Advanced Debugger for Arduino

Jan Dolinay[1], Petr Dostálek[2], Vladimír Vašek[3]
Faculty of Applied Informatics
Tomas Bata University in Zlín
Zlín, Czech Republic

***Abstract*—This article describes improved version of our source-level debugger for Arduino. The debugger can be used to debug Arduino programs using GNU debugger GDB with Eclipse or Visual Studio Code as the visual front-end. It supports all the functionally expected from a debugger such as stepping through the code, setting breakpoints, or viewing and modifying variables. These features are otherwise not available for the popular AVR-based Arduino boards without an external debug probe and modification of the board. With the presented debugger it is only needed to add a program library to the user program and optionally replace the bootloader. The debugger can speed up program development and make the Arduino platform even more usable as a tool for controlling various experimental apparatus or teaching computer programming. The article focuses on the new features and improvements we made in the debugger since its introduction in 2016. The most important improvement over the old version is the support for inserting breakpoints into program memory which allows debugging without affecting the speed of the debugged program and inserting breakpoints into interrupt service routines. Further enhancements include loading the program via the debugger and newly added support for Arduino Mega boards.**

Keywords*—Arduino; debugger; microcontroller; software debugging

I. Introduction

Arduino is a very popular prototyping platform with a microcontroller (MCU). It started as an educational tool in 2003 and evolved into a widespread platform for prototyping, controlling various devices and for teaching computer programming. It is now frequently used in courses focused on embedded systems, robotics, and the like. For example, [1] describes successful use of the platform in computer science capstone course, [2] used Arduino-based custom board to increase student's interest in programming courses and [3] utilized Arduino as the base for their educational mobile robot. A comprehensive review on this topic was presented e.g., by [4]. Arduino is also used in scientific laboratories as a low-cost multipurpose device for controlling various experimental apparatus in a wide range of areas. For example, [5] concludes that Arduino boards may be inexpensive tool for many psychological and neurophysiological labs, [6] based their device to abate tremors for patients with Parkinson's disease on this platform, [7] uses Arduino to generate pulsatile flow rate for biofluid dynamics research, [8] uses distributed network of Arduino boards acting as remote servers in a system controlling capacitive energy storage and [9] used Arduino for real-time monitoring of air quality in urban area. Yet another area of its use is the emerging technology of Internet of Things (IoT), as described by [10, 11] and others.

The Arduino hardware is a printed circuit board with a microcontroller. A program must be written, built, and uploaded to the MCU for the Arduino to be able to perform requested tasks. There is a software tool, integrated development environment (IDE), provided with the platform to accomplish this. It is also possible to use other tools to create the programs for Arduino, for example, Eclipse or Visual Studio Code. These tools offer more functionality than the simplistic Arduino IDE and are therefore preferred by many advanced users.

One feature which is commonly missed by advanced users is a source-level debugger. The Arduino IDE does not provide any interface for source-level debugging. The alternative IDEs do provide such interface, but the most popular Arduino boards based on AVR microcontrollers, such as Uno, Mega or Nano, cannot be debugged without an external debug probe and alteration of the board. Thus, most users debug their programs by printing textual messages to serial interface, which is usable, but not very effective or comfortable.

A debugger that lets the user stop the program, view, and modify variables or execute the program step by step can save significant amount of time in localizing problems, especially in more complex projects. It can also greatly improve the usability of the Arduino platform for teaching computer programming. For the novice programmers it is easier to create the mental model of the programming constructs they are learning if they can use a debugger to single step through lines of code, set breakpoints, and watch the internal state of the program as well as the outside effects, such as an LED turning on. Debugger is also an essential tool for teaching debugging skills, which is recognized as an important part of computing curricula [12, 13].

As follows from the above a source level debugger is a highly desirable feature, which can make the Arduino platform more usable both for teaching programming and for implementing various devices. We developed first version of such a debugger in 2016 [14]. This debugger had several limitations which affected the performance of the debugged program and thus the usability of the tool. In this article, we present a significantly improved version of the debugger, which overcomes limitations of the first version and offers features and comfort of use at the level expected from fully-fledged, hardware-based debugger.

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic within the National Sustainability Programme project No. LO1303 (MSMT-7778/2014) and also by the European Regional Development Fund under the project CEBIA-Tech No. CZ.1.05/2.1.00/03 xxx.

II. Debugger Performance Improvements

As already mentioned, in 2016 we developed a source-level debugger for the AVR-based Arduino boards. The unique feature of this debugger is that it makes it possible to debug programs at source level without any external tools. This can be particularly useful for educational purposes – there is no extra cost of buying a hardware debug probe for each workplace in the lab and no extra work with modifying the Arduino boards to be able to communicate with the probe.

However, the first version of the debugger had several limitations - it was implemented for Arduino Uno only, the program execution was significantly slower when any breakpoints were set, and it was not possible to set breakpoints into interrupt service routines (ISR). After receiving positive feedback from the users, we decided to improve the debugger to overcome these limitations.

A. Debugging Options for Arduino

There are three options for debugging the AVR-based Arduinos besides printing messages to serial line. First option is to use a hardware debugger – a debug probe, which connects to the debug pin of the MCU. Unfortunately, there is a capacitor attached to this pin on the Arduino boards which must be disconnected for the communication with the debug probe to work. Newer revisions of the board are equipped with a solder bridge which can be cut to enable this feature, making the modification relatively easy. Nevertheless, it is a modification which needs to be reverted if the normal program uploading via bootloader is needed. Another problem is that the debug protocol is proprietary and therefore a commercial debug probe must be obtained. The prices of such probes start at around $50 for the most affordable Atmel-ICE-PCBA tool.

The second option is to use VisualMicro Arduino IDE for Visual Studio, which contains a tool called Serial debugger. This debugger is based on inserting code into the user program to communicate with the IDE. This added code is not visible to the user and the user experience is quite satisfactory, but there are major limitations to this approach - it is not possible to insert breakpoints during debugging; a rebuild and re-upload is required. Also, stepping through the code is not possible - the program can only be run from one breakpoint to the next one. Moreover, the VisualMicro is a paid software; the prices start at $12 for a one-year student license or $49 for a permanent license. All this, together with the fact that it is only available as an extension for Visual Studio, a complex and hardware intensive IDE, probably makes it less than ideal solution for many users.

The third option is a debugger based on GDB stub mechanism, which is described in the next section. The advantage of this approach is that the features of such a debugger are very similar to a hardware debugger, including stepping through the code, inserting breakpoints in runtime, and viewing and modifying the variables. Also, the stub is not limited to certain IDE; it can be used with GDB in command line as well as with various IDEs. We verified the solution with Eclipse IDE, Visual Studio Code and PlatformIO which are all free, multiplatform, and relatively light-weight environments. As far as we know, our GDB stub presented here is the only such stub for the AVR based Arduinos available.

To summarize, to be able to debug Arduino programs at source level, the other options besides our solution are either modifying the board and buying a debug probe for approx. $50 or buying the Visual Micro software solution which has limited features. We believe that our solution is an attractive option especially for educators, as it is completely free, can be used in Windows, Linux or Mac and works with the Arduino board as-is, without any hardware modification.

B. Principle of Operation

The debugger is based on so-called debugger stub for the GNU debugger GDB. Debugger stub is a small program that runs on the debugged computer and communicates with the debugger running on development (host) computer [15].

To enable debugging, users must insert a program library (the stub) into their code and then they are able to connect to the running program and debug it with the GDB. We first presented this solution with Eclipse IDE, but it can also be used from command line or with any IDE that can integrate with GDB, notably the popular open-source editor Visual Studio Code. The principle of communication is shown in Fig. 1. Our software component, GDB stub, is part of the user program running on the Arduino board (target system). The stub handles serial communication with the GDB debugger running on the host system (desktop computer). This way the GBD can control the program, view the memory, etc.

C. New Breakpoints Implementation

The key new feature described in this paper is the ability to set breakpoints into program memory. In general, breakpoints are implemented by modifying the program code at the position where the execution should be suspended. The original instruction is replaced by another instruction that redirects the execution to the debugger. This is easily achieved on desktop computers, as the program is located in RAM memory, which it is easy to modify. The problem with using this technique in microcontrollers is that the program memory is typically based on flash technology and cannot be easily rewritten in runtime.

MCUs are commonly equipped with a debug module which takes care of inserting the breakpoints, single stepping etc. However, to access this module a special hardware - a debug probe is required. Moreover, the AVR-based Arduino boards can only be used with such a probe after modification of the printed circuit board, as mentioned in Section A.

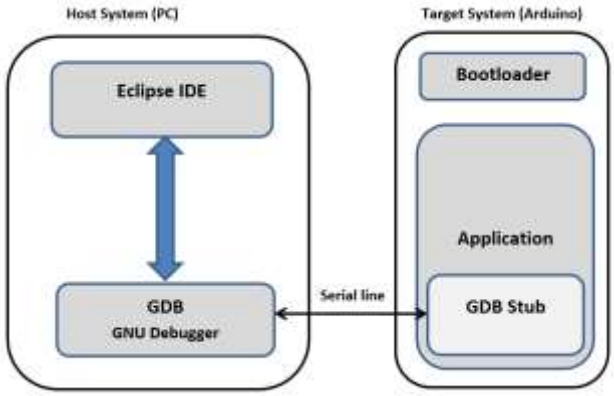

Fig. 1. Principle of Debugging with our GDB Stub.

We aimed to provide accessible debugging option without the relatively expensive hardware probe and modification of the board. That is why we implemented the debugger using the debugger stub technique mentioned above, without using the debug module of the MCU. The first version of our debugger introduced in [14] did not insert breakpoints into the program memory because it seemed too complicated at that time. Instead, breakpoints were implemented by comparing the current position in the program with a list of desired breakpoint addresses after executing every instruction of the CPU. Obviously, this considerably decreases the execution speed of the debugged program, but in many situations, it is not a problem and the debugger can be successfully used. The advantage of this approach is that the users simply add a library to their program; no other action is required.

However, setting breakpoint directly to the program memory promises significant benefits and we decided to implement this feature in the new version. In the following text we refer to these new breakpoints as "flash breakpoints" and to the older breakpoints as "RAM breakpoints". The advantage of the flash breakpoints is that the program can run at full speed, with virtually no intrusion, until a breakpoint is hit. Moreover, flash breakpoints can be set into interrupt service routines (ISR), which is not possible with the RAM breakpoints. This follows from the principle of operation of the RAM breakpoints – after each instruction of the main program an ISR must be executed to examine current position of the program - and the AVR CPU can only execute one ISR (which is used by the debugger) and the main program in this mode. On the other hand, flash breakpoints are implemented by replacing the original instruction at the position where the program should stop with another instruction that redirects execution back to the debugger. Thus, they do not require any use of an interrupt for monitoring current position of the program which would grade the execution speed. The details of the implementation are provided in the following section.

III. Results and Discussion

In this section we describe the implementation of the new version of the debugger stub which can be useful for better understanding of the features and limitations of this solution. We also present sample cases of running the debugger.

A. New Breakpoints and Program Load Implementation

To implement the flash breakpoints, we needed to solve two problems. First problem is that on the ATmega328 MCU it is only possible to rewrite the flash memory from code running in special memory section called the bootloader section. This section already contains the Arduino bootloader which handles loading user programs from the IDE. In normal course of operation, user programs (including our debugger stub) cannot be loaded into this section.

To solve this problem, we developed custom version of the bootloader which works in the same way as the standard bootloader - it allows uploading the user program without debugging, but additionally it provides service to the debugger stub to write to program memory. The principle is depicted in Fig. 2. When the stub needs to set a breakpoint in the program memory it calls a routine in the custom bootloader to modify the flash memory. The bootloader also provides a routine for loading new application program as described later.

The second problem was how to replace the original instruction in the memory when setting a breakpoint. To stop the program, we need to execute some code that will pass the control from the debugged program to the debugger. Often, there is a special instruction in the instruction set of the target CPU to break the execution and jump to the debugger, or an instruction for software interrupt which can be used to pass the control to an appropriate ISR handled by the debugger. However, in the AVR architecture neither of these are available.

The simple solution would seem to be to replace the original instruction with a jump to the debugger, but such a jump would require overwriting several bytes of the memory – the jump instruction together with its target address. Yet we can only replace single instruction by the breakpoint; we cannot overwrite the following instruction. Consider the case of setting a breakpoint one instruction before the location which is the target of a jump or a subroutine call. If the breakpoint replaces not only the intended instruction but also the following one, the program will crash when it jumps to the now-damaged location after the breakpoint. From this it follows that the instruction to be used as a breakpoint must not be longer than the shortest instruction of the CPU – which is just one word. This significantly limits the available options.

Our first solution was to use an external interrupt which would be asserted all the time but disabled in the peripheral, and to replace the original instruction with an instruction to enable the interrupt. Such an instruction fits into single word but as we found out the execution of the program does not stop immediately at the instruction which enables the interrupt; the program only stops at the next instruction. This would be unacceptable and thus this solution had to be abandoned.

The working solution proved to be using a relative jump instruction (RJMP) with -1 as the target address, which is a jump to itself (an endless loop). Thus, the program stops in an infinite loop at the position of the breakpoint. To pass the control to the debugger we use periodic interrupt that checks whether the program is currently at the address of a breakpoint and if so, it calls the debugger code. The watchdog peripheral is utilized to generate the periodic interrupt leaving all the timers free for use by the Arduino software.

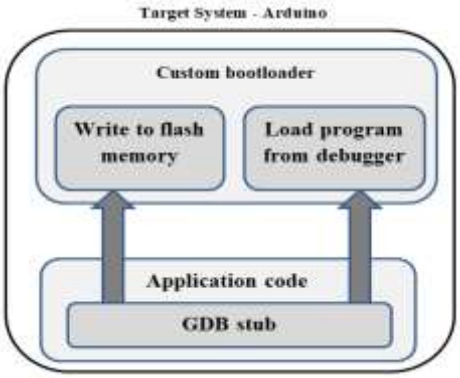

Fig. 2. Interaction of the GDB Stub with the Bootloader.

As follows from the above, to use the flash breakpoints the users need to replace the default bootloader in their Arduino board with our bootloader. Replacing the bootloader is relatively easy and the procedure is well documented by the Arduino community. However, should it be a problem, the debugger can still be used without replacing the bootloader, with RAM breakpoints only. This mode is supported as a compile-time option in the code. It has two advantages – it is readily available for Arduino boards without any modification, and it does not cause wear of the flash memory, as discussed in the next section. In some use-cases, such as school lab for a basic programing course, the RAM-breakpoints mode may be sufficient and preferred.

On the other hand, if the bootloader is replaced and thus writing to flash memory from the user program is enabled, the users can take advantage of another new feature we implemented – the support for loading the program via the debugger. Without this feature the typical workflow for debugging a program is as follows:

- Build the program.
- Upload the program (through the Arduino bootloader).
- Attach the debugger and debug.

With the load support it is possible to upload the program and start debugging with a single click in the IDE. Our debugger stub takes care of receiving the new program and writing it to the program memory of the MCU.

B. Flash Memory Wear Considerations

As already mentioned, the flash breakpoints are implemented by replacing one instruction in the program by a code which transfers the control to the debugger stub. Thus, setting a breakpoint requires overwriting part of the program memory of the MCU. This memory is based on flash technology, which can only endure certain number of write cycles. For the ATmega328 MCU the manufacturer guarantees flash endurance of ten thousand cycles. Although the number is high, memory wear should be considered when using the debugger. Let us briefly discuss this topic.

From user's perspective there are two debugger commands to control execution of the program – a Step command which moves the program to the next line and Continue (Run) command which lets the program run until a breakpoint is hit.

To perform the Step command, the debugger must execute one or more CPU instructions – consider that one line of code in C language may correspond to several CPU instructions. We implement the Step command in the same way as RAM breakpoints – one instruction is executed and then an interrupt is triggered to check whether desired position in the program has been reached. Consequently, the flash memory is not rewritten during step commands.

The Continue command requires writing a breakpoint to program memory - the program should run until a breakpoint is hit. Besides user-defined breakpoints there are also some breakpoints inserted automatically by the debugger. For example, when stepping over a function the GDB places temporary breakpoint at the next instruction after the function call. GDB also removes all breakpoints when the program stops on a breakpoint so that the user can see the stopped program with the original instructions in place and so that the original instruction can be executed when continuing from a breakpoint. When the user resumes the program, all active breakpoints must be written back to memory because the debugger does not know which one will be hit next.

To reduce the memory wear, we implemented a simple optimization so that the breakpoints are written and removed only if it is necessary. When our stub receives command from GDB to remove a breakpoint it notes this request but does not remove the breakpoint (rewrites the flash memory) until the continue command is received. In many cases the breakpoint is removed and then replaced by GDB, but the stub leaves the breakpoint in place thus saving two flash write cycles. Even with this optimization one should keep in mind that the flash memory is overwritten often. It is possible to analyze the number of writes if a compile-time option is enabled in the code of the stub; then there is a global variable which tracks the number of writes to flash memory.

C. Running the Debugger

In the following sections we show the usage of the debugger with focus on the new features. For detailed explanation of the basic setup and usage please refer to our earlier article [14]. The results presented here were obtained on a Window 10 desktop computer with Eclipse Oxygen IDE, 64-bit, version 4.7.3a.

We assume an Arduino Uno board in the original state as it left the factory. First step is to replace the bootloader. For this an in-circuit AVR programmer (ISP programmer) is required. Such programmers are available in many variants for low prices. It is also possible to use another Arduino board as a programmer. The modified bootloader can be found in the source package as a .hex file. This file needs to be loaded into the MCU memory instead of the original bootloader and so-called fuses need to be changed to take into account different size of the new bootloader – the fuse BOOTSZ (size of the bootloader region of the MCU) needs to be set to 1024 words, which means the bootloader occupies 2 kB of memory. After replacing the bootloader, we are ready to use the new features.

D. Building the Program

We will use the typical introductory program which blinks the on-board LED on Arduino pin 13. The user first needs to set up the Eclipse IDE to be able to develop programs for Arduino. The procedure is described in the documentation provided with the source code.

Once the Eclipse project is set up to build the blink program, we can add the code of our debugger stub. This code is located in four files in the avr8-stub folder: avr8-stub.c, avr8-stub.h, app_api.c and app_api.h. The two app_api files provide communication with the bootloader and are only needed if the flash breakpoints or load-via-debugger options are enabled.

As the next step we configure the debugger stub. The constant AVR8_BREAKPOINT_MODE determines whether the flash breakpoints should be used. Default value 1 results in using RAM breakpoints only. Changing it to 0 enables the flash breakpoints.

The constant AVR8_LOAD_SUPPORT determines whether it should be possible to load the program via the debugger. We set the value to 1 to enable this feature. Now we can build the program.

E. Debugging the Program

Once the program is built and an Eclipse debug configuration is created the program can be debugged. Even with the load-via-debugger option enabled we still need to upload the program to the MCU in the standard way (using avrdude tool and bootloader) for the first time because the code of the GDB stub is not yet present in the MCU to be able to receive new programs directly.

After uploading the program to the MCU, we are ready to start debugging. In our earlier article we described using a TCP-to-serial converter on Windows 7 to connect the GDB and the debugger stub because direct serial connection was unstable. With Windows 10 or Linux systems direct serial connection can be used.

Once the code is uploaded to the MCU we can connect to the running program using the Debug button in the IDE. When the connection is established, we should see the program stopped in the debugger – as shown in Fig. 3. The Debug window at the top shows the call stack. The program is stopped in the loop function. Below, in the source window, the line to be executed next is highlighted – it is a call to the delay function. It is now possible to either step into the function and debug the code inside, or step over and execute the function at once.

There is also a variable "counter" which we can examine or modify. Fig. 4 shows the value of the variable as displayed in the IDE when hoovering cursor over the variable name.

To illustrate the benefit of the flash breakpoints - that they do not slow down the execution of the debugged program, we compare the speed of the program when running with flash breakpoints and with RAM breakpoints. First, we insert a breakpoint into the setup function. It will never be hit because the setup function is only executed once when the program starts. However, as described earlier, the presence of the breakpoint should nevertheless slow down the program when using the RAM breakpoint mode.

With the breakpoint set, we resume the program so that it runs at full speed and measure the period of the blinking of the LED. Using the configuration described above we obtain approximately 2 seconds period, as expected given the two 1000 milliseconds delays in the code. This shows that the program speed is not affected by the presence of the breakpoint.

Now we switch the configuration to RAM breakpoints by setting AVR8_BREAKPOINT_MODE to 1 in avr8-stub.h file. After loading the program and running it without a breakpoint we obtain 2 seconds period as in the previous case. However, after setting the breakpoint into the setup function as in the previous case, we obtain period of 7.2 seconds. This means that the program is slowed down by a factor of nearly 4. If a busy loop is used instead of the timer-based Arduino delay function the program is slowed down even more by a factor of 350.

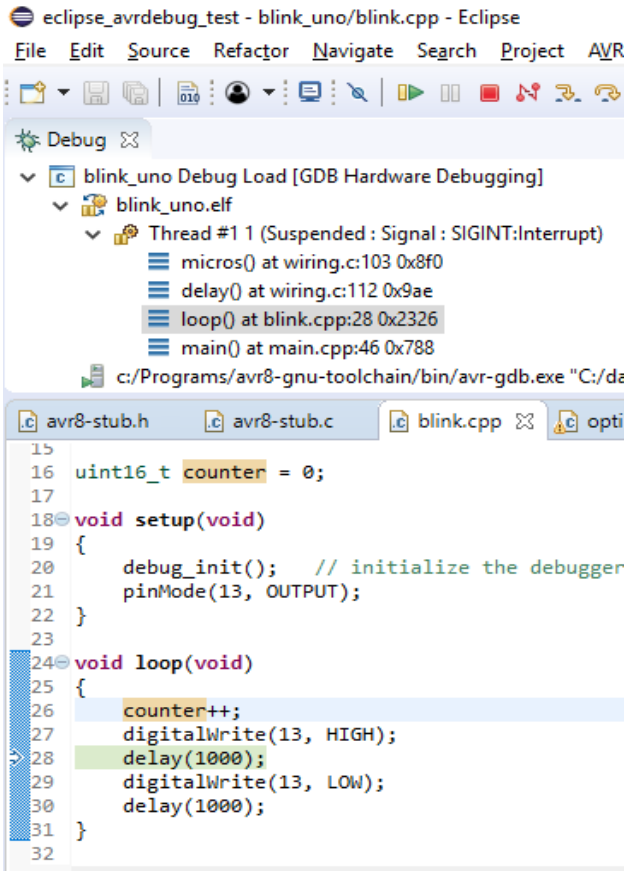

Fig. 3. Sample Program Stopped in the Debugger.

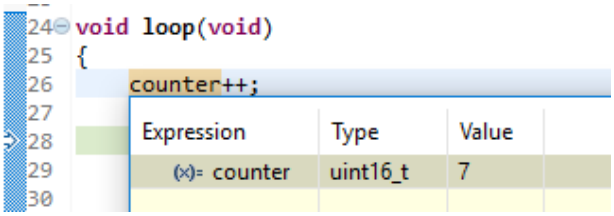

Fig. 4. Viewing Value of a Variable.

F. Loading New Program via the Debugger

With the new feature of loading the program via debugger it is possible to modify and reload the program faster while debugging; just edit the code, click the Debug button and the IDE will save changes, build the program, load it to the MCU and start it in the debugger.

For this to work our debugger stub must be able to receive the program from the IDE and write it to the flash memory of the MCU. To be more precise the GDB debugger issues a load command which our stub supports to load the executable into the target MCU. To enable this feature, the constant AVR8_LOAD_SUPPORT must be defined with value of 1 as described in section D above. Then we need to edit the debug configuration in Eclipse IDE to enable loading the program. This is done by checking the "Load image" box in the Startup tab of the debug configuration. There is detailed description of the procedure in the documentation provided with the source code.

After this we can start the program by clicking the Debug button in the IDE and selecting the debug configuration with load program enabled. To modify the code, we can just stop the program, edit the code, and click the debug button again to upload and debug the modified program.

G. Space Requirements

Table I shows the size of the debugger stub for various configurations of the breakpoints and load support. The exact values will depend on compiler version and configuration, but the presented numbers can be used as estimates of how much space is required to add the debugger support to the program. As can be seen the full-featured version with both breakpoints in flash and load-via-debugger enabled uses about 5.5 kB of program memory. Together with the increased size of the modified bootloader which is required for this configuration (2 kB) adding debugger support to a program requires 7.5 kB out of the total 32 kB of program memory available in the MCU.

H. Use of the Debugger

As already mentioned, presented debugger can be employed without any additional hardware and is free of cost which makes it suitable for use in programming courses which already have the necessary hardware - an Arduino board, and wish to extend the course with introduction to debugging. The improved version of the debugger described here provides better user experience than the old version and allows debugging even time-sensitive code. Besides the educational courses the debugger can also be useful to anyone developing Arduino programs who wants to use a debugger yet is not ready to invest the time and money to switching to a professional development environment.

The current version of the debugger can be used with Arduino boards with the ATmega328 microcontrollers, which includes Uno, Nano and Micro and for the Arduino Mega boards with ATmega2560 and ATmega1280 MCUs.

We are still seeking the ideal development environment to be used with the debugger. The environment based on Eclipse IDE shown above provides complete control of the processes but is quite complicated to set up. A promising alternative seems to be Visual Studio Code editor with Arduino extension which is considerably easier to set up and use for beginner programmers.

TABLE I. CODE AND DATA SIZE FOR DEBUGGER CONFIGURATIONS

Configuration	Program size in bytes	Data size in bytes
Flash breakpoints with load enabled	5446	347
Flash breakpoints with load disabled	5374	307
RAM breakpoints with load enabled	4904	342
RAM breakpoints with load disabled	4658	277

IV. CONCLUSION

In this article we presented new version of the source-level debugger for Arduino. The most important of the new features is the support for writing breakpoints to program memory and loading the program via the debugger. To implement these features a custom version of the Arduino bootloader was created which makes it possible for our debugger stub to write to the program memory. We also had to develop a way to replace the original instruction of the program with a breakpoint to transfer the control to the debugger with the constraint of not overwriting more than one instruction of the original program. Furthermore, to reduce the wear of the flash memory from unnecessary insertion and removal of breakpoints by the GDB debugger, we implemented an algorithm in the code which only rewrites the memory if necessary. Yet another new feature is that the debugger now works also with Arduino Mega boards, which extends its range of use into the area of larger program with many inputs and outputs.

We believe that with these new features the debugger offers user experience similar to a fully-fledged hardware debugger. The advantage of this solution is that, unlike the hardware debug probe, it is free and requires no changes in the Arduino board. It can be helpful in embedded programming courses, student's projects, for controlling lab experiments or in any other project based on the Arduino platform. In future we would like to add support for more Arduino boards and simplify the process of setting up the development environment for debugging. The source code of the debugger stub and detailed instructions for use can be found at https://github.com/jdolinay/avr_debug.

REFERENCES

[1] P. Bender and K. Kussmann, "Arduino based projects in the computer science capstone course", Journal of Computing Sciences in Colleges, Vol. 27, no. 5, pp. 152-157, 2012.

[2] I. Perenc, T. Jaworski and P. Duch, "Teaching programming using dedicated Arduino Educational Board", Comput Appl Eng Educ., vol. 27, no. 4, 943-954, 2019.

[3] F. M. López-Rodríguez and F.J. Cuesta, "Andruino-A1 Low-Cost Educational Mobile Robot Based on Android and Arduino", Journal of Intelligent & Robotic Systems, vol. 81, no. 1, 63-76, 2016.

[4] M. El-Abd. "A Review of Embedded Systems Education in the Arduino Age: Lessons Learned and Future Directions", International Journal of Engineering Pedagogy, vol. 7, no. 2, 79-93, 2017.

[5] A. D'Ausilio, "Arduino, a low-cost multipurpose lab equipment", Behavior Research Methods, vol. 44, no. 2, 305-313, 2012.

[6] J. Hinostroza-Qui˜nones and M. Vasquez-Cunia, "Non-invasive Device to Lessen Tremors in the Hands due to Parkinson's Disease," International Journal of Advanced Computer Science and Applications, vol. 11, no. 8, pp. 735-738, 2020.

[7] M. R. Najjari and M.W. Plesniak, "PID controller design to generate pulsatile flow rate for in vitro experimental studies of physiological flows", Biomedical Engineering Letters, vol. 7, no. 4, 339-344, 2017.

[8] K. I. Mekler, A.V. Burdakov, D.E. Gavrilenko and S. S. Garifov, "A new control system for the capacitive energy storage of the GOL-3 multiple-mirror trap", Instruments and Experimental Techniques, vol. 60, no. 3, 345-350, 2017.

[9] J. Balen, S. Ljepic, K. Lenac and S. Mandzuka, "Air Quality Monitoring Device for Vehicular Ad Hoc Networks: EnvioDev", International Journal of Advanced Computer Science and Applications, vol. 11, no. 5, pp. 580-590, 2020.

[10] P. Diogo, N. V. Lopes and L. P. Reis, "An ideal IoT solution for real-time web monitoring", Cluster Computing, vol. 20, no. 3, pp. 2193-2209, 2017.

[11] M. M. Soto-Cordova, M. Medina-De-La-Cruz and A. Mujaico-Mariano, "An IoT based Urban Areas Air Quality Monitoring Prototype", International Journal of Advanced Computer Science and Applications, vol. 11, no. 9, pp. 711-716, 2020.

[12] F. Kazemian and T. Howles, "Teaching Challenges - Testing and Debugging Skills for Novice Programmers", Software Quality Professional, vol. 11, no. 1, 2008.

[13] R. Chmiel and M.C. Loui, "Debugging: From Novice to Expert", ACM SIGCSE Bulletin, vol. 36, no. 1, 2004.

[14] J. Dolinay, P. Dostálek and V. Vašek, "Arduino Debugger", IEEE Embedded Systems Letters, vol. 8, no. 4, pp. 85-88, 2016.

[15] H. Li, Y. Xu, F. Wu and C. Yin, "Research of "Stub" remote debugging technique", Proceedings of 2009 4th International Conference on Computer Science & Education, Nanning, China, pp. 990-993, 2009.