Applied static firmware anlysis in an CI environment for increased safety and code review insights

Paul Würtz, Julius Kolb and Martin Eckardt

Abstract

Although formal tests of program logic matching functional requirements are often automated into the CI of a project with i.e. unit test, this is often not the case for other non-functional constraints, like memory usage and process time[1]. Not activly watching these constraints during development can end in mission-critical failures if not detected before testing on-device, and delay the insight that a controller's computational and memory capacity might not be sufficient for a given application. Evaluating these constraints at runtime demands a physical setup of all possible built targets with triggers for all possible events. Such setup, it's maintainance and throughput-limits add time- and require many resources to a project. Static analysis can yield a good approximation of these metrics for task timing and memory use without the costs of a physical setup and can be integrated into a continuous integration environment, similar to a functional test.

Tools for static firmware analysis are mostly proprietary and behind big pricewalls limited to few security critical industries, there is a big gap to open and free tools. One open and free candidate in that category that shows a lot of potential is puncover. In this work we want to present an extended use-case of how we can track memory constraints, especially in stack usage and other insights over the lifetime of a firmware, for various targets at build time and increase sensibility for firmware size metrics during code reviews and continuous integration runs. Furthermore, we want to take an outlock for further applications for RTOS and library development and take an outlock to more fields to analyze.

Keywords

code review, continous integration, firmware, static analysis, stack size, RTOS, tooling, Zephyr, ZephyrRTOS

1. Introduction

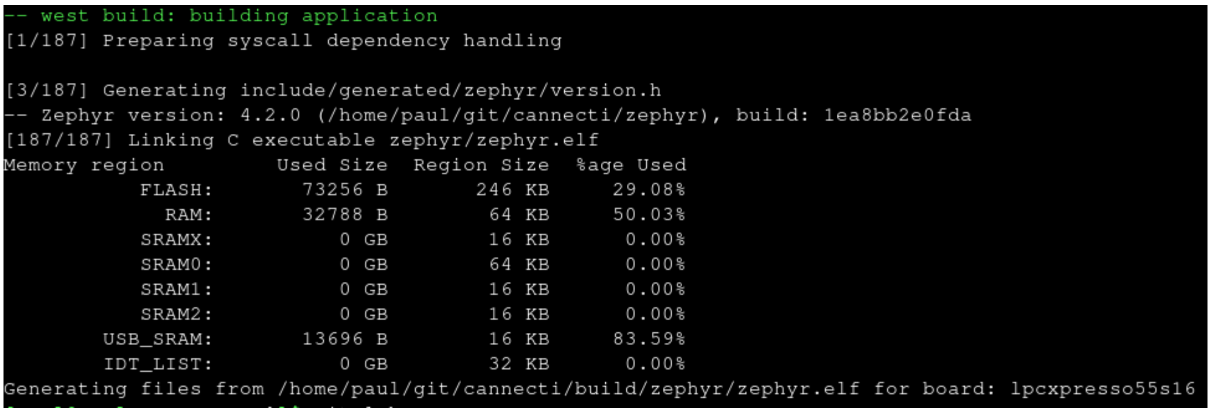
Manual memory layout and management in resource-constrained embedded devices, like microcontrollers with only a few kilobytes of Flash and RAM, using real-time operating systems is a typical use case. For flash memory - storing the program code - and global static RAM usage, the developer gets feedback after the firmware is compiled and linked as shown in figure 1. If any of the global memories are not sufficient in size the build will fail with a linker error. Then for cutting down on memory usage a more detailed overview can be helpful. On the other hand, if a task configured a too small stack size is configured for a task, this will often be discovered painfully during runtime. With static analysis and the knowledge of the configured stack size a stack overflow can be detected on the build process and an early failure could avoid a long error search. Since on-device runtime errors can be less clear, an early build error with an explicit error message of exceeded stack capacity supports the development process and also reduces later execution test efforts.

A considerable part of RAM can be used by tasks stack sizes. Knowing an exact limit of how much memory a task consumes on the stack lets the developer allocate the least amount of RAM possible - configuring a sufficient but not wasteful stack size. A viable stack size is often determined on the target device at run-time. For runtime evaluation, it is important to enter all program branches. Triggers to all possible events to enter all use cases of the controller are needed. This process needs to be repeated for continuous feedback on each task memory footprint. Typically, the complexity to reproduce all programm conditions increases with the development time. Often this elaborate setup and experiment is avoided, and stack sizes are chosen as defaults, by experience, or raised by arbitrary amounts after crashes caused by stack overflows. This either leads to wasted memory or can cause crashes in the field.

Zephyr in Science and Education, September 23-24, 2025, Jena, NY

paulwuertz@posteo.de (P. Würtz); juliuskolb@posteo.de (J. Kolb); marek@posteo.org (M. Eckardt)
https://github.com/paulwuertz (P. Würtz)

In this paper we want to explore how to tackle both issues, detecting stack overflows early in the build process and giving the developer feedback about sensible stack size by the results of static analysis.



```
-- west build: building application
[1/187] Preparing syscall dependency handling

[3/187] Generating include/generated/zephyr/version.h
-- Zephyr version: 4.2.0 (/home/paul/git/cannecti/zephyr), build: 1ea8bb2e0fda
[187/187] Linking C executable zephyr/zephyr.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:       73256 B       246 KB     29.08%
             RAM:       32788 B        64 KB     50.03%
           SRAMX:           0 GB        16 KB      0.00%
           SRAM0:           0 GB        64 KB      0.00%
           SRAM1:           0 GB        16 KB      0.00%
           SRAM2:           0 GB        16 KB      0.00%
        USB_SRAM:       13696 B        16 KB     83.59%
        IDT_LIST:           0 GB        32 KB      0.00%
Generating files from /home/paul/git/cannecti/build/zephyr/zephyr.elf for board: lpcxpresso55s16
```

Figure 1: Global memory use shown after building an application in zephyrRTOS

2. Current static analysis in puncover

Static program analysis is a wide field with diverse applications. Without executing a program, but by just analyzing the program source code or binary data, combined with knowledge of the programming languague and/or target computing platform, certain statements can be proven and runtime conditions can be formally verified, like correctness, runtime or memory consumption.

For this work we are interested in the memory footprint of the program - especially the space a task consumes on a stack. On an embedded controller running a RTOS usually several tasks run concurrently. Each task execution state is saved on a seperate thread with an associated stack in the controllers RAM. An illustration of a controllers RAM layout with such a memory layout is shown in figure 2. In this section, methods used to analyze a firmware build that give us the possibility to get estimation for each tasks stack usage are described.

2.1. Binary analysis by dissassembly

To use puncover the firmware binary with symbols as an ELF file and a compiler toolchain for the target are needed. The ELF is parsed and broken down into individual variables and functions and the program flow by all static calls between functions is extracted from the ELF. This is done by iterating line-by-line and do pattern-matching on each line of the ELF file's disassembly. Functions and variables are saved in a list of symbols. If they cannot be reconstructed by the regex-pattern the symbol is not added currently.

For each symbol its name, the address in memory, whether it is a variable or a function, the assembly instructions, the associated source file and linenumber where the symbol was defined are extracted by one regex. For these the compiler toolchain obtains the disassembly by calling *objdump -dslw ELF_FILE*.

The symbol sizes in program memory (typically flash memory) are extracted with another regular expression, applied on the output of a call to the compiler toolchain *nm -Sl ELF_FILE*. Each line-entry in the output of *nm* is iterated and matched and added to existing symbols or creates a new entry if new symbols are yielded by this step.

Further steps of enhancing the symbol data are:

- the disassembly calls from one function to others are found by also trying to match them with regular expressions to already found symbols - constructing a call tree
- if a build directory is given, *.su* files are searched and if found, each functions stack-usage and type is extracted - an example of a *.su* file is given in listing 1 - this requires build options and artifacts, described in the next section

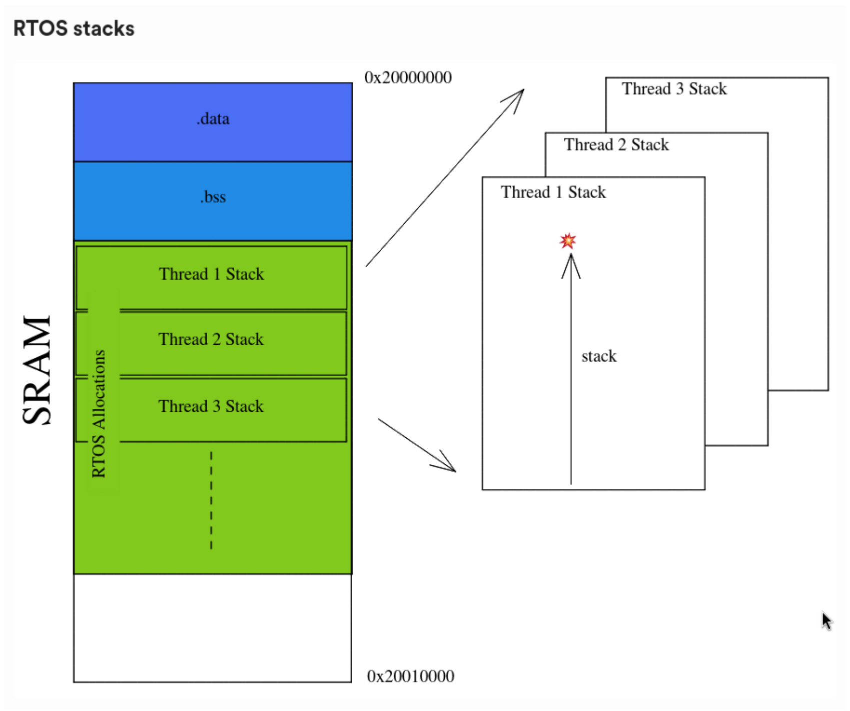


Figure 2: Typical memory layout of a microcontroller running a RTOS with multitasking, each thread with its own stack, picture source: https://interrupt.memfault.com/blog/measuring-stack-usage

- normalization of file paths and demangling of C++ symbol names
- finding neighbouring symbols - that are next to each other in the firmware address space and linking them on a list by association of the *next_symbol*

This results in a list of symbols - functions and variables, and a mapping of the elements in the source files to their compiled equivalents by address and size in the binary with a static call graph between function.

2.2. Additional compiler options for functions static worst-case stack-usage

The stack-usage of a function in the current implementation of puncover is not retreived by analyzing the ELF file assembly. It would be possible, but as a simpler method the output of the compiler is used. This needs specific compiler options and the user to provide the path to the build directory.

Listing 1: Stack usage file as implemented in [2]

```
fsu.c:4:foo  1040  static
fsu.c:7:bar  16  dynamic
fsu.c:10:main  32  dynamic,bounded
```

In [2] the introduction of the *-fstack-usage* flag is described. During the compilation, the size of all variables that get pushed onto the stack to generate the instructions that pass variables between functions and initialize stack-local variables must be known. With the said compiler option, this internal calculation can be output to files. Stack usage can be either *static*, *dynamic,bounded* or *dynamic* - to come to a worst-case memory usage everything but *dynamic* gives us a finite worst-case limit. For each compilation unit a *.su* file with one line for each contained function with its name, the stack size a call to this function consumes and the type of stack usage.

2.3. Known limitations and sources of errors

Missmatches in regular expressions are not handled or raised as warnings. A test regression suit, against a variety of firmwares and tracking the number of warnings already generated remains an open point.

The analysis time for certain projects can be very long: https://github.com/HBehrens/puncover/issues/101. In the original paper[2] *-fcallgraph-info* is introduced to get all the information out of GCC to calculate worst-case stack usage just by the output of GCC. In puncover the results of the dissassembly of the ELF file is used so far. Using the GCC provided output in puncover might be a way to improve analysis runtime and could in the future be added as an alternative provider for call information, in case the complete build environment is available additionally to just the ELF file.

Path matching is not working reliably with all forms of relative paths, and some symbols result with unknown stack size. Executing puncover this yields in warning messages ***WARNING Couldn't find symbol for FILE LINENUMBER SYMBOL_NAME***.

2.4. Stack Worst-Case Scenarios calculation

To calculate the worst or biggest use of stack, that a function is involved in a recursive search through the functions callers and callees is performed. The sum of both maxima and the function's own stack size yields the worst case.

An examplary vizualisation of some task's call-tree is shown in figure 3. For the right branches of task C stack-size are rendered with each leave representing decending callee paths branching out from the task.

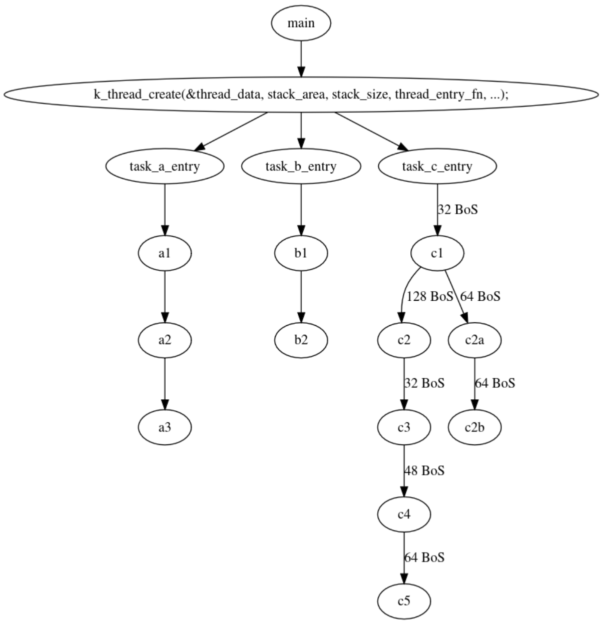


Figure 3: Example of firmware starting three threads, highlighting stack usage in the callee's of task c with how many Bytes of Stack (BoS) each call consumes

3. Additions to puncover

A positive of puncover is it can be called with few prerequisits. When in a more defined programming environment, we can add more information to get more context out of the analysis results. Some additions where implemented in this work and are presented in this section. A config via yaml, taking already existing and newly added arguments to puncover was added. An example of such config is shown in listing 2. elf_file, build_dir, src_root and port were already options, the others were added.

3.1. RTOS awareness - defining task entries

Calculating the worst-case stack scenarios on any firmware is already useful feedback for the developer. In the context of a multitasking operating system, this can be refined and applied to each task and the defined stack size of it. The memory layout of the RAM in such a system is shown in figure 2. If any thread stacks overflow during runtime, data in other stacks or content of variables can be over-written as an undesired consequence and the application state be damaged. This leads to undefined behavior and most often to a reboot by some hardware fault. Discovering the reason of the fault can be non-trivial as not the overflow itself but the effect of over-written memory can cause the reset. By knowing the size and entry point of each task by calculating the stack's worst-case size a tasks stack overflow can be detected at build time and the user can be informed directly with a descriptive error message.

To handle this kind of warning and errors related to stack overflows a list of task entries and configured stack sizes can be passed in the puncover conig as *report-max-static-stack-usage* added as an argument to puncover. For it to be easily usable on the command line the name of the function is serpated by ::: to the tasks stack size.

Listing 2: Puncover config file for the canectivity firmware

```
elf_file: /home/p4w5/git/cannecti/build/zephyr/zephyr.elf
build_dir: /home/p4w5/git/cannecti/build/
src_root: /home/p4w5/git/cannecti/
port: 5001
feature_version: "cannectivity-1.0"
generate-report: true
report-type: json
report-filename: report
report-max-static-stack-usage: [
  bg_thread_main:::1024,
  led_thread:::1024,
  gs_usb_tx_thread:::1024,
  gs_usb_rx_thread:::1024,
  log_process_thread_func:::768,
]
add-dynamic-calls: [
    z_impl_can_send->can_mcan_send,
    can_mcan_send->gs_usb_can_tx_callback,
]
error_on_exceeded_stack_usage: true
warn_threshold_size_for_max_static_stack_usage: 100
```

3.2. Early warnings and errors in continous integration

To enable an error when any task exceeds it's configured stack size the *error_on_exceeded_stack_usage*-option was added to throw an error with returncode 1. To warn developers earlier *warn_threshold_size_for_max_static_stack_usage* can set to a value in bytes. When in any task this amount or less of space is left as the difference of the static analysis worst-case stack-usage and the configured stack size an warning is triggered by returncode 79. To leaverage this to an CI pipeline an example step for a gitlab CI file is given listing 3.

Listing 3: Fragment of a gitlab-ci file to configure warnings and and errors after the build step

```
test_job_1:
  script:
    - puncover -c config.yaml
  allow_failure:
    exit_codes: 79
  cache:
    - key:
        files:
          - staticReport.json
```

3.3. User-defined additional information - dynamic callbacks

In many places dynamic function calls like callbacks in driver stacks are common in firmware. Especially in the zephyrRTOS driver stack this is a common abstraction interface for implementation stacks. The lack of dynamic callbacks limits the result of static analysis, hiding many call paths, that the firmware will be executing on the device. The developer configuring these callbacks can supply the static analysis with this information to improve the analysis.

To do so add-dynamic-calls was added as an argument to the puncover config. Each entry to this list is a string with the calling functions, followed by a '->' and the callbacks function name like: **z_impl_can_send->can_mcan_send** from listing 2 line 16. This call relationship will then be added by puncover and explored for the worst case stack calculation.

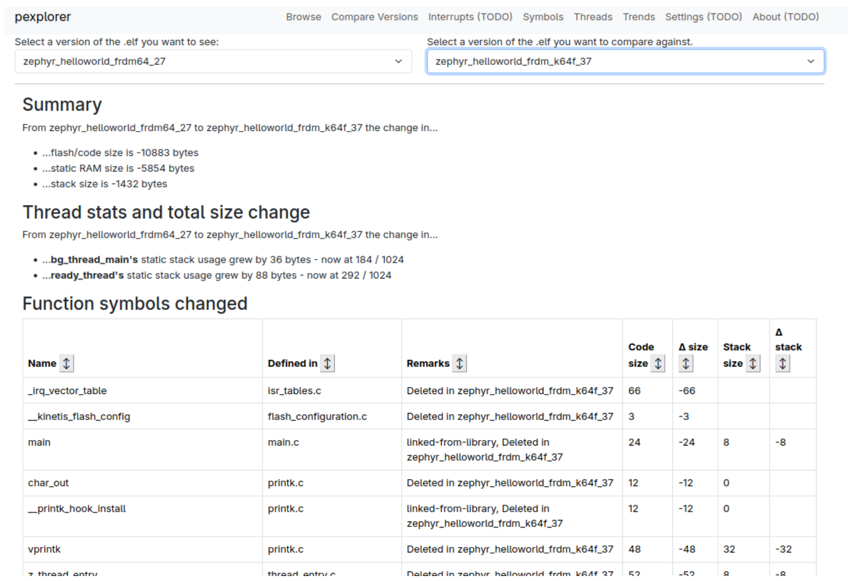
3.4. Seperating data generation and presentation

The development loop present in puncover started the analysis and served the presentation of the result on a live webserver. This made testing new presentational improvements time intensive in testing, since disassembly had to be repeated and took upto 30 minutes for some ELF files. Therefore a symbol and analysis result export to JSON was defined, allowing for running an analysis once and iterate on the presentation quicker with svelte as a frontend that supports also hot-reloading, minimizing the restart time to see code changes refelct on the UI representation.

4. Analysis presentation in pexplorer

4.1. Compare memory usage differences across build versions

To compare two build direct changes on the binary level a diff-view was added. All changes between the two compared versions global flash, RAM and cumulative stack-usage are listed. The changes to each tasks worst-stack size usages are listed as well. And a detailed list of symbols that were, added, delted or changed in size of flash or stack size between the two versions are rendered in two sortable tables, one for functions and another for variables.



Select a version of the .elf you want to see: zephyr_helloworld_frdm64_27

Select a version of the .elf you want to compare against. zephyr_helloworld_frdm_k64f_37

Summary

From zephyr_helloworld_frdm64_27 to zephyr_helloworld_frdm_k64f_37 the change in...

- ...flash/code size is -10883 bytes
- ...static RAM size is -5854 bytes
- ...stack size is -1432 bytes

Thread stats and total size change

From zephyr_helloworld_frdm64_27 to zephyr_helloworld_frdm_k64f_37 the change in...

- ...**bg_thread_main's** static stack usage grew by 36 bytes - now at 184 / 1024
- ...**ready_thread's** static stack usage grew by 88 bytes - now at 292 / 1024

Function symbols changed

Name	Defined in	Remarks	Code size	Δ size	Stack size	Δ stack
_irq_vector_table	isr_tables.c	Deleted in zephyr_helloworld_frdm_k64f_37	66	-66		
__kinetis_flash_config	flash_configuration.c	Deleted in zephyr_helloworld_frdm_k64f_37	3	-3		
main	main.c	linked-from-library, Deleted in zephyr_helloworld_frdm_k64f_37	24	-24	8	-8
char_out	printk.c	Deleted in zephyr_helloworld_frdm_k64f_37	12	-12	0	
__printk_hook_install	printk.c	linked-from-library, Deleted in zephyr_helloworld_frdm_k64f_37	12	-12	0	
vprintk	printk.c	Deleted in zephyr_helloworld_frdm_k64f_37	48	-48	32	-32
z_thread_entry	thread_entry.c	Deleted in zephyr_helloworld_frdm_k64f_37	52	-52	8	-8

Figure 4: Diff view in pexplorer

4.2. Detailed memory reports and trends

As shown in figure 1 overall memory usage is an essential health feedback developers get when building a firmware. This metric is important to understand during the development process, since the choice for a particular controller is a reflection of a projects requirements and choosing a controller with sufficient program and RAM memory is not a decision not fully informed based on assumptions and experience.

If these planning assumptions for the right controller turn out while development processes and a product matures. Also changing requirements can increase compute and memory prerequisits. If code reuse or libraries and RTOSes are used for the project knowing more detailed flash and RAM requirements is key for good early estimations. For later stage size optimization and code triage knowing the memory footprint of each software component is essential data for informed choices.

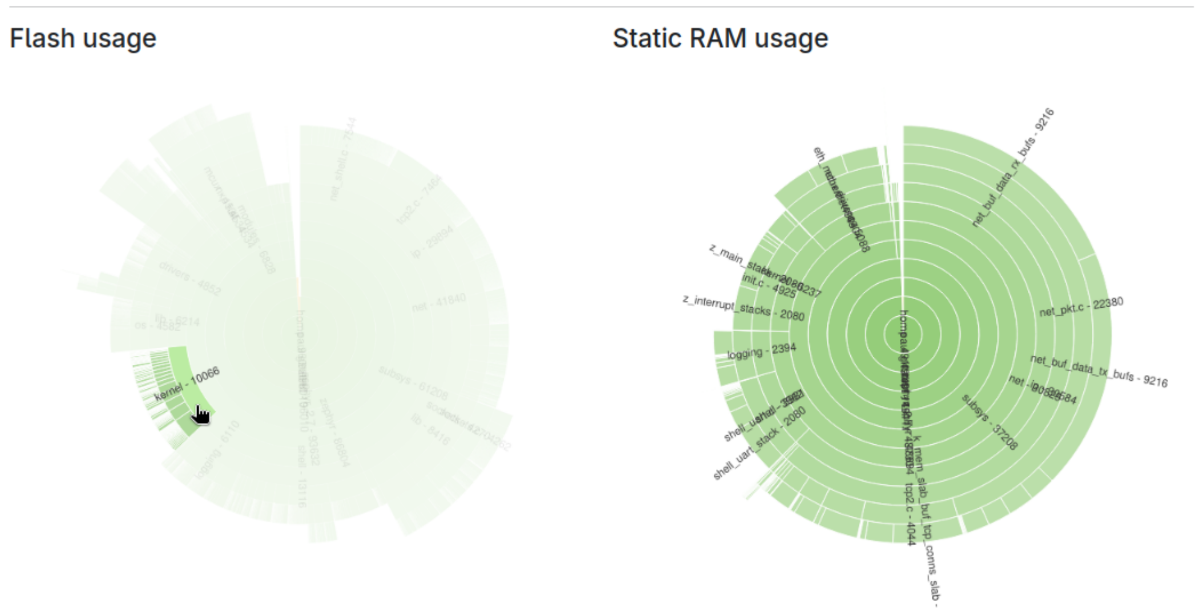


Figure 5: Detailed flash and stack use by folder, file down to symbol on interactive sunburst charts

To give development and project managment a better overview of memory footprints an hierarchical representation in a sunburst chart is proposed, as can be seen in figure 5. This reflects the current memory consumtion for flash and RAM usage.

4.3. Other stats and trends

For progressing development and release trends of memory usage trends of the memory usage over a series of firmware releases is displayed. In figure 6 a trend of static stack usage over some releases is shown in the upper plot. On the lower plot a trend for dynamic allocation are displayed, as these were of engineering interest, since dynamic allocations were undesired in the project where the idea for this paper came from and many dynamic allocation were included involuntarily by

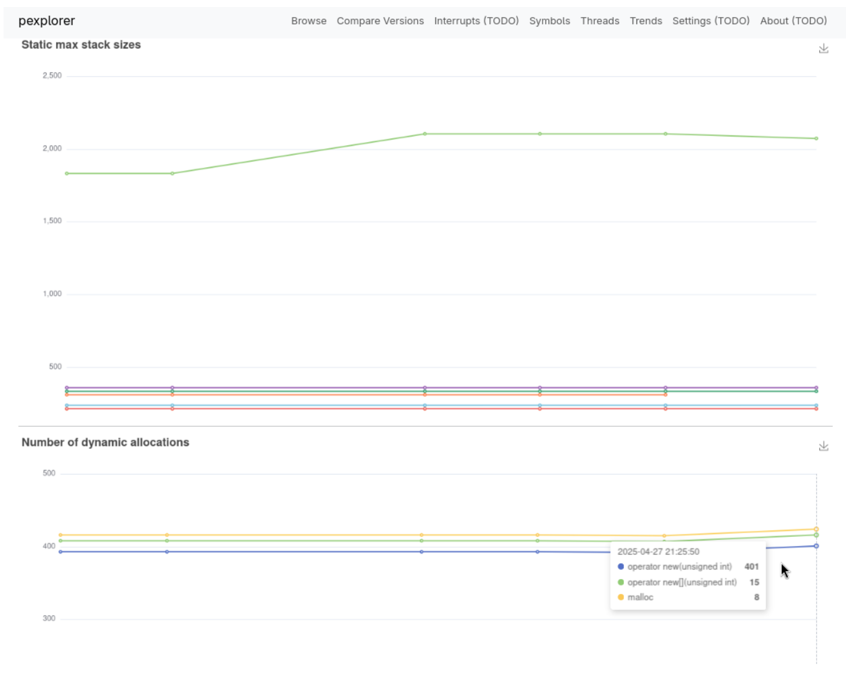
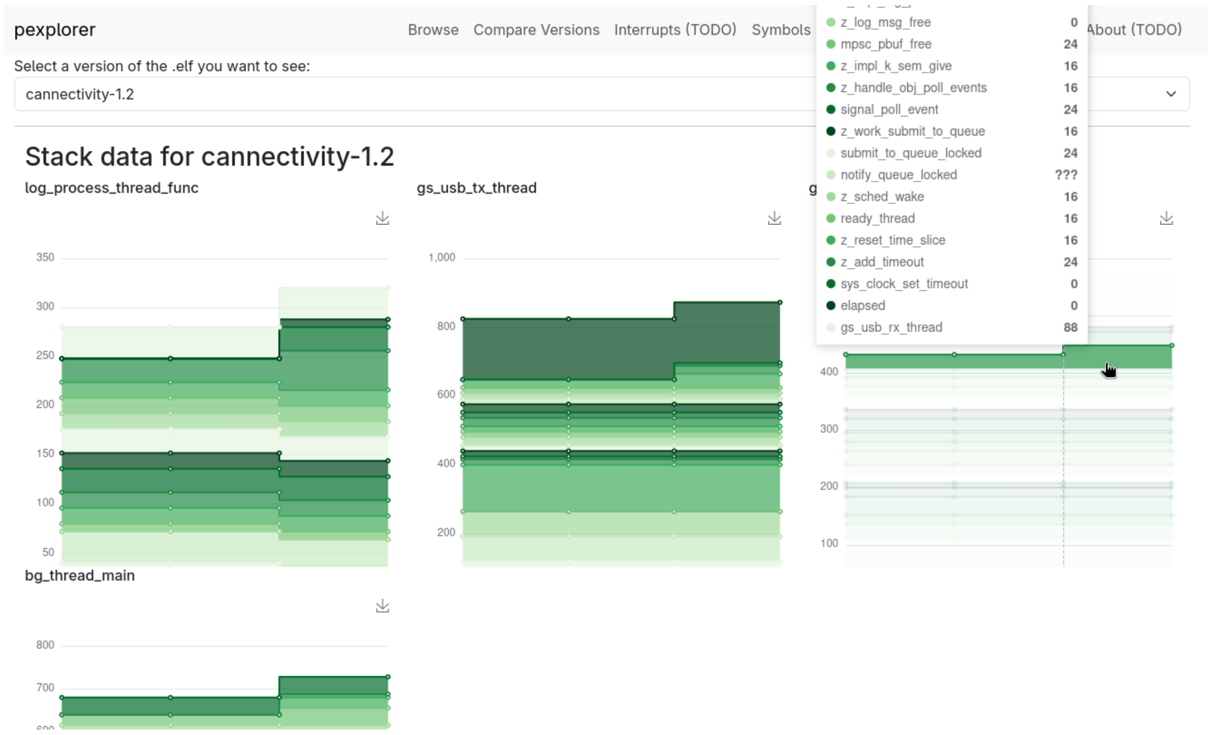


Figure 6: Trends page with graphical overview of each stack-usage and number of dynamic memory allocations found in the firmware



● z_log_msg_free	0
● mpsc_pbuf_free	24
● z_impl_k_sem_give	16
● z_handle_obj_poll_events	16
● signal_poll_event	24
● z_work_submit_to_queue	16
● submit_to_queue_locked	24
● notify_queue_locked	???
● z_sched_wake	16
● ready_thread	16
● z_reset_time_slice	16
● z_add_timeout	24
● sys_clock_set_timeout	0
● elapsed	0
● gs_usb_rx_thread	88

Figure 7: Overview of changing worst stack-usage call path with highlighted changed of all involved callees

the C++ std-library. The trend of these and other metrics are possible by analyzing several static analysis results together and can help engineers to keep track on development goals by defining simialar metrics, tracking them and integrate the analysis into the CI pipeline for releases and merge requests.

As a addition to the textual worst-stack usage of the current firmware a historical overview of all loaded firmware analysos in one graphic for each task and the stacked changes of it's worst case call path with changing size information of each function is displayed - an example is shown in figure 7.

5. Outlock

This paper only scratches the possibilities of using static analysis to benefit firmware development.

Some issues are raised in Section 2.3.

The quality and correctness of produced analysis with puncover is though interesting and useful up for formal debate, as no reference test suit and comparison with measured dynamic stack usage has formally been done and remains an open task.

The introduced user-defined dynamic callbacks could be automated partially in build environments like Zephyr's West for parts of driver abstractions in various stacks - so that e.g. configured driver implementations known at compile time but implemented as dynamic function calls could generate a configuration file some dynamic callbacks present this way. Also, the user could be guided and warned about jump instructions leading outside of functions that could not be mapped to any static functions to be potential dynamic calls and a user interface to associate them with a function, to make the manual part easier and less error-prone.

Extending and integrating stack usage into the twister build helper, especially the *–footprint-report* and *–footprint-threshold* options are interesting candidates, so that the reports for many build targets could be generated in one call and the key results be evaluated automatically by elevating existing program options.

Further ideas to process and condition the data for data vizualisation for further insight over the development cycle are open to be explored.

Adding Embedded Software Timing as another analysis item could be an attractive option. The results of further static analysis tools such as platin [3] could be compiled into a common extended report format to be displayed in the presented pexplorer as a single interface.

Acknowledgments

Thanks to the sonnen GmbH for the self-study time to think of ways to cut debugging time and search for ideas how to automatically identify memory problems early, to enhance puncover and to watch certain firmware metrics more easily.

Thank you Tobi and Stephan for encouraging to further work and present the topic.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] Gliwa, Embedded software timing (2021). URL: https://www.gliwa.com/book/.

[2] E. Botcazou, C. Comar, O. Hainque, Compile-time stack requirements analysis with gcc motivation, development, and experiments results (2007).

[3] E. J. Maroun, E. Dengler, C. Dietrich, S. Hepp, H. Herzog, B. Huber, J. Knoop, D. Wiltsche-Prokesch, P. Puschner, P. Raffeck, M. Schoeberl, S. Schuster, P. Wägemann, The Platin Multi-Target Worst-Case Analysis Tool, in: T. Carle (Ed.), 22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024), volume 121 of *Open Access Series in Informatics (OASIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2024, pp. 2:1–2:14. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2024.2. doi:10.4230/OASIcs.WCET.2024.2.

A. Online Resources

The sources for the ceur-art style are available via

- GitHub - puncover fork,
- GitHub - pexplorer,
- Presentation slides,