Agenda

• What is SCP Firmware
• Basic Building Blocks
• SCP Firmware execution stages
• Inter Module Communication
• Threading
• Guidelines for new Module Design
• Status & Next Steps
What is SCP Firmware

Introduction

• To abstract power and system management tasks away from Application Processors (APs).

• Compliant to Arm System Control & Management Interface (SCMI) specification.

• Code is opensource: https://github.com/ARM-software/SCP-firmware.
  • Contributions are welcome!

• Execution Environment agnostic. Can be run on RTOS or bare-metal environments.
Basic Building Blocks

Overall Layout

- 3 layers:
  - Architecture:
    - Provides execution environment dependent functionality like threading, interrupts, memory management etc.
  - Framework:
    - Provides common services to all modules like Initialization, Event, Notification & Interrupt handling.
      - Depends on architecture layer for execution environment dependent services
    - Architecture and execution environment agnostic
    - Drives the initialization, orchestration of, and interactions between modules
  - Modules:
    - Architecture agnostic
    - A module performs a well-defined set of operations.

Diagram:

- Framework
- Configuration
- Layers
- Architecture Specific Implementation
- Architecture & Execution Environment Agnostic
- Product Specific
Basic Building Blocks

Modules (struct fwk_module)

• **Types of Modules**
  • **Hardware Abstraction Layer:**
    – Generic interface to a class of drivers, e.g. sensor.
    – Other modules use platform drivers through HAL APIs
  • **Driver:**
    – Controls a specific device.
    – May implement APIs defined by a HAL module.
    – Driver can opt to not use HAL.
  • **Protocol:**
    – Provide protocol specific interfaces, e.g. arbitration of messaging channels, to other modules
  • **Service**
    – Work or functionality that is not related to hardware devices.
    – May be self contained and not expose any APIs to other modules

• **Binding**
  • Binding enables modules to use another module’s set of APIs.
  • Each set of APIs provided by a module is uniquely identified.
    – Module elements may provide different implementations of the same set of APIs

Combining Modules into a Product

• Product consists of a product.mk file that defines one or more firmware targets.
• Each Firmware target is a binary image built when building the product.
• A firmware target is fully defined by its set of modules and their configuration data through struct fwk_module_config.
Basic Building Blocks

Elements & Sub-Elements

• Elements
  • A resource that is owned and governed by a module.
  • An abstraction to refer to a device, a protocol or a service instance.
  • E.g., elements of a driver type module may represent each hardware device instance it controls.
  • Elements are optional.
  • Element description.
    – One per element.
    – Contains element configuration data.
  • Elements are defined by:
    – a structure containing a pointer to a name string
    – the number of sub-elements associated with the element
    – a void pointer to data that is in a module-defined format

• Sub-Elements
  • Resource owned and governed by an element.
  • Don’t have descriptors.

EXAMPLE

• SENSOR HAL is a Module.
• PVT and Thermal Sensor Drivers are Modules that use Sensor HAL.
• PVT and Thermal Sensors are arranged in several groups. Each group is an element having its own configuration.
• Each sensor within a group is a sub-element.
SCP Firmware Execution Stages

Pre-Runtime & Runtime

- Pre-Runtime Stages: 5 Stages in fixed order

  - Module Initialization: \( \text{init}() \) function of module called by framework with module configuration data.
  
  - Element Initialization: \( \text{element_init}() \) function of module called by framework with element configuration data. This stage is only valid if module has elements.
  
  - Post-Initialization: \( \text{post_init}() \) function of module called by framework. Any extra initialization after element data has been provided to modules. Optional Stage.
  
  - Bind: \( \text{bind}() \) function of module called by framework. Modules & elements bind to other modules & elements. Optional Stage.
  
  - Start: \( \text{start}() \) function of module called by framework. Modules can use resources of other modules to complete initialization. Optional Stage.

- Runtime Stage
SCP Firmware Execution Stages

Pre-Runtime & Runtime

• Pre-Runtime Stages: 5 Stages in fixed order
  • **Module Initialization**: `.init()` function of module called by framework with module configuration data.

• Runtime Stage
SCP Firmware Execution Stages

Pre-Runtime & Runtime

- **Pre-Runtime Stages:** 5 Stages in fixed order
  - **Module Initialization:** `.init()` function of module called by framework with module configuration data.
  - **Element Initialization:** `.element_init()` function of module called by framework with element configuration data. This stage is only valid if module has elements.

- **Runtime Stage**
SCP Firmware Execution Stages

Pre-Runtime & Runtime

- **Pre-Runtime Stages**: 5 Stages in fixed order
  - **Module Initialization**: `.init()` function of module called by framework with module configuration data.
  - **Element Initialization**: `.element_init()` function of module called by framework with element configuration data. This stage is only valid if module has elements.
  - **Post-Initialization**: `.post_init()` function of module called by framework. Any extra initialization after element data has been provided to modules. Optional Stage.

- **Runtime Stage**
SCP Firmware Execution Stages

Pre-Runtime & Runtime

• Pre-Runtime Stages: 5 Stages in fixed order
  • **Module Initialization**: `.init()` function of module called by framework with module configuration data.
  • **Element Initialization**: `.element_init()` function of module called by framework with element configuration data. This stage is only valid if module has elements.
  • **Post-Initialization**: `.post_init()` function of module called by framework. Any extra initialization after element data has been provided to modules. Optional Stage.
  • **Bind**: `.bind()` function of module called by framework. Modules & elements bind to other modules & elements. Optional Stage.

• Runtime Stage
SCP Firmware Execution Stages

Pre-Runtime & Runtime

• **Pre-Runtime Stages**: 5 Stages in fixed order
  - **Module Initialization**: `init()` function of module called by framework with module configuration data.
  - **Element Initialization**: `element_init()` function of module called by framework with element configuration data. This stage is only valid if module has elements.
  - **Post-Initialization**: `post_init()` function of module called by framework. Any extra initialization after element data has been provided to modules. Optional Stage.
  - **Bind**: `bind()` function of module called by framework. Modules & elements bind to other modules & elements. Optional Stage.
  - **Start**: `start()` function of module called by framework. Modules can use resources of other modules to complete initialization. Optional Stage.

• **Runtime Stage**
SCP Firmware Execution Stages

Pre-Runtime & Runtime

• **Pre-Runtime Stages:** 5 Stages in fixed order
  - **Module Initialization:** `.init()` function of module called by framework with module configuration data.
  - **Element Initialization:** `.element_init()` function of module called by framework with element configuration data. This stage is only valid if module has elements.
  - **Post-Initialization:** `.post_init()` function of module called by framework. Any extra initialization after element data has been provided to modules. Optional Stage.
  - **Bind:** `.bind()` function of module called by framework. Modules & elements bind to other modules & elements. Optional Stage.
  - **Start:** `.start()` function of module called by framework. Modules can use resources of other modules to complete initialization. Optional Stage.

• **Runtime Stage**
  - Normal execution flow directed primarily by interaction between modules.
  - Event, notifications and responses generated and processed.
Inter-Module communication

Events & Notifications

• Events: Abstraction for communicating requests/responses.
  • Mechanism for implementing a logical task in callee context.
  • Module provides .process_event() handler which gets called by Framework when event targeting module is found.
  • When the task associated to a request is completed, a response event may be sent. A response may be sent as part of the processing of an event or later.
    – Delayed Response: Response is sent later and not immediately after processing the Event
    – Standard Response: Framework generates response as soon as .process_event() returns.
    – Response is an Event with response flag set. Processed by Firmware in same manner as Events.

• Notifications: Events with notification field set.
  • Modules can subscribe to notifications from other modules.
  • Notifications are broadcast to all subscribed modules by the framework.
  • Can be used to implement dependency chain.
    – e.g., if before a System Power Transition, we may need to change a clock or setup some wakeup handling. The modules can use notifications from System Power Module.

• Responsiveness
• Scalability (Implicit queueing)
• Execution Environment Independence
• Lockless

• Events (fwk_event)
  • Response Required
    • Delayed Response
    • Standard Response
  • No response Required

• Notifications (fwk_event)
  • Response Required
    • Delayed Response
    • Standard Response
  • No response Required
Event Handling

Creating an Event

```
Start
Called from ISR?  
Yes  
Insert Event at tail of ISR Event Queue  
No  
Delayed Response?  
Yes  
Get event from delayed response list  
No  
Allocate Event Node  
Thread Wakeup Event?  
Yes  
Insert Target Thread to head of runnable thread queue  
No  
Insert Event into tail of Target Thread’s event queue  
Common Thread waiting as ISR Event Queue is empty?  
Yes  
Signal ISR_EVENT to common thread  
No  
Stop
```

`put_event()`
Event Handling
Creating an Event

- put_event_and_wait()
- Module does not use the common/framework thread.
- The thread blocks till event is processed and response is generated.
Event Handling

Processing an Event

Start

Get Event from head of current thread’s event queue

Event is notification?

Yes

Call process_notification() callback of target module

No

Call process_event() callback of target module

Event requires response?

Yes

Create & Insert response event in the tail of the source thread event queue.

Insert source thread at tail of runnable thread queue

Yes

Insert event into delayed response list to be processed by the target thread

No

Delay Response?

Yes

No

Stop

Events Processed in Framework/Common Thread or Module Thread Context
Threading

Features of SCP Threading Model

• Soft-realtime scheduling.
• Support both single and multi-threaded environments with equi-priority threads (no-preemption).
• Support for co-operative scheduling like in CMSIS compliant RTX RTOS.
• No Multi-processor support.
• Independence from direct RTOS calls through threading APIs defined by the framework.
  • These APIs are mapped to CMSIS for now.

Hybrid co-operative scheduling model

• The scheduling is co-operative among threads at same priority level.
• Eliminates the need for locks
• Makes the code simpler and avoids cases of deadlock.
• It removes execution-context/RTOS dependency and prevents overheads.
• Events are processed sequentially within a thread context.
Threading

Single Threaded Mode

- Simplest mode of operation and suits almost all non-RTOS-based execution environments.
- No Threading Overheads.
- BUILD_HAS_MULTITHREADING not defined.
- Framework thread is the only thread and services all events.
- Modules do not have own threads.
- Single event queue for all events, responses and notifications.
  - When an interrupt occurs, it gets serviced.
    - If part of the interrupt processing needs to be postponed (bottom half) an event gets inserted in the ISR Event Queue.
- When the Event Queue is empty, a single event in fetched from the ISR Event Queue and pushed into the tail of the Event Queue.

```
Start
Event Queue empty?
  Yes
  ISR Queue empty?
    Yes
    Move Event at head of Queue to Event Queue
    No
    No
    Process Event at head of Queue
  Yes
  No

Yes
```

Scheduling Model
Single Threaded Mode

Execution Flow

Queue of interrupt events

Event Driven – No Priority

Queue of framework events
Single Threaded Mode

Execution Flow

Event Driven – No Priority

Queue of interrupt events

Queue of framework events
Single Threaded Mode

Execution Flow

Event Driven – No Priority

Queue of interrupt events

Queue of framework events
Single Threaded Mode

Execution Flow

Event Driven – No Priority

Queue of interrupt events

Queue of framework events

I2 M12
Single Threaded Mode

Execution Flow

Event Driven – No Priority

Queue of interrupt events

Queue of framework events
Single Threaded Mode

Execution Flow

Event Driven – No Priority

Queue of interrupt events

Queue of framework events
Single Threaded Mode

Execution Flow

Event Driven – No Priority

Queue of interrupt events

Queue of framework events
Threading
Multi-Threaded Mode

• Suitable for RTOS based environments.
• BUILD_HAS_MULTITHREADING defined.
• One primary thread called the ‘common thread’.
  • In addition, each module can create its own thread.
• Common thread processes all events targeted to the modules which do not have their own thread.
• Each thread has its dedicated event queue.
• Threads are used to provide separate Module context for Event Processing.
  • Running arbitrary thread functions is disallowed.
• There is only one thread which is signaled to run at any point of time.

Scheduling Model

• All threads of same priority.
• All threads have same thread function – provided by the framework.
• Threads wait for a signal from another thread to wakeup and handle an event targeting it.
Threading

Multi-Threaded Mode

Common Thread Flowchart

Start

Get next Runnable Thread

Get event from next Runnable Thread’s Event Queue head.

Runnable Thread found?

No

ISR Event Queue Empty?

Yes

Wait for Signal on next ISR Event generation

Signal

No

Runnable thread == common thread?

Yes

Process Event

Signal target thread and yield on signal.

No

Put Event from ISR Event Queue to Target Thread Queue. Push target thread into head of runnable queue.

ISR_EVENT

Yes

Wait for Signal on next ISR Event generation

Signal ISR_EVENT

Module Thread Flowchart

Start

Wait on Event signal

Signal

Process Event

Runnable Thread found?

Yes

Runnable thread == itself?

Yes

Signal runnable thread and yield on signal.

No

Signal common thread (NO_READY_THREAD) and yield on signal.

No

Get next runnable thread and next event from next Runnable Thread’s Event Queue head.
Multi Threaded Mode

Execution Flow

- Event Driven – No Priority
- Soft real time constraints (no thread priority)

Queue of interrupt events

Common thread

M5-E2 specific thread

M17 specific thread
Multi Threaded Mode

Execution Flow

- Event Driven – No Priority
- Soft real time constraints (no thread priority)

Queue of interrupt events

- I1
- M2
- M1

- I3
- M17
- I2
- M17

Common thread

M17 specific thread

M5-E2 specific thread
Multi Threaded Mode

Execution Flow

- Event Driven – No Priority
- Soft real time constraints (no thread priority)

Queue of interrupt events

Common thread

M17 specific thread

M5-E2 specific thread

I3 M17
I2 M17
Multi Threaded Mode

Execution Flow

- Event Driven – No Priority
- Soft real time constraints (no thread priority)

Queue of interrupt events

- Common thread
- M17 specific thread
- M5-E2 specific thread

I3 M17

I2 M17
Multi Threaded Mode

Execution Flow

+ Event Driven – No Priority
+ Soft real time constraints (no thread priority)

Queue of interrupt events

Common thread

M5-E2 specific thread

M17 specific thread

E1
M5-E2

I3
M17
Multi Threaded Mode

Execution Flow

- Event Driven – No Priority
- Soft real-time constraints (no thread priority)

Queue of interrupt events

Common thread

M5-E2 specific thread

M17 specific thread

© 2019 Arm Limited
Multi Threaded Mode

Execution Flow

- Event Driven – No Priority
- Soft real time constraints (no thread priority)

Queue of interrupt events

Common thread

M5-E2 specific thread

M17 specific thread

© 2019 Arm Limited
Multi Threaded Mode

Execution Flow

- Event Driven – No Priority
- Soft real time constraints (no thread priority)

Queue of interrupt events
Multi Threaded Mode

Execution Flow

- Event Driven – No Priority
- Soft real time constraints (no thread priority)

Queue of interrupt events
Multi Threaded Mode

Execution Flow

• Event Driven – No Priority
• Soft real time constraints (no thread priority)

Queue of interrupt events
Multi Threaded Mode

Execution Flow

- Event Driven – No Priority
- Soft real time constraints (no thread priority)

Queue of interrupt events
Guidelines for New Module Design

• Generic Modules must support both single threaded and multi-threaded mode.
  • Product specific modules can choose to support any threading mode.

• Modules should not:
  • block in single threaded mode without a timeout.
  • perform long-running tasks in caller context. It should generate an event and split long-running tasks into smaller tasks via events.
  • have shared contexts with other modules which require locking.
  • allocate memory which will not be used later. The firmware does not have a memory “free” API. Dynamic memory management is not implemented to keep the firmware simple and efficient.
Status & Next Steps

• Neoverse N1 reference design, SGM-775, SGI-575 are supported – SCP Release v2.5.0
• SCP Idle (Turn OFF SCP when Idle): Under Design
• SCMI Protocols support in Single Threaded Mode: Under Design
• Full PM Support on reference Hardware (Arm Juno Board): Target Q4 2019
  • Power Domains, DVFS, Clocks, Sensors, PMIC over I2C, Thermal, etc.
• Work ongoing to enable SCP Firmware as Secure World resident firmware for Power and Performance Management – Linaro/STMicroelectronics/Arm.
Useful Links


• SCP Firmware: https://github.com/ARM-software/SCP-firmware
The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks