RM00285

uPower API Reference Manual

Rev. 1 — 22 December 2023 **Reference manual**

Document information

Information	Content
Keywords	i.MX, Linux, uPower, API, RM00285
Abstract	This API is intended to be used by the OS drivers (Linux, FreeRTOS, etc.) as well as bare metal drivers to command and use services from the uPower. It aims to be OS-independent.

1 uPower API Overview and Concepts

This API is intended to be used by the OS drivers (Linux, FreeRTOS, etc.) as well as bare metal drivers to command and use services from the uPower. It aims to be OS-independent.

The API functions fall in three categories:

- Initialization/start-up
- Service requests
- Auxiliary

The communication with the uPower is mostly made through the Message Unit (MU) IP. uPower provides one MU for each CPU cluster in a different power domain. An API instance runs on each CPU cluster.

The API assumes each SoC power domain/CPU cluster receives two interrupts from the uPower MU:

- TX/RX, which is issued on both transmission and reception.
- Exception interrupt, to handle critical alarms, catastrophic errors, etc. This interrupt should have a high priority, preferably a Non-Maskable Interrupt (NMI).

The normal uPower operation is done by service requests. There is an API function for each service request, and all service requests send back a response, at least to indicate success/failure. The service request functions are non-blocking, and their completion can be tracked in two ways:

- By a callback registered when the service request call is made by passing the callback function pointer. A NULL pointer may be passed, in which case no callback is made.
- By polling, using the auxiliary functions `upwr_req_status` or `upwr_poll_req_status`. Polling must be used if no callback is registered, but callbacks and polling are completely independent.

Note: *A service request must not be started from a callback.*

uPower service requests are classified in Service Groups. Each Service Group has a set of related functions, named `upwr_XXX_`, where XXX is a 3-letter service group mnemonic. The service groups are:

- Exception Service Group: `upwr_xcp_*`
  Gathers functions that deal with errors and other processes outside the functional scope.
- Power Management Service Group: `upwr_pwm_*`
  Functions to control switches, configure power modes, set internal voltage, etc.
- Delay Measurement Service Group: `upwr_dlm_*`
  Delay measurements function using the process monitor and delay meter.
- Voltage Measurement Service Group: `upwr_vtm_*`
  Functions for voltage measurements, comparisons, alarms, power meter, and setting PMIC rail voltage.
- Temperature Measurement Service Group: `upwr_tpm_*`
  Functions for temperature measurements, comparisons, and alarms.
- Current Measurement Service Group: `upwr_crm_*`
  Functions for current and charge measurement.
- Diagnostic Service Group: `upwr_dgn_*`
  Functions for log configuration and statistics collecting.

Service requests follow this "golden rule": No two requests run simultaneously for the same service group, on the same domain.

They can run simultaneously on different domains (RTD/APD), and can also run simultaneously if they belong to different service groups (even on the same domain). Therefore, requests to the same service group on the same domain must be serialized. A service request call returns error if there is another request on the same service group pending, waiting a response (on the same domain).

A request for continuous service does not block the service group. For instance, a request to "measure the temperature each 10 milliseconds" responds quickly, unlocks the service group, and the temperature continues to be measured as requested, every 10 milliseconds from then on.

Service Groups have a fixed priority in the API, from higher to lower:

1. Exception
2. Power Management
3. Delay Measurement
4. Voltage Measurement
5. Current Measurement
6. Temperature Measurement
7. Diagnostics

The priority above only affects the order in which requests are sent to the uPower firmware: request to the higher priority Service Group is sent first, even if the call was made later, if there is an MU transmission pending, blocking it. The service priorities in the firmware depend on other factors. It is outside the scope of this document.

Services are requested using API functions. A service function returns with no error if a request is successfully made, but it doesn't mean the service was completed. The service is executed asynchronously, and returns a result (at least success/fail) through a callback or polling for service status. The possible service response codes are:

- `UPWR_RESP_OK = 0`: No error.
- `UPWR_RESP_SG_BUSY`: The service group is busy.
- `UPWR_RESP_SHUTDOWN`: The services are not up or shutting down.
- `UPWR_RESP_BAD_REQ`: Invalid request (usually invalid arguments).
- `UPWR_RESP_BAD_STATE`: The system state does not allow to perform the request.
- `UPWR_RESP_UNINSTALLD`: The service or function is not installed.
- `UPWR_RESP_UNINSTALLED`: The service or function is not installed (alias).
- `UPWR_RESP_RESOURCE`: The resource is not available.
- `UPWR_RESP_TIMEOUT`: The service times out.

`upwr_callb`

Generic function pointer for a request return callback.

Prototype:

```
typedef void (*upwr_callb)(upwr_sg_t     sg,
         uint32_t      func,
         upwr_resp_t   errcode,
         int           ret);
```

Arguments:

- `sg`: request service group.
- `func`: service request function ID.
- `errcode`: error code.
- `ret`: return value, if any. Note that a request may return a value even if service error is returned (`errcode != UPWR_RESP_OK`). This depends on the specific service.

Context: no sleep, no locks taken/released.

Return: none (void).

1.1 Initialization and configuration

A reference uPower initialization sequence goes as follows:

1. The host CPU calls `upwr_init`.
2. (Optional) The host checks the ROM version and SoC code calling `upwr_vers()` and optionally performs any configuration or workaround accordingly.
3. The host CPU calls `upwr_start` to start the uPower services, passing a service option number. If no RAM code is loaded or it has no service options, the launch option number passed must be 0, which will start the services available in ROM. `upwr_start` also receives a pointer to a callback called by the API when the firmware is ready to receive service requests. The callback may be replaced by polling, calling `upwr_req_status` in a loop or `upwr_poll_req_status`. In this case, the callback pointer may be NULL. A host may call `upwr_start` even if the services were already started by any host: If the launch option is the same, the response will be ok, but will indicate error if the services were already started with a different launch option.
4. The host waits for the callback calling, or polling finishing. If no error is returned, it can start making service calls using the API.

Variations on that reference sequence are possible:

- The uPower services can be started using the ROM code only, which includes the basic Power Management services, among others, with launch option number = 0. The code RAM can be loaded while these services are running and, when the loading is done, the services can be re-started with these 2 requests executed in order: `upwr_xcp_shutdown` and `upwr_start`, using the newly loaded RAM code (launch option > 0).

Note:

The initialization call `upwr_init` is not effective and returns error when called after the uPower services are started.

1.1.1 upwr_init

API initialization; must be the first API call after reset.

Prototype:

```
typedef void* (*upwr_malloc_ptr_t)(long unsigned int); /* malloc function ptr */
typedef void* (*upwr_phyadr_ptr_t)(const void*);   /* pointer->physical address
                                                      conversion function ptr */
/*
 * upwr_lock_ptr_t: pointer to a function that prevents MU interrupts
 * (if argrument lock=1) or allows it (if argument lock=0).
 * The API calls this function to make small specific code portions thread safe.
 * Only MU interrupts must be avoided, the code may be suspended for other
 * reasons.
typedef void  (*upwr_lock_ptr_t)(int lock);
typedef void (*upwr_isr_callb)(void);
typedef void (*upwr_inst_isr_ptr_t)(upwr_isr_callb txrx_isr,
           upwr_isr_callb excp_isr);
int upwr_init(  soc_domain_t              domain,
    struct MU_tag*             muptr,
    const  upwr_malloc_ptr_t   mallocptr,
    const  upwr_phyadr_ptr_t   phyadrptr,
    const  upwr_inst_isr_ptr_t isrinstptr,
    const  upwr_lock_ptr_t     lockptr);
```

Arguments:

- `domain`: SoC-dependent CPU domain ID; identifier used by the firmware in many services. Defined by SoC-dependent type `soc_domain_t` found in `upower_soc_defs.h`.
- `muptr`: pointer to the MU instance.
- `mallocptr`: pointer to the memory allocation function.
- `physaddrptr`: pointer to the function to convert pointers to physical addresses. If NULL, no conversion is made (pointer=physical address).
- `isrinstptr`: pointer to the function to install the uPower ISR callbacks. The function receives the pointers to the MU TX/RX and Exception ISRs callbacks, which must be called from the actual system ISRs. The function pointed by `isrinstptr` must also enable the interrupt at the core/interrupt controller, but must not enable the interrupt at the MU IP. The system ISRs are responsible for dealing with the interrupt controller, performing any other context save/restore, and any other housekeeping.
- `lockptr`: pointer to a function that prevents MU interrupts (if argument=1) or allows it (if argument=0). The API calls this function to make small specific code portions thread safe. Only MU interrupts must be avoided, the code may be suspended for other reasons. If no MU interrupts can happen during the execution of an API call or callback, even if enabled, for some other reasons (e.g., interrupt priority), then this argument may be NULL.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if failed to allocate memory, or use some other resource.
- **-2** if any argument is invalid.
- **-3** if failed to send the ping message.
- **-4** if failed to receive the initialization message, or was invalid.

1.1.2 upwr_start

Starts the uPower services.

Prototype:

```
typedef void (*upwr_rdy_callb)(uint32_t vmajor,uint32_t vminor,uint32_t vfixes);
int upwr_start(uint32_t                launchopt,
    const upwr_rdy_callb    rdycallb);
```

Arguments:

- `launchopt`: a number to select between multiple launch options, that may define, among other things, which services will be started, or which services implementations, features, etc. `launchopt = 0` selects a subset of services implemented in ROM. Any other number selects service sets implemented in RAM, launched by the firmware function `ram_launch`. If an invalid launchopt value is passed, no services are started, and the callback returns error (see below).
- `rdycallb`: pointer to the callback to be called when the uPower is ready to receive service requests. NULL if no callback needed. The callback receives as arguments the RAM firmware version numbers. If all 3 numbers (`vmajor`, `vminor`, `vfixes`) are **0**, that means the service launching failed. Firmware version numbers will be the same as ROM if `launchopt = 0`, selecting the ROM services.

`upwr_start` can be called by any domain even if the services are already started: It has no effect, returning success, if the launch option is the same as the one that actually started the service, and returns error if called with a different option.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not. A callback may not be registered (NULL pointer), in which case

polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_EXCEPT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if a resource failed.
- **-2** if the domain passed is the same as the caller.
- **-3** if called in an invalid API state.

1.2 Exception service group

1.2.1 upwr_xcp_config

Applies general uPower configurations.

Prototype:

```
int upwr_xcp_config(const upwr_xcp_config_t* config, const upwr_callb callb);
```

Arguments:

- `config`: pointer to the uPower SoC-dependent configuration struct `upwr_xcp_config_t` defined in `upower_soc_defs.h`. NULL may be passed, meaning a request to read the configuration, in which case it appears in the callback argument ret, or can be pointed by argument retptr in the `upwr_req_status` and `upwr_poll_req_status` calls, casted to `upwr_xcp_config_t`.
- `callb`: pointer to the callback to be called when the uPower has finished the configuration, or NULL if no callback needed (polling used instead).

Some configurations are targeted for a specific domain (see the struct `upwr_xcp_config_t` definition in `upower_soc_defs.h`). This call has implicit domain target (the same domain from which is called).

The return value is always the current configuration value, either in a read-only request (`config = NULL`) or after setting a new configuration (`non-NULL config`).

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not. A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_EXCEPT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.2.2 upwr_xcp_sw_alarm

Makes uPower issue an alarm interrupt to given domain.

Prototype:

```
int upwr_xcp_sw_alarm(soc_domain_t     domain,
          upwr_alarm_t     code,
```

```
            const upwr_callb callb);
```

Arguments:

- `domain`: identifier of the domain to alarm. Defined by SoC-dependent type `soc_domain_t` found in `upower_soc_defs.h`.
- `code`: alarm code. Defined by `SoC-dependent` type `upwr_alarm_t` found in `upower_soc_defs.h`.
- `callb`: pointer to the callback to be called when the uPower has finished the alarm, or NULL if no callback needed (polling used instead).

The function requests the uPower to issue an alarm of the given code as if it had originated internally. This service is useful mainly to test the system response to such alarms, or to make the system handle a similar alarm situation detected externally to uPower.

The system ISR/code handling the alarm may retrieve the alarm code by calling the auxiliary function `upwr_alarm_code`.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not. A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_EXCEPT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.2.3 upwr_xcp_set_ddr_retention

Cortex-M33/A35 can use this API to set/clear DDR retention.

Prototype:

```
int upwr_xcp_set_ddr_retention(soc_domain_t     domain,
                        uint32_t enable,
                        const upwr_callb callb);
```

Arguments:

- `domain`: identifier of the caller domain. `soc_domain_t` found in `upower_soc_defs.h`.
- `enable`: **true** means to set ddr retention, and **false** means to clear DDR retention.
- `callb`: NULL.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_EXCEPT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.2.4 upwr_xcp_set_rtd_use_ddr

Cortex-M33 calls this API to inform uPower that Cortex-M33 is using DDR.

Prototype:

```
int upwr_xcp_set_rtd_use_ddr(soc_domain_t     domain,
                        uint32_t enable,
                        const upwr_callb callb);
```

Arguments:

- `domain`: identifier of the caller domain. `soc_domain_t` found in `upower_soc_defs.h`.
- `enable`: not **0**, **true** means that RTD is using DDR. **0**, **false** means that RTD is not using DDR.
- `callb`: NULL

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_EXCEPT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.2.5 upwr_xcp_set_rtd_apd_llwu

Cortex-M33/A35 can use this API to set/clear `rtd_llwu` or `apd_llwu`.

Prototype:

```
int upwr_xcp_set_rtd_apd_llwu(soc_domain_t     domain,
                        uint32_t enable,
                        const upwr_callb callb);
```

Arguments:

- `domain`: set which domain (RTD_DOMAIN, APD_DOMAIN) LLWU. `soc_domain_t` found in `upower_soc_defs.h`.
- `enable`: **true** means to set `rtd_llwu` or `apd_llwu`, **false** clear `rtd_llwu` or `apd_llwu`.
- `callb`: NULL

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_EXCEPT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.2.6 upwr_xcp_shutdown

Shuts down all uPower services and power mode tasks.

Prototype:

```
int upwr_xcp_shutdown(const upwr_callb callb);
```

Arguments:

- `callb`: pointer to the callback to be called when the uPower has finished the shutdown, or NULL if no callback needed (polling used instead).

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not. A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_EXCEPT` as the service group argument.

At the callback the uPower/API is back to initialization/start-up phase, so service request calls return error.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.2.7 upwr_xcp_i2c_access

Performs an access through the uPower I2C interface.

Prototype:

```
int upwr_xcp_i2c_access(uint16_t         addr,
      int8_t           data_size,
      uint8_t          subaddr_size,
      uint32_t         subaddr,
      uint32_t         wdata,
      const upwr_callb callb);
```

Arguments:

- `addr`: I2C slave address, up to 10 bits.
- `data_size`: determines the access direction and data size in bytes, up to 4. Negetive `data_size` determines a read access with size `-data_size`. Positive `data_size` determines a write access with size `data_size`. `data_size=0` is invalid, making the service return error `UPWR_RESP_BAD_REQ`.
- `subaddr_size`: size of the sub-address in bytes, up to 4. If `subaddr_size` = 0, no subaddress is used.
- `subaddr`: sub-address, only used if `subaddr_size` > 0.
- `wdata`: write data, up to 4 bytes. Ignored if `data_size` < 0 (read).
- `callb`: pointer to the callback to be called when the uPower has finished the access, or NULL if no callback needed (polling used instead).

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not. A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_EXCEPT` as the service group argument.

The service performs a read (`data_size` < 0) or a write (`data_size` > 0) of up to 4 bytes on the uPower I2C interface. The data read from I2C comes through the callback argument ret, or written to the variable pointed by retptr, if polling is used (calls `upwr_req_status` or `upwr_poll_req_status`). `ret` (or `*retptr`) also returns the data written on writes.

Sub-addressing is supported, with sub-address size determined by the argument `subaddr_size`, up to 4 bytes. Sub-addressing is not used if `subaddr_size` = 0.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.2.8 upwr_xcp_set_mipi_dsi_ena

Sets/Clears MIPI DSI ENA.

Prototype:

```
int upwr_xcp_set_mipi_dsi_ena(soc_domain_t     domain,
                        uint32_t enable,
                        const upwr_callb callb)
```

Arguments:

- `domain`: identifier of the caller domain. `soc_domain_t` found in `upower_soc_defs.h`.
- `enable`: **true** means to set `mipi_dsi_ena`; **false** means to clear `mipi_dsi_ena`.
- `callb`: pointer to the callback to be called when the uPower has finished the access, or NULL if no callback needed (polling used instead).
  A callback can be optionally registered, and is called upon the arrival of the request response from the uPower firmware, telling if it succeeds or not. A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling upwr_req_status or upwr_poll_req_status, using UPWR_SG_EXCEPT as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.2.9 upwr_xcp_get_mipi_dsi_ena

Gets MIPI DSI ENA status.

Prototype:

```
int upwr_xcp_get_mipi_dsi_ena(soc_domain_t domain, const upwr_callb callb)
```

Arguments:

- `domain`: identifier of the caller domain. `soc_domain_t` found in `upower_soc_defs.h`.
- `callb`: pointer to the callback to be called when the uPower has finished the access, or NULL if no callback needed (polling used instead).

A callback can be optionally registered, and is called upon the arrival of the request response from the uPower firmware, telling if it succeeds or not. A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling upwr_req_status or upwr_poll_req_status, using UPWR_SG_EXCEPT as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.2.10 upwr_xcp_set_osc_mode

Sets uPower OSC mode.

Prototype:

```
int upwr_xcp_get_mipi_dsi_ena(soc_domain_t domain, const upwr_callb callb)
```

Arguments:

- `domain`: identifier of the caller domain. `soc_domain_t` found in `upower_soc_defs.h`.
- `osc_mode`: 0 means low frequency; not 0 means high frequency.
- `callb`: pointer to the callback to be called when the uPower has finished the access, or NULL if no callback needed (polling used instead).
  A callback can be optionally registered, and is called upon the arrival of the request response from the uPower firmware, telling if it succeeds or not. A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling upwr_req_status or upwr_poll_req_status, using UPWR_SG_EXCEPT as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.3 Power management service group

1.3.1 upwr_pwm_dom_power_on

Commands uPower to power on the platform of other domain (not necessarily its core(s)); does not release the core reset.

Prototype:

```
int upwr_pwm_dom_power_on(soc_domain_t      domain,
                          int               boot_start,
                          const upwr_callb  pwroncallb);
```

Arguments:

- `domain`: identifier of the domain to power on. Defined by SoC-dependent type `soc_domain_t` found in `upower_soc_defs.h`.

- `boot_start`: must be **1** to start the domain core(s) boot(s), releasing its (their) resets, or **0** otherwise.
- `pwroncallb`: pointer to the callback to be called when the uPower has finished the power on procedure, or NULL if no callback needed (polling used instead).

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not. A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_PWRMGMT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.3.2 upwr_pwm_boot_start

Commands uPower to release the reset of other CPU(s), starting their boots.

Prototype:

```
int upwr_pwm_boot_start(soc_domain_t domain, const upwr_callb bootcallb);
```

Arguments:

- `domain`: identifier of the domain to release the reset. Defined by SoC-dependent type `soc_domain_t` found in `upower_soc_defs.h`.
- `bootcallb`: pointer to the callback to be called when the uPower has finished the boot start procedure, or NULL if no callback needed (polling used instead).

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not. A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_PWRMGMT` as the service group argument.

The callback calling does not mean the CPUs boots have finished: it only indicates that uPower released the CPUs resets, and can receive other power management service group requests.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.3.3 upwr_pwm_param

Changes Power Management parameters.

Prototype:

```
int upwr_pwm_param(upwr_pwm_param_t* param, const upwr_callb callb);
```

Arguments:

- param: pointer to a parameter structure upwr_pwm_param_t, SoC-dependent, defined in upwr_soc_defines.h. NULL may be passed, meaning a request to read the parameter set, in which case it appears in the callback argument ret, or can be pointed by argument retptr in the upwr_req_status and upwr_poll_req_status calls, casted to upwr_pwm_param_t.
- callb: response callback pointer; NULL if no callback needed.

The return value is always the current parameter set value, either in a read-only request (param = NULL) or after setting a new parameter (non-NULL param).

Some parameters may be targeted for a specific domain (see the struct upwr_pwm_param_t definition in upower_soc_defs.h); this call has implicit domain target (the same domain from which is called).

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not. A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling upwr_req_status or upwr_poll_req_status, using UPWR_SG_PWRMGMT as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.3.4 upwr_pwm_chng_reg_voltage

Changes the voltage at a given regulator.

Prototype:

```
int upwr_pwm_chng_reg_voltage(uint32_t reg, uint32_t volt, upwr_callb callb);
```

Arguments:

- reg: regulator ID.
- volt: voltage value; value unit is SoC-dependent, converted from mV by the macro UPWR_VOLT_MILIV, or from micro-Volts by the macro UPWR_VOLT_MICROV, both macros in upower_soc_defs.h.
- callb: response callback pointer; NULL if no callback needed.

The function requests uPower to change the voltage of the given regulator. The request is executed if arguments are within range, with no protections regarding the adequate voltage value for the given domain process, temperature, and frequency.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling upwr_req_status or upwr_poll_req_status, using UPWR_SG_PWRMGMT as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.3.5 upwr_pwm_freq_setup

Determines the next frequency target for a given domain and current frequency.

Prototype:

```
int upwr_pwm_freq_setup(soc_domain_t domain, uint32_t rail, uint32_t stage,
 uint32_t target_freq, upwr_callb callb);
```

Arguments:

- `domain`: identifier of the domain to change frequency. Defined by SoC-dependent type `soc_domain_t` found in `upower_soc_defs.h`.
- `rail`: PMIC regulator number for the target domain.
- `stage`: DVA adjust stage refer to `upower_defs.h` "DVA adjust stage"
- `target_freq`: target adjust frequency, accurate to MHz. Refer to `upower_defs.h` structure definition `upwr_pwm_freq_msg`.
- `callb`: response callback pointer; NULL if no callback needed.

The DVA algorithm is broken down into two phases. The first phase uses a look up table to get a safe operating voltage for the requested frequency. This voltage is guaranteed to work over process and temperature.

The second step of the second phase is to measure the temperature using the uPower Temperature Sensor module. This is accomplished by doing a binary search of the TSEL bit field in the Temperature Measurement Register (TMR). The search is repeated until the THIGH bit fields in the same register change value. There are 3 temperature sensors in i.MX 8ULP (APD, AVD, and RTD).

The second phase is the fine adjust of the voltage. This stage is entered only when the new frequency requested by application was already set as well as the voltage for that frequency. The first step of the fine adjust is to find what is the current margins for the monitored critical paths, or, in other words, how many delay cells will be necessary to generate a setup-timing violation. The function informs uPower that the given domain frequency has changed or will change to the given value. uPower firmware will then adjust voltage and bias to cope with the new frequency (if decreasing) or prepare for it (if increasing). The function must be called after decreasing the frequency, and before increasing it. The actual increase in frequency must not occur before the service returns its response.

Therefore, for increase clock frequency case, users need to call this API twice, the first stage gross adjust and the second stage fine adjust.

For reduce clock frequency case, users can only call this API once, full stage (combine gross stage and fine adjust).

The request is executed if arguments are within range.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_PWRMGMT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.

- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.3.6 upwr_pwm_power_on

Powers on (not off) one or more switches and ROM/RAMs.

Prototype:

```
int upwr_pwm_power_on(const uint32_t swton[],
          const uint32_t memon[],
          upwr_callb     callb);
```

Arguments:

- `swton`: pointer to an array of words that tells which power switches to turn on. Each word in the array has 1 bit for each switch. A bit = 1 means the respective switch must be turned on, bit = 0 means it will stay unchanged (on or off). The pointer may be set to NULL, in which case no switch will be changed, unless a memory that it feeds must be turned on.
  Note: *`swton` must not point to the first shared memory address.*
- `memon`: pointer to an array of words that tells which memories to turn on. Each word in the array has 1 bit for each switch. A bit = 1 means the respective memory must be turned on, both array and periphery logic; bit = 0 means it will stay unchanged (on or off). The pointer may be set to NULL, in which case no memory will be changed.
  Note: *`memon` must not point to the first shared memory address.*
- `callb`: pointer to the callback called when configurations are applied. NULL if no callback is required.

The function requests uPower to turn on the PMC and memory array/peripheral switches that control their power, as specified above. The request is executed if arguments are within range, with no protections regarding the adequate memory power state related to overall system state.

If a memory is requested to turn on, but the power switch that feeds that memory is not, the power switch will be turned on anyway, if the pwron array is not provided (that is, if pwron is NULL).

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_PWRMGMT` as the service group argument.

Callback or polling may return error if the service contends for a resource already being used by a power mode transition or an ongoing service in another domain.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-2** if the pointer conversion to physical address failed.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.3.7 upwr_pwm_power_off

Powers off (not on) one or more switches and ROM/RAMs.

Prototype:

```
int upwr_pwm_power_off(const uint32_t swtoff[],
           const uint32_t memoff[],
           upwr_callb     callb);
```

Arguments:

- `swtoff`: pointer to an array of words that tells which power switches to turn off. Each word in the array has 1 bit for each switch. A bit = 1 means the respective switch must be turned off, bit = 0 means it will stay unchanged (on or off). The pointer may be set to NULL, in which case no switch will be changed.
  Note: *`swtoff` must not point to the first shared memory address.*
- `memoff`: pointer to an array of words that tells which memories to turn off. Each word in the array has 1 bit for each switch. A bit = 1 means the respective memory must be turned off, both array and periphery logic; bit = 0 means it will stay unchanged (on or off). The pointer may be set to NULL, in which case no memory will be changed, but notice it may be turned off if the switch that feeds it is powered off.
  Note: *`memoff` must not point to the first shared memory address.*
- `callb`: pointer to the callback called when configurations are applied. NULL if no callback is required.

The function requests uPower to turn off the PMC and memory array/peripheral switches that control their power, as specified above. The request is executed if arguments are within range, with no protections regarding the adequate memory power state related to overall system state.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_PWRMGMT` as the service group argument.

Callback or polling may return error if the service contends for a resource already being used by a power mode transition or an ongoing service in another domain.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-2** if the pointer conversion to physical address failed.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.3.8 upwr_pwm_mem_retain

Configures one or more memory power switches to retain its contents, having the power array on, while its peripheral logic is turned off.

Prototype:

```
int upwr_pwm_mem_retain(const uint32_t mem[], upwr_callb callb);
```

Arguments:

- mem: pointer to an array of words that tells which memories to put in a retention state. Each word in the array has 1 bit for each memory. A bit = 1 means the respective memory must be put in retention state, bit = 0 means it will stay unchanged (retention, fully on or off).
- callb: pointer to the callback called when configurations are applyed. NULL if no callback is required.

The function requests uPower to turn off the memory peripheral and leave its array on, as specified above. The request is executed if arguments are within range.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling upwr_req_status or upwr_poll_req_status, using UPWR_SG_PWRMGMT as the service group argument.

Callback or polling may return error if the service contends for a resource already being used by a power mode transition or an ongoing service in another domain.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-2** if the pointer conversion to physical address failed.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.3.9 upwr_pwm_chng_switch_mem

Turns on/off power on one or more PMC switches and memories, including their array and peripheral logic.

Prototype:

```
int upwr_pwm_chng_switch_mem(const struct upwr_switch_board_t  swt[],
           const struct upwr_mem_switches_t  mem[],
           upwr_callb                        callb);
```

Arguments:

- swt: pointer to a list of PMC switches to be opened/closed. The list is structured as an array of struct upwr_switch_board_t (see upower_defs.h), each one containing a word for up to 32 switches, one per bit. A bit = 1 means switch closed, bit = 0 means switch open. Struct upwr_switch_board_t also specifies a mask with 1 bit for each respective switch: mask bit = 1 means the open/close action is applied; mask bit = 0 means the switch stays unchanged. The pointer may be set to NULL, in which case no switch will be changed, unless a memory that it feeds must be turned on.
  Note: *swt must not point to the first shared memory address.*
- mem: pointer to a list of switches to be turned on/off. The list is structured as an array of struct upwr_mem_switches_t (see upower_defs.h), each one containing 2 word for up to 32 switches, one per bit, one word for the RAM array power switch, and the other for the RAM peripheral logic power switch. A bit = 1 means switch closed, bit = 0 means switch open. Struct upwr_mem_switches_t also specifies a mask with 1 bit for each respective switch: mask bit = 1 means the open/close action is applied, mask bit = 0 means the switch stays unchanged. The pointer may be set to NULL, in which case no memory switch will be changed, but notice it may be turned off if the switch that feeds it is powered off.
  Note: *mem must not point to the first shared memory address.*
- callb: pointer to the callback called when the configurations are applied. NULL if no callback is required.

The function requests uPower to change the PMC switches and/or memory power as specified above. The request is executed if arguments are within range, with no protections regarding the adequate switch combinations and overall system state.

If a memory is requested to turn on, but the power switch that feeds that memory is not, the power switch will be turned on anyway, if the swt array is not provided (that is, if swt is NULL).

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_PWRMGMT` as the service group argument.

Callback or polling may return error if the service contends for a resource already being used by a power mode transition or an ongoing service in another domain.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-2** if the pointer conversion to physical address failed.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.3.10 upwr_pwm_pmode_config

Configures a given power mode in a given domain.

Prototype:

```
int upwr_pwm_pmode_config(soc_domain_t   domain,
        abs_pwr_mode_t pmode,
        const void*    config,
        upwr_callb     callb);
```

Arguments:

- `domain`: identifier of the domain to which the power mode belongs. Defined by SoC-dependent type `soc_domain_t` found in `upower_soc_defs.h`.
- `pmode`: SoC-dependent power mode identifier defined by type `abs_pwr_mode_t` found in `upower_soc_defs.h`.
- `config`: pointer to an SoC-dependent struct defining the power mode configuration, found in `upower_soc_defs.h`.
- `callb`: pointer to the callback called when configurations are applied. NULL if no callback is required.

The function requests uPower to change the power mode configuration as specified above. The request is executed if arguments are within range, and complies with SoC-dependent restrictions on value combinations.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_PWRMGMT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-2** if the pointer conversion to physical address failed.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.3.11 upwr_pwm_reg_config

Configures the uPower internal regulators.

Prototype:

```
int upwr_pwm_reg_config(const struct upwr_reg_config_t* config,
      upwr_callb   callb);
```

Arguments:

- `config`: pointer to the struct defining the regulator configuration; the struct `upwr_reg_config_t` is defined in the file `upower_defs.h`.
- `callb`: pointer to the callback called when configurations are applied. NULL if no callback is required.

The function requests uPower to change/define the configurations of the internal regulators.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_PWRMGMT` as the service group argument.

The service may fail with error `UPWR_RESP_RESOURCE` if a power mode transition or the same service (called from another domain) is executing simultaneously. This error should be interpreted as a "try later" response, as the service will succeed once those concurrent executions are done, and no other is started.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-2** if the pointer conversion to physical address failed.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.3.12 upwr_pwm_chng_dom_bias

Changes the domain bias.

Prototype:

```
int upwr_pwm_chng_dom_bias(const struct upwr_dom_bias_cfg_t* bias,
```

```
        upwr_callb                          callb);
```

Arguments:

- `bias`: pointer to a domain bias configuration struct (see `upower_soc_defs.h`).
- `callb`: pointer to the callback called when configurations are applied. NULL if no callback is required.

The function requests uPower to change the domain bias configuration as specified above. The request is executed if arguments are within range, with no protections regarding the adequate value combinations and overall system state.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_PWRMGMT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.3.13 upwr_pwm_chng_mem_bias

Changes a ROM/RAM power bias.

Prototype:

```
int upwr_pwm_chng_mem_bias(soc_domain_t                     domain,
        const struct upwr_mem_bias_cfg_t* bias,
        upwr_callb                        callb);
```

Arguments:

- `domain`: identifier of the domain upon which the bias is applied. Defined by SoC-dependent type `soc_domain_t` found in `upower_soc_defs.h`.
- `bias`: pointer to a memory bias configuration struct (see `upower_soc_defs.h`).
- `callb`: pointer to the callback called when configurations are applied. NULL if no callback is required.

The function requests uPower to change the memory bias configuration as specified above. The request is executed if arguments are within range, with no protections regarding the adequate value combinations and overall system state.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_PWRMGMT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.4 Voltage management service group

1.4.1 upwr_vtm_pmic_cold_reset

Requests cold reset the PMIC will power cycle all the regulators.

Prototype:

```
int upwr_vtm_pmic_cold_reset(upwr_callb callb);
```

Arguments:

- `callb`: response callback pointer; NULL if no callback needed.

The function requests uPower to cold reset the PMIC. The request is executed if arguments are within range, with no protections regarding the adequate voltage value for the given domain process, temperature and frequency.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_VOLTM` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.4.2 upwr_vtm_set_pmic_mode

Requests uPower to set PMIC mode

Prototype:

```
int upwr_vtm_set_pmic_mode(uint32_t pmic_mode, upwr_callb callb);
```

Arguments:

- `pmic_mode`: The target mode needs to be set.
- `callb`: Response callback pointer; NULL if no callback needed.

The function requests uPower to set PMIC mode. The request is executed if arguments are within range, with no protections regarding the adequate voltage value for the given domain process, temperature and frequency.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_VOLTM` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.4.3 upwr_vtm_chng_pmic_voltage

Changes the voltage of a given rail.

Prototype:

```
int upwr_vtm_chng_pmic_voltage(uint32_t rail, uint32_t volt, upwr_callb callb);
```

Arguments:

- `rail`: PMIC rail ID.
- `volt`: The target voltage of the given rail, accurate to uV. If pass volt value **0**, means to power off this rail.
- `callb`: Response callback pointer; NULL if no callback needed.

The function requests uPower to change the voltage of the given rail. The request is executed if arguments are within range, with no protections regarding the adequate voltage value for the given domain process, temperature and frequency.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_VOLTM` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.4.4 upwr_vtm_get_pmic_voltage

Gets the voltage of a given ral.

Prototype:

```
int upwr_vtm_get_pmic_voltage(uint32_t rail, upwr_callb callb);
```

Arguments:

- `rail`: PMIC rail ID.
- `callb`: Response callback pointer; NULL if no callback needed. (polling used instead)

The function requests uPower to get the voltage of the given rail. The request is executed if arguments are within range, with no protections regarding the adequate voltage value for the given domain process, temperature and frequency.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_VOLTM` as the service group argument.

The voltage data read from uPower through the callback argument ret, or written to the variable pointed by retptr, if polling is used (calls `upwr_req_status` or `upwr_poll_req_status`). `ret` (or `*retptr`) also returns the data written on writes.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.4.5 upwr_vtm_power_measure

Requests uPower to measure power consumption.

Prototype:

```
int upwr_vtm_power_measure(uint32_t ssel, upwr_callb callb);
```

Arguments:

- `ssel`: This field determines which power switches will have their currents sampled to be accounted for a current/power measurement. Supports 0 - 7.

SSEL	Power switch
0	Cortex-M33 core complex/platform/peripherals
1	Fusion Core and Peripherals
2	Cortex-A35[0] core complex
3	Cortex-A35[1] core complex
4	3DGPU
5	HiFi 4
6	DDR Controller (PHY and PLL NOT included)
7	PXP, EPDC

- `callb`: Response callback pointer; NULL if no callback needed. (polling used instead)

The function requests uPower to measure current consumption. The request is executed if arguments are within range, with no protections regarding the adequate voltage value for the given domain process, temperature and frequency.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_VOLTM` as the service group argument.

The power consumption data read from uPower through the callback argument ret, or written to the variable pointed by retptr, if polling is used (calls `upwr_req_status` or `upwr_poll_req_status`). `ret` (or `*retptr`) also returns the data written on writes. uPower FW needs to support cocurrent request from Cortex-M33 and Cortex-A35.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.4.6 upwr_vtm_vmeter_measure

Requests uPower to measure voltage.

Prototype:

```
int upwr_vtm_vmeter_measure(uint32_t vdetsel, upwr_callb callb);
```

Arguments:

- `vdetsel`: Voltage Detector Selector, supports 0 - 3.
  - 00b - RTD sense point
  - 01b - LDO output
  - 10b - APD domain sense point
  - 11b - AVD domain sense point

  Refer to `upower_defs.h`.
- `callb`: Response callback pointer; NULL if no callback needed. (polling used instead)

The function requests uPower to use vmeter to measure voltage. The request is executed if arguments are within range, with no protections regarding the adequate voltage value for the given domain process, temperature, and frequency.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_VOLTM` as the service group argument.

The voltage data read from uPower through the callback argument `ret`, or written to the variable pointed by retptr, if polling is used (calls `upwr_req_status` or `upwr_poll_req_status`). `ret` (or `*retptr`) also returns the data written on writes. uPower FW needs to support cocurrent request from M33 and A35.

Refer to RM COREREGVL (Core Regulator Voltage Level) uPower return VDETLVL to user. Users can calculate the real voltage:

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.4.7 upwr_vtm_pmic_config

Configures the SoC PMIC (Power Management IC).

Prototype:

```
int upwr_vtm_pmic_config(const void* config, uint32_t size, upwr_callb callb);
```

Arguments:

- `config`: pointer to a PMIC-dependent struct defining the PMIC configuration.
- `size`: size of the struct pointed by config, in bytes.
- `callb`: pointer to the callback called when configurations are applied. NULL if no callback is required.

The function requests uPower to change/define the PMIC configuration.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_PWRMGMT` as the service group argument.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-2** if the pointer conversion to physical address failed.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower. Sample code: Please must notice that, it will take very long time to finish, because it will send many I2C commands to PMIC chip.

1.5 Temperature management service group

1.5.1 upwr_tpm_get_temperature

Requests uPower to get temperature of one temperature sensor.

Prototype:

```
int upwr_tpm_get_temperature(uint32_t sensor_id, upwr_callb callb);
```

Arguments:

- sensor_id: temperature sensor ID, supports 0 - 2.
  - 00b: RTD temperature sensor
  - 01b: APD temperature sensor
  - 10b: AVD temperature sensor
- callb: response callback pointer; NULL if no callback needed. (polling used instead).

The function requests uPower to measure temperature. The request is executed if arguments are within range, with no protections regarding the adequate voltage value for the given domain process, temperature, and frequency.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling upwr_req_status or upwr_poll_req_status, using UPWR_SG_TEMPM as the service group argument.

The temperature data read from uPower through the callback argument ret, or written to the variable pointed by retptr, if polling is used (calls upwr_req_status or upwr_poll_req_status). ret (or *retptr) also returns the data written on writes.

uPower return TSEL to the caller (M33 or A35), caller calculate the real temperature Tsh = 0.000002673049*TSEL[7:0]^3 + 0.0003734262*TSEL[7:0]^2 + 0.4487042*TSEL[7:0] - 46.98694

uPower FW needs to support concurrent request from Cortex-M33 and Cortex-A35.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.6 Delay management service group

The Critical Path Delay Meter is used for Dynamic Voltage Adjust (DVA). This module builds a replica of the eight most critical paths in the device, and therefore it is device specific. These critical paths should be built next to the actual critical paths using same cells, with the same voltage supply and similar temperature. It measures the delay of this path while varying the voltage supply.

1.6.1 upwr_dlm_get_delay_margin

Requests uPower to get delay margin.

Prototype:

```
int upwr_dlm_get_delay_margin(uint32_t path, uint32_t index, upwr_callb callb);
```

Arguments:

- path: critical path.
- index: use which delay meter.
- callb: response callback pointer; NULL if no callback needed. (polling used instead)

The function requests uPower to get delay margin. The request is executed if arguments are within range, with no protections regarding the adequate voltage value for the given domain process, temperature, and frequency.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_DELAYM` as the service group argument.

The delay margin data read from uPower via the callback argument `ret`, or written to the variable pointed by `retptr`, if polling is used (calls `upwr_req_status` or `upwr_poll_req_status`). `ret` (or `*retptr`) also returns the data written on writes. uPower FW needs to support concurrent request from Cortex-M33 and Cortex-A35.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.6.2 upwr_dlm_set_delay_margin

Requests uPower to set delay margin.

Prototype:

```
int upwr_dlm_set_delay_margin(uint32_t path, uint32_t index, uint32_t
 delay_margin, upwr_callb callb);
```

Arguments:

- `path`: critical path
- `index`: uses which delay meter
- `delay_margin`: the value of delay margin
- `callb`: response callback pointer; NULL if no callback needed. (polling used instead)

The function requests uPower to set delay margin. The request is executed if arguments are within range, with no protections regarding the adequate voltage value for the given domain process, temperature, and frequency.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_DELAYM` as the service group argument.

The result of the corresponding critical path, failed or not read from uPower through the callback argument ret, or written to the variable pointed by retptr, if polling is used (calls `upwr_req_status` or `upwr_poll_req_status`). `ret` (or `*retptr`) also returns the data written on writes. uPower FW needs to support cocurrent request from Cortex-M33 and Cortex-A35.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.6.3 upwr_dlm_process_monitor

Requests uPower to do process monitor.

Prototype:

```
int upwr_dlm_process_monitor(uint32_t chain_sel, upwr_callb callb);
```

Arguments:

- `chain_sel`: Chain Cell Type Selection Select the chain to be used for the clock signal generation. Support two types chain cell, 0 - 1.
- `callb`: response callback pointer; NULL if no callback needed. (polling used instead)

The function requests uPower to do process monitor. The request is executed if arguments are within range, with no protections regarding the adequate voltage value for the given domain process, temperature, and frequency.

A callback can be optionally registered, and will be called upon the arrival of the request response from the uPower firmware, telling if it succeeded or not.

A callback may not be registered (NULL pointer), in which case polling has to be used to check the response, by calling `upwr_req_status` or `upwr_poll_req_status`, using `UPWR_SG_DELAYM` as the service group argument.

The result of process monitor, failed or not read from uPower via the callback argument `ret`, or written to the variable pointed by `retptr`, if polling is used (calls `upwr_req_status` or `upwr_poll_req_status`). `ret` (or `*retptr`) also returns the data written on writes. uPower FW needs to support cocurrent request from Cortex-M33 and Cortex-A35.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

Note that this is not the error response from the request itself: it only tells if the request was successfully sent to the uPower.

1.7 Diagnose service group

1.7.1 upwr_dgn_mode

Sets the diagnostic mode. It is for our NXP internal use for development.

Prototype:

```
int upwr_dgn_mode(upwr_dgn_mode_t mode, const upwr_callb callb);
```

Arguments:

- `mode`: diagnostic mode, which can be:
  - `UPWR_DGN_NONE`: no diagnostic recorded
  - `UPWR_DGN_TRACE`: warnings, errors, service, internal activity recorded
  - `UPWR_DGN_SRVREQ`: warnings, errors, service activity recorded
  - `UPWR_DGN_WARN`: warnings and errors recorded
  - `UPWR_DGN_ALL`: trace, service, warnings, errors, task state recorded
  - `UPWR_DGN_ERROR`: only errors recorded
  - `UPWR_DGN_ALL2ERR`: record all until an error occurs, freeze recording on error
  - `UPWR_DGN_ALL2HLT`: record all until an error occurs, executes an ebreak on error, which halts the core if enabled through the debug interface
  - `callb`: pointer to the callback called when mode is changed. NULL if no callback is required.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-1** if service group is busy.
- **-3** if called in an invalid API state.

1.8 Auxiliary calls

1.8.1 upwr_rom_version

Informs the ROM firwmware version.

Prototype:

```
uint32_t upwr_rom_version(uint32_t *vmajor, uint32_t *vminor, uint32_t *vfixes);
```

Arguments:

- `vmajor`: pointer to the variable to get the firmware major version number.
- `vminor`: pointer to the variable to get the firmware minor version number.
- `vfixes`: pointer to the variable to get the firmware fixes number.

Context: no sleep, no locks taken/released.

Return: SoC ID.

1.8.2 upwr_ram_version

Informs the RAM firwmware version.

Prototype:

```
uint32_t upwr_ram_version(uint32_t* vminor, uint32_t *vfixes);
```

Arguments:

- `vminor`: pointer to the variable to get the firmware minor version number.
- `vfixes`: pointer to the variable to get the firmware fixes number.

The 3 values returned are **0** if no RAM firmwmare was loaded and initialized.

Context: no sleep, no locks taken/released.

Return: firmware major version number.

1.8.3 upwr_req_status

Tells the status of the service group request, and returns a request return value, if any.

Prototype:

```
/* service request status */
typedef enum {
  UPWR_REQ_OK,      /* request succeeded */
  UPWR_REQ_ERR,     /* request failed */
  UPWR_REQ_BUSY     /* request execution ongoing */
} upwr_req_status_t;
upwr_req_status_t upwr_req_status(upwr_sg_t     sg,
          uint32_t*     sgfptr,
          upwr_resp_t*  errptr,
          int*          retptr);
```

Arguments:

- `sg`: service group of the request.
- `sgfptr`: pointer to the variable that will hold the function ID of the last request completed; can be NULL, in which case it is not used.
- `errptr`: pointer to the variable that will hold the error code; can be NULL, in which case it is not used.
- `retptr`: pointer to the variable that will hold the value returned by the last request completed (invalid if the last request completed didn't return any value); can be NULL, in which case it is not used. Note that a request may return a value even if service error is returned (`*errptr != UPWR_RESP_OK`): that is dependent on the specific service.

This call can be used in a poll loop of a service request completion in case a callback was not registered.

Context: no sleep, no locks taken/released.

Return: service request status: succeeded, failed, or ongoing (busy)

1.8.4 upwr_poll_req_status

Polls the status of the service group request, and returns a request return value, if any.

Prototype:

```
upwr_req_status_t upwr_poll_req_status(upwr_sg_t     sg,
               uint32_t*     sgfptr,
               upwr_resp_t*  errptr,
               int*          retptr,
               uint32_t      attempts);
```

Arguments:

- `sg`: service group of the request .
- `sgfptr`: pointer to the variable that will hold the function id of the last request completed; can be NULL, in which case it is not used.
- `errptr`: pointer to the variable that will hold the error code; can be NULL, in which case it is not used.
- `retptr`: pointer to the variable that will hold the value returned by the last request completed (invalid if the last request completed didn't return any value); can be NULL, in which case it is not used. Note that a request

may return a value even if service error is returned (`*errptr != UPWR_RESP_OK`): that is dependent on the specific service.
- `attempts`: maximum number of polling attempts; if `attempts` > 0 and is reached with no service response received, `upwr_poll_req_status` returns `UPWR_REQ_BUSY` and variables pointed by `sgfptr`, `retptr`, and errptr are not updated; if `attempts` = 0, `upwr_poll_req_status` waits "forever".

This call can be used to poll a service request completion in case a callback was not registered.

Context: no sleep, no locks taken/released.

Return: service request status: succeeded, failed, or ongoing (busy)

1.8.5 upwr_alarm_code

Returns the alarm code of the last alarm occurrence. The value returned is not meaningful if no alarm was issued by uPower.

Prototype:

```
upwr_alarm_t upwr_alarm_code(void);
```

Context: no sleep, no locks taken/released.

Return: alarm code, as defined by the type `upwr_alarm_t` in `upwr_soc_defines.h`.

1.9 Transmit/Receive primitives

1.9.1 upwr_tx

Queues a message for transmission.

Prototype:

```
int upwr_tx(const uint32_t*         msg,
            unsigned int            size,
            UPWR_TX_CALLB_FUNC_T    callback);
```

Arguments:

- `msg` : pointer to the message sent.
- `size`: message size in 32-bit words
- `callback`: pointer to a function to be called when transmission done; can be NULL, in which case no callback is done.

This is an auxiliary function used by the rest of the API calls. It is normally not called by the driver code, unless maybe for test purposes.

Context: no sleep, no locks taken/released.

Return: number of vacant positions left in the transmission queue, or **-1** if the queue was already full when `upwr_tx` was called, or **-2** if any argument is invalid (like size off-range).

1.9.2 upwr_rx

Unqueues a received message from the reception queue.

Prototype:

```
int upwr_rx(char *msg, unsigned int *size);
```

Arguments:

- `msg`: pointer to the message destination buffer.
- `size`: pointer to variable to hold message size in 32-bit words.

This is an auxiliary function used by the rest of the API calls. It is normally not called by the driver code, unless maybe for test purposes.

Context: no sleep, no locks taken/released.

Return: number of messages remaining in the reception queue, or **-1** if the queue was already empty when `upwr_rx` was called, or **-2** if any argument is invalid (like mu off-range).

1.9.3 upwr_rx_callback

Sets up a callback for a message receiving event.

Prototype:

```
int upwr_rx_callback(UPWR_RX_CALLB_FUNC_T callback);
```

Arguments:

- `callback`: pointer to a function to be called when a message arrives; can be NULL, in which case no callback is done.

This is an auxiliary function used by the rest of the API calls. It is normally not called by the driver code, unless maybe for test purposes.

Context: no sleep, no locks taken/released.

Return:

- **0** if OK.
- **-2** if any argument is invalid (mu off-range).

1.9.4 msg_copy

Copies a message.

Prototype:

```
void msg_copy(char* dest, char* src, unsigned int size);
```

Arguments:

- `dest`: pointer to the destination message.
- `src`: pointer to the source message.
- `size`: message size in words.

This is an auxiliary function used by the rest of the API calls. It is normally not called by the driver code, unless maybe for test purposes.

Context: no sleep, no locks taken/released.

Return: none (void)

2 i.MX 8ULP-dependent uPower API Definition

This chapter describes the API definitions that are specific to the i.MX 8ULP SoC. These services are provided by uPower ROM. Sometimes, users can choose not to load uPower firmware.

2.1 Initialization and configuration

i.MX 8ULP provides only one Message Unit (MU) for each core domain: Real Time Domain (RTD) and Application Domain (APD), which has two Cortex-A35 cores. Both Cortex-A35 cores in APD must share the same API instance, meaning `upwr_init` must be called only once for each domain. The API does not provide any mutually exclusion or locking mechanism for concurrent accesses from both APD cores, so any API arbitration, if needed, must be implemented by the API user code.

A domain must not go to Power Down (PD) or Deep Power Down (DPD) power modes with any service still pending (response not received).

The following sections describe the i.MX 8ULP particularities of service calls.

2.1.1 uPower ROM function APIs

i.MX 8ULP ROM provides only the launch option 0, which has no power mode transition support and provides the following services:

- `upwr_xcp_config`
- `upwr_xcp_sw_alarm`
- `upwr_pwm_param`
- `upwr_pwm_power_on`
- `upwr_pwm_power-off`

i.MX 8ULP RAM firmware provides 2 launch options:

1. Starts all tasks, services and power mode ones. This is the full-featured firmware option.
2. Starts only the power mode tasks. Services are not available with this option, and further calls to `upwr_start` (from either domain) have no response. This option is mostly used to accelerate power mode mixed-signal simulations, and not intended to be used with silicon.

Note: *Option 0 is also available if the RAM firmware is loaded.*

2.2 Exception service group

This group services are for NXP internal debug use.

2.2.1 upwr_xcp_config()

The i.MX 8ULP uPower configuration struct contains the following bitfields:

- `ALARM_INT` (1 bit): tells which RTD MU interrupt should be used for alarms; 1= MU GPI1; 0= MU GPI0; APD alarms always use GPI0.
- `CFG_IOMUX` (1 bit): determines if uPower configures i.MX 8ULP IOMUX for I2C and mode pins used to control an external PMIC; 1= uPower firmware or PMIC driver configures i.MX 8ULP IOMUX and mode pins; 0= i.MX 8ULP IOMUX and mode pins not configured by uPower.
- `DGNBUFBITS` (4 bits): determines the diagnostic buffer size according to the formula: size = 2^(DGNBUFBITS +3) bytes.

Defaults are all zeroes. All the other bits are reserved, and must be written **0**.

2.2.2 upwr_xcp_sw_alarm()

Argument code is defined by the enum `upwr_alarm_t`, with the values:

- `UPWR_ALARM_INTERNAL`: internal software error
- `UPWR_ALARM_EXCEPTION`: uPower core exception, either illegal instruction or bus error
- `UPWR_ALARM_SLACK`: delay path too slow, meaning a timing violation occurred or is iminent.
- `UPWR_ALARM_VOLTAGE`: one of the measured voltages is below safety margins.

Note that this service emulates an alarm that would normally be issued by uPower when it detects one of the causes above. A request to alarm the APD domain when it is powered off returns success, but is ineffective.

3 Note About the Source Code in the Document

Example code shown in this document has the following copyright and BSD-3-Clause license:

4 Revision History

This table summarizes the revisions to this document.

Revision history

Document ID	Release date	Description
RM00285 v.1	22 December 2023	Initial public release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents

For more information, please visit: https://www.nxp.com

Date of release: 22 December 2023
Document identifier: RM00285