Studio with help files
  - o http://www.atmel.com/products/AVR/
- WINAVR GCC compiler
  - o http://winavr.sourceforge.net/
- Atmel IAR Embedded Workbench® compiler
  - o http://www.iar.com/

10 Atmel Technical Support Center

Atmel has several support channels available:

- o Web portal: http://support.atmel.no/ All Atmel microcontrollers
- o Email: avr@atmel.com All Atmel AVR products
- o Email: avr32@atmel.com All 32-bit AVR products

Please register on the web portal to gain access to the following services:

- o Access to a rich FAQ database
- o Easy submission of technical support requests
- o History of all your past support requests
- o Register to receive Atmel microcontrollers' newsletters
- o Get information about available trainings and training material

ATMEL ®