

ARM Trusted Firmwareの BL31を単体で使う！

@Vengineer
2016/10/10 (まだ途中)



Vengineer DEATH

無限ゲームのなか

いつものように、
よろしくお願いします。

@Vengineer に居ます

ARM Trusted Firmware

ARMv8では重要なソフトウェア

Githubにて、仕様およびソースコードが公開されている

<https://github.com/ARM-software/arm-trusted-firmware>

サポートしているSoC

- ・ARM FVP (シミュレーション・モデル)

<http://www.arm.com/ja/products/tools/models/foundation-model.php>

Foundation_Platform (Version 9.4, Build 9.4.59)

FVP_Base_AEMv8A-AEMv8A (V.7.0, Build 0.8.7004)

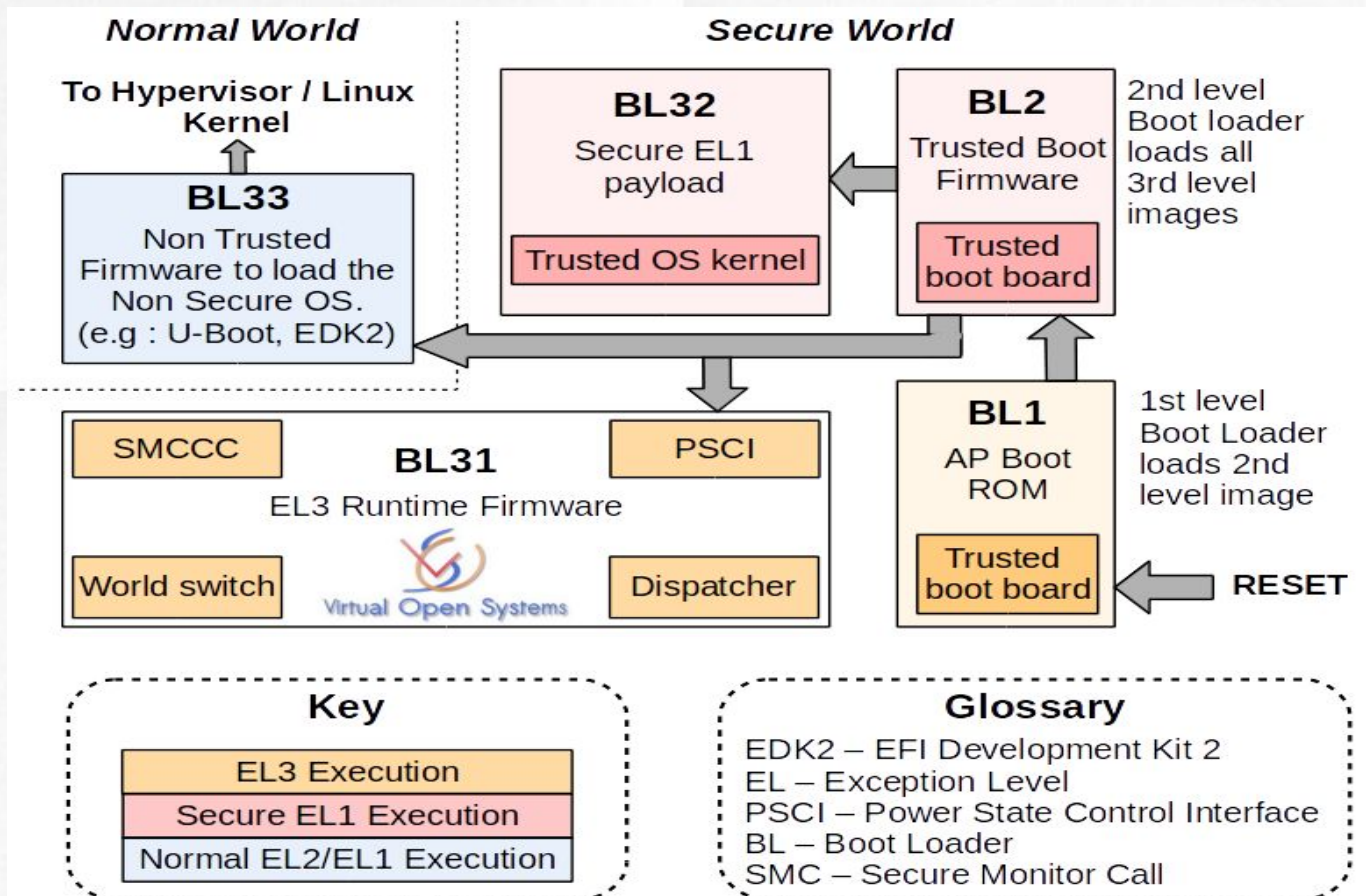
FVP_Base_Cortex-A57x4-A53x4 (同上)

FVP_Base_Cortex-A57x1-A53x1 (同上)

FVP_Base_Cortex-A57x2-A53x4 (同上)

- ・ARM Juno (A72x2 or A57x2 + A53x4)
- ・Mediatek MT6795 / MT8173
- ・NVIDIA Tegra-K1 (†132:DENVER) / Tegra-X1 (†210)
- ・RockChip RK3368 / RK3399
- ・Xilinx Zynq UltraScale+ MPSoC

ARM Trusted Firmwareの構造



各階層の簡単な説明

Secure World

- ・BL1 : AP Boot ROM (通常、非公開)
- ・BL2 : Trusted Boot Firmware
- ・BL31 : EL3 Runtime Firmware
- ・BL32 : Secure EL1 payload

Normal World

- ・BL33 : Non Trusted Firmware to load the non Secure OS (U-Boot, EDK2)

ドキュメント

Docsディレクトリに以下のものがある

`auth-framework.md`

`cpu-specific-build-macros.md`

`firmware-design.md` / `fimware-update.md`

`interrupt-framework-design.md`

`platform-migration-guide.md`

`porting-guide.md`

`psci-lib-integration-guide.md` / `psci-pd-tree.md`

`reset-design.md`

`rt-svc-writers-guide.md`

`trusted-board-boot.md`

`user-guide.md`

reset-design.md

ARM Trusted Firmware Reset Design

1. Introduction
2. General reset code flow
3. Programmable CPU reset address
4. Cold boot on a single CPU
5. Programmable CPU reset address, Cold boot on a single CPU
6. Using BL31 entrypoint as the reset address

RESET_TO_BL31

Makefile内の変数(RESET_TO_BL31)を1にすることで、BL31を単独に使うことができる

```
% make RESET_TO_BL31=1 bl31
```

ただし、CPUコアのRVBAR_EL3 (reset vector base address)に、BL31のジャンプ先アドレスを事前に設定しておく必要がある

RVBAR_EL3

Home > System Control > AArch64 register descriptions > Reset Vector Base Address, EL3

4.3.60 Reset Vector Base Address, EL3

Defines the address that execution starts from after reset when executing in the AArch64 state.

RESET_TO_BL31=1 な SoC

Xilinx Zynq UltraScale+ MPSoC

`plat/xilinx/zynqmp/platform.mk`

```
RESET_TO_BL31 := 1
```

```
....
```

```
ifneq (${RESET_TO_BL31},1)
```

```
$(error "Using BL31 as the reset vector is only one  
option supported on ZynqMP. Please set RESET_TO_BL31  
to 1.")
```

```
endif
```

Zynq UltraScale+ MPSoCの場合

BL1 : BootROM

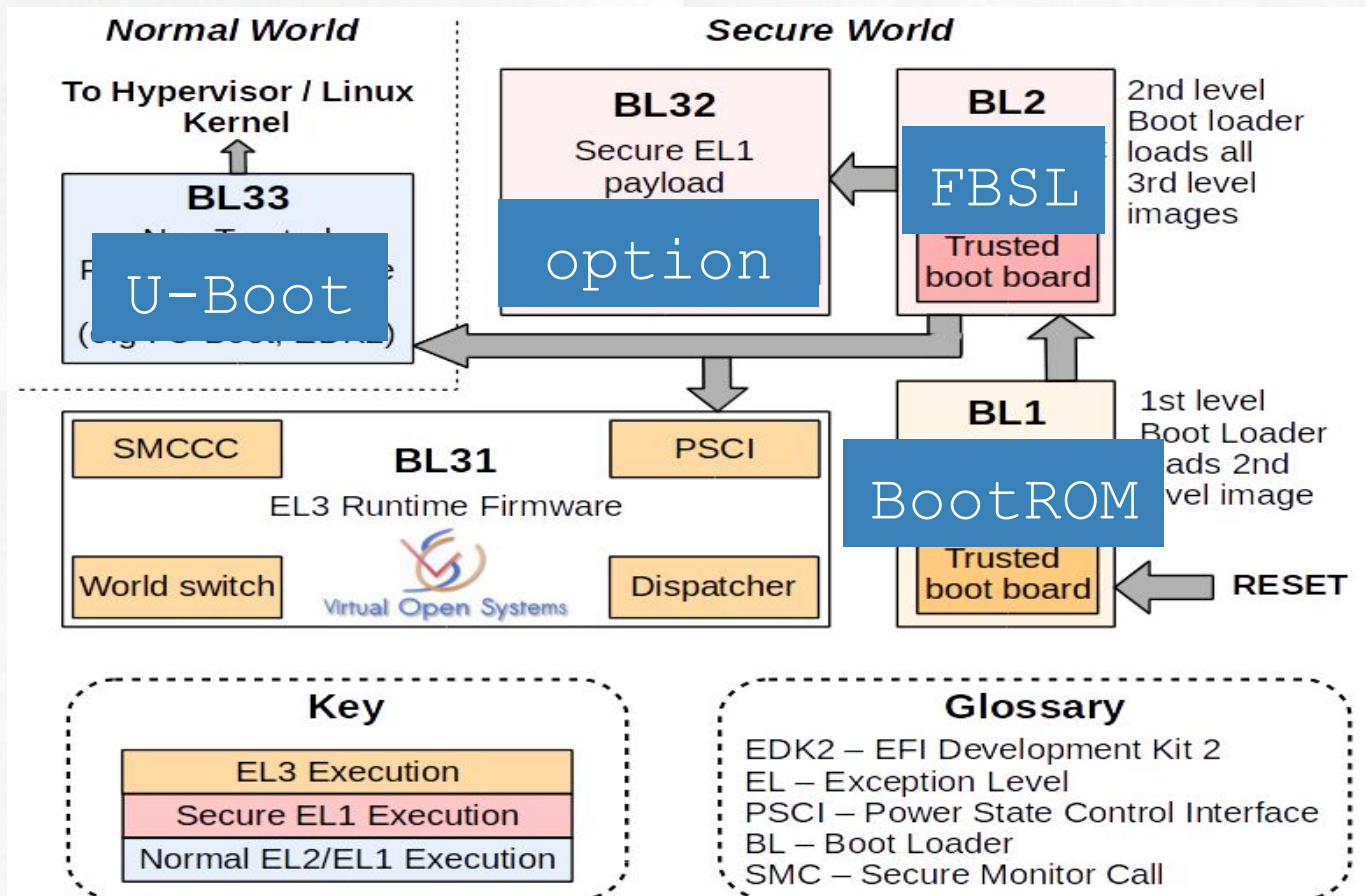
BL2 : FSBL (First Stage Boot Loader)

BL31 : ATF

BL32 : オプション (Secure Payload)

BL33 : U-Boot

Zynq UltraScale+ MPSoCの場合



BL31をビルドするには

```
% make ERROR_DEPRECATED=1 \  
  RESET_TO_BL31=1 \  
  CROSS_COMPILE=aarch64-none-elf- \  
  PLAT=zynqmp bl31
```

RESET_TO_BL31=1 にしている
クロスコンパイラは、aarch64-none-elf-
ターゲットは、bl31 のみ

FSBLからATFへのパラメータ渡し方

The FSBL populates a data structure with image information for the ATF.

The ATF uses that data to hand off to the loaded images. The address of the handoff data structure is passed in the `PMU_GLOBAL.GLOBAL_GEN_STORAGE6` register.

The register is free to be used by other software once the ATF is bringing up further firmware images.

ソースコード解析

plat/xilinx/zynqmp

porting-guide.md

BL31では、下記の関数が各プラットフォームとして必要

plat/xilinx/zynqmp/bl31_zynqmp_setup.c

bl31_early_platform_setup() [mandatory]

bl31_plat_arch_setup() [mandatory]

bl31_platform_setup() [mandatory]

bl31_plat_runtime_setup() [optional]

bl31_plat_get_next_image_ep_info() [mandatory]

plat/xilinx/zynqmp/aarch64/zynqmp_common.c

plat_get_syscnt_freq2() [mandatory]

(この関数については、ここでは解析していません)

bl31/bl31.ld.S

```
ENTRY(bl31_entrypoint)
```

...

リセット解除後、実行されるコードは、bl31_entrypoint

```
.text . : {  
    __TEXT_START__ = .;  
    *bl31_entrypoint.o(.text*)  
    *(.text*)  
    *(.vectors)  
    . = NEXT(4096);  
    __TEXT_END__ = .;  
} >RAM
```


処理の流れ

```
bl31_entrypoint
// 前処理
  ▪ 例外処理の設定 (runtime_exceptions)
  ▪ bl31_early_platform_setup
  ▪ bl31_plat_arch_setup
// メイン部
  ▪ bl31_main
    ▪ bl31_platform_setup
    ▪ bl31_plat_get_next_image_ep_info
    ▪ bl31_plat_runtime_setup
// EL3を抜け、BL32 or BL33 (U-Boot) へ
  ▪ el3_exit
```

bl31/aarch64/bl31_entrypoint.S

```
func bl31_entrypoint
// 各種前処理
el3_entrypoint_common \
    _set_endian=1 \
    _warm_boot_mailbox=!PROGRAMMABLE_RESET_ADDRESS
\
    _secondary_cold_boot=!COLD_BOOT_SINGLE_CPU \
    _init_memory=1 \
    _init_c_runtime=1 \
    _exception_vectors=runtime_exceptions
```

例外が発生すると、runtime_exceptionsが呼ばれる
(bl31/aarch64/runtime_exceptions.S)

bl31_entrypoint.Sの続き

```
// RESET_TO_BL31=1の場合は、  
// bl31_early_platform_setup関数の引数(x0/x1)を  
// 0でクリアする
```

```
mov    x0, 0  
mov    x1, 0
```

```
// Zynq UltraScale+ MPSoC用のセットアップ  
bl     bl31_early_platform_setup  
bl     bl31_plat_arch_setup
```

```
// BL31のmain関数を実行  
bl     bl31_main
```

bl31_early_platform_setup

```
void bl31_early_platform_setup(  
    bl31_params_t *from_bl2,  
    void *plat_params_from_bl2)  
{  
    // コンソールの設定  
    console_init(ZYNQMP_UART_BASE,  
                 zynqmp_get_uart_clk(),  
                 ZYNQMP_UART_BAUDRATE);  
    // Zynq UltraScale+ MPSoC固有の設定  
    zynqmp_config_setup();  
    // 引数が0 (NULL) であることをチェック  
    assert(from_bl2 == NULL);  
    assert(plat_params_from_bl2 == NULL);  
}
```

bl31_early_platform_setupの続き

// BL32イメージ情報の初期化

```
SET_PARAM_HEAD(&bl32_image_ep_info,  
               PARAM_EP, VERSION_1, 0);  
SET_SECURITY_STATE(bl32_image_ep_info.h.attr,  
                   SECURE);
```

// BL33イメージ情報の初期化

```
SET_PARAM_HEAD(&bl33_image_ep_info,  
               PARAM_EP, VERSION_1, 0);  
SET_SECURITY_STATE(bl33_image_ep_info.h.attr,  
                   NON_SECURE);
```


bl31_early_platform_setupの続き

```
if (zynqmp_get_bootmode() == ZYNQMP_BOOTMODE_JTAG (
{
    ...
} else { // fsbl_atf_handover関数から情報をゲット
    fsbl_atf_handover(
        &bl32_image_ep_info,
        &bl33_image_ep_info);
}
NOTICE("BL31: Secure code at 0x%lx\n",
        bl32_image_ep_info.pc);
NOTICE("BL31: Non secure code at 0x%lx\n",
        bl33_image_ep_info.pc);
}
```

fsbl_atf_handover

```
void fsbl_atf_handover(entry_point_info_t *bl32,
                      entry_point_info_t *bl33)
{
    ...
    Atf_handoff_addr = mmio_read_32(
        PMU_GLOBAL_GEN_STORAGE6);
    assert((atf_handoff_addr < BL31_BASE) ||
        (atf_handoff_addr > (uint64_t)&__BL31_END__));
    if (!atf_handoff_addr) {
        ERROR("BL31: No ATF handoff structure passed\n");
        panic();
    }
}
```

FSBLからATFへのパラメータ渡し方

The FSBL populates a data structure with image information for the ATF.

The ATF uses that data to hand off to the loaded images. The address of the handoff data structure is passed in the `PMU_GLOBAL.GLOBAL_GEN_STORAGE6` register.

The register is free to be used by other software once the ATF is bringing up further firmware images.

fsbl_atf_handoverの続き

// パラメータのチェック

ATFHandoffParams =

```
(struct xfsbl_atf_handoff_params *) atf_handoff_addr;
if ((ATFHandoffParams->magic[0] != 'X') ||
    (ATFHandoffParams->magic[1] != 'L') ||
    (ATFHandoffParams->magic[2] != 'N') ||
    (ATFHandoffParams->magic[3] != 'X')) {
    ERROR("BL31: invalid ATF handoff structure at %lx\n",
          atf_handoff_addr);
    panic();
}
VERBOSE("BL31: ATF handoff params at:0x%lx, entries:%u\n",
        atf_handoff_addr, ATFHandoffParams->num_entries);
if (ATFHandoffParams->num_entries > FSBL_MAX_PARTITIONS) {
    ERROR("BL31: ATF handoff params: too many partitions (%u/%u)\n",
          ATFHandoffParams->num_entries, FSBL_MAX_PARTITIONS);
    panic();
}
```

fsbl_atf_handoverの続き

```
#define FSBL_MAX_PARTITIONS    8
```

```
struct xfsbl_partition {  
    uint64_t entry_point;  
    uint64_t flags;  
};
```

```
struct xfsbl_atf_handoff_params {  
    uint8_t magic[4];    // Magic Number [XLNX]  
    uint32_t num_entries;  
    struct xfsbl_partition  
        partition[FSBL_MAX_PARTITIONS];  
};
```

パーティションは、最大8個

fsbl_atf_handoverの続き

```
/*  
we loop over all passed entries  
but only populate two image structs (bl32, bl33).  
I.e. the last applicable images in the handoff  
structure will be used for the hand off  
*/  
for(size_t i=0; i<ATFHandoffParams->num_entries;  
                                i++) {  
    // ここで、BL32/BL33に関する情報を獲得する  
  
}
```

bl31_plat_arch_setup

```
// RESET_TO_BL31=1の場合は、  
// bl31_early_platform_setup関数の引数(x0/x1)を  
// 0でクリアする
```

```
mov    x0, 0  
mov    x1, 0
```

```
// Zynq UltraScale+ MPSoC用のセットアップ  
bl     bl31_early_platform_setup  
bl     bl31_plat_arch_setup
```

```
// BL31のmain関数を実行  
bl     bl31_main
```

bl31_plat_arch_setup

```
void bl31_plat_arch_setup(void)
{
    plat_arm_interconnect_init();
    plat_arm_interconnect_enter_coherency();

    arm_setup_page_tables( // ページテーブルの設定
        BL31_BASE,        BL31_END - BL31_BASE,
        BL_CODE_BASE,     BL_CODE_LIMIT,
        BL_RO_DATA_BASE,  BL_RO_DATA_LIMIT,
        BL31_COHERENT_RAM_BASE,
        BL31_COHERENT_RAM_LIMIT);

    enable_mmu_el3(0);
}
```

bl31_main

```
// RESET_TO_BL31=1の場合は、  
// bl31_early_platform_setup関数の引数(x0/x1)を  
// 0でクリアする
```

```
mov    x0, 0  
mov    x1, 0
```

```
// Zynq UltraScale+ MPSoC用のセットアップ  
bl     bl31_early_platform_setup  
bl     bl31_plat_arch_setup
```

```
// BL31のmain関数を実行  
bl     bl31_main
```

bl31_main

```
void bl31_main(void)
{
    NOTICE("BL31: %s\n", version_string);
    NOTICE("BL31: %s\n", build_message);

    bl31_platform_setup(); // BL31の準備

    bl31_lib_init();       // BL31用ライブラリの設定

    INFO("BL31: Initializing runtime services\n");
    runtime_svc_init(); // ランタイム・サービスの初期化
```


bl31_platform_setup

```
void bl31_platform_setup(void)
{
    // GIC関連の初期化
    plat_arm_gic_driver_init();
    plat_arm_gic_init();

    // テスト用の設定 (ZYNQMP_TESTINGが定義されている時)
    zynqmp_testing_setup();
}
```

bl31_mainの続き

```
// BL32が設定されている場合
if (bl32_init) {
    INFO("BL31: Initializing BL32\n");
    (*bl32_init)();
}

// 次に実行するイメージエントリの準備
bl31_prepare_next_image_entry();

bl31_plat_runtime_setup();
}
```

bl31_prepare_next_image_entry

```
void bl31_prepare_next_image_entry(void)
{
    image_type = bl31_get_next_image_type();

    next_image_info =
        bl31_plat_get_next_image_ep_info(image_type);

    INFO("BL31: Preparing for EL3 exit to %s world\n",
        (image_type == SECURE) ? "secure" : "normal");

    print_entry_point_info(next_image_info);
    cm_init_my_context(next_image_info);
    cm_prepare_el3_exit(image_type);
}
```

bl31_plat_get_next_image_ep_info

```
entry_point_info_t*
bl31_plat_get_next_image_ep_info(uint32_t type)
{
    assert(sec_state_is_valid(type));

    // typeが Non Secure だと、BL33なので
    if (type == NON_SECURE)
        return &bl33_image_ep_info;

    return &bl32_image_ep_info;
}
```

bl31_prepare_next_image_entry

```
void bl31_prepare_next_image_entry(void)
{
    image_type = bl31_get_next_image_type();

    next_image_info =
        bl31_plat_get_next_image_ep_info(image_type);

    INFO("BL31: Preparing for EL3 exit to %s world\n",
        (image_type == SECURE) ? "secure" : "normal");

    print_entry_point_info(next_image_info);
    cm_init_my_context(next_image_info);
    cm_prepare_el3_exit(image_type); // EL3を抜ける準備
}
```


cm_prepare_el3_exit

```
/*  
    If execution is requested to EL2 or hyp mode,  
    SCTLR_EL2 is initialized  
    If execution is requested to non-secure EL1  
    or svc mode, and the CPU supports  
    EL2 then EL2 is disabled by configuring  
    all necessary EL2 registers.  
    For all entries, the EL1 registers  
    are initialized from the cpu_context  
*/  
  
void cm_prepare_el3_exit(uint32_t security_state)  
{
```

bl31_mainの続き

```
// BL32が設定されている場合
if (bl32_init) {
    INFO("BL31: Initializing BL32\n");
    (*bl32_init)();
}

// BL31のイメージエントリの準備
bl31_prepare_next_image_entry();

bl31_plat_runtime_setup();
}
```

bl31_plat_runtime_setup

```
void bl31_plat_runtime_setup(void)
{
    // 特に何もやっていない
}
```

bl31_entrpoint.Sの続き

```
adr    x0, __DATA_START__  
adr    x1, __DATA_END__  
sub    x1, x1, x0  
bl     clean_dcache_range // DATA領域Cache Clean
```

```
adr    x0, __BSS_START__  
adr    x1, __BSS_END__  
sub    x1, x1, x0  
bl     clean_dcache_range // BSS領域Cache Clean
```

```
b      el3_exit // EL3を抜ける  
endfunc bl31_entrpoint
```

el3_exit

```
func el3_exit
```

```
/* Save the current SP_EL0  
   i.e. the EL3 runtime stack which will be used  
   for handling the next SMC. Then switch to
```

```
SP_EL3
```

```
*/
```

```
// 現在のSP_EL0をセーブする
```

```
mov      x17, sp
```

```
msr      spsel, #1
```

```
str      x17, [sp, #CTX_EL3STATE_OFFSET  
               + CTX_RUNTIME_SP]
```


el3_exitの続き

```
/* Restore SPSR_EL3, ELR_EL3
   and SCR_EL3 prior to ERET
*/
// セーブしておいたSCR_EL3/SPSR_EL3/ELR_EL3を
// リストアする
ldr      x18, [sp, #CTX_EL3STATE_OFFSET
               + CTX_SCR_EL3]
ldp      x16, x17, [sp, #CTX_EL3STATE_OFFSET
                   + CTX_SPSR_EL3]
msr      scr_el3, x18
msr      spsr_el3, x16
msr      elr_el3, x17
```

el3_exitの続き

```
// restore_gp_registers_eret (context.S)  
// セーブしておいた汎用レジスタをリストアして、  
// eretを実行しているので、ここには戻ってこない  
b    restore_gp_registers_eret
```

```
endfunc el3_exit
```

eret命令は、例外からの復帰。

SPSR_ELn に基づいてプロセッサ状態を復元し、ELR_ELn に分岐
ここで、n は現在の例外レベル

リセットからU-bootが立ち上がるまで

FSBLが行うこと

- ・BL31をDRAMにロード
- ・U-BootイメージをDRAMにロード
- ・RVBAR_EL3にbl31_entrypointを設定


BL31にジャンプ

eret命令

ATF BL31

- ・例外ベクタの設定

BL33
U-Boot



まだ、途中