ARM SCP firmware porting

Marek Vasut

February 1st, 2026

SCP – motivation – why

- Cortex-A core(s) running Linux or another full OS
- Optional Cortex-M core(s) running RTOS
- Possibly other cores
- All cores share resources, clock, pin controller, RAM, ...
- Traditional embedded systems:
  - Cortex-A is the primary controller of the system
  - Shared resources depend on well behaved components
- Contemporary embedded systems:
  - Cortex-A is one part of the system, so is Cortex-M RTOS ...
  - Shared resources cannot depend on well behaved components
  - Central arbiter for resource access – SCP

SCP – implementation – what

- SCP is an abbreviation for System Control Processor
- SCP is another core in the system, often Cortex-M or Cortex-R
- SCP runs firmware, which exposes interfaces for other cores
- SCP implements resource access policy
- Other cores interact with SCP to configure pinmux, clock, ...
- Other cores cannot directly configure pinmux, clock, ...

SCP – interfaces – how

- SCP exposes interfaces through which cores communicate with it
- Communication channel often some sort of mailbox and SHMEM
- Communication is bidirectional, A2P and P2A
- A2P – Agent to Platform (Linux to SCP, requests and responses)
- P2A – Platform to Agent (SCP to Linux, notifications)
- Protocol on top, usually SCMI

SCMI

- SCMI – System Control and Management Interface (clock, pinmux, regulators . . .)
- ARM DEN 0056 [LINK]
- SCMI contains multiple protocols in it, is discoverable, and can be extended with vendor extras
- SCMI protocols are request/response based, each have a few commands and parameters
- SCMI base protocol – Contains version, used for protocol discovery
- SCMI PD, system PM, performance, clock, sensor, reset, voltage . . . protocols
- SCMI protocols use IDs to identify its objects (clock IDs, reset IDs), this is exposed to other agents and is therefore a firmware ABI!

SCP firmware

- Software that runs on the SCP
- The other side of the SCMI link, handles SCMI requests
- Handles general platform management
- Responsible for request synchronization and concensus
- Various implementations exist, some closed, some open
- ARM SCP firmware is BSD-3-Clause [LINK]

ARM SCP firmware

- Sources at [LINK]
- SCP implementation meant to run mainly on Cortex-M
- Largely self-contained, but depends on arm-none- toolchain and newlib
- Base code is simple, set up the Cortex-M and enter main loop
- Every extension to the base is added via modules
- Modules implement various SCMI protocols, power management, all of it
- Many readily available modules are in tree

ARM SCP firmware port options

- SCP does platform initialization:
  - SCP acts as BL2
  - Requires much more code
  - SCP build process generates two payloads
  - Better leave BL2 to U-Boot SPL ...
- SPL is started after platform initialization:
  - SCP acts as SCP only
  - Requires less code, is less complicated
  - SCP build process generate only SCP payload
  - This is used further in this text

Terminology

- Read `doc/framework.md` for more details, in short:
- Framework ... Common SCP code
- Module ... Encapsulated generic code for driver/service/...
  (e.g. UART driver)
- Element ... Instance of module
  (e.g. UART driver instance for `uart@0x12340000`)
- Event ... Message queue and passing between elements
- Notifications ... Message broadcast from modules
  (special event)

Porting ARM SCP firmware

- Clone sources at [LINK]
- Set up matching toolchain, arm-none-eabi- is also in Debian
- The easy next step is to fork existing `product`:
  - `product/synquacer/` is a good choice
  - Synquacer SCP is simple, meant for Cortex-M3
  - Synquacer SCP is built as both BL2 and SCP, ignore `scp_romfw` BL2
  - The `scp_ramfw` is a good starting point template
  - Duplicate `product/synquacer/` into `product/yourboard/`, rename as needed
  - Select CPU core in `product/yourboard/scp_ramfw/Toolchain*`
- Use git, create checkpoints often:
  `git add -u ; git commit -sm checkpoint`
- Compile the renamed result, to verify it builds

```
git clean -fqdx
make -j$(nproc) -f Makefile.cmake PRODUCT=yourboard MODE=release
```

Init process

- Boot has two phases, pre-runtime and runtime
- Pre-runtime contains Module/Element init, bind and start
- Runtime is the main loop
- Most things during porting go wrong during pre-runtime
- The interesting core files for Cortex-M are `arch/arm/arm-m/src/arch_main.c main()` and `framework/src/fwk_arch.c fwk_arch_init()`
- The `main()` function calls `fwk_arch_init()`
- The `fwk_arch_init()` does init work and ultimately lands in main loop `__fwk_run_main_loop()`

Early printing

- When porting SCP, it is helpful to get early signs of life
- SCP has logging facility, but it becomes available too late to debug early stages
- SCP logging facility does not print immediatelly, which makes printf() debugging harder
- Make use of the non-BL2 port, let BL2 initialize UART and simply feed data into UART TX FIFO
- Roll your own custom print function

```
fwk_mmio_write_32(UART_TX_FIFO_ADDR, 'x');
// Poll for TX FIFO empty, to assure characters is out of FIFO
fwk_mmio_write_32(UART_TX_FIFO_ADDR, 'y');
// Poll for TX FIFO empty, to assure characters is out of FIFO
fwk_mmio_write_32(UART_TX_FIFO_ADDR, 'z');
// Poll for TX FIFO empty, to assure characters is out of FIFO
fwk_mmio_write_32(UART_TX_FIFO_ADDR, '\r');
// Poll for TX FIFO empty, to assure characters is out of FIFO
fwk_mmio_write_32(UART_TX_FIFO_ADDR, '\n');
// Poll for TX FIFO empty, to assure characters is out of FIFO
```

Create UART driver module I

- UART driver module goes into `product/yourboard/module/uart`
- Do not forget `Module.cmake` and `CMakeLists.txt` to build the module
- Stream adapter – module logging facility

```
const struct fwk_module module_uart = {
    .type = FWK_MODULE_TYPE_DRIVER, // ...... Module type -- driver

    .init = mod_uart_init, // ............... Module init callback
    .element_init = mod_uart_element_init, // Element init callback

    .adapter = (struct fwk_io_adapter){ // .. Stream adapter
        .open = mod_uart_io_open,
        .putch = mod_uart_io_putch,
        .close = mod_uart_close,
    },
};
```

Create UART driver module II

```c
static int mod_uart_init(fwk_id_t module_id, unsigned int element_count, const void *data)
{
    /* Module init on boot */
    return FWK_SUCCESS;
}

static int mod_uart_element_init( fwk_id_t element_id, unsigned int unused, const void *data)
{
    /* Hardware instance init on boot */
    return FWK_SUCCESS;
}

static int mod_uart_io_open(const struct fwk_io_stream *stream)
{
    /* Start the hardware instance */
    return FWK_SUCCESS;
}

int mod_uart_io_putch(const struct fwk_io_stream *stream, char ch)
{
    /* Write character to FIFO */
    return FWK_SUCCESS;
}

int mod_uart_close(const struct fwk_io_stream *stream)
{
    /* Flush FIFO etc. */
    return FWK_SUCCESS;
}
```

Instantiate UART driver element I

- Include module in `product/yourboard/scp_ramfw/Firmware.cmake`
- Include module config in `product/yourboard/scp_ramfw/CMakeLists.txt`
- Implement module config

```
# CMakeLists.txt
target_sources(
    yourboard-bl2
    PRIVATE "${CMAKE_CURRENT_SOURCE_DIR}/config_uart.c"
    ...
```

```
# Firmware.cmake
list(PREPEND SCP_MODULE_PATHS "${CMAKE_CURRENT_LIST_DIR}/../module/uart")
list(APPEND SCP_MODULES "uart")
...
```

Instantiate UART driver element II

- Include module in `product/yourboard/scp_ramfw/Firmware.cmake`
- Include module config in `product/yourboard/scp_ramfw/CMakeLists.txt`
- Implement module config

```
#include <fwk_element.h>
#include <fwk_id.h>
#include <fwk_macros.h>
#include <fwk_module.h>

struct fwk_module_config config_uart = {
    .elements = FWK_MODULE_STATIC_ELEMENTS({
        [0] = {
            .name = "UART0",
            .data = &((struct mod_plat_user_element_cfg) {
                /* Use these passed data in module */
            }),
        },

        [1] = { 0 }, // Sentinel
    }),
};
```

Immediate printing using facilities

- Test the UART module, print something
- It is possible to force print outside of logging facilities
- This is a hack:

```c
// Print buffer is local and on stack
char pb[256];
// Print string into print buffer
snprintf(pb, 256, "%s[%d]\r\n", __func__, __LINE__);
// Push the result out through the UART driver
fwk_io_puts(fwk_io_stdout, pb);
```

Next steps

- Implement mailbox driver to communicate with other CPUs
- Implement clock, power domain, . . . drivers
- Include generic "transport" and "scmi" modules
- Instantiate generic modules which includes SCMI base protocol
- Depending on what SCMI protocols are needed:
  - Implement driver for that IP and instantiate it
  - Include and instantiate generic "scmi-*" module
  - Connect the generic SCMI code with IP

Conclusion

- ARM SCP firmware port to existing hardware SCP is possible
- Code is publicly available, BSD-3-Clause
- Port can be implemented incrementally
- Be mindful of SCMI protocol ID allocation, this is ABI

End

Thank you for your attention

Marek Vasut <marek.vasut+fosdem26@mailbox.org>