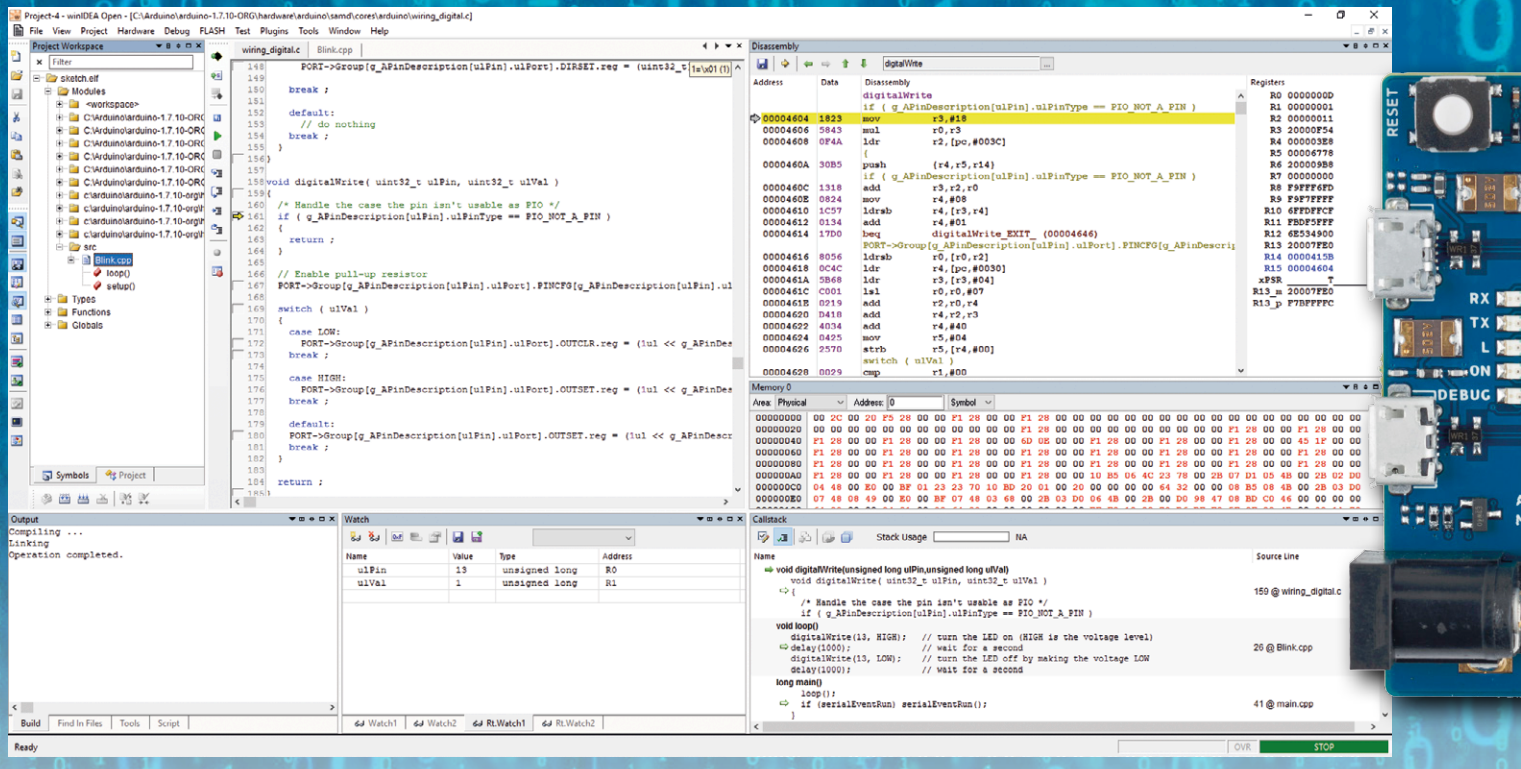
Debugging the Arduino Zero & M0 Pro

Delving deeper into the world of Arduino



By **Stuart Cording** (iSystem, Germany)

Boards like the Arduino M0 Pro and DUE provide, respectively, the same pin-out as their 8-bit Arduino Uno and Mega cousins, but with the advantage of significantly more SRAM, flash memory and CPU clock speed. Perhaps the only disadvantage or challenge with these boards is the 3.3-volt restriction on the input and output pins. However, these advantages, combined with support for many of the available shields on the market and the established base of libraries, make the 'upgrade' simple and fast.

The biggest challenge remaining for the advanced hobbyist or professional using Arduino for rapid prototyping is the limited Integrated Development Environment (IDE). For entry into the world of Arduino, the Arduino IDE is perfect with its simplicity and clarity. Once a sketch becomes slightly more involved, however, and failures in the code cannot be found with a blinking LED or serial message output alone, the sheer power of a professional IDE is highly alluring.

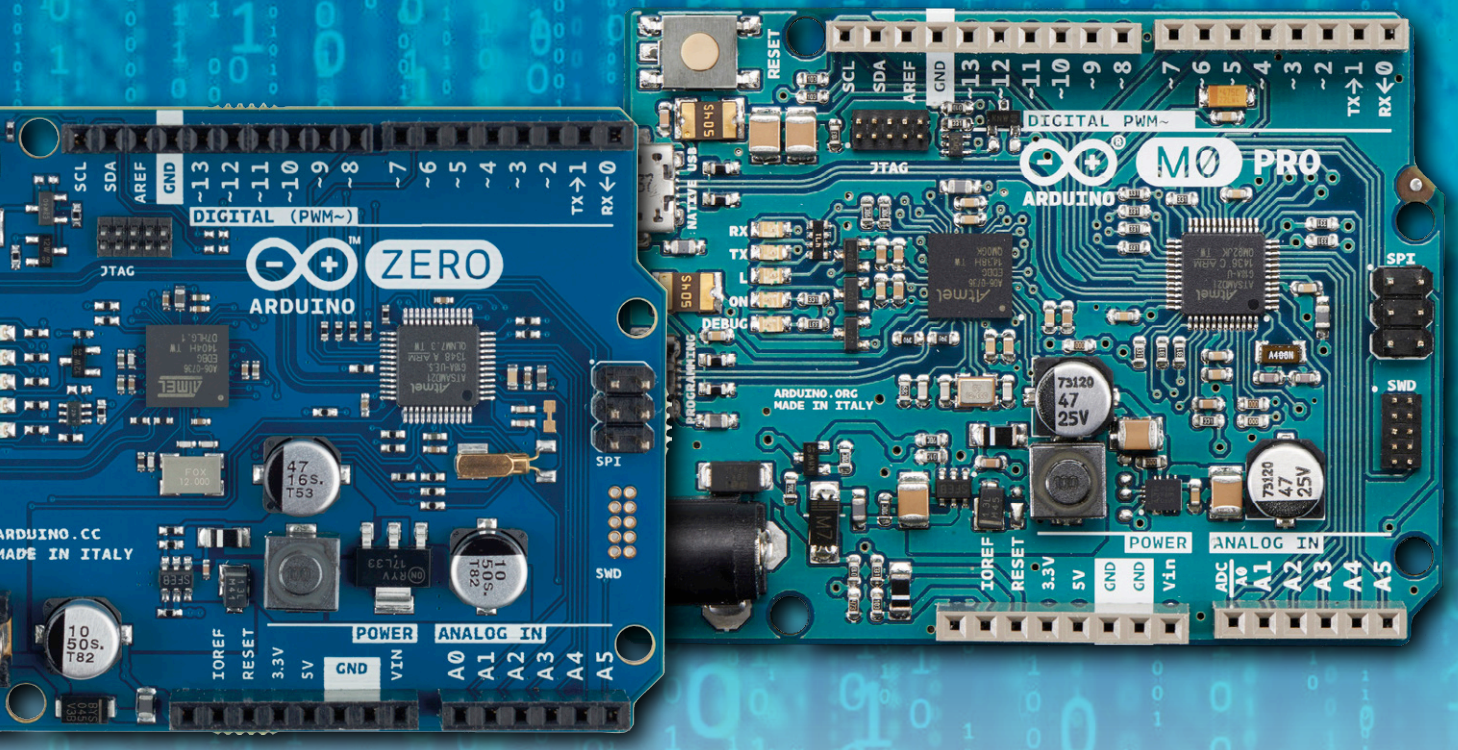
Debug (as) a Pro, Pay Nothing

The Arduino M0 Pro (from Arduino.org) as well as the Arduino/Genuino Zero (from Arduino.cc), based on the SAM D21G from Atmel (now Microchip), are kitted out with a slightly different programming interface compared to previous boards. The device behind the USB "programming" connecter is also an Embedded Debugger (EDBG) which can be used by many different development environments as an interface to not only program the board but additionally to interrogate the internal workings of the MCU at the heart of the system. With an appropriate IDE, it is then possible to debug not only your sketch, but the entire Arduino core code and libraries.

One such IDE is iSystem's winIDEA Open, the free version of their IDE targeted at MCUs that use ARM's Cortex-M processor technology. As a professional tool it is easy to get started, as, at the most basic level, all that is needed is the output file from a sketch built in the classic Arduino IDE. However, as we will discover, the build manager can also be invoked to speed up the compilation of sketches. Additionally, the integrated test environment testIDEA can be used to check whether any bugs have scurried into our project during development.

There are many details and nuances behind configuring and using such a tool. In order to provide both an overview for those simply interested in this concept as well as detailed instructions for those who would like to go hands on, the overview in this article is supplemented by a series of tutorials and code available online [1].

With the introduction of ARM Cortex-M-based 32-bit microcontrollers to the Arduino portfolio, the maker has some very powerful hardware in their hands. For entry into the world of Arduino, the simplicity and clarity of the Arduino code editor is perfect. However, once a sketch becomes more involved, the power of a professional development environment becomes highly alluring.



My First Project

When starting a new project for a SAM D21 MCU in other IDEs, such as Atmel Studio which has been covered by Elektor before [2][3], you really need to start from a template and have to include many pre-defined library source code files. The winIDEA Open IDE is different in that it forms the premise that the output of the compiler toolchain, for example the ELF file containing the binary code of your Arduino sketch, is the most important file of your project. From this single file, the IDE can find all of your source code and the core Arduino files and libraries needed to debug the application code. Thus the simplest method to debug an Arduino sketch is to create and build it in the Arduino IDE and then import and download the resulting ELF file into the Arduino M0 Pro using winIDEA Open.

In the first three projects of Tutorial 1 [1], the classic 'Blink' sketch is created in the standard Arduino IDE. Due to slight differences between the Arduino M0 Pro and Arduino/Genuino Zero, winIDEA Open needs to be configured slightly differently. However, for those who just want to get started, a preconfigured workspace is provided allowing the developer to simply open the appropriate workspace and download the ELF file that the Arduino IDE has created.

Once the code is in the MCU, it is then possible to start exploring the internal workings of the Arduino core code. In the "Project Workspace" window (**Figure 1**) all the files that belong to the project, along with all the used functions, are listed in a tree structure just like in Windows' File Explorer. So, if you have ever wondered what code lies behind the `delay()` function, simply expand the "Functions" folder, scroll down until you find `delay(unsigned long ms)` and double-click. In the editor window, the function will be displayed. If you are interested to see when the function is called, you can set a break-point on a line of code where a grey box is also shown in the editor's gutter (the grey region in the text editor to the left of the line numbers). Simply right-click with the mouse and select "Set Breakpoint". Once the code is restarted, the MCU will halt when it reaches this breakpoint, allowing you to analyse the content of variables, registers and memory.

Note that, despite the Arduino environment's apparent simplicity, the code in its core software makes good use of many of the advanced tricks the C programming language has to offer. As such, some 'symbols' (names of variables and functions) may appear listed in the Project Workspace window but may not actually be associated with any code of significance in the software.

Speeding Up the Workflow

One of the issues with the method discussed thus far is that the original `Blink.ino` cannot be debugged properly. The issue

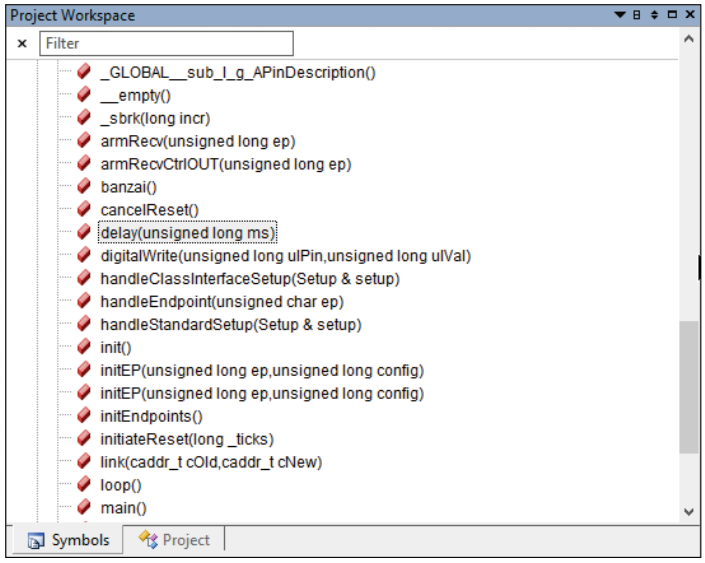


Figure 1. The 'Functions' view shows all the functions included in the sketch, even those belonging to the Arduino core and library code.

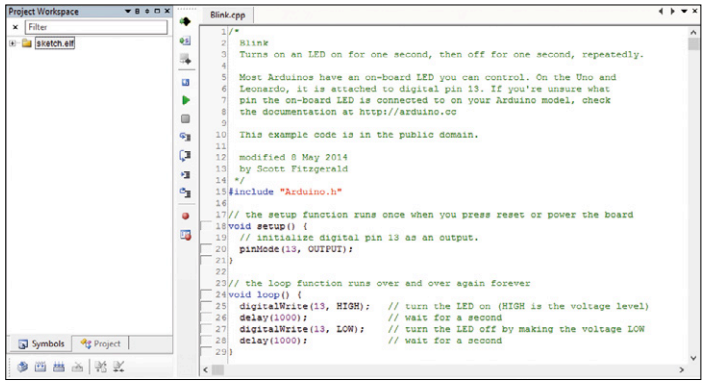


Figure 2. Blink.cpp in the winIDEA Open IDE.

Figure 3. The Blink sketch in the Arduino IDE.

relates back to how the Arduino IDEs compile the sketch. For the beginner, this is really no problem, but when wanting to go deeper into the code, it would be nice to be able to build a sketch outside of the Arduino IDE.

There is also another benefit to be gained from this. One of the most frustrating things about the standard Arduino IDEs is that every compilation rebuilds the entire project from scratch. Again, for smaller sketches this is no big deal. However, as soon as a couple of libraries are included and you have several source files, rebuilding and programming the board starts to become quite a laboured activity. Here again, a professional IDE can help by providing the means to build the project outside of the IDE.

In order to make the build process more intuitive and faster, Tutorial 2 [1] uses a Makefile to build our sketch. This requires a few changes to the creation of your sketch as follows:

- Your sketch will need to be stored in a file named `<SKETCH NAME>.cpp` and stored in a folder named 'src'.

Your sketch will need the line of code `#include "Arduino.h"` (see **Figure 2**) added to the top of it (before the setup() function call).

In addition, you will need the make utility which can be installed as part of MinGW. Detailed instructions on how to do this can be found in Tutorial 2, available from [1].

Now, instead of building the sketch in the Arduino IDE (**Figure 3**) and then simply programming and debugging the code in the winIDEA Open IDE, the whole process can be undertaken in winIDEA Open.

As a result, we can now debug our sketch. The project workspace works in a manner similar to that of Windows Explorer, allowing the elements listed to be "expanded" to show their contents in the same way folders can be expanded. In the Project Workspace simply expand the elements listed using the plus symbol to the left of each element in the following order: sketch.elf → Modules → src → Blink.cpp (**Figure 4**). Finally, double-click on `loop()` or `setup()` and the editor will open the source code at the respective line of code where the function is implemented. As we see in the gutter, each line of code has a grey box associated with it allowing us to set a breakpoint in the code execution if we wish. Simply right-click on the desired line of code and select "Set Breakpoint".

Tips & Tricks

Now is probably a good time to note the limitations of the debug features of the SAM D21. In total, only three breakpoints can be set at any one time so, at some point, you will get a window pop up explaining that all breakpoint resources are used up (**Figure 5**). The easiest method to work around this is to disable a current breakpoint (right-click on a line with a breakpoint and select 'Disable Breakpoint') and then set the new breakpoint at the location of your choosing. It is also possible to 'Clear Breakpoint', but disabling a breakpoint has the advantage of leaving a red marker in the gutter of the code editor as a reminder of where the breakpoint previously was. This is especially helpful as you jump back and forth in the code.

In the event that you are interested in the efficiency of the code generated by the Arduino tool chain (which incidentally is GCC), we can also open a disassembly window from the menu View

➞ Disassembly. If we click inside this window (**Figure 6**), any further single-stepping of the code will be executed instruction by instruction, instead of source code line by source code line as is the case in the editor window. In this way it is also possible to see how the CPU registers are used during the calling of functions to pass parameters, amongst other things.

Beyond Debugging

Typically, when developing an application, specific functionality will be broken out into separate functions. For small projects, where perhaps a single developer is responsible, it is relatively easy to keep track of what is working, what isn't and where a bug might be when the code fails. However, for larger, multi-programmer projects, it can help to have a battery of tests available that, when executed, ensure that the code is still operating as originally specified. For this purpose, winIDEA includes the Original Binary Code (OBC) unit testing tool testIDEA. Here we will use it to develop some tests for a function that evaluates an imaginary input value and returns a new time delay for our `delay()` function call.

The algorithm implemented in the code is quite simple (**Figure 7**). If the passed value is less than 50, it returns 150. If the passed value is between 50 and 99, it returns 1000. For all other values, it returns 1750. Having developed the code, it would make sense to develop a short test that could be executed to ensure that it still works, even if someone else makes changes and introduces a bug into the code.

Before writing tests for a function, it is worthwhile initially considering how you would go about testing it by writing the steps down on a piece of paper or making a small table of input and expected output values. In the **Table 1** we have focused on input values at the extremes of possible inputs as well as values that are close to the boundaries defined in the algorithm (50 and 100).

In order to develop our tests, from the menu bar, we select Test ➞ Launch testIDEA. Upon starting testIDEA, the tool suggests creating a 'test specification file'. This file is where our tests will be stored and the file extension is 'iyaml'. Here we explain how to create a test:

- From the menu, select Test ➞ New Test…
- Now we need to define the function to be tested. Before we can do this, we have to refresh the link to our winIDEA project by clicking the refresh symbol. Once complete we can select our function to be tested, `evaluateNumber()`, from the drop-down list.
- In the Parameters box we enter the value to be passed to the function. For test number 1, this is 0.
- Next we enter the Expected Result, first selecting the 'Default expression' radio button. In the field we enter 150, which is the expected response entered in our table.
- Click OK and the first test is complete.

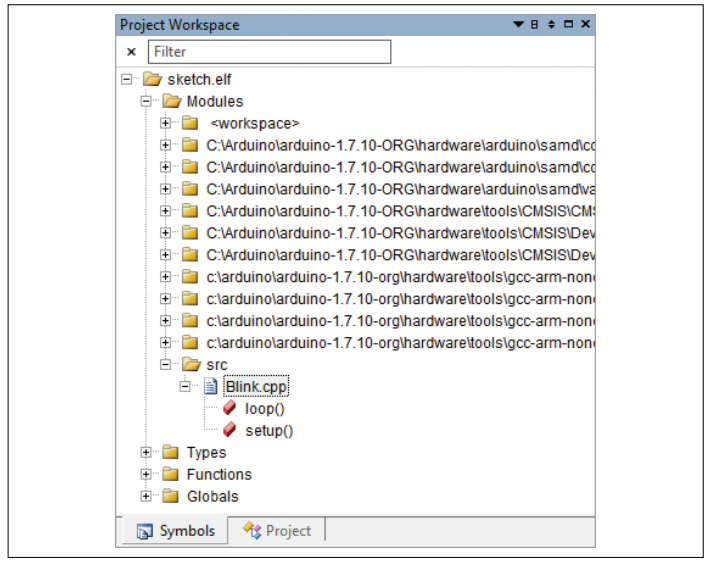


Figure 4. The sketch executable can be explored as if it was a folder tree.

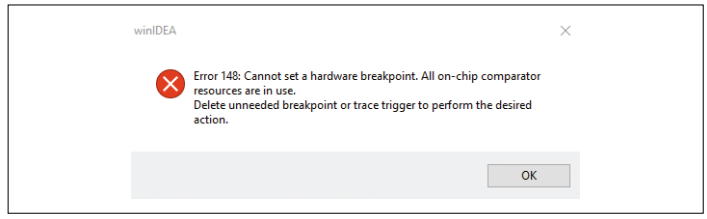


Figure 5. Message when no more breakpoints are available.

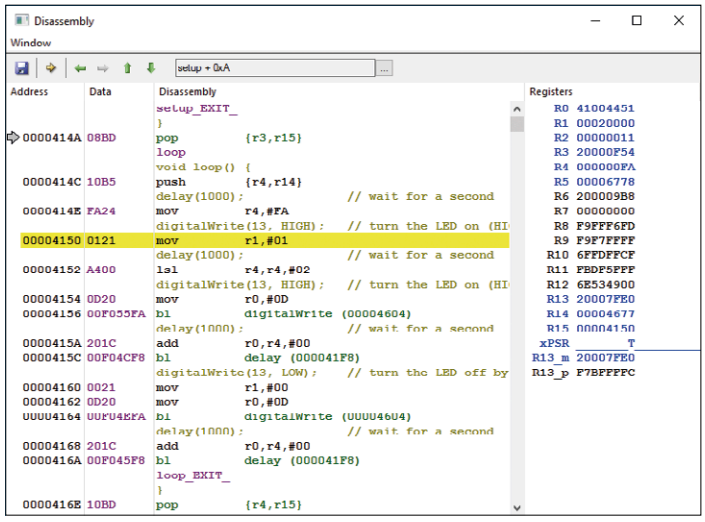


Figure 6. Disassembly view for the Blink sketch.

Table 1. Input values to exercise our algorithm and the expected output values.

Test Number	Input Value	Expected Response
1	0	150
2	1	150
3	48	150
4	49	150
5	50	1000
6	51	1000
7	98	1000
8	99	1000
9	100	1750
10	101	1750
11	150	1750
12	200	1750
13	255	1750

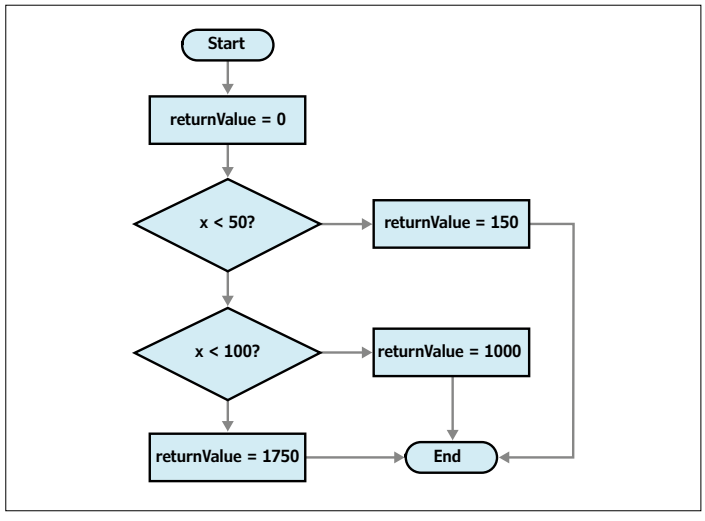


Figure 7. Flow diagram of a simple algorithm.

The uniqueness of OBC testing is that the testing itself all occurs on the MCU target in use and not, like in some other unit testing tools, in a simulation of the MCU. When we now execute this test, the test will be actually executed on the MCU target after downloading the code into the MCU's flash memory. This is performed by selecting Test ➞ Run All Tests. Hopefully, upon executing the tests, the result will be "OK", indicated by a small green checkbox next to the test in the Outline panel. In order to distinguish all the different tests from one another, we can add some meta data to our test. In the Form panel, select 'Meta' and add a Test ID such as 'Test_1'. We can now go on to add the remaining tests to our testing plan. Upon completion, we can run all the tests which should result in a pass in each case. Of course, the real value is in using the tool to find any bugs that have slipped in whilst we weren't looking. Let us assume that a colleague on the team misunderstands the specification and decides that all the less-than ('<') comparisons should actually be less-than-or-equal ('<=') comparisons and changes them in your code. The first time you become aware of the issue is when the project as a whole does not work as expected. To trial this potential mistake, replace the '<' with '<=' at lines 20 and 22 in the `Blink.cpp` source code (Tutorial 3, Project 5 [1]). In order to see where the bug may have occurred, simply rebuild the application in winIDEA, open up the test specification in testIDEA and run all tests. You should see that tests 5 and 9 now fail and, working from that result, it should be possible to determine the cause of the error.

Sounding off

The Cortex-M-based Arduino boards have enormous potential, especially in the area of rapid prototyping. STMicroelectronics recently announced a Cortex-M4 STM32-based board in the form of the STAR Otto. Such boards will likely be used for complex prototyping with Ethernet and Wi-Fi shields. A professional debugging environment, such as winIDEA Open, will certainly provide relief when searching for why a sketch is not quite working as expected. ⏮

(160228)

Web Links

[1] www.elektormagazine.com/160228

[2] www.elektormagazine.com/130392

[2] www.elektormagazine.com/140037

The SAMD21 is just one in the range of SAM D ARM Cortex-M0+ based, 32-bit microcontrollers from Atmel. These devices offer up to 256 kB of flash memory and 32 kB of SRAM. Additionally, the devices boast a wide range of interfaces, including full-speed USB, a real-time counter for implementing a clock, USART, SPI and I²C interfaces, and a 12-bit ADC. Optimal energy efficiency is also covered with the power consumption lying at 70 µA/MHz. The family also features an automotive grade version of the MCU and some devices can be used to implement a capacitive touch interface, making use of Atmel algorithms to implement buttons, sliders and wheel user interfaces.

www.atmel.com/products/microcontrollers/arm/sam-d.aspx