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 \texttt{ARM_ARCH_LEAST} macro.

\textbf{Save\_routine}

\begin{verbatim}
#ifdef ENABLE_FEAT_FGT
    mrs x13, HDFGRTR_EL2
#endif /* ENABLE_FEAT_FGT */

#ifdef 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 */
\end{verbatim}

\textbf{Restore\_routine}

\begin{verbatim}
#ifdef ENABLE_FEAT_FGT
    #ifdef 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 */
\end{verbatim}
• 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:

<table>
<thead>
<tr>
<th>FEAT_STATE</th>
<th>DESCRIPTION</th>
<th>BOOT SECTION</th>
<th>CONTEXT_MGMT</th>
<th>CONTEXT_SAVE_RESTORE</th>
<th>ORDER OF STRICTNESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Feat Disabled at Compile time</td>
<td></td>
<td></td>
<td></td>
<td>LOW</td>
</tr>
<tr>
<td>1</td>
<td>Feat Check at boot-time</td>
<td>Panic, if in case Feature enabled but not present in the hardware</td>
<td>Feature Enablement (set_bit)</td>
<td>Save and Restore feature specific registers</td>
<td>HIGH</td>
</tr>
<tr>
<td>2</td>
<td>Feat Check at Run-time</td>
<td></td>
<td></td>
<td>Feat_Check and allow bit enablement, if supported by hardware.</td>
<td>MEDIUM</td>
</tr>
</tbody>
</table>
• **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.
Implementation

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

<table>
<thead>
<tr>
<th>FeatureName</th>
<th>FeatureFlag</th>
<th>Boot Section (Bl31_main.c)</th>
<th>Context_Management Section (context_mgmt.c)</th>
<th>Registers Save &amp; Restore Section (Context.S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT_HCX</td>
<td>ENABLE_FEAT_HCX</td>
<td><code>detect_arch_features()</code></td>
<td>#if ENABLE_FEAT_HCX</td>
<td>Save routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td>scr_el3</td>
<td>= SCR_HXEn_BIT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read_feat_hcx();</td>
<td></td>
<td>mrs x14, hcrx_el2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read_feat_xxx();</td>
<td></td>
<td>str x14, [x0, #CTX_HCRX_EL2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#endif /* ENABLE_FEAT_HCX */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
<td>Restore routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void read_feat_hcx(void)</td>
<td></td>
<td>#if ENABLE_FEAT_HCX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td>ldr x14, [x0, #CTX_HCRX_EL2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#if (ENABLE_FEAT_HCX == FEAT_STATE_1)</td>
<td></td>
<td>msr hcrx_el2, x14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>feat_detect_panic(is_feat_hcx_present(), &quot;HCX&quot;);</td>
<td></td>
<td>#endif /* ENABLE_FEAT_HCX */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#endif</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>FeatureName</th>
<th>FeatureFlag</th>
<th>Boot Section (Bl31_main.c)</th>
<th>Context_Management Section (context_mgmt.c)</th>
<th>Registers Save &amp; Restore Section (Context.S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEAT_HCX</td>
<td>ENABLE_FEAT_HCX</td>
<td>#if ENABLE_FEAT_HCX #if (ENABLE_FEAT_HCX == FEAT_STATE_2) if (feat_hcx_present) #endif scr_el3</td>
<td>= SCR_HXEn_BIT; #endif</td>
<td>#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 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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 */</td>
</tr>
</tbody>
</table>

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 \{\text{FEAT\_STATE}=0,1\}\ which will read through all the enabled feature build flags, \text{ENABLE\_FEAT\_XXX}=1\ the respective feature flag will be validated.

• Further we will be handling the exceptional cases discussed earlier and handle \text{FEAT\_STATE}=2\ for all the features.

• Patch Links:
  • https://review.trustedfirmware.org/q/topic:jc/detect_feat