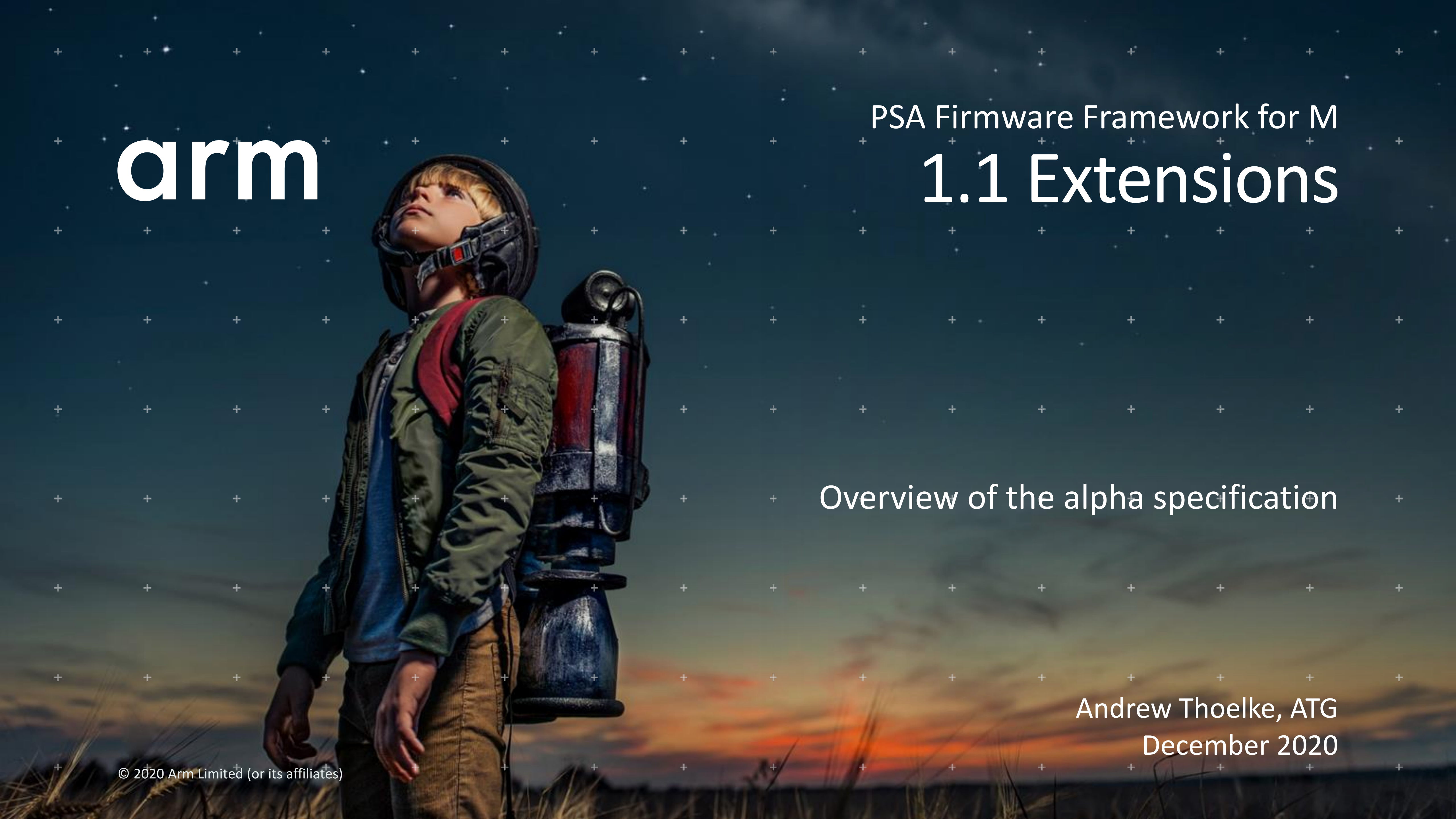

PSA Firmware Framework for M

# 1.1 Extensions

Overview of the alpha specification

Andrew Thoelke, ATG

December 2020

## PSA Firmware Framework for M – 1.1 Extensions

### Content

- Status
- Objectives
- Overview
- Feature details
- Next steps

## Status of the specification

- The proposed set of 1.1 Extensions is complete
- We would now like wider review of the proposed update
- This is an ALPHA quality specification
  - Significant changes to the architecture or API are still possible
  - This enables us to take community feedback of any ‘size’
- Integrating the 1.1 Extensions into the version 1.0 specification is complex
  - We will wait until we are ready with a BETA specification to make those changes
  - The ALPHA is a specification of the **changes** between version 1.0 and 1.1

- The 1.1 Extensions alpha specification should be available in a few weeks

# Objectives for version 1.1

- A framework specification that can target a smaller, simpler system
  - There should be continuity between any new lighter-weight framework and FF-M version 1.0, enabling the same RoT Service code to be used with different types of framework
- Provide optimized mechanisms for some common Secure Partition RoT Service development patterns
  - Improving efficiency for these use cases requires a reduction in flexibility or security mitigation
- Improve the support for secure peripheral drivers – the APIs in version 1.0 are inadequate for many systems:
  - The signal-based mechanism makes it very difficult to handle interrupts in a bounded time
  - The FF-M 1.0 API assumes that the peripheral MMIO interface is sufficient to control its interrupts
- Maintain compatibility for RoT Service code written for version 1.0
  - Make migration to version 1.1 frameworks, and adoption of 1.1 features easy

# Overview of the 1.1 Extensions

- Secure Functions
  - This introduces the SFN model as an option for each Secure Partition. Services are functions called by the framework, and use the IPC model APIs to read and write request parameters
- Stateless RoT Services
  - RoT Services that do not make use of connection features can be defined as *stateless*. A stateless RoT Service has a build-time handle value, used by a client to request one-shot services without making an explicit connection. Stateless RoT Services do not receive connection or disconnection messages
- Memory-mapped iovecs
  - This optional API introduces the ability for a service to directly read and write the client parameter memory. This will not work on all implementations, but is necessary for efficiency in simple systems
- Enhancements for peripheral drivers
  - Add a deprivileged, low-latency, interrupt handling capability to Secure Partitions
  - Provide interrupt management APIs
- A few smaller clarifications and improvements to the framework

## Secure Functions

## Secure Functions – programming model

- The Secure Function model (SFN model) is a new Secure Partition programming model
- The SFN model looks like a hybrid between the IPC model and the Library model
  - Secure services are implemented as Secure Functions (SFN) that are called by the framework
  - SFNs are invoked by a client call to `psa_connect()`, `psa_call()` and `psa_close()`
  - SFNs are provided with a client identity
  - SFNs access client parameters indirectly using APIs
- The framework invokes the SFNs directly to process a request
  - The framework provides the execution context for the SFN
  - There is no Secure Partition entry-point and signal handling loop
- Execution within and between Secure Partitions is the same as the IPC model:
  - A Secure Partition using the SFN model is single-threaded, SFNs within a Partition are run sequentially
  - The framework is permitted to run SFNs from different Secure Partitions concurrently
- The SFN model API is compatible with the IPC model API, not the Library model API

## Secure Functions – manifest changes

- The manifest file defines a new attribute `model`, to be either `"IPC"` or `"SFN"`
  - If it is `"SFN"`, then Secure Partition is using the SFN model and the following changes apply:

- The `entry_point` attribute is replaced with an optional `entry_init` attribute
  - If present, this identifies a function that is used to initialise the Secure Partition
- There are no RoT Service signals, and the RoT Service signal names are not defined
- Each RoT Service defined in the manifest has a Secure Function with the prototype:

```
psa_status_t «name»_sfn(const psa_msg_t* msg);
```

  - where «*name*» is the lowercase version of the RoT Service's `name` attribute

# Secure Functions – writing SFNs

- SFNs will still receive *connect*, *disconnect* and *request messages*, in the same way that these were delivered to an SP using the IPC model[1]
- A SFN processes the delivered message using the `psa_read()`, `psa_write()`, `psa_skip()`, MM-IOVEC and `psa_set_rhandle()` functions
- The return value from the SFN is used as the reply status for the message
- A SFN cannot use the `psa_get()` or `psa_reply()` functions, as this functionality is performed by the framework
- A SFN can use `psa_wait()` to wait for IRQ signals that are defined in the manifest, or the Secure Partition doorbell signal
- The remaining PSA FF-M APIs work in the same way as in the IPC model

[1] RoT Services that are defined as *stateless* do not receive *connection* and *disconnection messages*

## Stateless RoT Services

## Stateless RoT Services - context

- Many RoT Service APIs provide standalone operations that do not maintain any non-volatile state in the RoT Service, or do not expose any kind of context to the caller
- To implement these functions as a RoT Service using the version 1.0 API, the client side implementation of the service must use one of the following techniques:
  - In every function, use a transient connection handle with psa_connect() and psa_call() and psa_close()
  - Call psa_connect() once, and store the connection handle for reuse by all the other service functions
- The first technique has a high runtime overhead as it requires three calls to the RoT Service for every operation
- The second technique removes the overhead, but needs to store the connection handle. Using a shared connection variable will not always have the right isolation and security properties for the service, depending on the framework implementation
- Changes in the framework are needed to enable portable and efficient implementation of standalone RoT Service operations

# Stateless RoT Services – programming model

- In version 1.1, each RoT Service is either *connection-based* or *stateless*
- Connection-based RoT Services are already defined in version 1.0
- Stateless RoT Services do not use connections:
  - There is no call to psa_connect() or psa_close() by the client
  - There is no corresponding connection and disconnection message delivered to the RoT Service
  - The RoT Service cannot use the rhandle functionality
- Requests to the service are made by calling psa_call() using a fixed *stateless handle* value for the RoT Service
  - The identifier symbol for the stateless RoT Service handle is based on the RoT Service name
  - The value of the handle is determine by the implementation
  - If required, the implementation incorporates the RoT Service version into the *stateless handle*
- Implementations must support at least 32 stateless RoT Services in a system
- Stateless RoT Services can be used in Secure Partitions using IPC model and SFN model

## Memory-mapped iovecs

## Memory-mapped iovecs – programming model

- This is an optional API for frameworks implementing version 1.1
  - Direct access to client parameters is important for smaller, simpler systems
  - Direct access might be unacceptable in secure systems as it bypasses some memory-safety mitigations
  - Direct access can be difficult to implement in systems with complex memory architectures
- Availability of the API is discoverable using a pre-processor macro
- An RoT Service must explicitly enable MM-IOVEC in the manifest file to use the API
  - This limits the potential for exploiting this feature
- Each parameter can **either** be mapped using psa_map_invec() or psa_map_outvec(), or accessed using the existing psa_read() or psa_write() functions
  - There is limited value in mixing these APIs, and the restriction simplifies the implementation
- Unmapping parameters is automatic, but explicit unmapping is required for output parameters to provide the length of output that was written
- Memory mappings might be imprecise – implementations must document behaviour

# Enhancements for peripheral drivers

## Peripheral support - context

- The v1.0 API was designed to be simple and easy to use securely. It avoided concurrent execution within the Secure Partition by requiring the interrupt handler to run within the execution thread
  - Working around the limitations of this API for lower-latency interrupt requirements is painful, complex and error-prone
  - Framework support for immediate, asynchronous interrupt handling is needed in these use cases
- The v1.0 API assumed that peripheral interrupts are all ‘behaved well’:
  - The interrupt can be disabled at-source using a peripheral control register
  - The peripheral resets with all interrupts disabled
- In reality, not all peripherals meet these requirements either due to design decisions or implementation defects. PSA-FF-M needs to accommodate this reality
  - This requires changes to the interrupt model, and the addition of interfaces to support managing the interrupts at the CPU/Interrupt Controller

## Peripheral support – programming model

- Many operating systems have the concept of two levels of interrupt handling
  - Initial handling that occurs within the context of the interrupt exception, blocking all normal scheduled execution. Any interrupt response activity with bounded response time requirements is done here. The initial handling may run in a privileged or deprivileged context
  - Deferred handling that occurs within a thread/task context, subject to normal scheduling. This is often triggered from the initial handler, and can use the full set of OS functionality

- In version 1.1 we use the term *First-level interrupt handling* (FLIH) for the initial handling, and *Second-level interrupt handling* (SLIH) for the deferred activity

- The specification defines an architectural model for interrupt handling
  - This enables framework implementations to provide consistent behaviour across different systems

## Peripheral support – FLIH

- An interrupt specified in a Secure Partition manifest selects FLIH or SLIH handling
- SLIH mode works the same way as interrupt handling in version 1.0
  - The interrupt exception sets the interrupt signal, which is detected in a call to `psa_wait()`
  - The Secure Partition thread context calls `psa_eoi()` when interrupt processing is complete
- FLIH mode requires the developer to provide a callback function for the interrupt
  - The FLIH function is executed asynchronously when the interrupt occurs
  - Execution occurs within the Secure Partition memory context – it is deprivileged
  - Functionality is limited
    - FLIH functions cannot block, and most of the Secure Partition API is not available
  - Execution can pre-empt other code in the same Secure Partition
    - Care is required to avoid race conditions
  - End-of-interrupt processing occurs when the FLIH function returns
  - The FLIH function can optionally set the interrupt signal in the Secure Partition
    - Enables the Secure Partition thread context to process interrupt results, for example by replying to a message

## Peripheral support – interrupt control

- Provide a flexible IRQ control API
  - Allows Secure Partition code to enable and disable interrupts belonging to the Secure Partition
  - Also allows for implementation-specific extensions for managing other interrupt features
- Interrupts are disabled before first entry to any Secure Partition code
  - If the Secure Partition manifest specifies that it is using version 1.1
  - This is a change from the version 1.0 interrupt model, but was typically required in implementations anyway, which provided an implementation-specific API to manage the interrupt
  - The Secure Partition must enable the interrupt explicitly to start receiving interrupt FLIH callbacks or interrupt signals, after initializing the peripheral
- This extension also provides MMIO register access functions
  - These abstract reads and writes of peripheral control registers
  - They enable an implementation to provide essential platform-specific behaviour, for example, memory access control, or memory barriers.

## Other changes

# Other changes and clarifications for version 1.1

- Revision of the definition for RoT Service
  - Distinguishing between a generic RoT Service, and one that is implemented in a Secure Partition
- Clarification of the rules for a PSA RoT Service
  - Version 1.0 had conflicting rules about RoT Services that are deployed in the PSA Root of Trust
- Replacement of the term ‘reverse handle’ with ‘rhandle’ to refer to this feature
- Relaxation of the memory access rules for Constant data
  - Permitting execute access to Constant data is an acceptable approach in some systems
- Making the PSA Lifecycle API available to the NSPE
- Permitting the use of symbolic definitions for stack_size and heap_size attributes in the manifest file

## Next steps

- Public release of the PSA FF-M 1.1 Extensions alpha specification
- Please provide feedback on the 1.1 Extensions in the TF-M mailing list, or to arm.psa-feedback@arm.com
- Feedback and future updates to the specification can be discussed on the TF-M mailing list, and will also feature in future Technical Forums

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