# System optimization for enabling debugging

Anudhi Jain, Raju Udava Siddappa, Tushar Vrind, Venkat Raju Indukuri
*Samsung Semiconductor India R&D*
*Samsung Electronics*
Bangalore, India

***Abstract*— Embedded systems are designed with low on-chip RAM and low processing power to reduce cost and power consumption, so software running on it should be optimized to utilize less resources. At the same time it is important to allow room for adding debug information so that failures can be analyzed quickly in order to reduce overall software development cost. Usual methods of adding debug logs in a software has both memory and performance overheads. In this paper we discuss mechanisms through which the debug log mechanisms memory overhead can be reduced by nearly 100%, and its processing overhead can be reduced by over 85%. Through this optimization it is expected that the development cost will be reduced, and in addition it will improve the systems scalability to meet future incremental requirements.**

***Keywords*—system optimization, debug logging mechanism, overlay**

## I. Introduction

Embedded software, because of its nature of existence – doesn't have the luxury of having vast amount of memory and high processing power at its disposal; thus the judicious use of both resources becomes a necessity. As the software goes through a cycle of design, implementation, testing and validation, it is possible that during the stage of validation - it may re-enter design phase or implementation phase because of a particular failure. Usually around 60% of defects exist at design time [1], and reworking on defects consumes around 50%-60% of the total software development cost [2].

Software for embedded systems such as routers, access points or wireless modems are designed around standard driven protocol specifications and such software needs to be upgraded with newer features to inter-operate and scale up to ever growing market needs. These incremental changes will require additional system resources – both processing and memory.

In order to understand the behavior and flow of the software transactions during failures, debug traces or prints are added in the code. Such traces are transferred via a debug interface to a separate process, normally residing on a host PC. Debug trace logging mechanism comes with a trade-off in the form of additional memory requirement (static strings included in prints) and processing overhead (to transfer logs through debug interface).

Through this paper, we try to address these two problems. First, by building an optimal logging mechanism that allows enabling of extensive debugging - we reduce the development cost as the test-debug cycle is expected to be iterated lesser number of times. Second, by optimizing performance and reducing memory footprint we create room for new features to be added to the software.

The rest of the paper is organized as follows. Section II discusses problems with existing debug logging mechanisms. Section III discusses solution and captures processing gains derived out of the proposal. Section IV discusses memory optimizations derived out of the proposal. Section V discusses the conclusion.

## II. Software Diagnosis

Debugging information is usually gathered by adding debug traces in the implementation which invokes an available platform specific logging API (application program interface). The API internally transfers the gathered information through the debug interface to a separate process residing on the same device as shown in Fig. 2 or on a host PC as shown in Fig. 1. The debug interface could be a serial interface like UART or USB, or a standard File I/O.

A typical debug trace would be of below format:

PRINT (<format string>, < variable argument list >)

where, <format string> is a typically a static string, which becomes part of (constant) read-only area or text area of the binary image [4] and <variable argument list> indicates run time parameters for the format string.

In a standard implementation of PRINT, the debug trace is formatted along with the arguments with in the device and the formatted output (including the string and the arguments) is transferred over the logging interface, so that the same is visible on external Diagnostic Monitor (DM). This is depicted in Fig. 3 where the formatted logs are stored in the log store. Logs are subsequently polled by the background task and transferred via the logging interface to DM.

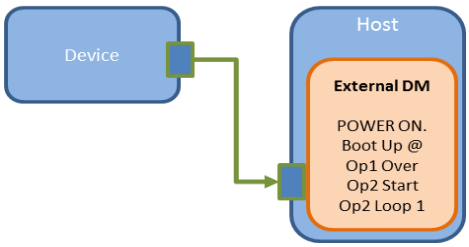


Fig. 1. External Diagnostic Monitor

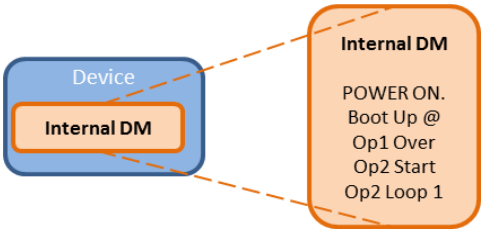


Fig. 2. Internal Diagnostic Monitor

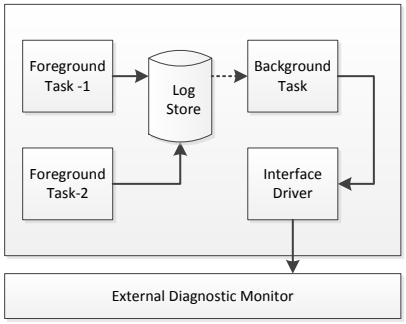


Fig. 3. General design of Logging Utility

Such a design inevitably comes with below drawbacks:

- Performance degradation due to formatting of the string within the device (PRINT, printf or similar routines).
- Memory consumption (RO area) due to many such constant strings (format strings) in entire software.

A program (or binary image of any system) to be executed is loaded into RAM by the boot-loader (static loading) or by the operating system (dynamic loading). A typical memory layout of a program, after loading into RAM is shown Fig. 4. The size of the generated binary (Text + Data + BSS) directly affects the available run time resources like stack and heap.

It's also desirable to not restrict a developer from using debug trace logs (restriction in terms of their count and size), and still get away with the memory and processing overhead.

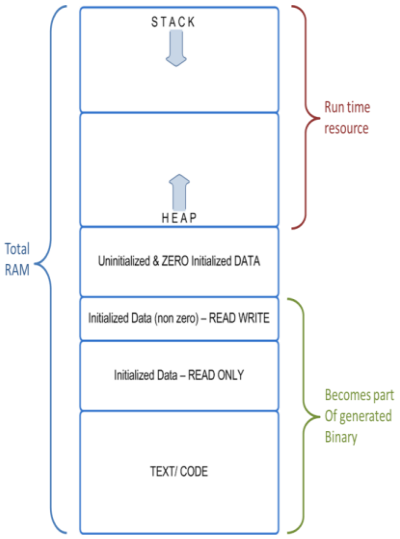


Fig. 4. RAM view for a software system

## III. Performance Optimization

In proposed mechanism, the logging API won't format the print within the device as shown in byte indexed Table I; instead it would send the reference of the unformatted string along with its arguments, as shown in byte indexed Table II.

For example if there is a print statement (ANSI C) "Completed initialization: %d" and its run time argument value is '0', then a reference to the unformatted string and the value '0' will be transferred to the DM.

TABLE I. TRANSFERRED DEBUG TRACE (BEFORE)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| C | o | m | p | l | e | t | e |
| d | | i | n | i | t | i | a |
| l | i | z | a | t | i | o | n |
| : | 0 | | | | | | |

TABLE II. TRANSFERRED DEBUG TRACE (AFTER)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| String Reference | | | | 0 | | | |

The formatting is done by host DM, so that the final print is visible on the host as "Completed initialization: 0".

The reference passed over the logging interface could be an agreed index to a table say 'INDEX_1' which indicates a string "Completed initialization: %d" to be used at DM. This table would be common and shared between the software on the device and external DM on host system.

The reference could also be an address pointing to the physical location of the string in binary. External DM can use software image (binary file), compute and generate string location offset in binary by using the reference address and read the corresponding string present in the binary file.

Adopting this mechanism leads to significant processing gain, as the formatting is offloaded to DM. Additionally, size of data to be transferred over the debug interface is reduced as shown in Table I and Table II. In our Test bed which is based on a mobile platform like [5], we measured the average gain of around 85% processing reduction per debug trace as shown in Table III.

TABLE III. GAINS OF PERFORMANCE OPTIMIZATION

| Parameter | Old method | Proposed method |
|---|---|---|
| Processing time | 20 uS | 3uS |

## IV. Memory Optimization

Debug strings would linearly increase as and when new features/ enhancements are added into the system, thereby increasing time to boot-up in addition to adding memory overhead.

In proposed mechanism, only a reference of the format string is required as the software does not format print strings before transferring it over the logging interface; unformatted strings aren't accessed at run time. Thus, static memory occupied by constant strings becomes redundant and therefore be used for other purpose (as heap/ code area).

As such format strings are cluttered throughout the binary and are needed to be clubbed into a single area/cluster before re-claiming the area occupied by them. Identification of such constant format strings during compilation can be done by using compiler specific syntax, as in ARM compiler [3].

Syntax example: _Pragma ("arm section rodata = \"TRACE_STRING_AREA\"")

Identified regions can then be grouped together at linking time by using linker specific scatter loading as in Fig. 5, and placed in contiguous memory area (say "TRACE_MEM").

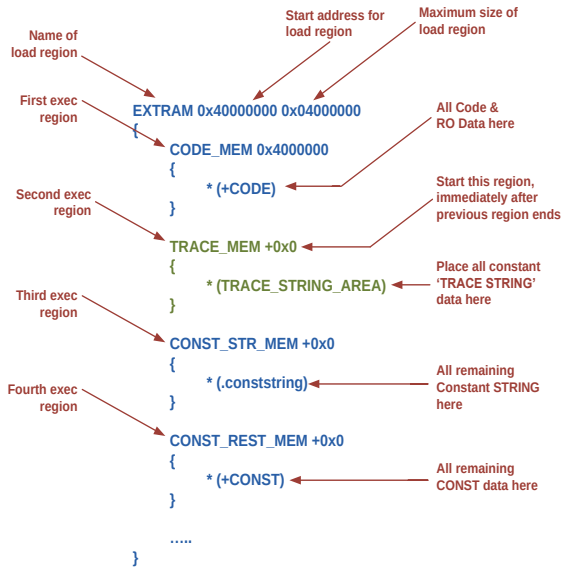


Fig. 5. Scatter RAM depiction showing TRACE_MEM section creation

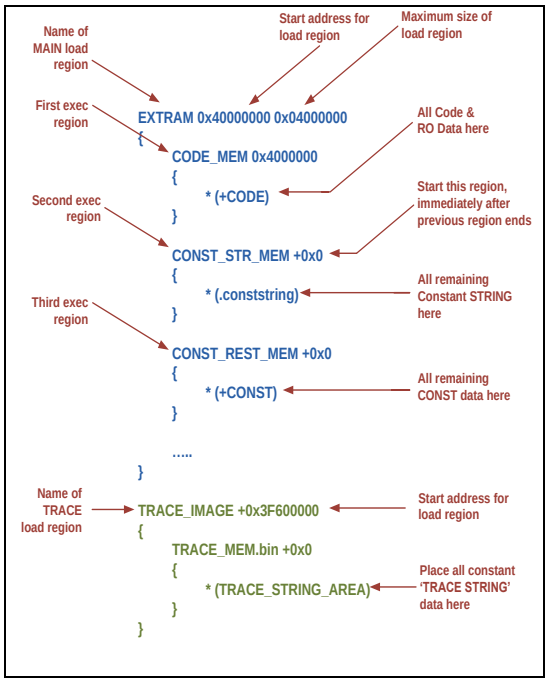


Fig. 6. Scatter RAM showing Trace memory moved out of main memory

'TRACE_MEM' memory region can then be:

1. Reused as Run time dynamic memory pool (heap) after boot up initialization is completed
2. Moved out of the main memory map using linker specific scatter loading technique as in Fig. 6 so that it doesn't occupy any physical RAM area. Two load regions will get generated as a result of this. 'TRACE_MEM' will continue to be part of the software image (binary file), so DM can use binary file to resolve the reference to the unformatted string references passed in the debug trace.
3. In an alternative solution for memory reuse the concept of OVERLAYs [6] can be exploited, and we have exemplified the same using scatter loading principles for an ARM based test bed [5].

'OVERLAY' attribute can be used in a scatter file to place multiple code/data blocks at the same memory location [7]. Generally Linker throws an error when more than one memory region has the same memory address but when memory region is specified with OVERLAY attribute in a scatter file, the code or data to be placed in this region will be linked. However, the loading of code/data in the execution region needs to be managed by 'overlay manager' at run time. Fig. 7 explain the OVERLAY concept, and how regions marked as overlays can be swapped by the overlay manager on a need basis. This is dynamic process, and was utilized in some of the old operating systems like RSX [8] before demand paging was widely adopted.

A typical scatter-loading concept based on a scatter loader description file for ARM [6] is illustrated in Fig. 8, where the initial copy from load region to execute region is taken care by the scatter loader functionality in the initialization/boot up sequence routine. However, if a region has an 'OVERLAY' attribute then it needs to be loaded by the overlay manager as shown in Fig. 9. In this example region called as RAM1 (comprising of Constants), and region called as RAM2 (comprising of Read/Write (data) and ZI (BSS)) are overlaid and have the same execute address 0x10000.

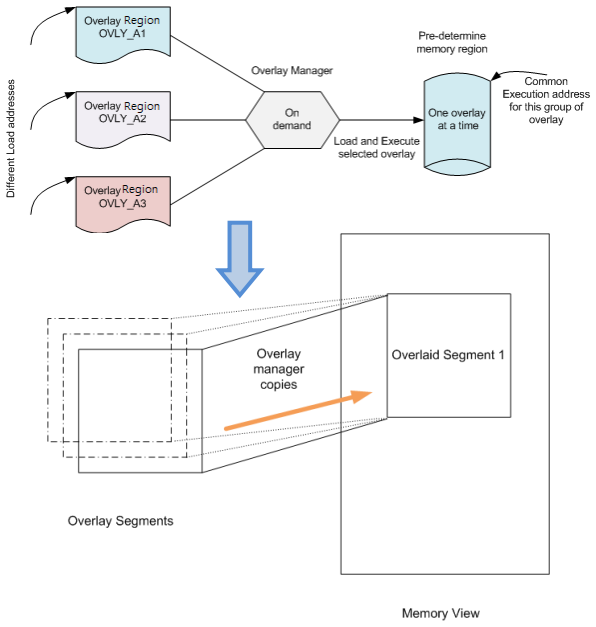


Fig. 7. Overlay concept: overlay region and overlay manager

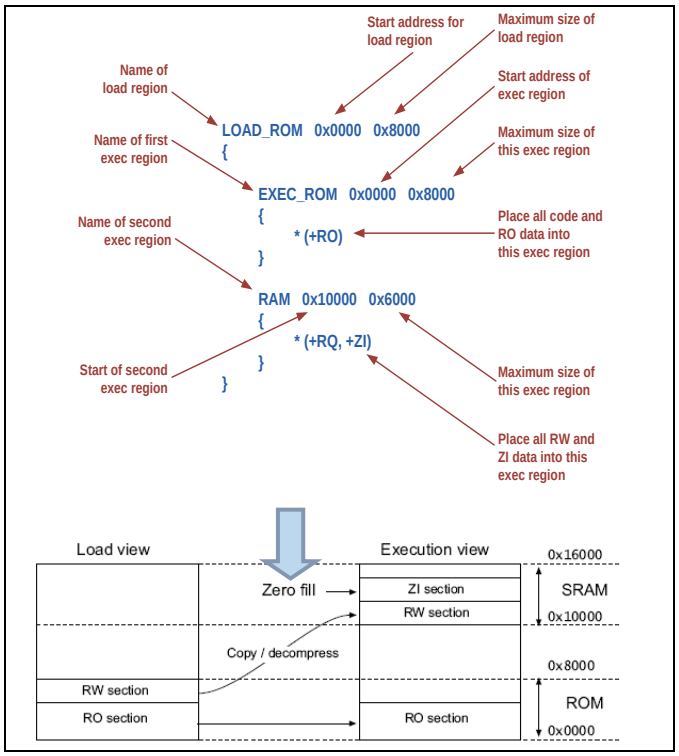


Fig. 8. Scatter loading illustration

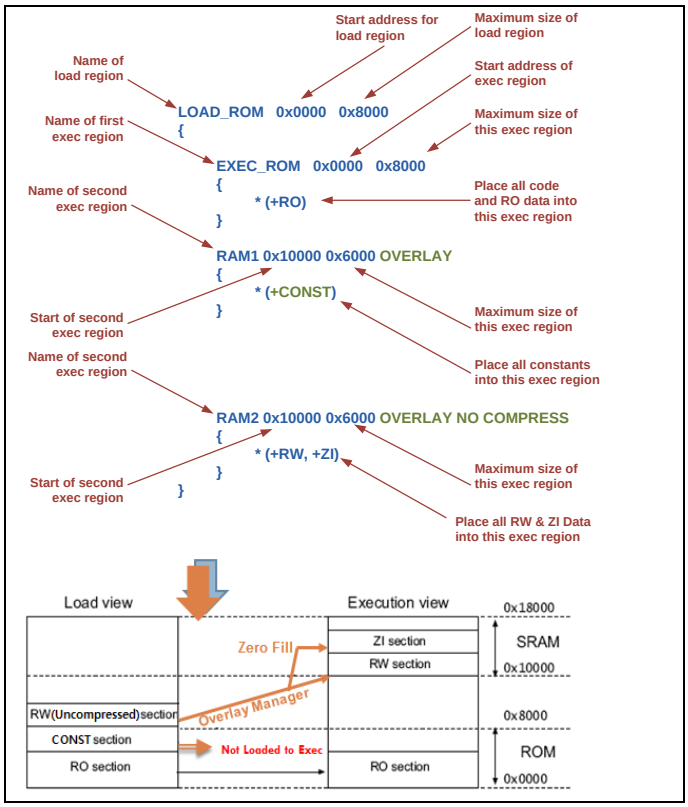


Fig. 9. Scatter loading for Overlay

If we overlay 'TRACE_MEM' with ZI region, we can avoid the need for dynamism associated with overlay swapping since 'TRACE_MEM' is a redundant memory region (possible after the performance optimization implemented in Section III). This redundancy ensures that it would never be required during execution.

Thus, we introduce a concept called a 'Fixed or Pre-loaded Overlay', where there is no need for an overlay manager to exist, and the initial loading of the region is taken care by the usual scatter loader functionality found in the standard initialization library. Conceptually this scheme can be used in cases, when only a memory reuse (dual use) is expected, and perhaps a dynamic change of contents of an overlay region is not expected (though allowed). This is illustrated in Fig. 10. In order to realize this concept, we let region RAM1 and RAM2 share the same execute address even though region RAM2 doesn't have an attribute of an 'OVERLAY'. Using this mechanism, copy from load region to execute region for RAM2 is taken care by the scatter loader functionality in the initialization/boot up sequence as part of normal standard procedure.

Table IV analyses the proposed memory optimization schemes and concludes that the 'Fixed/Preloaded' Overlay mechanism outperforms the other two mechanisms from maintainability, reuse and ease of application perspectives.

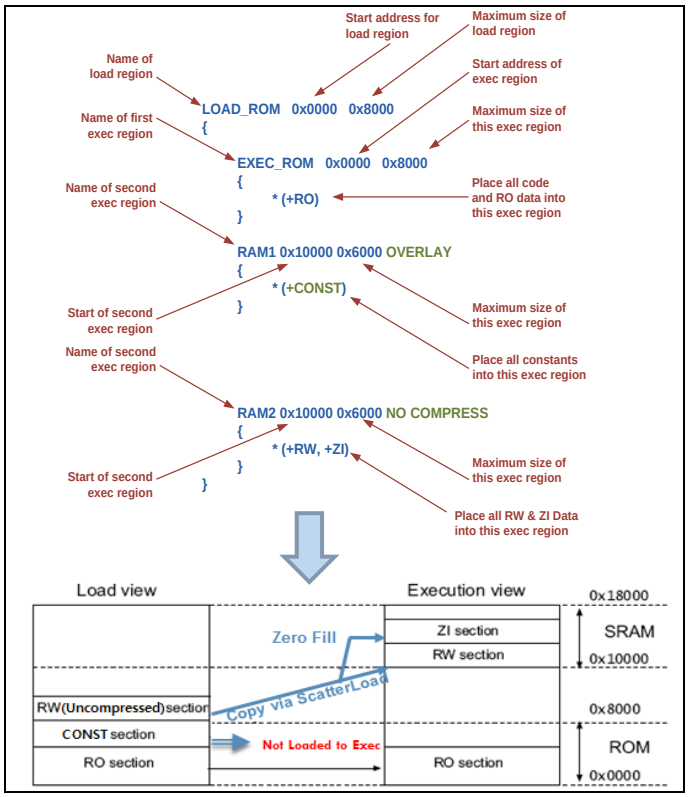


Fig. 10. Fixed or pre- loaded overlay

TABLE IV. ANALYSIS OF MEMORY OPTIMIZATION SCHEME

| Scheme | Re-claim at runtime | Relocate base address | Fixed/Preloaded Overlay |
|---|---|---|---|
| Device Binary Size | No Change | Decreases | No Change |
| DM Binary size | No Change | Increase | No Change |
| Reusability | BSS only | BSS + TEXT + RW | BSS + TEXT+RW |
| Cons | Internal Fragmentation | Address space restriction | None |

## V. CONCLUSION

By using proposed mechanism for combined processing and memory optimization deployed in debug trace management, developers and manufacturers of software systems can reduce huge memory footprint induced by debug trace logs. In addition, system developers can do away with restriction on size/ length of format string as processing time would remain same for all logs irrespective of their length. The proposed approach provides several benefits to system developers by providing a method to get around 85% processing gains, and around 100% memory gains in the logging utility of the embedded software. With this, it is expected that the development cost will be reduced, and in addition it will improve the systems scalability to meet future incremental requirements.

## REFERENCES


[1] T. Gilb, "Principles of Software Engineering Management", Addison-Wesley, Wokingham, UK, 1988.

[2] B. W. Barry "Improving Software Productivity" IEEE Computer, September: 43-57; Jones, Capers, ed. 1986. Tutorial: Programming Productivity: Issues for the Eighties, 2nd ed. Los Angeles: IEEE Computer Society Press (1987)

[3] #pragma ARM section. [Online]. Available: http://www.keil.com/support/man/docs/armcc/armcc_chr1359124985290.htm

[4] ARM ELF File Format. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0101a/DUI0101A_Elf.pdf

[5] New Cortex™-R Processors for LTE and 4G Mobile Baseband. [Online]. Available: https://www.scribd.com/document/103438212/New-Cortex-r-Processors-for-Lte-and-4g-Mobile-Baseband-1

[6] O. Beckmann, *Operating Systems Concepts: Chapter 8: Memory Management.* Department of Computing, Imperial College London, 2005.

[7] RealView® Compilation Tools. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0206j/DUI0206J_rvct_linker_user_guide.pdf

[8] Introduction to RSX-11. [Online]. Available: http://bitsavers.informatik.uni-stuttgart.de/pdf/dec/pdp11/rsx11/RSX11M_V2/DEC-11-OMIEA-A-D_Intro_74.pdf