AVR Butterfly Training

Atmel Norway,
AVR Applications Group

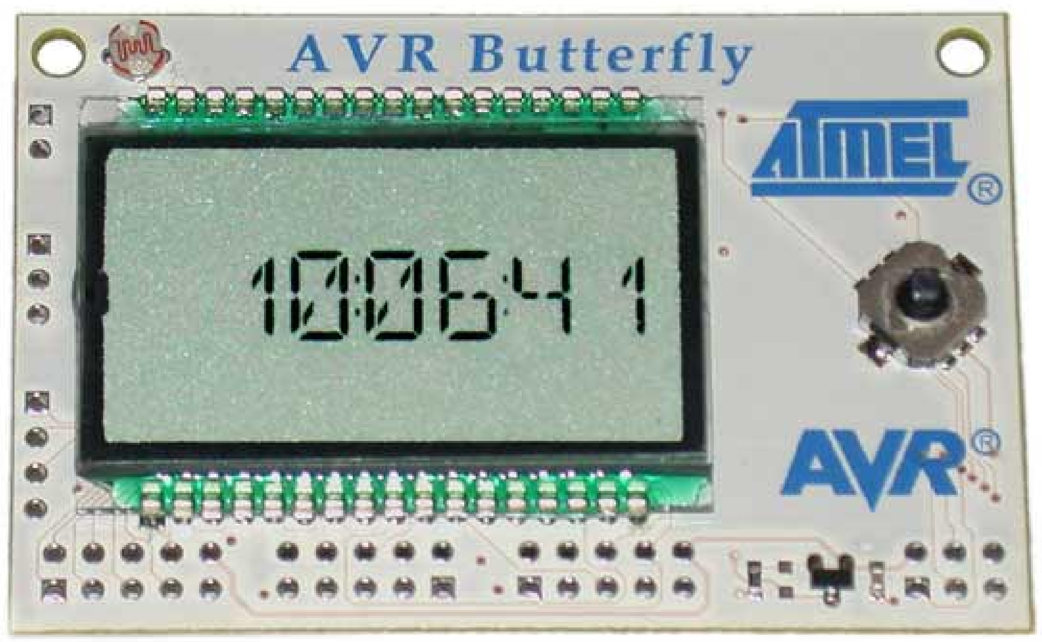


Table of Contents

Introduction

This Training is based on the ATmega169 LCD microcontroller and the AVR Butterfly board. This Board includes a JTAG connector which makes it very easy to use it as a development and debug platform. The Butterfly is powered by a battery cell, but we will in this exercise remove the battery and power it using a STK500 as a power supply (saving the battery for later).

This Training consists of 6 tasks. The 4 first tasks do not require you to write code, but is a tutorial; a step by step introduction to how to utilize the JTAG ICE to get the most out of the OCD system!

Quick overview of the Tasks in this training

Task 1: Getting Everything Running!
Here you will be guided through the initial setup, and given a description of the Butterfly features that we will use during this training.

Task 2: Breakpoints and IO view
General Breakpoint use, HW vs. SW Breakpoints and how to use the IO view to see what is going on inside the AVR

Task 3: Advanced Breakpoints and Stack
In this exercise we will look at a recursive function, and how we can use advanced breakpoints to catch a stack overflow

Task 4: Advanced Breakpoints and Masks
A closer look at advanced breakpoints and how to set up a mask to trap access to a range of memory locations

Task 5: Programming: Max and Min Values
Add a function that is able to store and display Max and Min values as well as currently measured value.

Task 6: Programming: Sound Player
Turn the AVR Butterfly into a musical instrument.

This Training is based on a strip down version of the Butterfly Source code, and the AVR064 and AVR065 application notes. The code has been stripped down to make it easier to read and debug.

Getting Started

This section contains useful information regarding the Butterfly hardware, and how to connect it to the STK500 to save your battery.

Required Software and Hardware

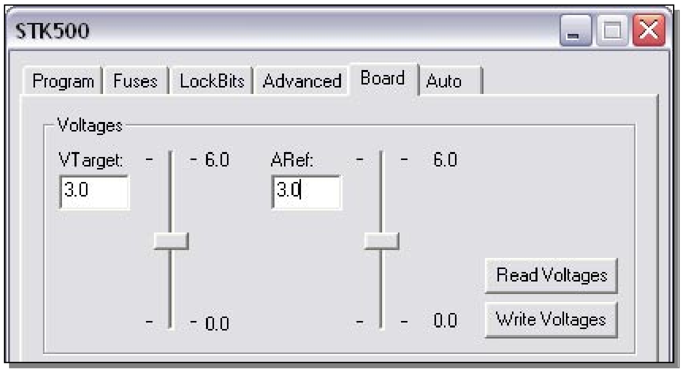
To complete this training you'll need the following equipment

- AVR Studio 4.07
- IAR EWAAVR 2.28a C-Compiler
- Butterfly Demo Board
- JTAG ICE
- STK500 Board
- Power Supply for both STK500 and JTAG ICE

Setting up the Hardware

Setting up the STK500

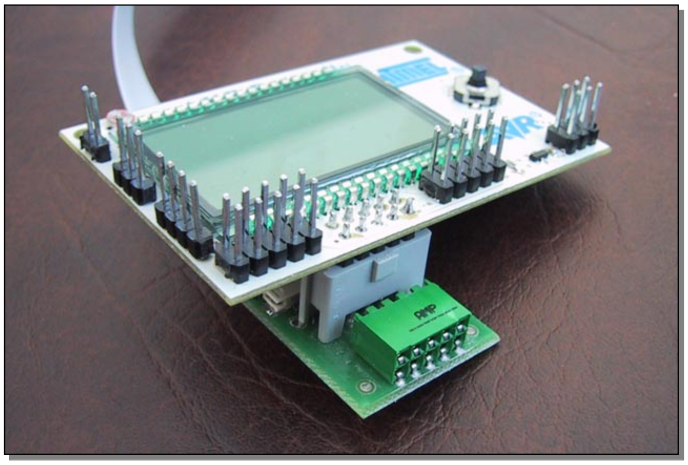
The Butterfly Demo is normally powered by a 3 volt battery cell. To use the STK500 Board it should be set up to provide VTG = 3 volt. This is done in the "Board" settings in the STK500 software (in AVR Studio 4)



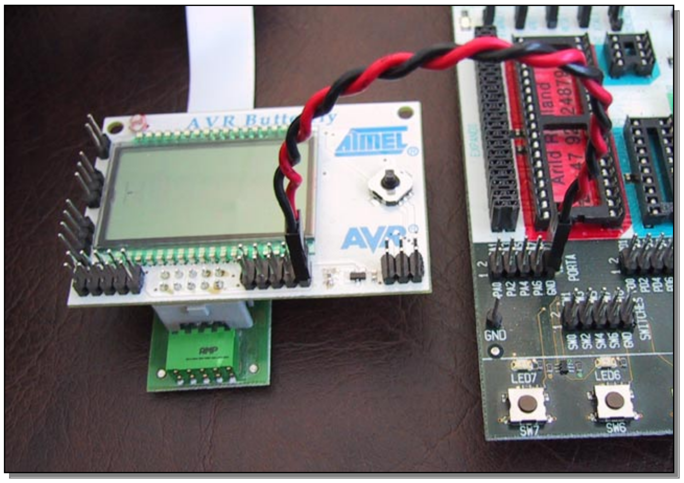
VTarget: 3.0
ARef: 3.0

Connecting the JTAG ICE

Once the STK500 board is set up to provide correct voltage, connect the JTAG ICE to the Butterfly board. The JTAG connector is the middle 10bit header connector. Connect it as shown in the picture.



Connecting Power
Power down the STK500 and JTAG ICE. Connect a 2-wire cable (power) between the STK500 and the Butterfly Demo board as shown below. (The pinout on the header connectors are the same, and we want to feed VTG and GND from the STK500 to the Butterfly.



Verify all connections, and when sure that the power supply cables are correct, power up the STK500, then the JTAG ICE.

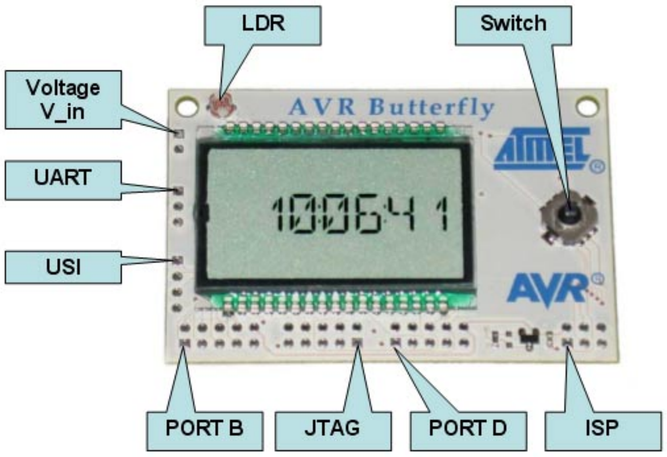
Setting up the software

Both AVR Studio 4 and IAR compiler need to be installed to complete this training. The training uses files located in a folder named "butterfly" in the C:\AVR folder. If you do not have a folder with this name, you should install it now. Ask for the install disk.

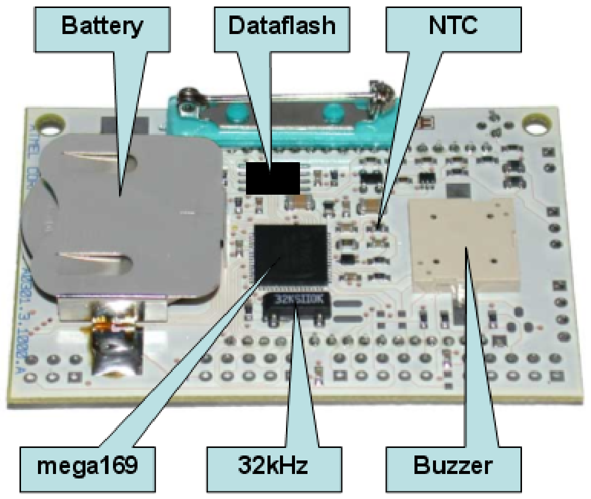
Each Task has its own subfolder named Task1 to Task6. Each containing its own set of files. After completing the first task, close it down, and open the next one to continue the training.

The Butterfly Hardware

Front view

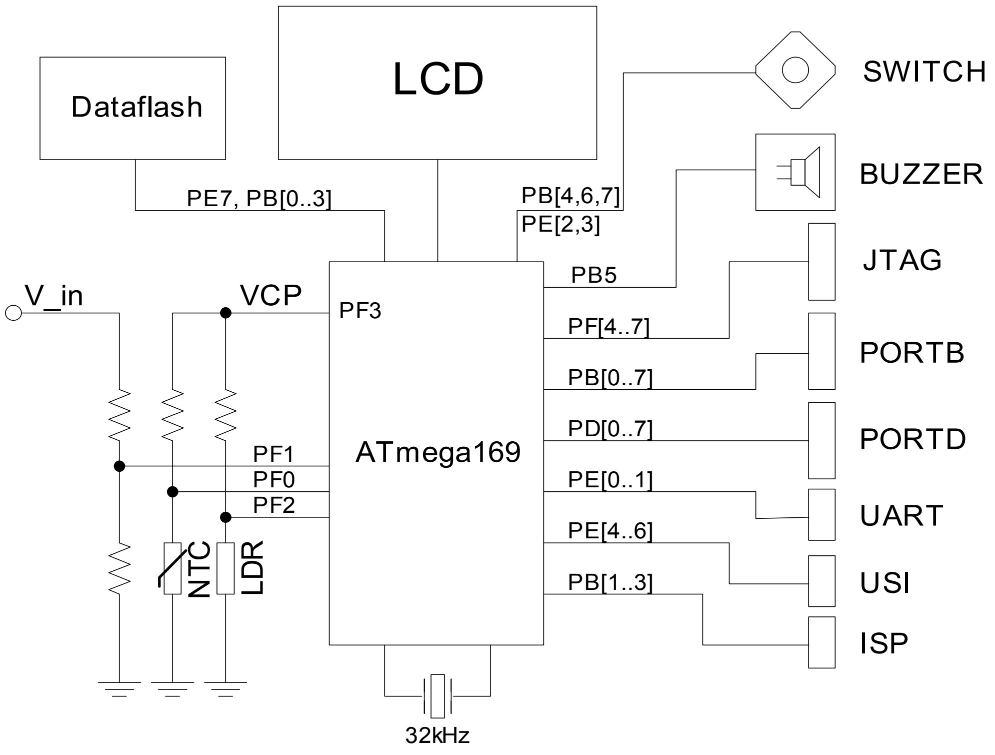


Backside View



Block Schematics

This is a (very) simplified schematic showing how the different peripherals are connected to the ATmega169.



Task 1: Getting Everything Running!

Introduction:

This exercise uses a fully working C code, which should compile and run without further modifications.
The purpose of this first task is to get all software and hardware up running, and to verify that everything is working correctly. Once running, use a few minutes to familiarize yourself with the code, and try to understand how it works.

Task:

Set up the hardware as explained in the Getting Started section. That is:

1. Remove Battery from the Butterfly Board
2. Set STK500 to deliver VTG = 3 Volt
3. power down STK500
4. Power off JTAG ICE and Connect it to your PC
5. Connect the JTAG ICE to the Butterfly Board
6. Connect Power strap from STK500 to Butterfly
7. Power up STK500
8. Power up JTAG ICE

Load **Task1** (task1.prj) project in IAR Embedded Workbench (IAREW) and compile by pressing F9 or selecting [Projects] | [Build All] from the pull down menu.
Files are located in the task1 folder at this location: **C:\AVR\BUTTERFLY**
Take a look at the Build output dialog window and verify that it compiles without errors or warnings.

Open AVR Studio and select [Open] button (upper right icon).
Browse to the **c:\avr\butterfly\task1\debug\exe** folder and open the **task1.d90** file.
This is the UBROF file which contains all necessary debug information for AVR Studio.

Select **JTAG ICE as Debug Platform** and **ATmega169 as Device**. If you know which COM port the JTAG ICE is connected to, you may specify this in the "Connect" pull down box, or just leave it as is. AVR Studio will then search for it on all available ports.
Press "Finish" button.
You should now see the C source code, and a yellow arrow indicating the position of the program counter.

Press F5 to run the program, and have a look at the LCD Display. It should show a number. This number is the 8 most significant bits of an ADC measurement on the Light Dependent Resistor (LDR) located in the upper left corner on the Butterfly board. Verify that the value changes depending on the light on the LDR.

Task 1 Completed

Task 2: Breakpoints and IO View

Introduction

We will in this exercise take a closer look at how the application works, and explain the difference between hardware and software breakpoints.
The goal of the exercise is to understand how modifying bits in the IO view affects operation, and some of the limitations on breakpoints

Theory

The JTAG ICE breakpoints differs from what you will find in the other AVR Emulators. First of all you only have 3 general purpose hardware breakpoints to use. The term hardware breakpoints referee to the internal OCD system. It contains 3 general breakpoint registers which can contain breakpoint addresses. If more breakpoints are needed, break instructions have to be added in the code. This will require a reprogramming of the flash page containing the braked instruction. The JTAG ICE does not support software breakpoints in the current version (Studio 4.07). Such support could be added in later versions though. The 3 general purpose registers can also be combined to form complex break conditions as will be used in Task 4.

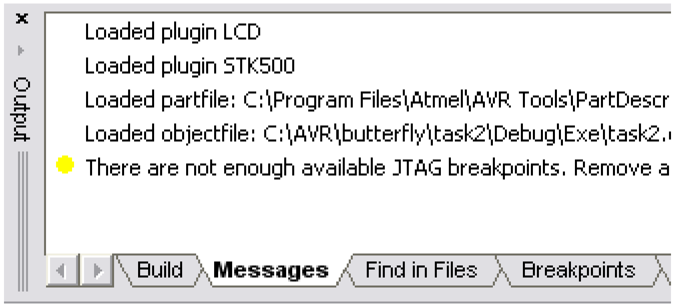
Task

Open and compile Task2 project in IAR Embedded Workbench (EW) Make sure you do not get any error messages or warnings.
Load the object file in AVR Studio, and run it to verify that it still works.
Restart the program emulation by pressing [SHIFT] | [F5] or pressing the "Reset" button in the toolbar.

Breakpoints

Place 3 breakpoints (press F9 or "Toggle Breakpoint" in the "Debug" menu) at 3 lines where you want to stop (break) program execution. The location of the breakpoints is not important. Start program execution by pressing [F5] or selecting [Run] in the [Debug] menu. Verify that program execution is stopped when the breakpoint is reached.

Now, try to add another breakpoint exceeding the 3 hardware breakpoints limit. What happens? A breakpoint is added. Now try to start program execution [F5]. You should notice that a warning appear in the "Output" window.
AVR Studio evaluates the number of breakpoints at runtime. This is done to allow the user to place the new breakpoints before removing the old ones.

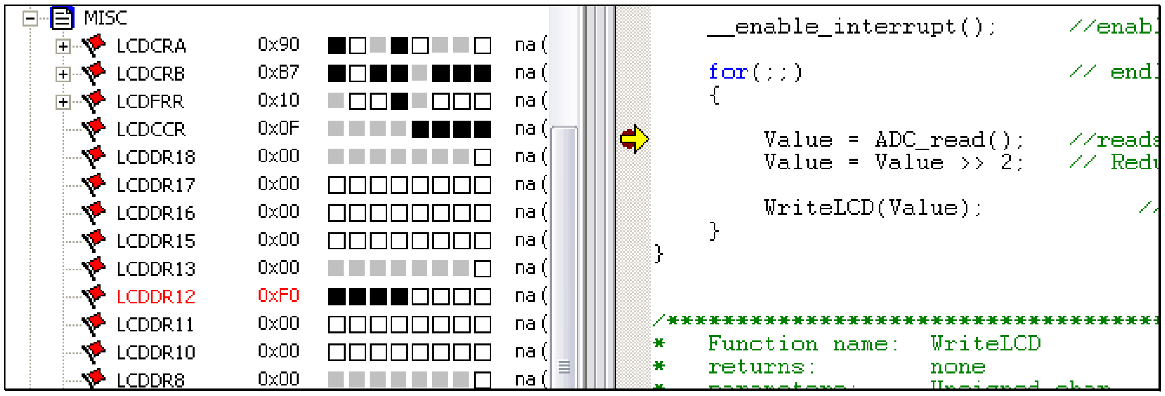


```
Loaded plugin LCD
Loaded plugin STK500
Loaded partfile: C:\Program Files\Atmel\AVR Tools\PartDescr
Loaded objectfile: C:\AVR\butterfly\task2\Debug\Exe\task2.
There are not enough available JTAG breakpoints. Remove a
```

Remove a breakpoint so you only have 3 breakpoints. Verify that the JTAG ICE now starts executing the code, and that it stops at the first breakpoint.

IO view

We are now going to take a look at the IO view, and how changes we make here affect operation.
Remove all previous breakpoints("Remove breakpoints" in the "Debug" menu), and place a single breakpoint on the first statement in the for(;;) loop as shown below. Run the program [F5]. The program will now run through all the initialization routines and stop in the main loop.


MISC

LCDCRA	0x90	■□▨■□▨▨□	na (
LCDCRB	0xB7	■□■■▨■■■	na (
LCDFRR	0x10	▨□□■▨□□□	na (
LCDCCR	0x0F	▨▨▨▨■■■■	na (
LCDDR18	0x00	▨▨▨▨▨▨▨□	na (
LCDDR17	0x00	□□□□□□□□	na (
LCDDR16	0x00	□□□□□□□□	na (
LCDDR15	0x00	□□□□□□□□	na (
LCDDR13	0x00	▨▨▨▨▨▨▨□	na (
LCDDR12	0xF0	■■■■□□□□	na (
LCDDR11	0x00	□□□□□□□□	na (
LCDDR10	0x00	□□□□□□□□	na (
LCDDR8	0x00	▨▨▨▨▨▨▨□	na (

```
    __enable_interrupt();      //enabl
    for(;;)                    // endl
    {
        Value = ADC_read();    //reads
        Value = Value >> 2;    // Redu

        WriteLCD(Value);            //
    }
}

/*************************************
*   Function name:  WriteLCD
*   returns:        none
```

Expand the "Misc" icon in the IO view. This contains the LCD registers. Use your mouse and check / uncheck some of the bits in the LCDDRxx registers. Notice that you now will turn on / off segments on the Butterfly LCD display even if the program execution is halted.

Why? Because changes done in the IO view are immediately updated in the AVR device and present when continuing running the device! (This has not always been the case. The first firmware version in the JTAG ICE did not support this functionality).

- Do you see the relation between the LCDDR registers and the LCD segments?
- Try changing the framerate and see how this affect the LCD (set LCDPS2 = 1)

Task 2 Completed

Task 3: Advanced Breakpoints and Stack

Introduction

In this exercise we will add a factorial calculation function to the Butterfly. This function is quite intensive on SRAM usage, thus perfectly suited to illustrate what happens when we get into a stack overflow situation. We will use advanced breakpoints to verify and trap the stack overflow.

Theory

Stack

From the IAR documentation regarding stack size:

> *"The compiler uses the internal data stack, CSTACK, for a variety of user program operations, and the required stack size depends heavily on the details of these operations. If the given stack size is too small, the stack will normally overwrite the variable storage which is likely to result in program failure. If the given stack size is too large, RAM will be wasted."*

The IAR compiler reserves an area in SRAM for variable storage during program execution. This reserved SRAM area is called CSTACK and is by default set to 0x20 (32) bytes. If your program tries to use more than the allocated amount, it will start writing other memory locations, with program failure as a likely result.

Factorial function

The factorial function: n! = 1 x 2 x 3 x 4 x … x n.
So: 3! = 1 x 2 x 3 = 6 and 12! = 479001600. This function is very easy to solve using recurstion. Simply have the function call itself with n-1 until we reach n=1. To calculate 3! F(3) = f(3 * f(2 * f(1))) = f(3 * f(2 * 1)) = f(3* 2) = 6.
Written as a C function it will look like this:

```
unsigned long fact(int x)
{
      if ( x == 1 )
            return(1);
      else
            return( x * fact( x-1 ) );
}
```

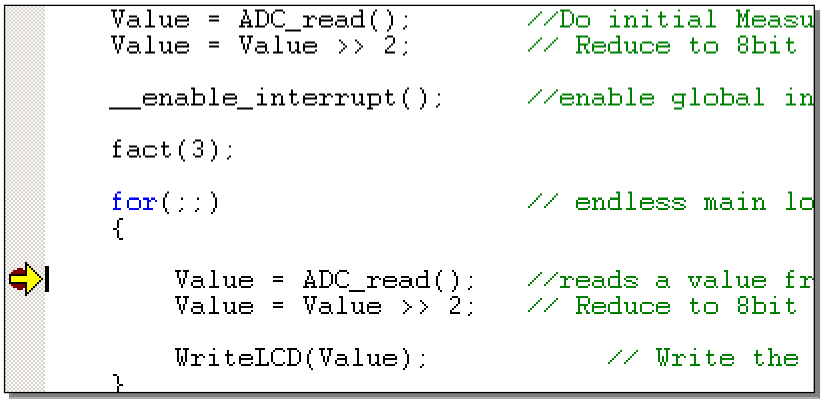
This C function will calculate the factorial x. The higher x is the more times it calls itself, thus eating up more of the CSTACK memory. At some point it will exceed the limits of CSTACK, and generate a stack overflow situation.

Task

Compile Task4 and open the object file in AVR Studio. Notice the fact(3); function located before the endless for loop. The CSTACK area is located after the extended IO area in the address range 0x100 to 0x120 (32 bytes is the default CSTACK size).

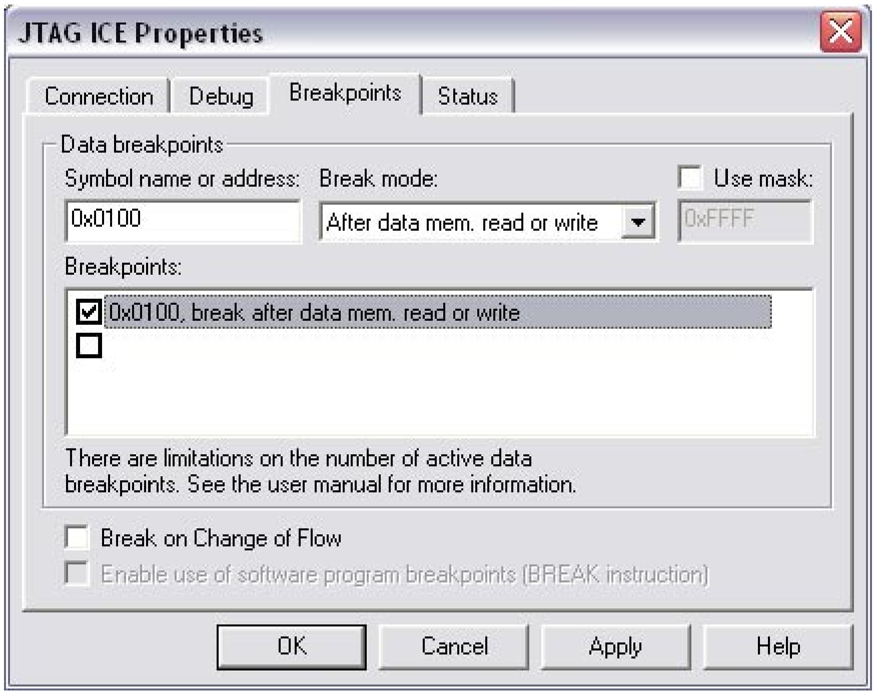
Place a breakpoint at the fist statement in the for(;;) loop as shown in the picture to the right.

Open the [Debug] | [JTAG Options] menu, select the [Breakpoints] tab, and add a Data breakpoint at location 0x0100 (see picture below). Set break mode to "After data memory read or writes".



```
    Value = ADC_read();          //Do initial Measu
    Value = Value >> 2;          // Reduce to 8bit

    __enable_interrupt();        //enable global in

    fact(3);

    for(;;)                      // endless main lo
    {

        Value = ADC_read();      //reads a value fr
        Value = Value >> 2;      // Reduce to 8bit

        WriteLCD(Value);              // Write the
    }
```

This advanced breakpoint will halt operation if location 0x0100 in data memory (SRAM) is written. This is the first location in SRAM, and it will be the last location written before the stack overflows and starts writing into the extended IO memory area.



JTAG ICE Properties

Connection | Debug | Breakpoints | Status

Data breakpoints

Symbol name or address: 0x0100

Break mode: After data mem. read or write

[ ] Use mask: 0xFFFF

Breakpoints:

[✓] 0x0100, break after data mem. read or write

[ ]

There are limitations on the number of active data breakpoints. See the user manual for more information.

[ ] Break on Change of Flow

[ ] Enable use of software program breakpoints (BREAK instruction)

Once both breakpoints are set press [F5] to execute the code. Notice that the program now reaches and stops at the breakpoint in the for(;;) loop. This indicates that the fact(3) does not reach the bottom the CSTACK space.

Now go back to the IAR EW and change the function to fact(10);
Recompile the code and restart the new code in AVR Studio. Run it, and you'll see that you now halt operation in disassembly mode. Change back to source view, do a single step, and you'll see that you are located in the fact function!
Single step the code and take a look at the LCD display. You'll notice that the function pushes data into the LCD registers!

Q: Why do we end up in disassembly mode?

A: ___________________________________

Q: What is the highest fact(n) that do not give you a stack overrun? (Tip: Use the memory view window to see how the SRAM is used by the fact(); function.

A: ___________________________________

As you can see, the advanced data breakpoints can be very useful when you want to verify that your pointers are not exceeding their valid boundaries.

Task 3 Completed

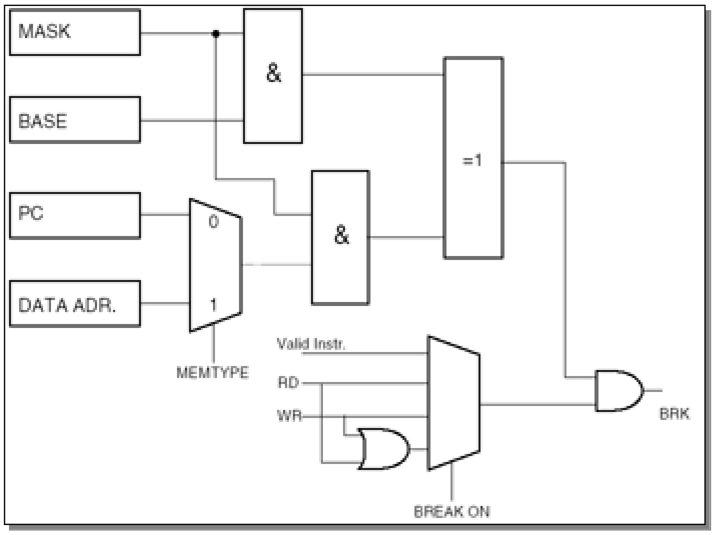
Task 4 Advanced Breakpoints and Masks

Introduction

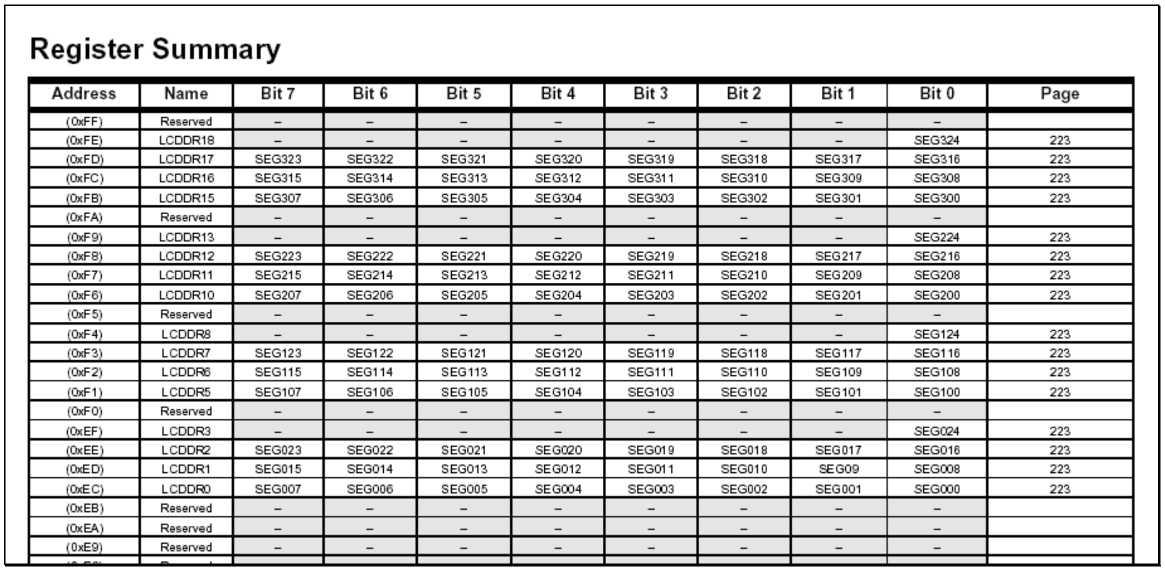
In the previous exercise we used advanced breakpoints to trigger a break when a specific data address was accessed. In this exercise we will look how to use the mask feature to trigger a break condition for a range of addresses.

Theory

The theory behind masked breakpoints are described in greater detail in the JTAG ICE User Guide (Technical Library CD) but can be summed up as follows:



When using masked breakpoints a base address and the mask is AND’ed together to form a set of break vectors. These are checked against the program counter or the data address AND’ed with the same mask to see it a valid break condition has occurred.



Register Summary

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(0xFF)	Reserved	–	–	–	–	–	–	–	–	
(0xFE)	LCDDR18	–	–	–	–	–	–	–	SEG324	223
(0xFD)	LCDDR17	SEG323	SEG322	SEG321	SEG320	SEG319	SEG318	SEG317	SEG316	223
(0xFC)	LCDDR16	SEG315	SEG314	SEG313	SEG312	SEG311	SEG310	SEG309	SEG308	223
(0xFB)	LCDDR15	SEG307	SEG306	SEG305	SEG304	SEG303	SEG302	SEG301	SEG300	223
(0xFA)	Reserved	–	–	–	–	–	–	–	–	
(0xF9)	LCDDR13	–	–	–	–	–	–	–	SEG224	223
(0xF8)	LCDDR12	SEG223	SEG222	SEG221	SEG220	SEG219	SEG218	SEG217	SEG216	223
(0xF7)	LCDDR11	SEG215	SEG214	SEG213	SEG212	SEG211	SEG210	SEG209	SEG208	223
(0xF6)	LCDDR10	SEG207	SEG206	SEG205	SEG204	SEG203	SEG202	SEG201	SEG200	223
(0xF5)	Reserved	–	–	–	–	–	–	–	–	
(0xF4)	LCDDR8	–	–	–	–	–	–	–	SEG124	223
(0xF3)	LCDDR7	SEG123	SEG122	SEG121	SEG120	SEG119	SEG118	SEG117	SEG116	223
(0xF2)	LCDDR6	SEG115	SEG114	SEG113	SEG112	SEG111	SEG110	SEG109	SEG108	223
(0xF1)	LCDDR5	SEG107	SEG106	SEG105	SEG104	SEG103	SEG102	SEG101	SEG100	223
(0xF0)	Reserved	–	–	–	–	–	–	–	–	
(0xEF)	LCDDR3	–	–	–	–	–	–	–	SEG024	223
(0xEE)	LCDDR2	SEG023	SEG022	SEG021	SEG020	SEG019	SEG018	SEG017	SEG016	223
(0xED)	LCDDR1	SEG015	SEG014	SEG013	SEG012	SEG011	SEG010	SEG09	SEG008	223
(0xEC)	LCDDR0	SEG007	SEG006	SEG005	SEG004	SEG003	SEG002	SEG001	SEG000	223
(0xEB)	Reserved	–	–	–	–	–	–	–	–	
(0xEA)	Reserved	–	–	–	–	–	–	–	–	
(0xE9)	Reserved	–	–	–	–	–	–	–	–	

In this exercise we will use masked breaks to halt operation if any of the LCD registers are written. To do this we have to construct a mask so that all accesses to these registers result in a valid break condition. Use the JTAG ICE User Guide and set up a suitable Mask and Base address to achieve this. To make it simpler; break on all accesses starting from 0xE0 to 0xFF (see register summary figure above, or use the ATmega169 datasheet on the TechLib CD)

Find a Base and Mask that will do this:

Base: ________________________

Mask: ________________________

Task

Compile and start Task4 project in AVR Studio. Enter the mask and base values in the JTAG ICE breakpoint dialog, and verify that the JTAG ICE halts operation when accessing locations as determined by your breakpoint mask and base. (We will use the same fact(10); function as we know this will overflow the stack and write to the LCD registers in extended IO space.

Task 4 Completed

Task 5 Max and Min Values

Introduction

In this exercise we will add two variables that will store the maximum and minimum values read from the LDR. We have added a function that reads the joystick and returns a value depending on movement (up, down, left, right, push). Use this function to display max value if pushed up, min value if pushed down, and currently measured value otherwise.

Task

Open task5, complete the code in IAR EW, and test it out on the JTAG ICE and Butterfly.

Hints

Will be given as you work (you might write them down for later reference here)

____

____

____

____

Task 5 Completed

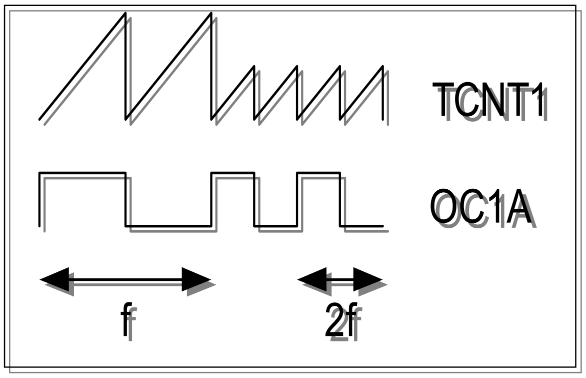
Task 6 Sound Player

Introduction

Now we are going to turn the Butterfly into a musical instrument! We will add a function that will change a tone played on the buzzer based on the measured LDR value.

Theory

In normal PWM mode we only change the duty cycle of the signal. So to be able to use the PWM for frequency generation we will make use of a special mode. The Clear Timer on Compare match (CTC) mode. In this mode the TCNT1 register will be cleared when it reaches the OCR1A value. By lowering the value in this register, the TCNT1 reaches the value quicker, thus generating a higher frequency on the OC1A pin (see figure). Note that the OC1A pin must be configured as output.

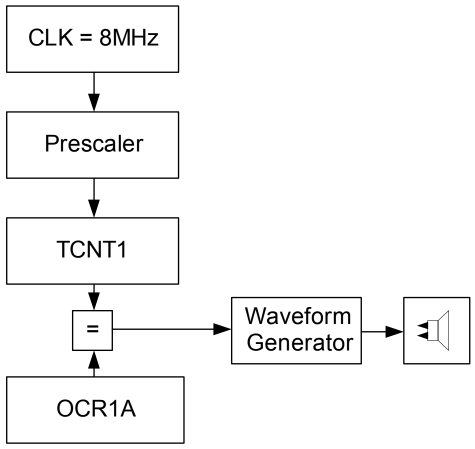


Task

Make a program that uses the measured value on the LDR as OCR1A value for the Timer Counter 1.

- Clear Timer on Compare Match (CTC) Top value in OCR1A
- Prescaler = 1
- Toggle OC1A on compare match
- Turn sound on by pressing stick up
- Turn sound off by pressing stick down

By changing the light on the LDR the buzzer will now play different frequencies.



Hints

(To set up the Timer Counter Register, you need to configure these registers: **TCCR1A**, **TCCR1B**, **OCR1AH/L** and **DDRB)**

Task 6 Completed ?