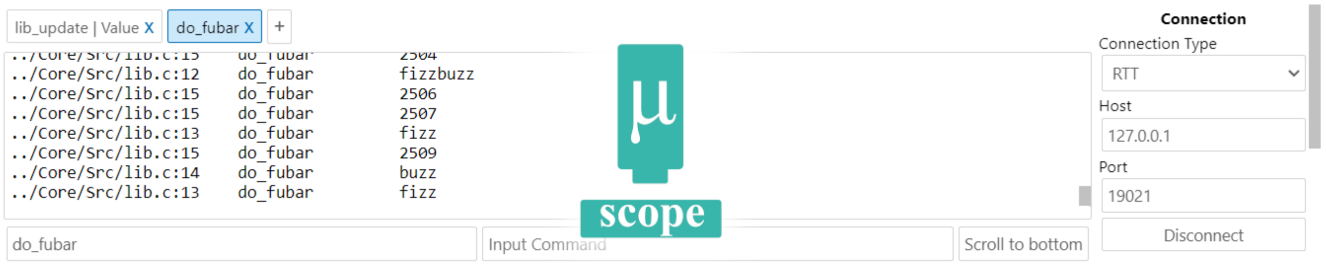
µScope: A Reusable Interface For Debugging STM32 Microcontrollers

TOM SMEETS, University of Twente, The Netherlands



```
../Core/Src/lib.c:15    do_fubar      2504
../Core/Src/lib.c:12    do_fubar      fizzbuzz
../Core/Src/lib.c:15    do_fubar      2506
../Core/Src/lib.c:15    do_fubar      2507
../Core/Src/lib.c:13    do_fubar      fizz
../Core/Src/lib.c:15    do_fubar      2509
../Core/Src/lib.c:14    do_fubar      buzz
../Core/Src/lib.c:13    do_fubar      fizz
```

Connection Type: RTT
Host: 127.0.0.1
Port: 19021
do_fubar

Fig. 1. Screenshot of µScope in action

Microcontrollers play a critical role in a wide range of industrial applications. Developing software for these devices is a challenging task that requires a powerful debugging environment. Existing solutions are insufficient for navigating the stream of messages emitted by the embedded software. In this paper we research and develop a tool to manage the stream of messages by working iteratively and gathering feedback from stakeholders. We conclude with a Visual Studio Code extension called "µScope" that can view and navigate these messages. Demcon applies µScope in a number of projects improving the developers experience. By publishing and open sourcing µScope we increase its reach and usability to the wider community.

Additional Key Words and Phrases: embedded systems, microcontroller, debug, STM32, RTT, SWO, Visual Studio Code, J-Link, ST-Link

INTRODUCTION

Embedded programs produce a wide range of informational debug messages. Visualising these messages is difficult, as every developer wants to have a different view of the output. Existing solutions for visualising these messages are limited. In this paper we will develop a solution tot this on behalf of the engineering company Demcon[1] by answering the following research question:

> "How can the stream of messages emitted by embedded software in microcontrollers be effectively managed and navigated to enhance the debugging process?"

This paper is structured in the following chapters. In "Existing Technology" we explore the current state of the art technology and explain concepts that are needed to understand this paper. In the "Problem Statement" chapter we will describe the research problem in detail, and propose a solution. We look and compare previous solutions in "Previous Work" and describe their features and their shortcomings. We explore related literature in "Literature Review" and describe the importance of our research. In the "Formal Requirements" chapter we provide a formal description of the requirements using the MoSCoW method. This approach enables us to make quantitative comparisons between the requirements and the resulting product. In the "Method" chapter, we provide a detailed description of the development setup and introduce our proposed solution called "µScope". In the "Evaluation" chapter, we reflect on the development process, compare the resulting product against our formal requirements, and evaluate its effectiveness in addressing the issues highlighted in the literature review section. In "Future Work" we propose possible improvements to µScope and describe alternative solutions.

[1]https://demcon.com

TScIT39, July 07, 2023, Enschede, The Netherlands

1 EXISTING TECHNOLOGY

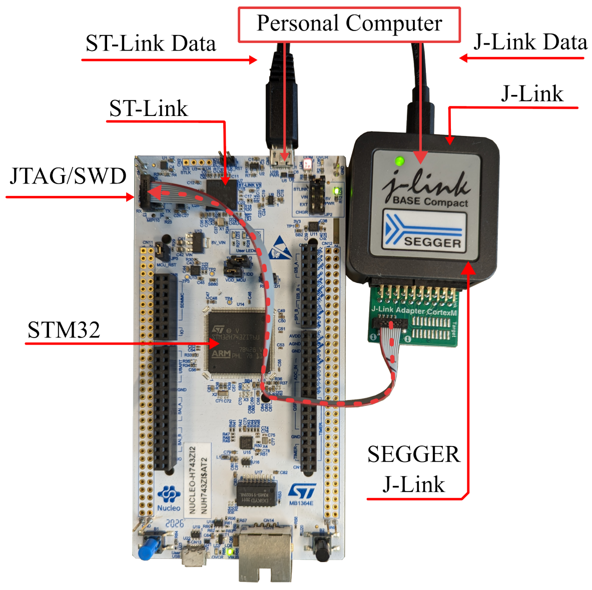


Fig. 2. STM32 Development Setup

Following is an overview of this system, what each component does and how they interact with each other. The components described below can also be seen in Figure 2 where we describe how they are connected.

1.1 STM32

Microcontrollers come in a number of different variants. The STM32 is a 32-bit ARM Cortex Microcontroller produced by STMicroelectronics [1]. This microcontroller can be programmed to control various external devices electronically. Once deployed, the microcontroller is fully integrated into a circuit board that incorporates the electronics of the whole system.

1.2 Nucleo Development Board

To effectively write software for the STM32, the use of a Development Board is essential. One popular option is the STM32 Nucleo Development board, as depicted in Figure 2. This development board offers easy access to all features of the microcontroller, eliminating the need of manual circuit design before deployment. By utilizing the provided pin headers, external electronic devices can easily be connected. Once the software development is complete, developers can seamlessly integrate the microcontroller into the final system.

1.3 JTAG and SWD

To write and deploy software for the microcontroller, developers must have the ability to interact with it effectively. This can be achieved through interfaces such as JTAG [2] or the more recent SWD [3] [4]. These interfaces allow for tasks such as uploading the program, starting stopping, and stepping through the code. Additionally, an important feature is the ability to directly read from and write to the microcontroller's memory.

1.4 J-Link

Fig. 3. A J-Link Debug Probe

To facilitate the communication with the Microcontroller over JTAG or SWD, an external module is required. The SEGGER J-Link [5] is such a device and can be connected over a USB cable to the developers computer. The J-Link is depicted Figure 3. Figure 2 shows a different variant connected over SWD with the STM32. All the features such as reading memory and controlling the current process are provided over this J-Link interface. They are accessible by using a program running on the developers machine called the "JLink GDB server" [6]. This program provides the features of the J-Link over various local TCP sockets. The primary socket that is commonly used for debugging is the GDB server socket to which a GDB client can connect and both control and inspect the running process on the microcontroller. GDB provides additional features such as allowing for multiple simultaneous connections and ability to read debugging symbols giving readable labels to memory locations.

1.5 ST-Link

The J-Link is not the only programmer that is able to do this. A device called the ST-Link is developed by STMicroelectronics and is significantly cheaper. In addition to an external tool, it also comes embedded into the Nucleo development board [7] as seen in Figure 2. This has the advantage that you don't need an additional USB cable.

1.6 SWO

A developer needs to be able to receive feedback of a running program to be able to find and solve issues that might arise. A common solution is to send the information as text messages to the developers computer for inspection. With microcontrollers such as the STM32 this is possible by connecting an external wire to one of the microcontrollers pins. After writing a small driver to transmit the data, it can be received on the developers computer. This is not optimal due to the need for a driver an additional cable.

Fortunately is such a feature directly integrated into the SWD interface with the inclusion of a Serial Wire Output (SWO) [8] pin. The text sent over SWO is accessible over the ST-Link and J-Link on the developers computer.

1.7 RTT

Sending data over SWO is not instantaneous and requires for the microcontroller to wait while the text is being sent over the cable to the developers machine, as measured by SEGGER [9]. This will cause inconsistent delays and is a big problem for time critical software. A solution for this is developed by SEGGER, called "Real Time Transfer" which advertises a significantly lower overhead for transmitting text [10]. The low overhead is achieved by writing the messages into a small circular buffer in the microcontroller's memory, which can be done with a much smaller delay [9]. The need for an additional SWO pin is now also removed as RTT works directly over the SWD memory interface. However RTT also has a number of disadvantages. One disadvantage is that it requires a region of memory, which may be limited on microcontrollers. Another big disadvantage is that RTT only works with the J-Link and not with the ST-Link.

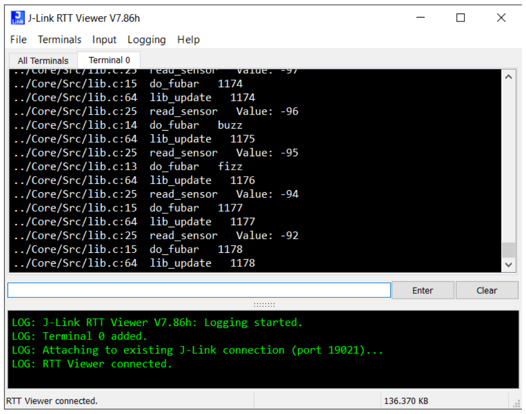
2 PROBLEM STATEMENT

Viewing and navigating the long stream of debug messages sent from microcontrollers such as the STM32 is difficult, due to the high throughput of the messages and the interleaving of messages from different tasks.

The current state of the art tools are lacking and do not provide a flexible interface for viewing these messages. This calls for a solution that can efficiently view and navigate debug messages. In the next chapter we will compare the state of the art tools that allow for viewing debug messages. We will also describe why they are insufficient.

3 PREVIOUS WORK

3.1 SEGGER J-Link RTT Viewer



```
../Core/Src/lib.c:15  do_fubar   1174
../Core/Src/lib.c:64  lib_update   1174
../Core/Src/lib.c:25  read_sensor   Value: -96
../Core/Src/lib.c:14  do_fubar   buzz
../Core/Src/lib.c:64  lib_update   1175
../Core/Src/lib.c:25  read_sensor   Value: -95
../Core/Src/lib.c:13  do_fubar   fizz
../Core/Src/lib.c:64  lib_update   1176
../Core/Src/lib.c:25  read_sensor   Value: -94
../Core/Src/lib.c:15  do_fubar   1177
../Core/Src/lib.c:64  lib_update   1177
../Core/Src/lib.c:25  read_sensor   Value: -92
../Core/Src/lib.c:15  do_fubar   1178
../Core/Src/lib.c:64  lib_update   1178
```

```
LOG: J-Link RTT Viewer V7.86h: Logging started.
LOG: Terminal 0 added.
LOG: Attaching to existing J-Link connection (port 19021)...
LOG: RTT Viewer connected.
```

RTT Viewer connected. 136.370 KB

Fig. 4. SEGGER J-Link RTT Viewer

As an external tool, SEGGER provides the J-Link RTT Viewer. This is a piece of software for viewing RTT messages sent over the USB cable to the developers computer [11]. This program lists all messages and groups them into a number of channels. However, these channels have to be setup beforehand on the microcontroller's program and cannot be changed during execution. A screenshot of this software can be seen in Figure 4.

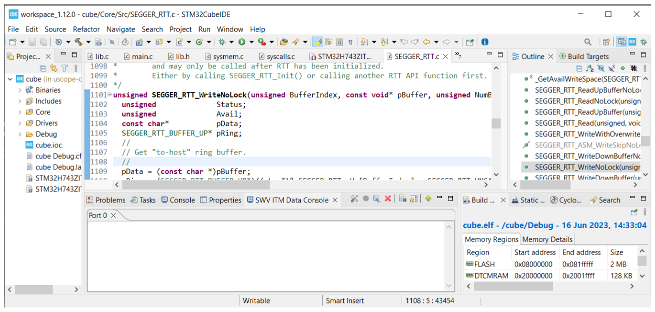
Allowing for only a number of predefined channels over which the output can be seen is limiting its usefulness. It is not clear how these channels should be allocated. Allocating a channel for each module separately will prevent the direct comparison of the output, as some modules might be related and require to be viewed together. Combining all messages into a single channel preserves the correct ordering, but will cause issues when unrelated messages are obstructing the view of the developer.

Improving the existing SEGGER RTT Viewer is not possible, the source code is not available for modification. Fortunately SEGGER provides an API that can be used to send and receive the RTT text by communicating over a local TCP socket [12]. We will use this in the Method chapter to develop a solution.

3.2 STM32 Cube IDE

STMicroelectronics provides an integrated development environment (IDE) called the STM32 Cube IDE [13]. This IDE has a number of features integrating the compiler, the debugger and additional tools into a single graphical interface. This IDE can connect to both the J-Link and ST-Link GDB servers. Text output is also supported over SWO and displayed in the SWV ITM Data Console, but requires some setup [14]. This console also supports multiple channels but have to be setup in advance on the microcontroller's program and cannot be changed during execution, just as the J-Link RTT Viewer.

The STM32 Cube IDE is great for getting started with STM32 development but lacks some modern features. This was noted by a stakeholder at Demcon. A more commonly preferred solution is



```
*         and may only be called after RTT has been initialized.
*         Either by calling SEGGER_RTT_Init() or calling another RTT API function first.
*/
unsigned SEGGER_RTT_WriteNoLock(unsigned BufferIndex, const void* pBuffer, unsigned NumB
  unsigned              Status;
  unsigned              Avail;
  const char*           pData;
  SEGGER_RTT_BUFFER_UP* pRing;
  //
  // Get "to-host" ring buffer.
  //
  pData = (const char *)pBuffer;
```

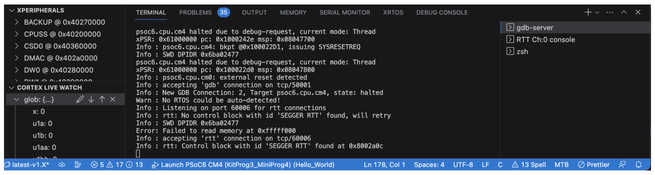
cube.elf - /cube/Debug - 16 Jun 2023, 14:33:04

Region	Start address	End address	Size
FLASH	0x08000000	0x081fffff	2 MB
DTCMRAM	0x20000000	0x2001ffff	128 KB

Fig. 5. STM32 Cube IDE

the Visual Studio Code IDE[15], which is a more flexible editor that allows installing extensions for each use case.

3.3 Cortex Debug



```
psoc6.cpu.cm4 halted due to debug-request, current mode: Thread
xPSR: 0x61000000 pc: 0x1000242e msp: 0x08047700
Info : psoc6.cpu.cm4: bkpt @0x100022D1, issuing SYSRESETREQ
Info : SWD DPIDR 0x6ba02477
psoc6.cpu.cm4 halted due to debug-request, current mode: Thread
xPSR: 0x61000000 pc: 0x100022d0 msp: 0x08047800
Info : psoc6.cpu.cm0: external reset detected
Info : accepting 'gdb' connection on tcp/50001
Info : New GDB Connection: 2, Target psoc6.cpu.cm4, state: halted
Warn : No RTOS could be auto-detected!
Info : Listening on port 60006 for rtt connections
Info : rtt: No control block with id 'SEGGER RTT' found, will retry
Info : SWD DPIDR 0x6ba02477
Error: Failed to read memory at 0xfffff000
Info : accepting 'rtt' connection on tcp/60006
Info : rtt: Control block with id 'SEGGER RTT' found at 0x8002a0c
```

Launch PSoC6 CM4 (KitProg3_MiniProg4) (Hello_World) Ln 178, Col 1 Spaces: 4 UTF-8 LF C

Fig. 6. Cortex Debug Visual Studio Code Extension

One of these extensions is Cortex-Debug [16], which integrates a number of tools into Visual Studio Code [15]. The extension provides integration with the ST-Link and J-Link GDB servers and allows for debugging the programs from within the editor. An additional window is provided that can receive both SWO and RTT messages from the microcontroller.

Compared to the J-Link RTT Viewer is the Cortex Debug more convenient for developers who already make use of the Visual Studio Code IDE. Integrating these tools into the Visual Studio Code editor has a number of advantages, streamlining the development setup. However, the viewer is even more limited than the J-Link RTT Viewer and allows for viewing only a single channel of messages. The filtering of messages is not possible. A screenshot of this software can be seen in Figure 6.

4 LITERATURE REVIEW

In "How developers debug" [17], the authors describe how "printf debugging" is often the very first tool developers use to start their search for a bug.

> "Interviewees praised printf as a universal tool that one can always resort back to, helpful when learning a new language ecosystem, in which one is not yet familiar with the tools of the trade." [17, p. 9]

Printf debugging means that a developer places statements in the program that write the current value of variables as text into an output buffer. Such a tool is widely available in many programming languages. The STM32 supports this either via RTT or SWO.

However, the authors also note that developers are aware of its shortcomings in concurrent programs.

> "that it is insufficient for concurrent programs, primarily because the [output] interleave[s]" [17, p. 9]

This issue can be resolved with correct locking but messages themselves are still interleaved. In the Evaluation section we describe how µScope addresses this shortcoming.

Layman et al explored professional debug challenges and needs of developers at Microsoft in "Debug Revisited"[18]. They found that the difference between the sequential thought process and the non-sequential execution of multithreadded environemnts provide a source of difficulty.

Velihorski et al analyzed the usability of the STM32CubeMonitor tool for remote development. [19] They concluded that using the correct tools it is possible to remotely debug embedded software.

5 FORMAL REQUIREMENTS

The requirements resulting from a number of meetings with the stakeholder at Demcon are compiled into a list using the MoSCoW [20] method. Due to the nature of this research it is difficult to predict the time requirement of researching and developing a solution. Using the MoSCoW method allows us to focus on the important features first and transition to lesser important features later. We choose this method to ensure that we remain in scope while still staying flexible with the time requirements.

The MoSCoW method is made of four parts. The "Must Haves" are the most important requirements that are needed to create the "Minimum Viable Product" [21]. The requirements listed as "Should Haves" are not essential for a working product but will greatly improve it. The "Could Haves" are additional features that are less important and should only be considered at the end. The features that are out of scope and should not be considered are listed in "Won't Haves".

5.1 Must Haves

Displaying Messages. The tool should be able to intercept debug messages sent by the microcontroller and display them on a screen. The messages should be remembered and viewable for a reasonable duration. The primary and suggested method to intercept the messages is by going over RTT.

Filtering. A solution is needed that allows for separating the output into the parts that are relevant to the developer. This can be done by filtering the messages on a given search term.

Reusability. The tool should not be dependant on any specific project at Demcon. Making the tool entirely independent allows for easy reuse into other projects. Demcon works on many different projects that could benefit from this tool.

5.2 Should Haves

Sending Commands. Sending custom commands to the microcontroller is useful in some cases, making it possible to directly instruct the embedded software to perform some predefined action. Examples are sending commands to read a sensor value or move an actuator.

Low Overhead. Sending debug messages should not interfere with the currently running process on the microcontroller. Because the microcontroller has a limited memory capacity and processor speed the program should stay as small and fast as possible. The viewing application has less strict requirements due to the more capable hardware of personal computers.

5.3 Could Haves

Visual Studio Code Extension. Integrating the tool into the Visual Studio Code ecosystem will give a number of advantages to the developers. Allowing for easy integration into the current toolset without the need of installing and starting external software.

Beyond J-Link. The tool could support other interfaces in addition to RTT, increasing the support to devices by other manufacturers. One example is to support the integrated ST-Link programmer in the Nucleo development board.

Tracking And Plotting Data. A communication channel can be used to send the state of variables, which can then be plotted in some graphical window.

5.4 Won't Haves

Full Debugging Environment. The tool will not provide a complete debugging environment. The focus of the tool will solely be visualizing and navigating debug messages sent by microcontrollers.

6 METHOD

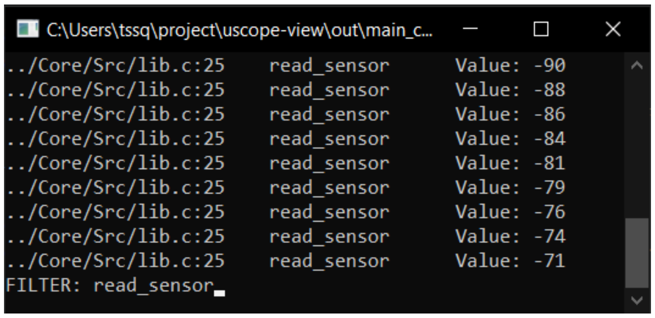
6.1 Experiment Setup

The following system setup was used during this Research. This was chosen because it is commonly used by Embedded Software Developers, including those at Demcon.

The setup consisted initially of a STM32 microcontroller embedded onto a Nucleo board. The Nucleo was connected to a laptop over a USB cable, providing power and data. The embedded ST-Link was communicating over SWD with the microcontroller. On the laptop we run the STM32 Cube IDE [13] which starts a ST-Link GDB Server over which GDB can communicate and SWO messages can be received. The STM32 Cube IDE allowed for quick development and deployment of an embedded example program that communicates with SWO and RTT. This required minimal setup. Later during the research we added a J-Link to the setup, also communicating over SWD with the microcontroller and connected over a second USB cable to the laptop. For using the J-Link we ran the official SEGGER J-Link GDB Server [6]. Which provided access to RTT, SWO and a GDB connection. The final setup that was used can be seen in Figure 2.

6.2 Prototype

The initial prototype is written in C and makes use of a console window to display the received messages. This interface has been chosen to keep the program simple, allowing for us to focus on the

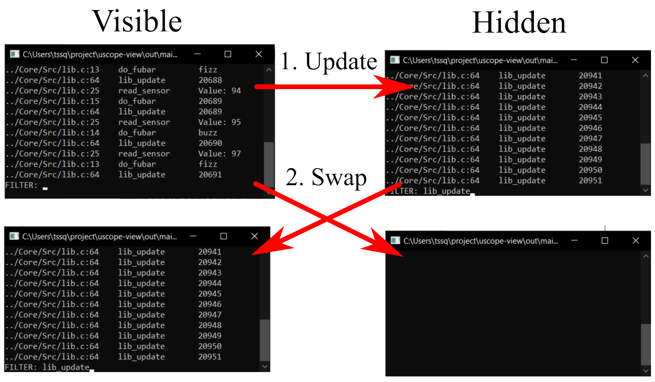


```
C:\Users\tssq\project\uscope-view\out\main_c...
../Core/Src/lib.c:25    read_sensor     Value: -90
../Core/Src/lib.c:25    read_sensor     Value: -88
../Core/Src/lib.c:25    read_sensor     Value: -86
../Core/Src/lib.c:25    read_sensor     Value: -84
../Core/Src/lib.c:25    read_sensor     Value: -81
../Core/Src/lib.c:25    read_sensor     Value: -79
../Core/Src/lib.c:25    read_sensor     Value: -76
../Core/Src/lib.c:25    read_sensor     Value: -74
../Core/Src/lib.c:25    read_sensor     Value: -71
FILTER: read_sensor
```

Fig. 7. µScope Prototype

research and development of the logic and communication code for RTT and SWO. A screenshot of the program in action is shown in Figure 7.

The program quickly gained features that required a more sophisticated method of displaying information on the screen. The screen is updated immediately once the user enters a character to change the filter. This is implemented by using the Windows Console API [22]. Initially, the redrawing was done by moving the terminal cursor using Virtual Terminal Sequences [23] and overwriting some sections of the screen. This quickly became complicated and caused flickering of the screen due to intermediate states of this process being visible. This resulted in the program transitioning to a different method called "Double Buffering" [24]. Here the screen is not directly updated like before, but fully re-rendered into a hidden buffer. Once finished, the hidden buffer is switched with the visible buffer, making the changes visible immediately. The previously visible buffer can be reused again as the next hidden buffer. A visualisation of this can be seen in Figure 8.



```
C:\Users\tssq\project\uscope-view\out\mai...
../Core/Src/lib.c:13    do_fubar        fizz
../Core/Src/lib.c:64    lib_update      20688
../Core/Src/lib.c:25    read_sensor     Value: 94
../Core/Src/lib.c:15    do_fubar        20689
../Core/Src/lib.c:64    lib_update      20689
../Core/Src/lib.c:25    read_sensor     Value: 95
../Core/Src/lib.c:14    do_fubar        buzz
../Core/Src/lib.c:64    lib_update      20690
../Core/Src/lib.c:25    read_sensor     Value: 97
../Core/Src/lib.c:13    do_fubar        fizz
../Core/Src/lib.c:64    lib_update      20691
FILTER:
```

```
C:\Users\tssq\project\uscope-view\out\mai...
../Core/Src/lib.c:64    lib_update      20941
../Core/Src/lib.c:64    lib_update      20942
../Core/Src/lib.c:64    lib_update      20943
../Core/Src/lib.c:64    lib_update      20944
../Core/Src/lib.c:64    lib_update      20945
../Core/Src/lib.c:64    lib_update      20946
../Core/Src/lib.c:64    lib_update      20947
../Core/Src/lib.c:64    lib_update      20948
../Core/Src/lib.c:64    lib_update      20949
../Core/Src/lib.c:64    lib_update      20950
../Core/Src/lib.c:64    lib_update      20951
FILTER: lib_update
```

```
C:\Users\tssq\project\uscope-view\out\mai...
../Core/Src/lib.c:64    lib_update      20941
../Core/Src/lib.c:64    lib_update      20942
../Core/Src/lib.c:64    lib_update      20943
../Core/Src/lib.c:64    lib_update      20944
../Core/Src/lib.c:64    lib_update      20945
../Core/Src/lib.c:64    lib_update      20946
../Core/Src/lib.c:64    lib_update      20947
../Core/Src/lib.c:64    lib_update      20948
../Core/Src/lib.c:64    lib_update      20949
../Core/Src/lib.c:64    lib_update      20950
../Core/Src/lib.c:64    lib_update      20951
FILTER: lib_update
```

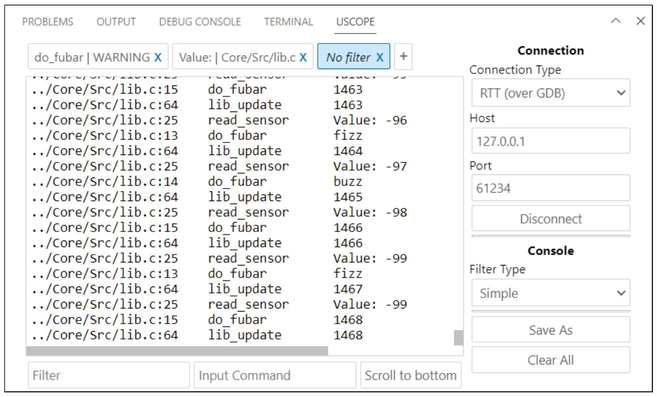
```
C:\Users\tssq\project\uscope-view\out\mai...
```

Fig. 8. Double buffering Console windows

While this was sufficient for a prototype and the minimum viable product, it was difficult to use, and made implementing some features such as tabs very difficult.

6.3 Visual Studio Code Extension

Initially we researched existing GUI libraries such as GTK [25] and QT [26], but decided to settle with a Visual Studio Code Extension, as this was one of the "Could Have" requirements. Making the tool quick and easy to use was one of the main goals and guided the interface design. In Figure 9 we show the current interface of this Visual Studio Code extension.



```
../Core/Src/lib.c:15    do_fubar        1463
../Core/Src/lib.c:64    lib_update      1463
../Core/Src/lib.c:25    read_sensor     Value: -96
../Core/Src/lib.c:13    do_fubar        fizz
../Core/Src/lib.c:64    lib_update      1464
../Core/Src/lib.c:25    read_sensor     Value: -97
../Core/Src/lib.c:14    do_fubar        buzz
../Core/Src/lib.c:64    lib_update      1465
../Core/Src/lib.c:25    read_sensor     Value: -98
../Core/Src/lib.c:15    do_fubar        1466
../Core/Src/lib.c:64    lib_update      1466
../Core/Src/lib.c:25    read_sensor     Value: -99
../Core/Src/lib.c:13    do_fubar        fizz
../Core/Src/lib.c:64    lib_update      1467
../Core/Src/lib.c:25    read_sensor     Value: -99
../Core/Src/lib.c:15    do_fubar        1468
../Core/Src/lib.c:64    lib_update      1468
```

Fig. 9. "µScope Interface"

Before being able to use the console we first have to create a connection to the microcontroller. A number of different methods are supported, such as RTT, SWO and RTT over GDB as seen in Figure 10. A more detailed explanation of RTT over GDB can be seen in chapter 6.6. Once a connection type is selected a default port and IP address are provided. It is possible to change them to connect to a different target. The host address "127.0.0.1" indicates the current computer, but can be changed to the IP address of an external computer if needed. A connection can be established by pressing the "Connect" button. All messages sent over RTT or SWO are now shown in the console pane on the left.

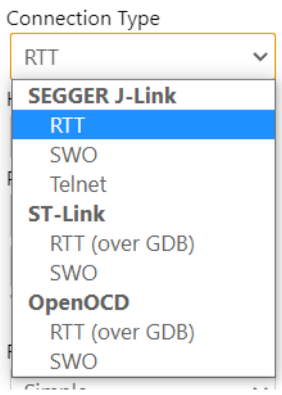


Fig. 10. Connection Type Selector

By including extra information in the debug messages, such as the current file, line and function, we can intelligently filter these messages. The following macro was used on the microcontroller to automatically insert these tags into the messages.

```
#define TO_STRING0(x) #x
#define TO_STRING(x) TO_STRING0(x)
```

```
#define dbg(fmt, ...) SEGGER_RTT_printf(0, __FILE__ \
   ":" TO_STRING(__LINE__) ":" fmt "\n", ## __VA_ARGS__)
```

It can be used just as any other ‘printf‘ method. An example usage is the following.

```
// create a simple incrementing counter
static u32 counter = 0;

// continiously call this to run the program
void update(void) {
    // print the counter and increment it
    dbg("Counter: %d", counter++);

    // blink the led on the microcontroller
    HAL_GPIO_TogglePin(LD1_GPIO_Port, LD1_Pin);
    HAL_Delay(100);
}
```

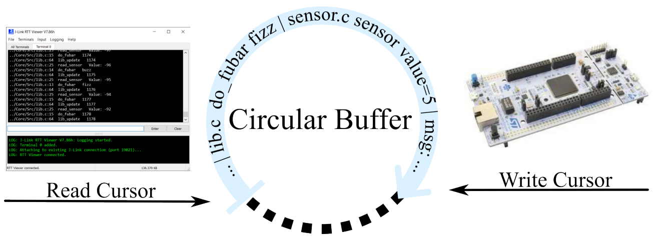
This information can then be used for filtering. The filter input box on the bottom left can be used for this. Changing the filter immediately updates the console for quick feedback. Multiple filter terms can be given by separating them with a boolean OR operator represented by the vertical pipe symbol "|". A boolean AND operator is also supported by separating terms with an ampersand "&". A filter text of "`lib & value | other`" matches messages that contain both "lib" and "value" at the same time, or messages that just contain "other". The AND operator binds more strongly than the OR. The implementation of this grammar is kept minimal and focusses on easy to write filters. If a more advanced grammar is required for a filter the user can select "Regex" in the filter type drop-down on the right. Then the filter for that tab will be interpreted as a regular expression allowing more sophisticated filters.

A prominent feature is the tab list at the top of the window. Each tab consists of a different filter set by the user. Switching between the tabs can be done by clicking on them. Tabs can be added and removed, and the filter dynamically updated. Saving the content of the console can be done by pressing the "Save As" button. The console can be cleared with the "Clear All" button.

6.4 How RTT Works

Communicating over RTT works by reading and writing directly to the microcontroller's memory [27]. The primary structure is the RTT control block which is located at a fixed location in memory defined by the linker. The address is fixed for the duration of the program but can be different for different programs. The structure contains a region of memory which is used as a circular buffer where all the text is written to. Additional variables in the control block are used to describe the buffer size and store the read and write cursor. This text written to the circular buffer can then be transmitted over SWD or JTAG by the J-Link. Because the linker can place this control block anywhere in memory it is not possible for the J-Link to directly locate it. By writing a unique id at the beginning of this structure it can be located by the J-Link by scanning the entire memory region looking for the id.

One of the provided methods by the RTT library on the microcontroller is `SEGGER_RTT_printf(...)`. This method will combine variables and text into a list of characters and write them into one of the circular buffers. While the microcontroller is writing text it is advancing a write-cursor telling the host exactly how much text is written. The host can then read this data and advance the read-cursor until it matches with the write-cursor. A visualisation can be seen in Figure 11.



```
../Core/Src/lib.c:15  do_fubar   1174
../Core/Src/lib.c:64  lib_update   1174
../Core/Src/lib.c:25  read_sensor   Value: -96
../Core/Src/lib.c:14  do_fubar   buzz
../Core/Src/lib.c:64  lib_update   1175
../Core/Src/lib.c:25  read_sensor   Value: -95
../Core/Src/lib.c:13  do_fubar   fizz
../Core/Src/lib.c:64  lib_update   1176
../Core/Src/lib.c:25  read_sensor   Value: -94
../Core/Src/lib.c:15  do_fubar   1177
../Core/Src/lib.c:64  lib_update   1177
../Core/Src/lib.c:25  read_sensor   Value: -92
../Core/Src/lib.c:15  do_fubar   1178
../Core/Src/lib.c:64  lib_update   1178
```

```
LOG: J-Link RTT Viewer V7.86h: Logging started.
LOG: Terminal 0 added.
LOG: Attaching to existing J-Link connection (port 19021)...
LOG: RTT Viewer connected.
```

Fig. 11. Illustration of a circular buffer

6.5 RTT and SWO

When developing for a microcontroller, the user has either a ST-Link or a J-link server running. These provide all the communication with the microcontroller over a number of interfaces, usually with local TCP sockets on a predefined port. For example, the J-Link server sends the RTT text on the port 1902. A viewer can connect to that port and receive the RTT messages. The ST-Link also hosts a similar interface for SWO but on a different port.

6.6 RTT over GDB

While the primary requirement was to provide an interface to RTT messages, this still required a SEGGER J-Link programmer and limited the compatibility of this tool. Before receiving the J-Link programmer we initially had only access to the embedded ST-Link programmer, which was present on the Nucleo development board.

First we attempted using the "ST-Link Reflash" tool that SEGGER provides, which can convert a ST-Link debugger into a J-link debugger by changing the firmware [28]. Despite following the instructions, we were not able to upload the new firmware onto the Nucleo development board. Not wanting to spend too much time on this issue we decided to investigate the ST-Link tooling and how to communicate with it.

We realized that it could be possible to read the RTT buffer using a ST-Link device by reading the memory directly. This would also increase the support for RTT to a wide range of other devices without needing to change the program running on the microcontroller. Initially we used the Open Source ST-Link toolset [29] using the `stlink_read_mem32()` method to read memory at a given address. While this worked well, it could not share the microcontroller connection with other programs such as the debugger, which was a big limitation.

A better method is reading memory directly with the debugger. The ST-Link starts a GDB Server over which a debugger can inspect and control a program. While usually used for integrating a debugging interface into an editor, it also allows for reading and writing memory. In contrast to the ST-Link library, the GDB server does support multiple connections.

To read the text from the microcontroller we start the GDB client and configure a number of settings to allow reading of memory while the program is still running. GDB supports scanning a memory region for a sequence of bytes with the `find` command. Using this we can find the initial “SEGGER RTT” id string and use that address to locate the buffer, as well as the read and write cursors. By polling the write cursor cursor we can discover when new text is written. Then, the portion of memory between the read-cursor and the write-cursor is read and forwarded to the console. Finally the read-cursor is advanced to match the write-cursor, allowing the microcontroller to write more data.

7 EVALUATION

7.1 Requirements

Both the Must Haves and the Should Haves requirements are fully implemented.

Displaying Messages. µScope is able to intercept debug messages sent by the microcontroller and display them on a screen. The messages are remembered and viewable for a reasonable duration. Messages are receivable over RTT which satisfies the first “Must Have”.

Filtering. Using µScope the developer is able to quickly filter messages into relevant parts.

Reusability. µScope is fully independent of any specific project. The tool is reusable and works on any platform supporting Visual Studio Code. By directly using the RTT and SWO message stream we remove the need for custom code on the microcontroller. This makes the tool immediately useable in any existing project using these interfaces.

Sending Commands. Sending commands from the interface works fully over RTT and SWO and can be received by any existing code on the microcontroller. Sending text is however not yet implemented in the “RTT over GDB” feature. This is a possible improvement for the future. The received text can be handled on the microcontroller, the implementation of these commands is out of scope and is specific for each project.

Low Overhead. µScope achieves a very low overhead by making direct use of the exiting RTT code on the microcontroller. A small memory section is required for RTT however, which is not the case for SWO. The developer can choose the best option for each project. The visual studio extension applies some tricks to be kept performant. Reducing the number of HTML elements and keeping the updates to a minimum. To keep the interface interactive the messages have to be limited eventually.

Transmitting debug messages should not interfere with the currently running process on the microcontroller. Because the microcontroller has a limited memory capacity and processor speed the program should stay as small and fast as possible. The viewing application has less strict requirements due to the more capable hardware of personal computers.

Visual Studio Code Extension. µScope is fully implemented as a Visual Studio Code extension and does not require additional installs. The extension can be set as a recommended extension in a workspace, allowing for easy installation of µScope during project setup [30].

```
{
  "recommendations": ["DEMCON.uscope"]
}
```

Beyond J-Link. µScope supports multiple interfaces such as the ST-Link and OpenOCD. RTT is supported on these platforms by manually implementing RTT over a GDB connection.

Tracking And Plotting Data. There was not enough time to implement data plotting due to the time constraints. This would require a significant time investment in the user interface and addition of data channel. We decided to not implement data plotting.

7.2 The MoSCoW Method

We setup the requirements using the MoSCoW Method and applied this method during the development. We made this decision deliberately, to allow for a more flexible development time frame.

However, some disadvantages were encountered as a result of developing this way. Initially we completed the MVP as a separate C program with all the “Must Have” features. This took a number of weeks of our time but resulted in a satisfactory result.

However, while transitioning to the “Should have” features we encountered a problem. Developing a Visual Studio Code Extension is not possible in C and required us to completely throw away our previous work and start anew in JavaScript. Fortunately we could still apply the acquired knowledge from our research into this new program. However, if we had started immediately with the Visual Studio code extension it would have cost us less time to develop.

7.3 Literature

In the Literature Review section we explored a number of real world use cases and issues with debugging.

In “How developers debug” [17, p .9] the subjects note that printf debugging is insufficient for concurrent programs due to the interleaving of the output. Concurrent programs are also present on the STM32 and are widely used in practice. RTT partially solves the interleaving problem by locking the output buffer during each printf statement. This prevents interleaving of the words inside a message. Messages are still interleaved per line and have an undefined ordering. This is addressed by µScope. Related messages of a given task can be viewed together by using filters over a number of tabs.

Velihorski et al described the benefit of remote debugging in “Remote Debugging of Embedded Systems in STM32CubeMonitor” [19]. By utilizing TCP connections we allow for the ability of remotely sending and receiving the debug messages.

8 FUTURE WORK

Some improvements could still be made to µScope. Currently we combine all messages and display them directly into the console. However, for a large amount of messages is the HTML renderer not performant enough by itself. The message limit could be increased, which could be achieved by dynamically creating and removing HTML elements while the user is scrolling. This could be an entire research topic of itself.

µScope is only available as a Visual Studio Code Extension. An external tool applying a similar interface could be useful for developers not using Visual Studio Code. Revisiting the original C program could be a good starting point.

The "RTT over GDB" feature does not support sending data back to the microcontroller. This would work similarly to receiving data and uses an additional circular buffer in memory.

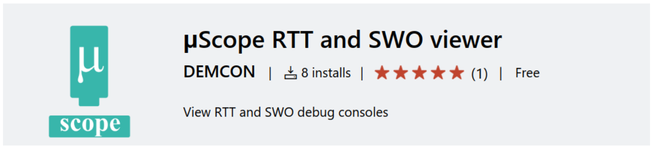
The plotting of variables was also not implemented in the end. This could be done by defining a syntax over the text interface that defines key-value pairs of variables. This allows us to re-use the same text channel for data, keeping the compatibility with the interfaces that allow for only a single channel. The tabs in the interface could have a "text-mode" and a "plotting-mode" setting, allowing for the same method of filtering.

The RTT over GDB feature could be abstracted away and provided as a full separate program. Connecting to a GDB session and opening a TCP socket for RTT to be received.

CONCLUSION

In this paper we solved a practical problem that Demcon employees were having during the development of embedded software. By iteratively working and gathering feedback from stakeholders we were able to develop and publish the "µScope" Visual Studio Code extension. Employees at Demcon have already successfully applied this tool and are using it in future projects. We are already looking into expanding the tool to support even more microcontrollers other than the STM32. By publishing and open sourcing the tool under a permissive license we increase its reach and usability to the wider community.

µScope is available for free on the Visual Studio Code Marketplace as "DEMCON.uscope" and on GitHub as a free and open source project at "https://github.com/DEMCON/uscope". Any contributions are welcome and can be submitted by creating an Issue or a Pull Request.



µScope RTT and SWO viewer

DEMCON | 8 installs | ★★★★★ (1) | Free

View RTT and SWO debug consoles

Fig. 12. µScope in the Visual Studio Code Marketplace

ACKNOWLEDGMENTS

I would like to thank my supervisor Dr. Ir. Vadim Zaytsev at The University of Twente and my supervisor Ir. Tim Klein Nijenhuis at Demcon.

REFERENCES

[1] ST, (2023), STM32 32-bit Arm Cortex MCUs, https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html
[2] Joint Test Action Group, (1990), IEEE Std 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture, 10.1109/IEEESTD.1990.114395
[3] ARM, (2006), Debug Interface v5 Architecture Specification, https://developer.arm.com/documentation/ihi0031/a/The-Serial-Wire-Debug-Port--SW-DP-/Introduction-to-the-ARM-Serial-Wire-Debug--SWD--protocol
[4] ARM, (2023) JTAG/SWD Interface, https://developer.arm.com/documentation/101636/0100/Debug-and-Trace/JTAG-SWD-Interface
[5] SEGGER, (2023), J-Link, The Market-leading debug probe, https://www.segger.com/products/debug-probes/j-link/.
[6] SEGGER, (2023), J-Link GDB Server, https://wiki.segger.com/J-Link_GDB_Server
[7] STMicroelectronics, (2023), ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32 https://www.st.com/en/development-tools/st-link-v2.html
[8] ARM, (2006), Debug Interface v5 Architecture Specification, https://developer.arm.com/documentation/ddi0314/h/Serial-Wire-Output/About-the-Serial-Wire-Output
[9] SEGGER, (2023), J-Link RTT Product Page, https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/
[10] SEGGER, (2023), J-Link RTT Wiki, Real Time Transfer, https://wiki.segger.com/RTT.
[11] SEGGER, (2023), J-Link RTT Viewer, https://www.segger.com/products/debug-probes/j-link/tools/rtt-viewer/.
[12] SEGGER, (2023), TELNET channel of J-Link software, https://wiki.segger.com/RTT#TELNET_channel_of_J-Link_software
[13] STMicroelectronics, (2023), STM32 Cube IDE, https://www.st.com/en/development-tools/stm32cubeide.html
[14] PCB Artists, (2021), Enable SWO Debug Output in STM32 CubeMX, https://pcbartists.com/firmware/stm32-firmware/debug-printf-stm32-using-swo-serial-wire/
[15] Microsoft, (2015-2023), Visual Studio Code, https://code.visualstudio.com/
[16] Marcel Ball, (2017), Cortex-Debug, Visual Studio Code extension for enhancing debug capabilities for Cortex-M Microcontrollers, https://marketplace.visualstudio.com/items?itemName=marus25.cortex-debug,
[17] Beller M, Spruit N, Zaidman A., (2017), How developers debug, PeerJ Preprints 5:e2743v1, doi:10.7287/peerj.preprints.2743v1
[18] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline and G. Venolia, (2013), Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers, 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 2013, pp. 383-392, doi:10.1109/ESEM.2013.43.
[19] O. Velihorskyi, I. Nesterov, M. Khomenko, (2020), Remote Debugging of Embedded Systems in STM32CubeMonitor (pp.22-25). International Scientific and Practical Conference Theoretical and Applied Aspects of Device Development on Microcontrollers and FPGAs (MC&FPGA), doi:10.35598/mcfpga.2020.007.
[20] Clegg, Dai and Barker, Richard, (1994), Case Method Fast-Track: A RAD Approach, 978-0-201-62432-8, https://dl.acm.org/doi/10.5555/561543.
[21] Wernham, Brian, (2012), Agile Project Management for Government, Maitland and Strong, ISBN 978-0957223400.
[22] Microsoft, (2023), Windows API Reference, Console Developer's guide & API Reference, https://learn.microsoft.com/en-us/windows/console/console-functions
[23] Microsoft, (2023), Windows API Reference, Virtual Terminal Sequences, https://learn.microsoft.com/en-us/windows/console/console-virtual-terminal-sequences
[24] OSDev.org, (2021), Double Buffering, https://wiki.osdev.org/Double_Buffering
[25] GNOME, (1997-2023), GTK, https://www.gtk.org/
[26] The Qt Company, (1995-2023), Qt, https://www.qt.io/
[27] SEGGER, (2023), How RTT works, https://wiki.segger.com/RTT
[28] SEGGER, (2023), Converting ST-LINK On-Board Into a J-Link, https://www.segger.com/products/debug-probes/j-link/models/other-j-links/st-link-on-board/
[29] st-link.org, (2020-2023), Open source STM32 MCU programming toolset, https://github.com/stlink-org/stlink
[30] Visual Studio Code Documentation, (2023), Workspace recommended extensions, https://code.visualstudio.com/docs/editor/extension-marketplace#_workspace-recommended-extensions