Feature Detection Mechanism

Jayanth Chidanand
21-04-2022

AGENDA

- Introduction
- Need for this mechanism
- Design
- Implementation
- Blockers (Impact on current boot-flow)
- Note for Partners
- Future Scope

What is Feature Detection ?

- A mechanism for identifying the features which are enabled (by software) but not implemented in the hardware.
- A diagnostic procedure to quickly check and get assured which features are not supported by the hardware at an early stage of booting.

Need for Feature Detection ?

- Accessing registers based on architectural version during context management.
- Registers are tightly coupled to features than the arch version.
- For a given version of the architecture, registers will be under the influence of both the optional and mandatory features.
- If the given version of implementation does not support both, unconditional access to such registers leads to undefined behaviour.
- Altogether saving and restoring of registers without verifying their actual presence in the PE leads to runtime EL-3 exceptions.

Code Snippet (earlier)

```
#if ARM_ARCH_AT_LEAST(8, 6)
   mrs    x13, HAFGRTR_EL2
    mrs    x14, HDFGRTR_EL2
    stp    x13, x14, [x0, #CTX_HAFGRTR_EL2]
    ……………….
    ……………….
#endif ARM_ARCH_AT_LEAST(8, 6)
```

Arm ARM reads:
"**HAFGRTR_EL2** register is present only when **FEAT_AMUv1** is implemented and **FEAT_FGT** is implemented. Otherwise, direct accesses to **HAFGRTR_EL2** are UNDEFINED."

But as per Arm ARM, for v8.6 **FEAT_FGT** is mandatory, **FEAT_AMUv1** is optional.

So, the above code, leads to runtime EL-3 exception. This needs to be handled.

Method 1:

Having an explicit feature specific build flag, to control the register access, rather than **ARM_ARCH_LEAST** macro.

Save_routine

```
#if ENABLE_FEAT_FGT
   mrs   x13, HDFGRTR_EL2
#if ENABLE_FEAT_AMUv1
      mrs   x14, HAFGRTR_EL2
      stp   x13, x14, [x0, #CTX_HDFGRTR_EL2]
#else
      str   x13, [x0, #CTX_HDFGRTR_EL2]
#endif /* ENABLE_FEAT_AMUv1   */
#endif /* ENABLE_FEAT_FGT   */
```

Restore_routine

```
#if ENABLE_FEAT_FGT
#if ENABLE_FEAT_AMUv1
   ldp   x13, x14, [x0, #CTX_HDFGRTR_EL2]
   msr   HAFGRTR_EL2, x14
#else
   ldr   x13, [x0, #CTX_HDFGRTR_EL2]
#endif /* ENABLE_FEAT_AMUv1   */
   msr   HDFGRTR_EL2, x13
#endif /* ENABLE_FEAT_FGT   */
```

- Having an explicit build flag for each feature is certainly a good option.
- **Pros:**
  - It avoids dynamic feature detection which will affect the performance as thousands of context switch happens at runtime between normal and secure worlds.
- **Cons:**
  - However, this still needs some refinement as the feature-specific macros are just the build options and accidentally still there is a possibility of these options getting enabled and they might still be able to not match the features and the issue can reappear at a later part.

- To handle these scenarios, build flags need to be validated at an early phase, which is done via Feature Detection Mechanism.

Method 2 (Feature Detection):

- The feature specific build flags need to be validated at an early stage, before their usage, by reading the specific ID registers to confirm the feature's presence.
- But reading ID registers to verify the presence is again an additional overhead.

- To simplify this, we have designed it in a way, wherein we have introduced feature detection mechanism at a central boot/initialization path, checking whether the enabled build options match the given hardware implementation (by reading all the ID registers) at once.

Design

- We have considered a tri-state approach for Feature enablement for EL3.
- The 3 states are:
  - **ENABLE_FEAT_xxx = 0:** The feature is disabled statically at compile time.
  - **ENABLE_FEAT_xxx = 1:** The feature is enabled and must be present in hardware. There will be hard panic if the feature is not present at cold boot.
  - **ENABLE_FEAT_xxx = 2:** The feature is enabled but dynamically enabled at runtime depending on hardware capability.

TRI-STATES:

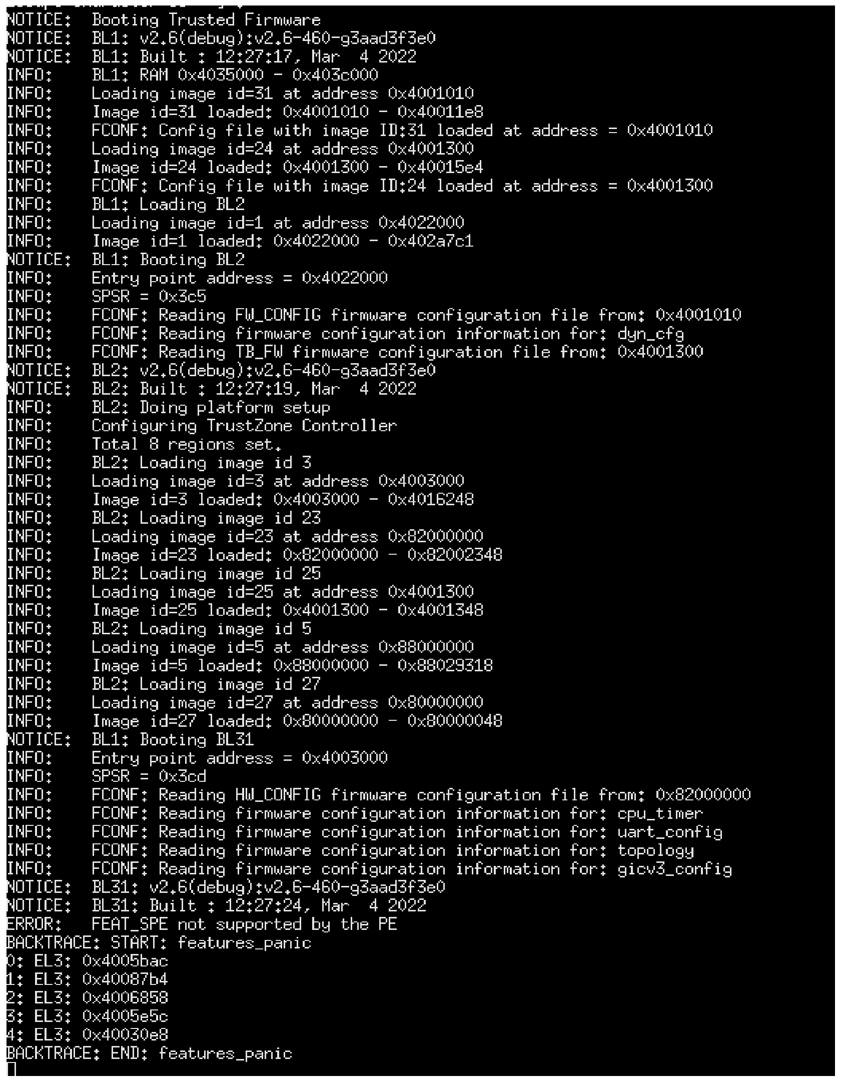
FEAT_STATE	DESCRIPTION	BOOT SECTION	CONTEXT_MGMT	CONTEXT_SAVE_RESTORE	ORDER OF STRICTNESS
0	Feat Disabled at Compile time	----	----	----	LOW
1	Feat Check at boot-time	Panic, if in case Feature enabled but not present in the hardware	Feature Enablement (set_bit)	Save and Restore feature specific registers	HIGH
2	Feat Check at Run-time	----	Feat_Check and allow bit enablement, if supported by hardware.	Feat_Check and perform save and restore routines if supported by hardware.	MEDIUM

- **Pros:**
  - The major advantage of using this approach is that we can run the same software on multiple variants/versions of hardware.
  - Also, we are not preventing the boot mechanism as we detect the features and enable them dynamically with FEAT_STATE=2.
  - Flexibility in fine tuning the mechanism.

- **Cons:**
  - Software is relatively less optimized due to many conditional checks (if **ENABLE_FEAT_STATE=2**).

Key features:

- An explicit build option to enable/disable it based on requirements.
- TRI_STATE: Allowing flexibility to fine tune the mechanism.
- Supports for both debug and release builds.
- Error logging support.



```
NOTICE:  Booting Trusted Firmware
NOTICE:  BL1: v2.6(debug):v2.6-460-g3aad3f3e0
NOTICE:  BL1: Built : 12:27:17, Mar  4 2022
INFO:    BL1: RAM 0x4035000 - 0x403c000
INFO:    Loading image id=31 at address 0x4001010
INFO:    Image id=31 loaded: 0x4001010 - 0x40011e8
INFO:    FCONF: Config file with image ID:31 loaded at address = 0x4001010
INFO:    Loading image id=24 at address 0x4001300
INFO:    Image id=24 loaded: 0x4001300 - 0x40015e4
INFO:    FCONF: Config file with image ID:24 loaded at address = 0x4001300
INFO:    BL1: Loading BL2
INFO:    Loading image id=1 at address 0x4022000
INFO:    Image id=1 loaded: 0x4022000 - 0x402a7c1
NOTICE:  BL1: Booting BL2
INFO:    Entry point address = 0x4022000
INFO:    SPSR = 0x3c5
INFO:    FCONF: Reading FW_CONFIG firmware configuration file from: 0x4001010
INFO:    FCONF: Reading firmware configuration information for: dyn_cfg
INFO:    FCONF: Reading TB_FW firmware configuration file from: 0x4001300
NOTICE:  BL2: v2.6(debug):v2.6-460-g3aad3f3e0
NOTICE:  BL2: Built : 12:27:19, Mar  4 2022
INFO:    BL2: Doing platform setup
INFO:    Configuring TrustZone Controller
INFO:    Total 8 regions set.
INFO:    BL2: Loading image id 3
INFO:    Loading image id=3 at address 0x4003000
INFO:    Image id=3 loaded: 0x4003000 - 0x4016248
INFO:    BL2: Loading image id 23
INFO:    Loading image id=23 at address 0x82000000
INFO:    Image id=23 loaded: 0x82000000 - 0x82002348
INFO:    BL2: Loading image id 25
INFO:    Loading image id=25 at address 0x4001300
INFO:    Image id=25 loaded: 0x4001300 - 0x4001348
INFO:    BL2: Loading image id 5
INFO:    Loading image id=5 at address 0x88000000
INFO:    Image id=5 loaded: 0x88000000 - 0x88029318
INFO:    BL2: Loading image id 27
INFO:    Loading image id=27 at address 0x80000000
INFO:    Image id=27 loaded: 0x80000000 - 0x80000048
NOTICE:  BL1: Booting BL31
INFO:    Entry point address = 0x4003000
INFO:    SPSR = 0x3cd
INFO:    FCONF: Reading HW_CONFIG firmware configuration file from: 0x82000000
INFO:    FCONF: Reading firmware configuration information for: cpu_timer
INFO:    FCONF: Reading firmware configuration information for: uart_config
INFO:    FCONF: Reading firmware configuration information for: topology
INFO:    FCONF: Reading firmware configuration information for: gicv3_config
NOTICE:  BL31: v2.6(debug):v2.6-460-g3aad3f3e0
NOTICE:  BL31: Built : 12:27:24, Mar  4 2022
ERROR:   FEAT_SPE not supported by the PE
BACKTRACE: START: features_panic
0: EL3: 0x4005bac
1: EL3: 0x40087b4
2: EL3: 0x4006858
3: EL3: 0x4005e5c
4: EL3: 0x40030e8
BACKTRACE: END: features_panic
```

Implementation

- Phase-1 (FEAT_STATE=0 , FEAT_STATE=1)
- Phase-2 (FEAT_STATE=2)

Example: FEAT_STATE=1

FeatureName	FeatureFlag	Boot Section (Bl31_main.c)	Context_Management Section (context_mgmt.c)	Registers Save & Restore Section (Context.S)
FEAT_HCX	**ENABLE_FEAT_HCX**	**detect_arch_features( )** **{** read_feat_hcx( ); read_feat_xxx( ); **}** void read_feat_hcx(void) { #if (ENABLE_FEAT_HCX == FEAT_STATE_1) **feat_detect_panic(is_feat_hcx_present(), "HCX");** #endif **}**	**#if ENABLE_FEAT_HCX** scr_el3 |= SCR_HXEn_BIT; **#endif**	Save routine: **#if ENABLE_FEAT_HCX** mrs x14, hcrx_el2 str x14, [x0, #CTX_HCRX_EL2] **#endif /* ENABLE_FEAT_HCX */** Restore routine: **#if ENABLE_FEAT_HCX** ldr x14, [x0, #CTX_HCRX_EL2] msr hcrx_el2, x14 **#endif /* ENABLE_FEAT_HCX */**

So, with the above example, it's evident that we are using the build flag in three sections.
If we validate them at an early phase, the subsequent sections will be safe.

Example: FEAT_STATE=2 (To do)

FeatureName	FeatureFlag	Boot Section (Bl31_main.c)	Context_Management Section (context_mgmt.c)	Registers Save & Restore Section (Context.S)
FEAT_HCX	**ENABLE_FEAT_HCX**		**#if ENABLE_FEAT_HCX** **#if (ENABLE_FEAT_HCX == FEAT_STATE_2)** **if** (feat_hcx_present) **#endif** scr_el3 |= SCR_HXEn_BIT; **#endif**	Save routine: **#if ENABLE_FEAT_HCX** **#if (ENABLE_FEAT_HCX == FEAT_STATE_2)** **if** (feat_hcx_present) **#endif** mrs x14, hcrx_el2 str x14, [x0, #CTX_HCRX_EL2] **#endif /* ENABLE_FEAT_HCX */** Restore routine: **#if ENABLE_FEAT_HCX** **#if (ENABLE_FEAT_HCX == FEAT_STATE_2)** **if** (feat_hcx_present) **#endif** ldr x14, [x0, #CTX_HCRX_EL2] msr hcrx_el2, x14 **#endif /* ENABLE_FEAT_HCX */**

So, here we dynamically enable the features allowing the platforms to boot and then detect and enable the features at runtime.

Mandatory Features

- TF-A supports majority of Arm architectural features (Mandatory & Optional).
- We provide an explicit build flag for each mandatory feature enablement, and this gets validated as part of mechanism.
- A given hardware implementation based on particular arch version, will support the mandatory features by default.
- Henceforth, TF-A enables those feature-specific build flags by default.

- Eg: FEAT_FGT (As per Arm ARM docs, it's a mandatory feature from v8.6+), so the build flag, will be set by default from 8.6 and higher versions. i.e (ENABLE_FEAT_FGT=1)

Mandatory Features Detection:

- If the feature is a mandatory feature for a particular arch version and upwards, the platform which is based on that arch version will implement it.
- Eg: Let's say **FEAT_FGT** which is mandatory from the 8.6 version. So if a platform is based on v8.6 it will implement this and this feature will be detected. So no issue here.
  If the platform is based on v8.5, this FEAT_FGT is not enabled by the TF-A. It gets enabled from 8.6. So here, in this case, the feature is disabled so nothing to worry about.

Optional Features

- TF-A provides support for optional features, like the way it does for mandatory.
- However, here we do not enable optional feature by default.
- We allow the platforms to decide based on their requirements.

Optional Feature Detection:

- **Let's say FEAT_NV2** which is an optional feature from arch version 8.4. is supported by TF-A. Since it is an optional one as per Arm ARM, TF-A implements and disables it by default and allows the platforms to decide and enable them as per their requirements.
- So here, if the platform enables it, it implies they are sure this feature is implemented. If not, this mechanism will help them by detecting it, so that they disable it in future. In general, this would not break the boot flow in all scenarios.

- But there is a minor deviation in the way this has been handled for couple of features.
- SPE and SVE.

Blockers

- Feature Detection mechanism runs through all the features and checks if there is any mismatch between software and hardware.
- Now, even though **FEAT_SPE and FEAT_SVE** are optional features they are enabled by default in TF-A build system.
- If we detect these optional features, they might not be found on certain platforms and eventually the booting halts.
- To make it comply with feature detection mechanism ideally it should be disabled by default and platform which are actually using this feature should enable it in their platform Makefile.
- The problem is, we do not know which all platforms are actually using these features : **need input from Platform owners.**

Note for TF-A Partner-platforms:

- To handle the previously mentioned scenarios, for now we have overlooked such features and introducing this mechanism as an experimental procedure.
- **FEATURE_DETECTION** build flag has been added to guard the entire implementation.
- As part of the 2.7 release, we have up streamed this implementation .
- We urge the platforms to enable this mechanism, test it and get used to its behaviour before it gets mandated.
- So, for now, it wouldn't cause any issue. But our plan is to make sure this mechanism runs by default.
- So, moving ahead, we will refactor the optional features, which are troublesome (mainly the ones which enabled by default) and ensure partners are aware of it.
- We are happy to hear any feedback from platform owners on handling this issue.

Future Scope

- Currently, we are in phase-1 delivery, wherein we have implemented for {FEAT_STATE=0,1} which will read through all the enabled feature build flags, **ENABLE_FEAT_XXX=1** the respective feature flag will be validated.
- Further we will be handling the exceptional cases discussed earlier and handle FEAT_STATE=2 for all the features.

- Patch Links:
  - https://review.trustedfirmware.org/q/topic:jc/detect_feat
  - https://trustedfirmware-a.readthedocs.io/en/latest/getting_started/build-options.html?highlight=FEATURE_DETECTION#common-build-options

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה