# Software Overlay for RISCV - HLD

Version 0.4-draft-20210322

## Revision History

| *Revision* | *Date* | *Contents* | *Author(s)* |
|---|---|---|---|
| 0.1 | Jan 20,2021 | Initial revision / Draft a | Ronen Haen<br>Ofer Shinaar<br>Craig Blackmore |
| 0.2 | Feb 03,2021 | Toochain compiler | Craig Blackmore |
| 0.3 | Feb 16,2021 | Toochain linker and debugger | Craig Blackmore |
| 0.4 | Feb 16,2021 | pdf fixups | Ofer Shinaar |

### List of Figures

### List of Tables

### Reference Documents

| Item # | Document | Revision Used | Comment |
|---|---|---|---|
| 1 | riscv-overlay-software-standard-draft.adoc | 0.2-draft-20201230 | N/A |
| 2 | | | |

## 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 |

# Chapter 1. Overview

Some systems (mostly embedded systems) have limited memory resources, and as a result, the total code footprint is bigger than the available memory. The concept of arranging code in ‘code overlays’ is quite old but still valid these days to resolve the code size issue. The following document specifies the requirement and design of an overlay manager engine for RISC-V.

# Chapter 2. High-Level Design

## 2.1. Block Diagram

The following figure describes a general firmware block diagram with the RT-Engine:

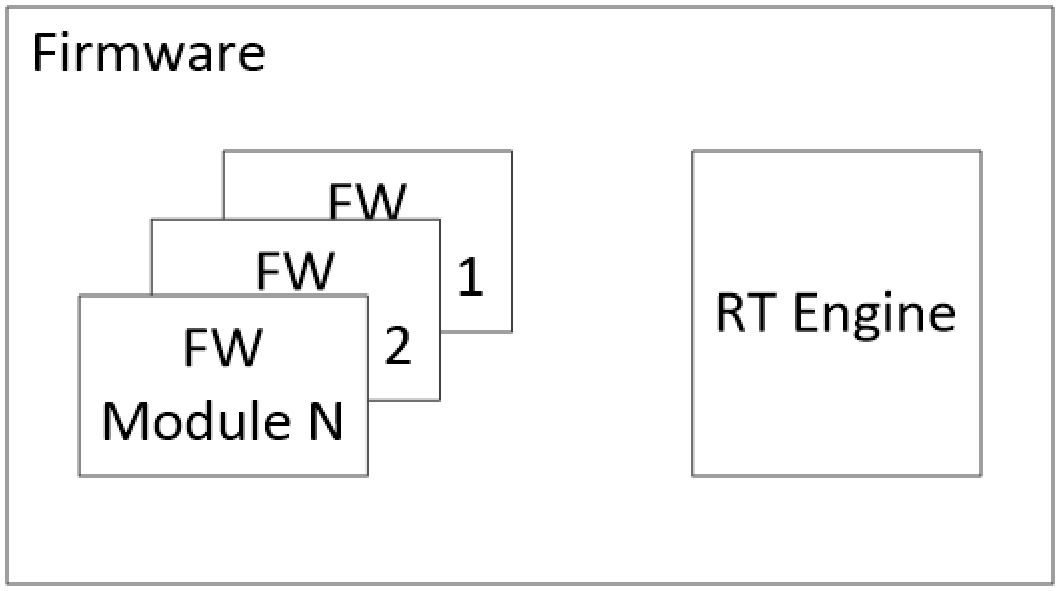


*Figure 1. Generic Firmware Block Diagram*

**Firmware** – this is the software being executed by the RISC-V core. It may contain several firmware modules that interact with each other or with different hardware components through firmware drivers. The firmware code can reside/execute in different memories, e.g., ROM, RAM, FLASH, etc.

**FW Module** – Firmware module that responsible for a specific operation in the program. One Firmware module may reside/execute in a different memory region than other firmware modules.

**RT-Engine** – this is the module responsible for managing the run time code load and execution. It is compiled and linked as all other firmware modules, and as such, it may also reside/execute in different memories. Any existing firmware module wishing to invoke a function defined as an overlay function will indirectly use the RT-Engine to dynamically load the function (if not already loaded) and invoke (call) it

## 2.2. Overlay functions

When developing firmware code for systems with memory constraints, the engineers will program the code to define which function is designated to be an overlay function. All marked overlay functions are gathered into overlay groups with a size range of 512B-4K each. An overlay group may contain one or more overlay functions, and it is the responsibility of the toolchain to create the overlay groups encapsulating overlay functions.

### 2.2.1. Overlay group structure

An overlay group size ranges from 512B – 4K and may contain several functions. Since an overlay group's boundary is always 512B, the group will be padded with the Overlay `Group ID` up to the upper 512B boundary. The structure of an overlay group is:

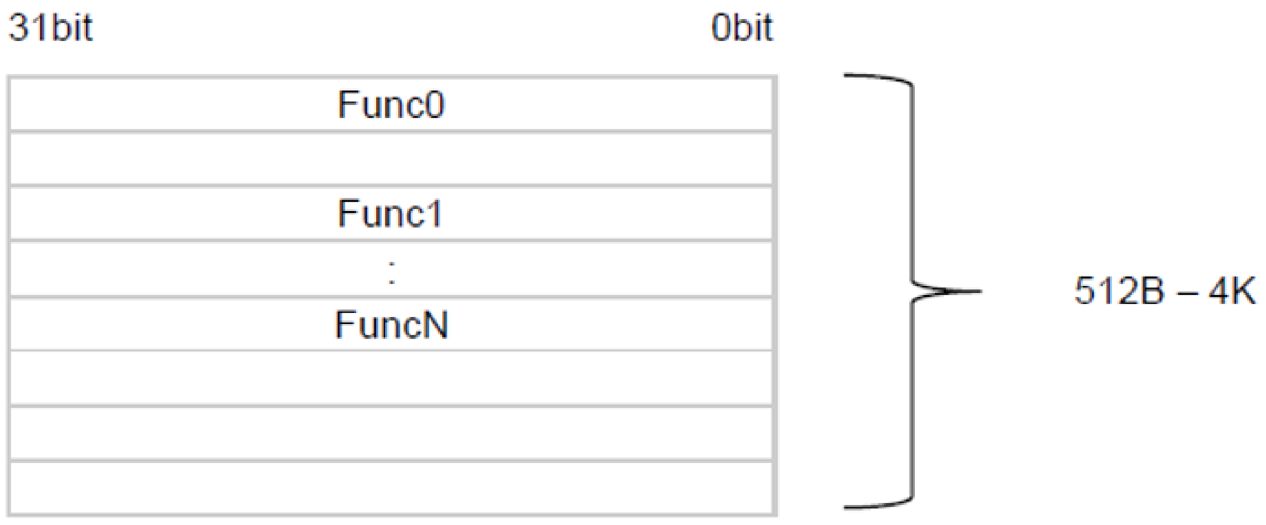


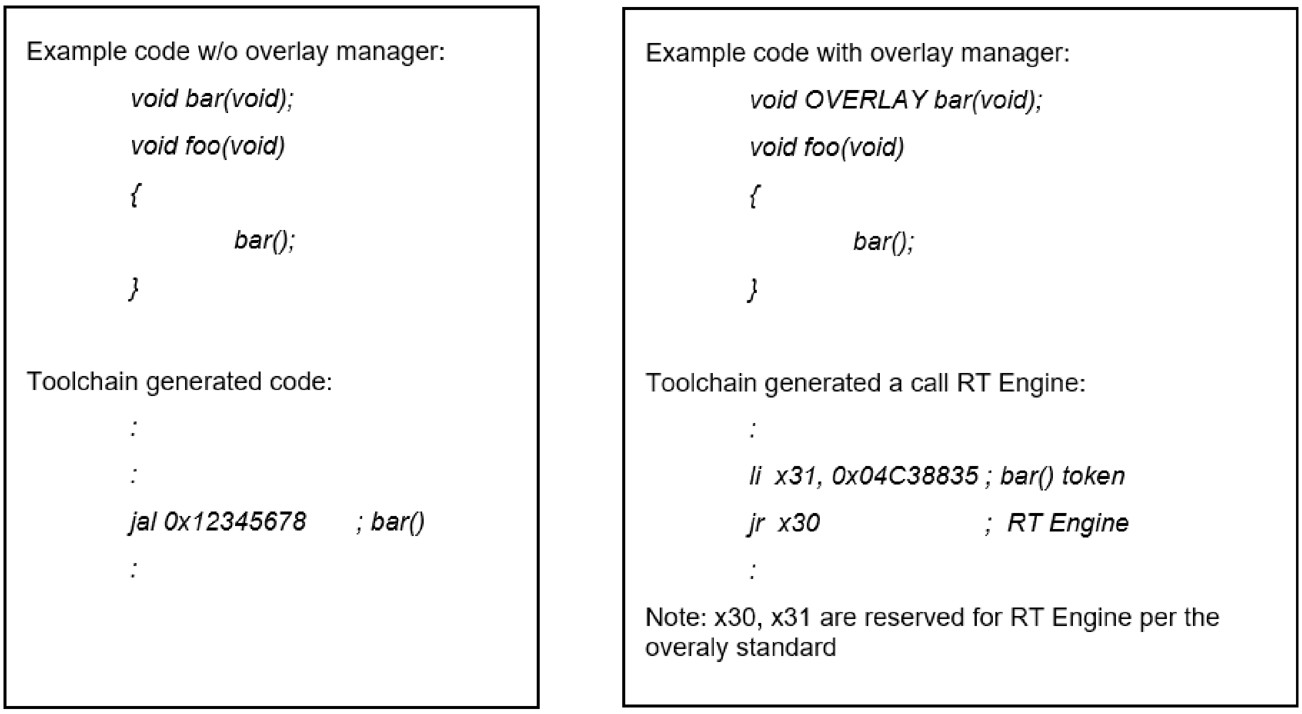
*Figure 2. Overlay group structure*

## 2.3. Overlay function call

On regular operation, when a given function foo() performs a call to function bar(), the toolchain generates a core-specific 'jump' instruction code and resolving the jump 'address'. In overlay design, if bar() function is defined as an overlay function, the compiler can generate a 'jump' instruction, but the linker will not be able to resolve the symbol's address since the 'jump' is not referring a fixed address in memory.

### 2.3.1. Implicit RT Engine invocation

Since the linker can't resolve the actual address of the overlay function bar(), and it does know the address of the RT Engine entry point, the compiler shall plant a 'jump' instruction to the RT Engine entry point instead of a 'jump' to bar(). To distinguish which overlay function is to be loaded and invoked, the linker will use an address token defining the bar() overlay function instead of the actual bar() address. Sharing a token will allow the RT Engine to prepare (load/invoke) the correct overlay group in memory along with the bar() function offset within the overlay group.


Example code w/o overlay manager:

```
        void bar(void);
        void foo(void)
        {
                bar();
        }
```

Toolchain generated code:

```
        :
        :
        jal 0x12345678      ; bar()
        :
```

Example code with overlay manager:

```
        void OVERLAY bar(void);
        void foo(void)
        {
                bar();
        }
```

Toolchain generated a call RT Engine:

```
        :
        li  x31, 0x04C38835 ; bar() token
        jr  x30                 ;  RT Engine
        :
```

Note: x30, x31 are reserved for RT Engine per the overaly standard

*Figure 3. Overlay operation example*

### 2.3.2. Implicit RT Engine invocation for a non-overlay function

When a function foo() is declared as an overlay function, and it is calling a non-overlay function bar(), there is a chance that when returning from bar(), foo() will already be evicted. That could be if additional overlay functions were loaded due to calling bar() or in another scenario, an OS context switch occurred, and overlay function calls were done from that context. Returning to an “already evicted” caller means that all non-overlay function calls that are made from within an overlay function must be done through the RT Engine. The toolchain replace the call to bar() with a call to the RT Engine and set the token value to point to bar() address. When the RT Engine is invoked, it will check if the token is a real token or an actual address; in this case, an actual address the RT Engine will directly jump to that address. When bar() completes, it will return to RT Engine, which will load foo() if not loaded, and return to it.

### 2.3.3. Address Token

An address token is an overlay function descriptor providing all the needed information for the RT Engine to load and invoke an overlay function. A regular address is always an even number. Therefore, to differentiate a token address from a standard address, the least significant bit of the address token shall be set to 1 (odd).

#### 2.3.3.1. Overlay address token structure

The overlay address token is a 32bit value defining a specific overlay function as follows –

*Table 1. Overlay token structure*

| 31 | 29 | 28 | 27 | 17 | 16 |
|---|---|---|---|---|---|

| Multi-group token | Heap ID | Reserved | Thunk call | Function offset | Overlay group ID ⇒ |
|---|---|---|---|---|---|
| | | | | **1** | **0** |
| | | | | ⇐ Overlay group ID | Overlay address token |

| | | |
|---|---|---|
| B31 | Multi-group token | B31 [1] – B16:1 specify a multi-group overlay ID<br>B31 [0] – B16:1 specify a regular overlay group ID |
| B30:29 | Heap ID | Heap region identification |
| B28 | Reserved | |
| B27 | Thunk call | Calling an overlay function through a function pointer |
| B26:17 | Function offset | Value defining the function offset from the beginning of the group; value expressed in 4 bytes granularity |
| B16:1 | Overlay group ID | Overlay group ID: regular overlay group ID (function resides in) or multi-group overlay ID (ID to a list of groups the function resides in) |
| B0 | Overlay address token | Overlay token indication:<br>B0 [1] – B31:0 define an overlay token address<br>B0 [0] – B31:0 define a memory address |

## 2.4. RT Engine Management Tables

The following management tables are required for the RT Engine operation:

### 2.4.1. Overlay offset table

This table is an array of overlay offsets prepared by the linker. A table index represents an overlay group ID; a table entry holds a specific overlay group's offset. For example, entry #1 defines the location offset of overlay-group ID #1. The offset is relative to the beginning of all existing overlays (Per overlay standard - "overlay area" ). There can be a case where several Overlay Offset Tables exist, and each such table refers to a different overlay heap location (Heap ID Table 1 – Overlay token). In run-time, the RT Engine shall get the overlay group ID from the address token and use it with this table to determine the overlay offset to be loaded. The overlay offset granularity is expressed in 512B units.

### 2.4.1.1. Overlay offset table structure

An entry in the overlay offset table is defined as follows –

*Table 2. Overlay offset table structure*

| **15 .. 0** | | |
|---|---|---|
| Group offset | | |
| B15:0 | Group offset | Offset from the begging of the overlay section; value expressed in 512B granularity |

Table size (number of entries) shall be equal to the number of overlay groups plus one unused entry [1]; a single table entry represents each overlay group. The group offset value is accumulative, and the overlay group size is calculated by subtracting the overlay offset of the x+1 entry with the overlay offset of x entry.e.g., in the following table, overlay group ID 2 is in offset 5632B (11 * 512B) from the beginning of the overlay section, and its size is 1024B

*Table 3. Overlay offset table example*

| Entry | Overlay offset table [size] |
|---|---|
| 0 | 0[3*512] |
| 1 | 3[8*512] |
| 2 | 11[2*512] |
| 3 | 13[x*512] |
| : | |
| n-1 | |
| n | |

The ‘Overlay Group ID’ field of the Overlay Address Token is used to access an entry in the table.

## 2.4.2. Overlay multi-group table

Any given function can be defined as a multi-group function meaning it will reside in more than one overlay group. The toolchain prepares this table; the table index defines the multi-group identifier; table entries are sub-lists of address tokens specifying all multi-group overlay functions; each sub-list defines the overlay groups of one function. In run-time, RT Engine shall use the input address token to determine if the token describes a multi-group token. If so, the overlay ID token field specifies the first index of the sub list in the overlay mulit-group table; RT Engine will iterate through the sub list and check if one of the address tokes in the sublist is already loaded. If none of them are loaded, the first entry of the sub list is used to specifies the default address token.

### 2.4.2.1. Overlay multi-group table structure

An entry in the overlay multi-group table is an Overlay Address Token (see *Tabel 1 - Overlay token structure*)). The table size depends on the number of multi-groups and the number of occurrences per function. A zeroed Address Token separates each mutli-group token list. e.g., if there is only one multi-group and that multi-group function appears in 3 overlay groups, it will mean we’ll have a single multi-group ID (ID 0). That Overlay multi-group table shall contain 4 entries (the last entry will be zero). In the following example, we see that there are 4 multi-groups with the IDs – 0, 3, 8, 11, and each multi-group contains 2, 4, 2, and 3 occurrences of each function, respectively.

*Table 4. Overlay Multi-Group example*

| Entry | Overlay Multi-Group table |
|---|---|
| 0 | Some Address Token footnote:declaimer[Each first entry is the default entry in case none of the Address Tokens of a specific group is loaded] |
| | Some Address Token |
| | 0 |
| 3 | Some Address Token footnote:declaimer[] |
| | Some Address Token |
| | Some Address Token |
| | Some Address Token |
| | 0 |
| 8 | Some Address Token footnote:declaimer[] |
| | Some Address Token |
| | 0 |
| 11 | Some Address Token footnote:declaimer[] |
| | Some Address Token |
| | Some Address Token |
| | 0 |

When RT-Engine received an Overlay Address token with the “Multi-group token” field set, it will extract the ‘Overlay Group ID’ field of the Overlay Address Token so it can access the sub-token list in the Overlay Multi-group table.

## 2.5. Reserved registers

As described in riscv-overlay-software-standard-draft.adoc the RT-Engine shall have 4 RV dedicated resisters solely to it. It means the compiler won't use those registers on the register-allocation stage. The following registers are being used RT-Engine:

*Table 5. Reserved registers*

| Register/ABI name | Register Name | Reserved for RT-Enginee |
|---|---|---|
| x31 (t6) | Holds the RT-Engine Entry point address | Yes |
| x30 (t5) | Holds the overlay descriptor/token | Yes |
| x29 (t4) | RT-Engine managing a pool of stack frames, the register will hold the pointer to this stack | Yes |
| x28 (t3) | Holds the stack register for the RT-Engine | Yes |
| x4 (tp) | Only on RTOS support: Holds RT-Engine dedicated stack-pointer, per task/thread | No |

**NOTE** x4 is not reserved. Currently, X4 is not being used by the compilers (GCC 10/LLVM 12). X4 holds the thread pointer on OS system. If compiler/RTOS uses this register in the future, we will need to allocate a different register.

### 2.5.1. RT Engine Entry Point Address register (x31)

The RT-Engine sets this register during firmware initialization time. It shall be set to the address of the RT-Engine entry point function. There are two cases where the compiler uses this register: i) When it encounters a call to an overlay function. ii) when it encounters a non-overlay function call from within an overlay function.In both cases, the compiler shall replace the call to overlay function with a 'JR' instruction where x31 is the jump register (rs1). This register content is fixed, and therefore there is no need to save/restore its value in case of context switch or interrupt handling.

### 2.5.2. RT Engine overlay descriptor/token (x30)

This register is read by the RT-Engine when it is called for determining which function is to be invoked. There are 2 cases where the toolchain sets this register: i) When an overlay function is invoked, the compiler/linker needs to set this register with the corresponding Overlay Address Token. ii) When a non-overlay function is called from within an overlay function, the compiler/linker needs to set this register to hold the non-overlay function address. Setting this register shall be done before the added 'jump' (to RT-Engine entry point address) instruction.

### 2.5.3. RT Engine Stack Frames Pool register (x29)

RT-Engine uses a designated stack to keep track of nesting function calls. This register holds the next available stack element, and each element holds token, return address, and offset to the previous element.

On compilation time, the user needs to define the stack max depth.

Example:

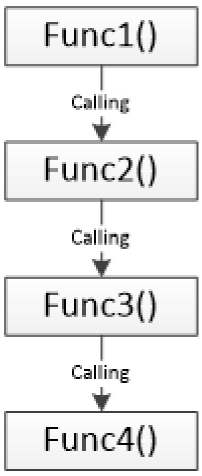


*Figure 4. Example of function deepest nesting*

If this is our deepest nesting calling in our application, we need to define the stack frame pool deep = 4 (+null element, end-of-list).

On RTOS based application, the user should take into consideration that this pool is shared between all tasks/threads. Therefore on a RTOS based application, the max nested calling depth that the user should take need to bes: **[sum of max nested calling depth per task]**.
That is the worst-case scenario.

### 2.5.4. RT Engine Stack register (x28)

This register holds the address of the RT-Engine designated stack of the main process. Each entry holds an allocated element address from the “Stack Frames Pool (x29)”.

On RTOS based application, this register is saved on a context switch time since each task/thread can have its own nesting function calling on the joint pull list (x29)

### 2.5.5. RT Engine task/thread dedicated stack-pointer (x4/tp)

This register is being used only on RTOS based application. Since the RT-Engine is defined to be none-blocking (as much as it can per riscv-overlay-software-standard-draft).

The logic flow of the RT-Engine can diverge if a context switch happened in the middle of its operation. Meaning we may not return to the PC we left when the context switch occurs. Due to this fact, we need to save all related registers to a stack. This stack is part of the Task/Thread stack, and x4 is pointing to it.

## 2.6. RT Engine

### 2.6.1. High-level flow

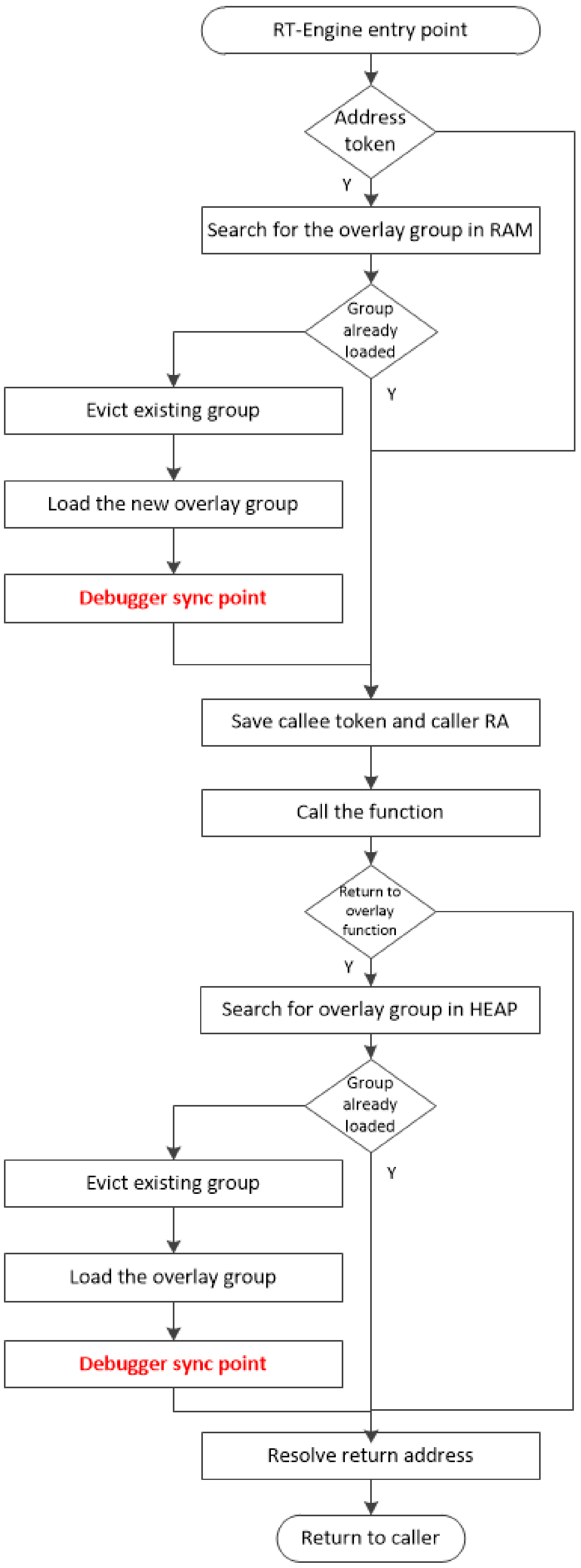


*Figure 5. RT-Engine High level flow*

#### 2.6.1.1. Address token query

The RT-Engine needs to understand what is required to do: i) Call an overlay function or ii) Call a non-overlay function. This information is provided by reading the Address Token register. If the function is an overlay function, bit0 is set, and the Address Token register holds the overlay function descriptor. The RT-Engine then uses this token to load and/or call the designated overlay function. If bit0 is cleared, the Address Token register holds a physical memory address of a non-overlay function, and the engine shall directly call this function (no need for other handlings).

#### 2.6.1.2. Search for an already loaded overlay group

When the Address Token register holds an overlay function descriptor/token, we first need to check whether the said function is already loaded in memory to avoid redundant load. The RT-Engine extract the Overlay Group ID field (bits[1:16]) from the Address Token register and search for it in the internal database for “loaded overlay groups”.

In some cases, the search is done twice; the first one is when a new call to an overlay function is performed, and the second search is done when returning to RT-Engine, and the return destination is also an overlay function.

#### 2.6.1.3. Eviction

When the required overlay group is not loaded in memory, and the overlay heap is entirely occupied, the RT-Engine needs to decide which overlay heap entry can be evicted and replaced with the new overlay group. The eviction needs to consider which group is less “hot” than others and the required heap size for the new entry. The eviction algorithm is LRU (least recently used).
The eviction decision may also accrue twice: i) When a new overlay function is called, ii) and after returning to the RT-Engine while the return destination is an overlay function,which was already evicted (can occur due to nested overlay function calls).

#### 2.6.1.4. Load

The load operation is initiated by RT-Engine and is implemented by the hosting application. The engine does not care how the load is performed; it can be done from different sources, e.g., volatile memory, non-volatile memory, or communication interface. The call to the load routine is blocking and does not return until the load is completed.

#### 2.6.1.5. Saving the callee token and caller return address

When the overlay function resides in memory, the engine must save the callee token and the caller return address before it is being invoked. When returning from callee to the caller, we first return to the RT-Engine to load the caller if it was evicted. Due to this paradigm, we need to save the caller’s return address, and it’s token, so the RT-Engine can load it if needed.

## 2.7. Toolchain

This section describes the high level design for the compiler, linker and debugger required to support overlays.

### 2.7.1. Compiler

The compiler support will be implemeted in Clang/LLVM.

#### 2.7.1.1. Flags

The `-moverlay` flag enables overlay support in the compiler. Specifically, this flag:

- Enables an overlay calling convention `RISCV_OverlayCall`.
- Reserves the registers required by the RT-Engine.
- Enables the use of attributes `overlaycall` and `overlaydata`.

#### 2.7.1.2. Relocations

In order to support linker token generation, custom relocations are needed to represent the token version of symbol addresses. These are currently placed in the custom extension space, as defined in the RISC-V psABI document, but will require moving after standardization.

The new relocations are as follows:

| Enum | ELF Reloc Type | Description |
|---|---|---|
| 220 | R_RISCV_OVL_HI20 | U-type (upper 20-bit) token value |
| 221 | R_RISCV_OVL_LO12_I | I-type (lower 12-bit) token value |
| 222 | R_RISCV_OVL32 | 32-bit overlay token value |
| 223 | R_RISCV_OVLPLT_HI20 | U-type (upper 20-bit) overlay plt address |
| 224 | R_RISCV_OVLPLT_LO12_I | I-type (lower 12-bit) overlay plt address |
| 225 | R_RISCV_OVLPLT32 | 32-bit overlay plt entry address |

#### 2.7.1.3. Calling convention

A new calling convention is required for calls involving overlay functions. This will be called `RISCV_OverlayCall`. Non-overlay functions follow a calling convention from the RISCV psABI. Calls between overlay and non-overlay functions must be compatible with the underlying calling convention for non-overlay functions - to ensure this compatibility:

- The RT-Engine is responsible for passing arguments to the callee, return values from the callee and preserving registers as required by the underlying ABI.
- The compiler must not apply optimizations that would break compatibility between overlay

function and non-overlay function calling conventions.

The compiler places overlay functions or data in their own sections so that they are self-contained and the linker can sort and group them. An overlay function or data with symbol `X` will be placed in section `.ovlinput.X`. There is no need to distinguish between functions and data in the section name as they are treated the same by the linker.

Any call in which the caller or callee is an overlay function must be invoked via the RT engine.

**Direct call**

For a call to an overlay function (i.e. callee has attribute `overlaycall`), the compiler must load the callee token into `t5` and then jump and link to the RT-Engine entry point via `t6`.

For example, for the following code:

```
int globalCount;

void __attribute__((overlaycall)) f1() {
  globalCount += 3;
}

int main() {
  f1();
  return 0;
}
```

`main` compiles and assembles to:

```
Disassembly of section .text:

00000000 <main>:
   0:   1141                    addi    sp,sp,-16
   2:   c606                    sw      ra,12(sp)
   4:   00000f37                lui     t5,0x0
                        4: R_RISCV_OVL_HI20     f1
   8:   000f0f13                mv      t5,t5
                        8: R_RISCV_OVL_LO12_I   f1
   c:   000f80e7                jalr    t6
  10:   4501                    li      a0,0
  12:   40b2                    lw      ra,12(sp)
  14:   0141                    addi    sp,sp,16
  16:   8082                    ret
```

and after linking:

```
204000e4 <main>:
204000e4:       1141                    addi    sp,sp,-16
204000e6:       c606                    sw      ra,12(sp)
204000e8:       00000f37                lui     t5,0x0
204000ec:       003f0f13                addi    t5,t5,3 # 3
204000f0:       000f80e7                jalr    t6
204000f4:       4501                    li      a0,0
204000f6:       40b2                    lw      ra,12(sp)
204000f8:       0141                    addi    sp,sp,16
204000fa:       8082                    ret
```

**Indirect call**

For an indirect call to an overlay function (i.e. callee has attribute `overlaycall`), the function pointer will contain the address of an entry in the overlay procedure linkage table (`.ovlplt`). A call via this function pointer will jump to the entry in the `.ovlplt` which will then load the overlay function token into `t5` and jump and link to the RT-Engine entry point via `t6`.

For example, for the following code:

```
int globalCount;

void __attribute__((overlaycall)) f2() {
  globalCount += 2;
}

void __attribute__((overlaycall)) (*fptr)();

int main() {
  fptr = f2;
  fptr();
  return 0;
}
```

`main` compiles and assembles to:

```
00000000 <main>:
   0:   1141                    c.addi  sp,-16
   2:   c606                    c.swsp  ra,12(sp)
   4:   00000537                lui     a0,0x0
                        4: R_RISCV_OVLPLT_HI20  f2
   8:   00050513                addi    a0,a0,0 # 0 <main>
                        8: R_RISCV_OVLPLT_LO12_I        f2
   c:   000005b7                lui     a1,0x0
                        c: R_RISCV_HI20 fptr
  10:   00a5a023                sw      a0,0(a1) # 0 <main>
                        10: R_RISCV_LO12_S      fptr
  14:   00000f37                lui     t5,0x0
                        14: R_RISCV_OVL_HI20    f2
  18:   000f0f13                addi    t5,t5,0 # 0 <main>
                        18: R_RISCV_OVL_LO12_I  f2
  1c:   000f80e7                jalr    ra,0(t6)
  20:   4501                    c.li    a0,0
  22:   40b2                    c.lwsp  ra,12(sp)
  24:   0141                    c.addi  sp,16
  26:   8082                    c.jr    ra
```

and after linking:

```
204000e4 <main>:
204000e4:       1141                    c.addi  sp,-16
204000e6:       c606                    c.swsp  ra,12(sp)
204000e8:       20400537                lui     a0,0x20400
204000ec:       34450513                addi    a0,a0,836 # 20400344
204000f0:       800005b7                lui     a1,0x80000
204000f4:       10a5a223                sw      a0,260(a1) # 80000104
204000f8:       00000f37                lui     t5,0x0
204000fc:       003f0f13                addi    t5,t5,3 # 3
20400100:       000f80e7                jalr    ra,0(t6)
20400104:       4501                    c.li    a0,0
20400106:       40b2                    c.lwsp  ra,12(sp)
20400108:       0141                    c.addi  sp,16
2040010a:       8082                    c.jr    ra
...
20400344 <.ovlplt>:
20400344:       08000f37                lui     t5,0x8000
20400348:       003f0f13                addi    t5,t5,3 # 8000003
2040034c:       000f8067                jalr    zero,0(t6)
```

**Return**

No special handling is required by the compiler.

#### 2.7.1.4. Overlay data

RO data can be marked as overlay with the `overlaydata` attribute, for example:

```
__attribute__((overlaydata)) const int foo;
```

Overlay data `foo` will be placed in `.ovlinput.foo`.

#### 2.7.1.5. Constraints

Static functions/data cannot be marked as `overlaycall`/`overlaydata` (this does not include class-static symbols), doing so will produce a compiler error.

The compiler will not inline overlay functions.

The compiler will not generate tail calls to or from overlay functions.

Overlay functions/data must be 4 byte aligned so that they can be addressed by overlay address tokens. The compiler will ensure this alignment.

Arithmetic cannot be done on overlay tokens, this will produce a compiler error.

### 2.7.2. Linker

The linker support will be implemented in GNU binutils.

The presence of `.ovlinput.*` sections will trigger the linker to enable overlay support. The presence of an overlay symbol in multiple groups will trigger multi-group support.

#### 2.7.2.1. Grouping

There are three ways in which an overlay symbol may be assigned to groups.

1. Manually, by providing a CSV grouping file:
   - `--grouping-file <filename>`.
2. By calling a grouping tool that populates a grouping file. Two flags control this:
   - `--grouping-tool <tool-cmd>` - command used to call the grouping tool.
   - `--grouping-tool-args <arg1>;<arg2>;…;<argN>` - arguments to be passed to the grouping tool. The required argument `--in-file <filename>` specifies the CSV file in which the linker should pass a list of symbols that require grouping to the grouping tool. The required argument `--out-file <filename>` specifies the CSV in which the grouping tool will output its groupings.
3. Linker autogrouping - the linker will put any overlay symbol that has not been assigned to a group into its own group.

#### 2.7.2.2. Sections

The linker will populate the following output sections:

*ovlgrps - contains each overlay group. Referred to as* `` `overlay area`` *in the*

requirements document. * The first group contains the overlay offset table followed by the multi-group table (if multi-groups are present). * Each input section `.ovlinput.*` for each overlay symbol is copied into the group(s) to which it has been assigned.

*ovlcache - the overlay heap into which overlay groups are loaded at runtime by*

the RT-Engine. Referred to as ` `heap area'' in the requirements document.

#### 2.7.2.3. Tables

*ovlplt - contains the overlay PLT.*

The linker will construct the overlay offset table and multi-group tables (if multi-groups are present).

#### 2.7.2.4. Tokens

For each overlay relocation, the linker will construct the required overlay token.

#### 2.7.2.5. Groups

Any symbol referred to by an overlay relocation must be assigned to one or more groups. Each group will be populated with the input sections for the symbols assigned to that group. Each group will be padded to the next overlay group page boundary, with the CRC32 value being placed in the last 32-bits of the group.

Minimum group size / overlay group page boundary is defined by symbol `OVERLAY_MIN_GROUP_SIZE`.

Maximum group size is defined by symbol `OVERLAY_MAX_GROUP_SIZE`.

#### 2.7.2.6. Overlay Procedure Linkage Table (PLT)

The overlay PLT contains an entry for calling each overlay function called via a function pointer. Since each entry contains three instructions, users should consider the code size overhead associated with indirect overlay calls (as well as the speed overhead from the indirection).

```
20400364 <.ovlplt>:
20400364:       08000f37                lui     t5,0x8000
20400368:       007f0f13                addi    t5,t5,7 # 8000007
2040036c:       000f8067                jr      t6
20400370:       08000f37                lui     t5,0x8000
20400374:       003f0f13                addi    t5,t5,3 # 8000003
20400378:       000f8067                jr      t6
2040037c:       08000f37                lui     t5,0x8000
20400380:       005f0f13                addi    t5,t5,5 # 8000005
20400384:       000f8067                jr      t6
```

#### 2.7.2.7. Relaxations

The linker is permitted to relax the materialization of overlay tokens, for example:

```
lui     t5, 0
addi    t5, t5, 3
```

could be relaxed to:

```
addi    t5, zero, 3
jalr    ra, 0(t6)
```

### 2.7.3. Debugger

The debugger support will be implemented in GDB.

#### 2.7.3.1. Awareness for overlays

When debugging a firmware with overlay functions, the end-user is not interested in the debug flow of RT-Engine (the implicit call to RT-Engine). Therefore when stepping in an overlay function, the debugger can automatically skip the RT-Engine (skip the search, evict, load, etc.) and set the breakpoint after entering the actual overlay function.

[1] Entry n does not represent an actual overlay; it exists to calculate the size of overlay ID n-1.