RISC-V Overlay “Software Standard Proposal” Version 0.2-draft-20201230

Table of Contents

Revision History

Revision	*Date*	*Contents*	*Author(s)*
0.1	Oct 07,2020	Initial revision / Draft a	Ronen Haen Ofer Shinaar Craig Blackmore
0.2	Dec 30,2020	Update changes gathered in the overlay-tg meetings	Ofer shinaar Craig Blackmore

List of Figures

List of Tables

Reference Documents

Item #	Document	Revision Used	Comment
1			

Abbreviations

Abbreviation	Description
SW	Software
HW	Hardware
MMU	Memory Management Unit
LRU	Least Recently Used
OS	Operation System
RT	Runtime
RT-Engine	Runtime Engine
RTOS	Real-Time Operating System
R/O or RO	Read Only

1. Overview

Physical memory is one of the most important resources in SW programming, since the program runs on it. Some embedded systems have limited memory resources and as a result the total code

footprint is often bigger than the available memory. To solve this problem, the program needs to use a technique to load executable code at runtime from virtual memory or to overwrite executable code that is not currently needed by the program.

On OS driven systems a virtual memory is very commonly used, the OS will use a HW MMU with direct mapping between virtual and the physical memory.

There is another technique that can be used to realize this paradigm in SW without the need for special HW. It is called ‘SW overlay”.

The concept of arranging code in ‘code overlays’ is quite old but still valid these days to resolve the code size issue. For small embedded systems (like IoT) footprint and code size is critical. Those systems usually will not be driven by a large OS (e.g. Unix) and will not include a MMU.

An overlay represents a function or read-only data that by some scheme will be loaded into heap area when the SW requires it (e.g. for a function call). The overlay solution offers an advantage over HW, since the SW controls what needs to be loaded and when.

2. Goals

This document specifies the requirements for all the relevant SW entities needed to implement SW overlays. We expect changes in the toolchain and a requirement for a RT-Engine that handles calls/returns involving overlay functions and loads overlay functions/data as needed. The following requirements will lay a base for everyone to implement their own RT-Engine or toolchain support and/or take just part of the tools and integrate it with their own design.

The main goal of this standard is to make the SW developer experience an easy and smooth one. From the developer’s point-of-view the request will be to just tag their designated overlay functions and overlay data with an attribute and the toolchain and RT-Engine will take care of the rest.

The implementation will be based on the RISC-V ISA and RISC-V tools but can be adapted to other targets. Moreover, the implementation will be open source, both for the RT-Engine and the selected toolchain.

3. Methods requirements

The following sections specify the requirements for:

1. RT-Engine – SW module which is responsible for managing the overlay heap. This module is designated to be part of operational SW.
2. Toolchain – broad collection of programming tools (e.g. compiler, linker, debugger, etc.) needed to develop SW applications.

3.1. RT-Engine

3.1.1. General

1. Since SW can be more flexible then HW, we should not use a direct mapping approach for overlays. For a small allocated overlay heap we can map any amount of code.
2. Functions and read-only data can be in overlays.
3. Functions and read-only data will be assigned to one or more overlay `groups`.
4. The RT-Engine will be aware of the functions in the group and how to address them.
5. The RT-Engine will manage the loading/eviction of groups via hooks to be implemented by the platform.
6. The RT-Engine can run on a bare metal system or under a RTOS and therefore should be aware of RTOS usage to ensure it is thread-safe, since any given thread can invoke overlay functions or use overlay data.

3.1.2. Groups

A 'Group' is a collection of overlay functions and overlay data. We should use groups to minimize the necessity of loading/evicting a singular function from the overlay heap.

1. Overlay group size can impact the RT-Engine and the toolchain so it must be selected pre-build.
2. Overlay group size ranges from 512B – 4K for both functions and RO data.
3. Group size will be decided in advance by the developer, and we will be provided at link-time.
4. An overlay function or overlay data must not be bigger than the maximum group size.
5. *Multi Group* – an overlay function or overlay data can be resident in N groups.
   Example: foo(void) can be located in $Group_1$, $Group_2$..., *GroupN*

3.1.3. Evict

Group eviction can be handled with similarity to HW cache concepts.

1. Eviction resolution will be at `group` granularity, meaning we can evict N groups per demand.
2. The search-algorithm for determining whether a group is loaded or not shall be defined at compile time.
3. The search-algorithm is open to interpretation; we recommend to have at least one, for example LRU.
4. The RT-Engine will provide a "group lock/free" API mechanism to prevent specific groups from being evicted.

NOTE	This section is optional. But it is ***recommended*** to have it if the design requires eviction, due to heap space limitations.

3.1.4. Load

The load area, “heap”, contains loaded overlay groups. It should have its own memory section definition, so that the RT-Engine and the toolchain can work on the same section.

1. The heap area should be defined pre-build.
2. The heap should have range limitation to be in sync with the RT-Engine and toolchain. The heap minimum size should be bigger or equal to the maximum pre-defined overlay group size. (***heap-min-size*** >= ***max group size***)
3. We can have multiple heaps to be controlled by a single/multiple RT-Engine(s).
4. A *Load-Function-Hook* [1] will be provided to the user for executing the load operation itself.
5. The RT-Engine should hold all information regarding the heap. Which area is allocated/free, sizes, and address.
6. Based on the given "heap information," the RT-Engine should provide a mechanism to lock segments in the heaps.
7. The heap information should be accessible by the application to get the status of the heap.

NOTE	[5-7] This ability can give the application a way to allocate memory from the heap

3.1.5. Invoke

The RT-Engine will be the entity to invoke the overlay function.

1. The RT-Engine should support invoking indirect function calls (i.e. calls via function-pointers)
2. After a function is loaded to the heap, the RT-Engine will be responsible for passing all requested arguments from the root caller to the callee.

   Therefore, the RT-Engine will apply the ABI rules.
3. We should follow the ABI in the matter of passing X numbers of arguments to `callee`. Per the current RISCV psABI the max numbers of registers to pass to callee is #8. above that, all arguments will be pushed stack.
4. Return values to `caller` should also respect the psABI.

3.1.6. Platform/Framework Hooks

Hooks implementation will be the responsibility of the platform since only the platform knows how

to implement them.
RT-Engine design may be dependent on platform resources (e.g. “enter critical” section) or may be able to leverage platform features to increase performance of the engine.
For those the engine will need to expose API hooks to be provided by the platform/framework.

There are several types of hooks that need to be standardized so they can be used in any implementation:

3.1.6.1. Load Function Hook

A hook triggered by the RT-Engine to request the load of a group.

The API will need to provide information which is understood by the engine and the user,
AKA Overlay Static table (*Linker section: Overlay Static Table*)

Example:

- Source: group location/referenced from the *‘Overlay Static Table’*
- Size of group
- Destination to load

3.1.6.2. Error-Hook

On encountering an error, the RT-Engine will call the Error-Hook. Error hook is **fatal** the system can not recover from it.

3.1.6.3. Instrumentation interface hooks

Instrumentation is needed for analysis, which can be used to improve the performance of overlay function calls. For example: user can catch a sequence of overlay-function-calls, from the instrumentation, and according to the result he can encapsulate the functions to a specific group.

Table 1. Instrumentation interface hooks

	Instrumentation name	Description
1.	Invoke callee + Load	Load overlay function and invoke it
2.	Invoke caller (return) + load	When returning to an overlay function, and re-loading of the ‘caller’ is needed
3.	Invoke callee + No load	The callee function is already loaded, we just need to invoke it
4.	Invoke caller (return) + No load	When returning from an overlay function and re-loading of the ‘caller’ is needed

3.1.6.4. RTOS hooks and porting

On RTOS based system, there are two needed hooks: 1) Critical section hooks and 2) Porting

Critical section hooks: The RT-Engine will provide hooks to protect its critical sections. Those hooks will be implemented by the application based on the RTOS selection. e.g. application can provide mutex, semaphores or disable/enable interrupt logic

Porting: RTOS may need some overlay porting; in these cases, the engine implementation should encapsulate the porting as much as possible. e.g., context switch notification to the overlay RT-Engine

3.1.7. RTOS

The RT-Engine should support a system bare metal design and/or RTOS system design.

1. The implementation with/without RTOS should be a compile-time option.
2. If RTOS is supported, the RT-Engine should be thread-safe and not block other threads due to overlay operations.
3. Blocking can be acceptable for short critical sections and only with inherent operations (e.g. mutex).
4. The RT-Engine should be agnostic to any specific RTOS, therefore hooks should be provided *(RTOS hooks)*.
5. Load operations should lock the designated memory region in the heap, to prevent a case where a higher priority task will take the region from the current running task.

3.2. Toolchain

The toolchain needs to be integrated with the overlay standard to support the usage of overlays. The compiler, linker and debugger all need to support the overlay mechanism in order for the user to use overlay functions and data and debug them. The following are the module-requirements per tool.

3.2.1. Compiler

The main compiler demands are related to generating a sequence code to enter the RT-Engine whenever the running code references an overlay symbol, which can be data usage or function call/return.

1. The compiler needs to generate code for any related overlay usage, the sequence will lead to entering to the RT-Engine which then manages the process of loading, evicting, etc…
2. The user will need to add a designated attribute to its target overlay function or data to make the compiler emit the designated sequence for example: "__*attribute*__ (overlaycall)" or "__*attribute*__ (overlaydata)"
3. Types of related overlay use cases:
    a. Direct call – just calling to the overlay function
    b. Indirect call – call is via function pointer

c. RO Data – read-only data which is marked as overlay should be referenced with the same sequence to enter the RT-Engine so that the data can be loaded as necessary.

4. We need to reserve N core registers. To be used only for the RT-Engine. Those registers will have a special purpose understood by the compiler, linker, debugger, and RT-Engine. Moreover, those registers form a *RT-Eng-Debugger-handshake* between compiler, RT code, and debugger. Any library linked with an application that uses overlay scheme must be compiled without using the reserved registers. Per RISCV psABI we should reserve **N *temp registers*** (x28-x31,x5-x7) On new eABI we should reserve **N *saved registers*** (preferred last ones x16-x31)

Table 2. Reserved N core registers

Register	Designation
Xa	Holds the RT-Engine Entry point address
Xb	Holds the overlay descriptor/token
Xc	RT-Engine managing a pool of stack frames, the register will hold the pointer to this stack
Xd	Holds the stack register for the RT-Engine
Xe	**Only on RTOS support**: Holds RT-Engine dedicated stack-pointer, per task/thread.

5. The compiler should pass a descriptor/token to the RT-Engine via an 'entry' sequence.
The descriptor will be materialized at link time.
6. Related debug information should be aligned with the compiler overlay scheme.

3.2.2. Linker

1. Overlay symbols cannot be referenced by a memory address, since they are not part of the physical memory. Therefore we should have a descriptor/token to describe the overlay symbol (e.g. specifying the group to which it belongs and its offset within the group).
2. The linker shall create an overlay section for each overlay symbol that appears in an object file (as a result of attributes added by the user in the source code).
3. Each overlay symbol is assigned to one or more **Groups** at link time, as the linker has full visibility of all overlay symbols.
4. The linker shall have the ability to encapsulate functions and read-only data into overlay groups.
5. There should be an ***"overlay area"*** that holds all of the groups in the program. This area is not for execution, it is the area from which the RT-Engine will load overlay groups and it is also for the linker to treat overlay functions as regular functions (for address allocation, optimization

etc…) and debugging information is associated with the contents of this area.

6. Multi-group

   The linker should deal with overlay symbols which can be resident in more then one group:

   a. An overlay function can be resident in more than one group.

   b. Overlay data can be resident in more than one group.

7. Overlay Static Table

   a. The linker shall create a group-offset-table to hold all the overlay group offsets. Each entry index in the table represents an overlay group ID. Each entry contents represent the zero base offset to the group.

   b. Overlay group IDs are numerical.

   c. This table can be read at runtime (e.g. by the RT-Engine, debugger or another utility) to provide a mapping to locate an overlay group.

   d. This table shall provide sufficient information for the RT-Engine, debugger or other utilities to find the requested group within the ***"overlay area"*** (for example, so that the FW can locate and load a group).

8. Overlay group size ranges from 512B – 4K for both functions and data.

NOTE	This table is targeted to be a spec between the running code and the low level driver for loading the overlay function (per group). Since the table is part of the code, the developer can manage it and allocate a placeholder for the overlay groups/functions in the storage for example (storage refers to any SW I/F that can fetch the code).

3.2.2.1. Linker flags

The linker will get all the necessary data for overlay symbols from: object files, the linker script and linker flags.

i. **Input file**

   An external file holds "group numbers" per function name (this is for manual grouping). Appendix to file format *(Appendix A - Grouping file syntax)*. Without providing this file, the linker will generate a group per function

ii. **Max / Min size of overlay group**

   For the linker to be aware of the selected group size, the user should specify the max/min size of an overlay group (512, 4096, etc …)

3.2.3. Debugger

Since our goal is to provide a comfortable experience for the SW developer we need support for key

debugging features (such as breakpoints and backtracing) on an overlay system where overlay functions and data may be mapped or unmapped (loaded/unloaded).

1. The debugger should give the overlay functions the same debugging capabilities as a non-overlay function (e.g. step, step instruction, skip, backtracing etc…)
2. *RT-Eng-Debugger-handshake:* The debugger and the RT-Engine will communicate during run-time. The information passed from the RT-Engine to the debugger will contain the status of the loaded/unloaded (mapped/unmapped) overlay groups.
3. Overlay RT-Engine awareness:
    a. For backtracing, the debugger should be able to unwind the stack with awareness of calls/returns through the RT-Engine.
    b. To give a comfortable debugging experience we should have an option to “skip” through the RT-Engine when doing a step on a function call or return. E.g. if we step at call to function myOverlayFoo(), the debugger should skip through the RT-Engine and the next PC we see will be the beginning of myOverlayFoo() and not within the RT-Engine. Similarly, if we step at a function return, the debugger should skip through the RT-Engine and the next PC we see will be at the return address in the caller.
    c. There should also be an option to disable this “skip” functionality to allow debugging of the RT-Engine.
4. The RT-Engine will have three anchors in the source code for debugger-engine synchronization. With those anchors, the debugger will be able to sync with RT-Engine logic-flow. Those anchors manifest by symbols and break-points: entering, exiting, data-base-sync-point.

    Table 3. RT-Engine and Debugger sync anchors

Sync point	**Description**
Enter RT-Engine	The entry point to the RT-Engine
Exit RT-Engine	The exit point from the RT-Engine
Data-base-sync-point	The sync point on which the RT-Engine refresh the loaded (mapping update) groups

5. The debugger will be agnostic to the existence of a RTOS, this means a context switch can happen during an overlay operation and the debugger should hold a valid sequence.
6. Changes in the debugger should be generic in such a way that all related *“RT-Eng-Debugger-handshake”* will be in an external file to hook into the debugger.
7. We shall have debug information for overlay functions and overlay data. That information should be symmetric if a function is placed in several groups (***multi group***).

4. Appendix A - Grouping file syntax

The linker can receive an input file to give it details about assignments of groups to functions. e.g. myFunction() should be in group 1. This file should be in comma-separate-value syntax (CSV), as described:

- Each new line represents a function
- First column holds a function name
- Each next column holds a group number to assign the function

Table 4. Example for Grouping file CSV

Function name	Group number	Group number	Group ...
OvlFuncA	1		
OvlFuncB	2	7	

```
* OvlFuncA is to be assigned to group 1
* OvlFuncB is to be assigned to group 2 and to group 7
```

[1] Hook implementation will be the responsibility of the platform since only the platform knows how to implement them. + Please refer to section ***3.1.6 Platform/Framework Hooks***