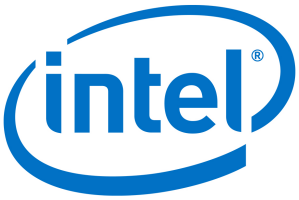


Intel® Curie™ Platform

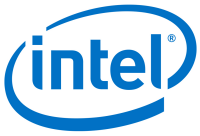
Software Architecture Guide

February 2016

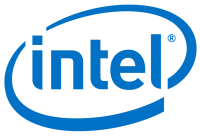
Revision 002

Intel Confidential

Document Number: 333185 rev. 002



Contents



Figures

Tables

Revision History

Revision	Description	Date
001	Initial release.	October 8, 2015
002	V2 Release updates	January 28, 2016

§

1 Introduction

This document provides a high level understanding of the Intel® Quark™ SE Software architecture. The purpose of this document is to present an overview of the different components present in Intel® Curie™ module from the software prospective.

1.1 Terminology

Table 1 Terminology

Term	Definition
ACM	Abstract control model
ADC	Analog to digital converter
ARC	Argonaut RISC core
BAS	Battery Service
BT	Bluetooth
BTLE/BLE	Bluetooth Low Energy
CPU	Central processing unit
DFU	Device firmware upgrade
DIS	Device Information Service
GPIO	General purpose input output
I2C	Inter-integrated circuit
IPC	Interprocess communication
ISPC	Intel® Software Platform for Curie™
ISPP	Intel Serial Port Protocol
JTAG	Joint Test Action Group
LED	Light emitting diode
OS	Operating system
OTA	Over-the-air Programming
OTP	One-time programmable
RTOS	Real time operating system
SOC	System on chip
SPI	Serial peripheral interface
SS	Sensor subsystem
UART	Universal asynchronous receiver/transmitter
USB	Universal serial bus

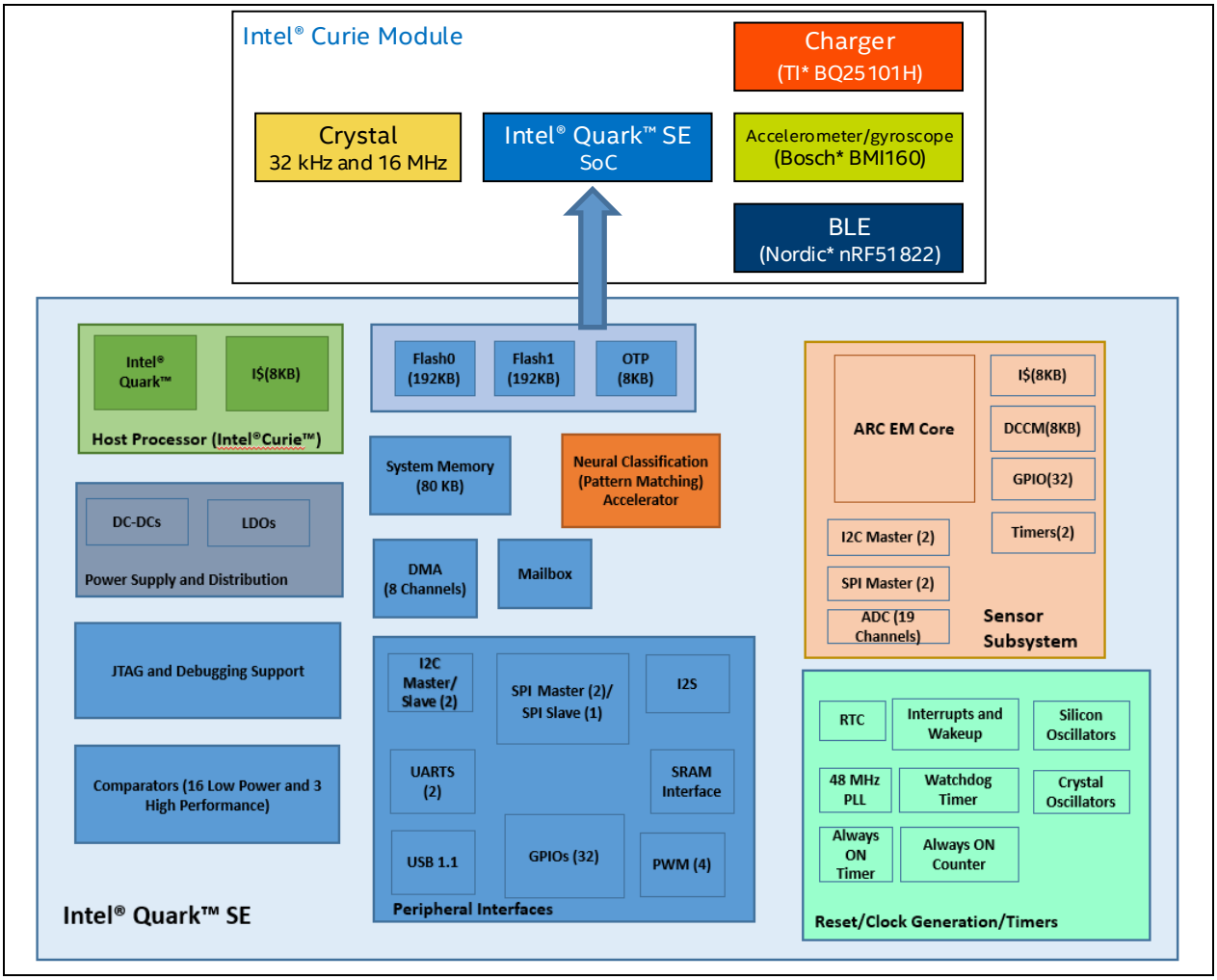
§

2 Software Architecture Overview

The Intel® Curie™ hardware module embeds Intel® Quark™ SE system-on-chip, Bluetooth low energy chip, battery charger, accelerometer/gyroscope, and crystals.

Intel® Quark™ SE is a highly integrated SoC targeted for Wearable devices. The Intel® Quark™ SE SoC contains 2 cores: one MinuteIA (Intel® Quark™) targeted for application code, and one ARC (Sensor Subsystem) for specific sensor management code. It also features a "Pattern Matching Technology", a new IP that implements a neural network aimed at executing pattern matching algorithms.

Figure 1 Intel® Curie™ hardware module and SoC overview

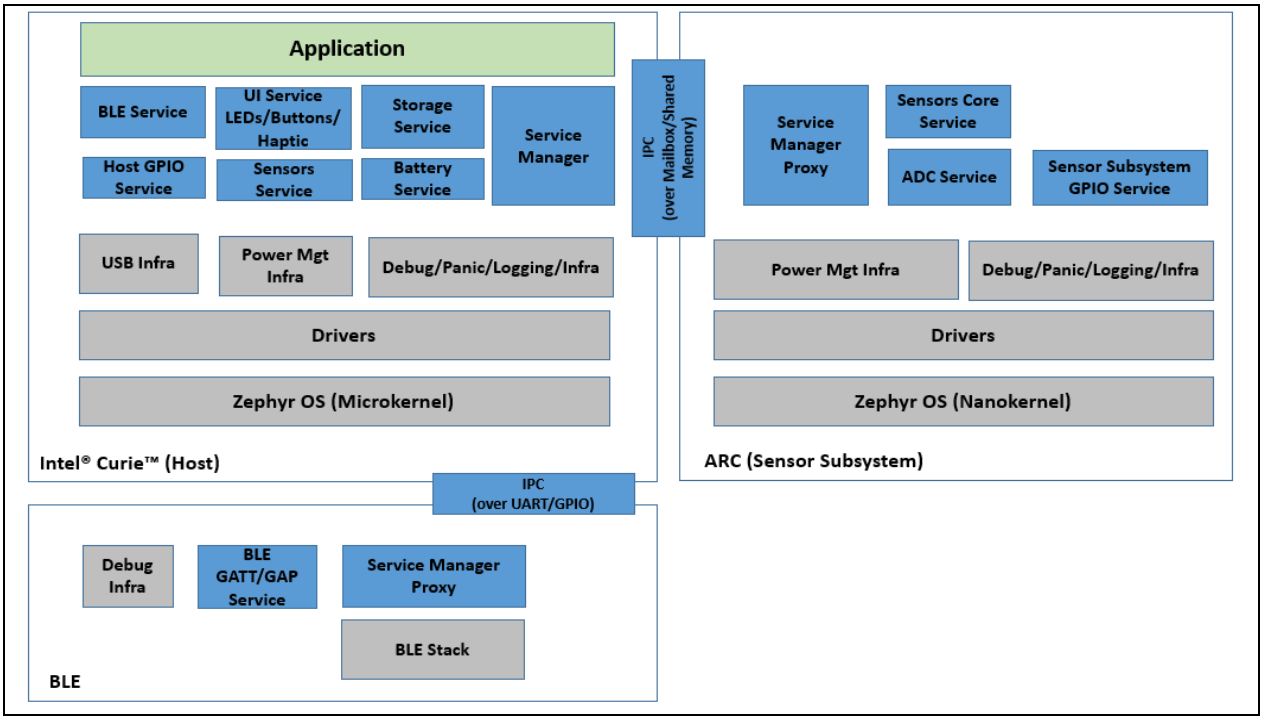


Intel® Quark™ SE software architecture is based on a component model framework. The component model allows an easy split of the software blocks into components with bounded responsibility and allows a high level of reuse of the component.

2.1 High level architecture diagram

Figure 2 shows the software architecture of the platform. Each of the three subsystems (Intel® Quark™ SE SoC, ARC, and BLE) exposes its services to the other two subsystems through a framework that abstracts where the service itself is deployed.

Figure 2 High-level software architecture



2.2 Actors description

2.2.1 Intel® Quark™ SE SoC

The Intel® Quark™ processor is the main processor inside Intel® Quark™ SE SoC, and is in charge of executing device end-user application, handling all communications with the BT chip, and managing the states of the sensor system using the ARC chip. More specifically, it provides following functionality:

- ***UI (User Interaction) Service*** for managing LEDs, buttons, haptics.
- ***Storage Service*** for managing embedded Intel® Quark™ SE on-die nonvolatile memory and SPI flash if present.
- ***BLE Service*** for managing BLE chip and exposing supported Bluetooth services and profiles.
- ***Battery Service*** for managing the battery state of charge estimation and charging state.
- ***GPIO Service*** for managing GPIOs accessible by the Intel® Quark™ processor.
- ***Power Management Infrastructure APIs*** for managing power states of the Intel® Quark™, and the Intel® Curie™ module (active, sleep, off).
- **USB Infrastructure APIs** for managing USB.
- **Debug/Panic/Logging APIs** for handling logging.

The Device OS on the Intel® Quark™ comprises of all the services that run under the supervision of the service manager, infrastructure software components, and peripheral drivers, which collectively run on the Zephyr RTOS.

Note: The Zephyr RTOS microkernel running on the Intel® Quark™ is configured in multithreaded mode.

2.2.2 ARC

The ARC processor is in charge of the sensor subsystem, and handles all sensors and ARC-only resources.

More specifically, it provides the following functionality:

- Sensor service and sensor core service for managing all sensors and sensor-related algorithms
  - Within the scope of Intel® Curie™ module, it handles the gyroscope/accelerometer.
  - Reads data from the sensor system, preprocesses the data, feeds the data to the Pattern Matching Accelerator, and provides the results to the Intel® Quark™ processor.
- ADC service for managing ADCs.
- Sensors subsystem GPIO service for managing dedicated sensors subsystem GPIOs on the ARC.
- Power management infrastructure APIs for managing power states of the ARC.
- Debug/panic/logging APIs for logging activities.

The Device OS on the ARC comprises of all the services that run under the supervision of the service manager proxy, infrastructure software components, and peripheral drivers, which collectively run on the Zephyr RTOS.

Note: Zephyr RTOS nanokernel running on the ARC is configured in single-threaded mode.

2.2.3 BLE

The Nordic BLE chip manages all Bluetooth-related functions and exposes a raw GATT (server/client)/GAP service to Intel® Quark™ SE. The BLE controller is connected with the Intel® Quark™ core using UART BLE supports peripheral and central mode concurrently.

§

3 Wearable Device Framework Software Layers

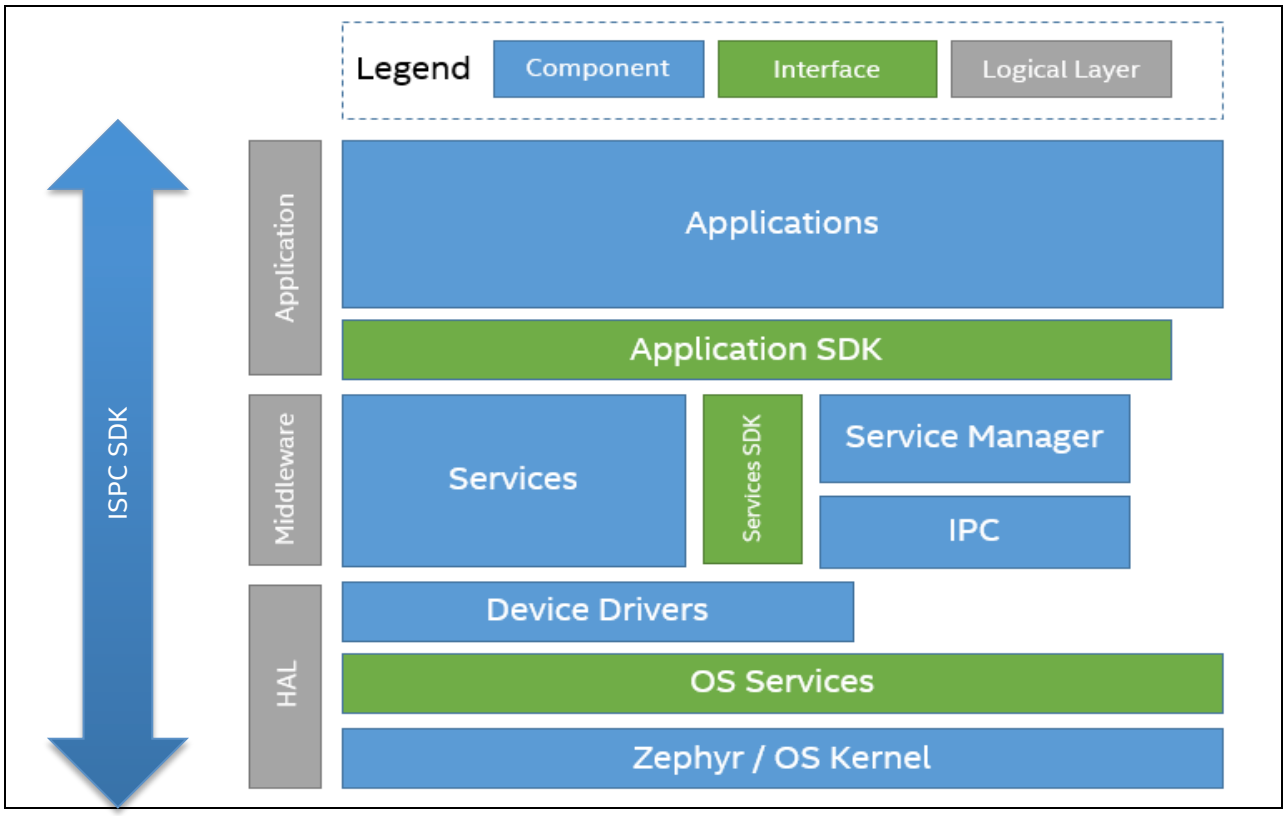
The Wearable Device software stack is a cross-project framework for embedded devices based on the Intel® Curie™ module. The framework abstracts different software layers, which helps in creating and interacting with services.

The Wearable Device software is composed of various components:

- **Hardware Abstraction Layer (HAL)**: The HAL is composed of a Zephyr RTOS kernel and its related services. The OS services abstract OS features to allow OS-agnostic high-level framework and expose APIs to be used in the middleware layer. Similarly, the drivers abstract specific device interfaces.
- **Middleware:** The middleware is again composed of various subcomponents. The main software components in this layer are services, service manager, and IPC.
- **Applications:** The application is the component that makes use of the application SDK and allows it to implement the specific behavior of the device.

The component framework (CFW) is the core of the **Wearable Device** software stack. It allows definition of services and client applications than can run virtually on any CPU of the hardware platform. The component framework is composed of the following layers, described in the following diagram and paragraphs.

Figure 3 Software layers



3.1 Application SDK / framework

The application SDK / framework is the main building block for a custom application, it consists of:

- Service APIs
- Service Manager API
- Main loop handler that wraps the messaging and exposes a functional / callback interface to the application.

An application always needs to create a thread managed by the framework in order to be interfaced with it. All communication is managed with the framework thread for interfacing with the rest of the system.

3.2 Middleware

3.2.1 Services

A ***service*** is a software component that offers a software functionality to applications. Services are multi-client and work in an asynchronous manner. It is the building block for devices applications. It abstracts features that can be hardware optimized or software implemented by hiding them behind an API. As soon as a service is registered to the platform, it is available for use on all the CPUs of the platform.

Services are composed of:

- An API, request / callback based.
- An Implementation / message based.

Service implementations are message receivers. A service should never be blocked on an IO operation and hence all services are implemented asynchronously. As an example, if a service makes use of a driver, it should make the driver send a message to itself in order to be scheduled and react to the event from the driver in its main loop. This way a service is always ready to handle requests from clients.

3.2.2 Service manager

The service manager is the software component that knows how to contact the services, allow services to be registered in the system, and allows application or services to get handles on other services.

The service manager APIs allows any client/service to connect to other services. It will establish a route between the client and the services in order to allow direct communication between client and services.

There is only one instance of the service manager on the platform. All other CPUs of the platform implement a service manager proxy that uses the IPC link to communicate with the actual service manager. The service manager has a centralized knowledge of the services that are instantiated on the platform. It is used to retrieve the route to a particular service and once this is resolved the client has a direct route to the service's implementation.

3.2.3 IPC

IPC (Inter process communication) is the layer that allows communication between cores. It defines an interface that can be implemented over the underlying physical layer. The IPC is the mechanism that allows messages to be sent across CPU boundaries. This is hidden by the service manager that knows where the requested service lies, and knows how to route the message so it can be sent to its destination.

The Intel® Quark™/ARC IPC is using shared memory in order to transport messages and Mailbox registers to interrupt the cores, whereas Intel® Quark™/BLE IPC is using UART to transport messages and GPIOs to wake up the cores from deep sleep.

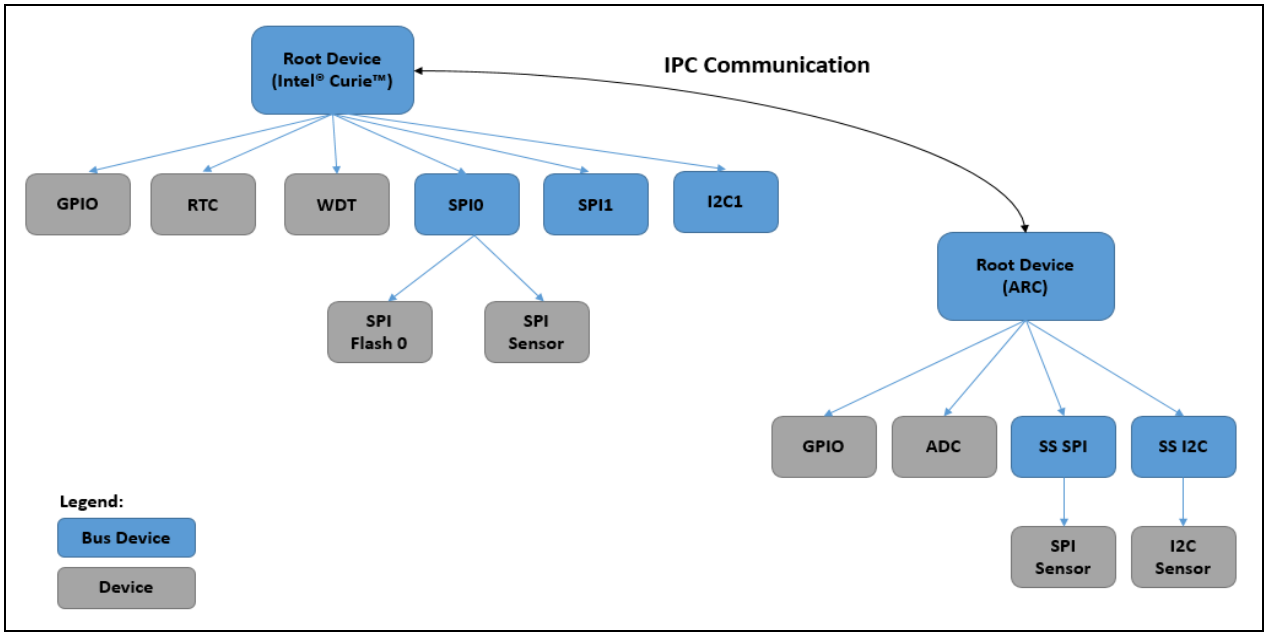
3.3 HAL

3.3.1 Device drivers

The device tree is built upon two structures:

- The *device* structure
- The *bus_device* structure

Figure



A bus device can have several (bus) device children, while a normal device cannot.

3.3.1.1 Device tree declaration

The device tree is declared only once and cannot be modified afterward (it does not support device hot plug). The user must declare one pointer to an array of *struct device* containing all the devices of the platform. The ordering of this array is very important as it must respect initialization order. For example, a spi_flash device must be positioned after the spi bus device using the following code:

```
// SPI Bus device
struct device my_spi_bus = {
    .id = SBA_SOC_SPI_0_ID,
    .driver = &my_spi_driver,
    .priv = NULL,
};
struct device my_spi_device = {
    .id = SPI_DEVICE1D,
    .driver = &my_spi_device1_driver,
    .parent = &my_spi_bus,
};
static struct device *all_platform_devices[] = {
    &my_spi_bus,
    &my_spi_device,
};
```

3.3.1.2 Implementing a device driver

When implementing a new device driver, the developer has to use the *driver* structures.

This structure contains three hooks that can be implemented if needed:

- ***init*** hook is called at platform initialization
- ***suspend*** hook is called when suspend/shutdown is requested
- ***resume*** hook is called to resume device after a deep sleep

The *driver* structure also contains a *priv* field. It is commonly used to store a pointer to a structure that handles the device data. This allows the developer to point to a custom structure, so that the driver can access the custom data. Configuration data can also be set when the structure is declared.

Here is an example for an UART driver:

```
{
    .id = UART0_PM_ID,
    .driver = &ns16550_pm_driver,
    .priv = &(struct ns16550_pm_device) {
        .uart_num = 0,
        .init_info =  &(struct uart_init_info) {
            .regs= COM1_BASE_ADRS,
            .sys_clk_freq = UART_XTAL_FREQ,
            .baud_rate = COM1_BAUD_RATE,
            .options = 0,
            .irq = COM1_INT_LVL,
            .int_pri = COM1_INT_PRI,
        },
    },
},
```

3.3.2 OS services / OS abstraction layer

The device OS Services provides an abstraction for low level kernel objects (semaphores, tasks, queues). It allows the application code and driver code to not include OS-specific definitions.

This layer defines functions to handle many OS level objects

- Semaphores (count)
- Mutex (binary)
- Scheduling
- Queues
- Interrupts
- Time/timer
- Memory allocation
- Tasks

3.3.3 Zephyr OS

The Device OS in the Intel® Curie™ platform uses Zephyr microkernel in multithreaded mode on the Intel® Quark™ and Zephyr nanokernel in single-threaded mode on the ARC core.

Execution contexts:

- **Tasks:** Tasks are preemptible execution contexts; they are the threads with a given priority. Whenever a task with higher priority than current is ready, it will preempt the current task. The microkernel allows creation of multiple tasks. The nanokernel has only one task (that calls the *main()* function).

- **Fiber:** A fiber is like a thread that is not pre-emptible. Its execution will continue until it releases the CPU by sleeping, waiting on a semaphore / mutex. Only interrupts can interrupt a fiber. All fibers are of higher priority than the task.
- **Interrupt:** An interrupt is executed in a specific context, no processing should be done inside the interrupt function. The generic pattern of an interrupt is either to signal a semaphore to wake up a sleeping fiber/ task if there is no data to pass to the non-interruptible context, or to allocate and send a message to the non-interruptible context when there is data to pass.

As the system resources are small in the Intel® Quark™ SE SoC, the number of tasks / fibers should be limited to a minimum. Sharing a queue for all the logical applications of the platform is a way to minimize the resource usage. This means that all applications are running in the same thread, by using the same queue for CFW client definition. All events will be multiplexed to the same queue. If an application needs to implement heavy processing of data, then it could make sense to create a dedicated task for this processing with a low priority so it runs when no other service / applications needs the CPU.

Fiber vs Tasks

A fiber is a lightweight, non-preemptible thread of execution that implements a portion of an application's processing. It is normally used when writing device drivers and other performance critical work.
On DSP Hub (ARC) , A Nano kernel which is Nano threaded is running, we can only use fibers for critical tasks.
Example: Sensor core is running in ***opencore_fiber***

```
int sensor_core_create()
{
                ipc_svc_core_create();
                SensorCoreInit();
                task_fiber_start(&stack[0], STACKSIZE,

   (nano_fiber_entry_t)opencore_fiber, 0, 0, 7, 0);
                return 0;
}
```

On Quark CPU which is running a Microkernel, developers have the choice to implement Tasks or Fiber. Again if tasks require to perform critical work , it is recommended to use Fiber.

Due to flash size limitation, we recommend to be diligent on thread creation.

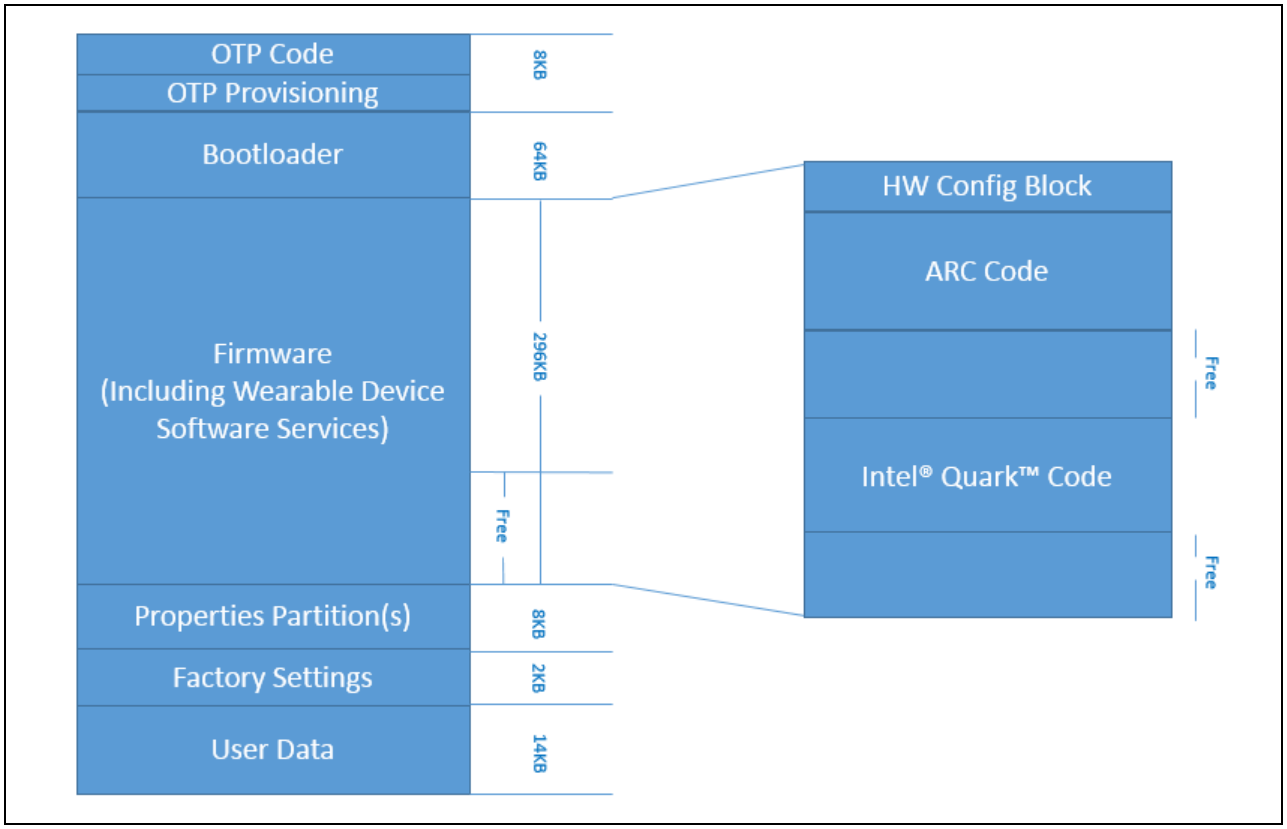
§

4 Memory/Boot/Debug System Architecture

4.1 Memory organization

The main processors (Intel® Quark™ and ARC) share the same on-die nonvolatile memory for code/data and SRAM for volatile data, and optionally an SPI flash for storing user data. The BLE chip has its own dedicated nonvolatile memory and SRAM.

Figure 4 Memory layout without external SPI flash



HW Config block (first part of the firmware flash block) consists of:

- Start address of ARC code.
- Start address of the context restoration procedure (for waking from deep sleep) on the ARC subsystem.
- Start address of Intel® Quark™ code.
- Start address of the context restoration procedure (for wake from deep sleep) for the Intel® Quark™ processor.
- SHA1 of the eventual part of FW that goes beyond 192 kB.

OTP Code/Data block consists of:

- The one-time-programming memory area for boot code/data can't be updated and can't be erased
- Intel-OTP Provisioning Data contains data that identifies the Intel® Curie™ module, keys etc. and in general all data that shall not be updated.
- Customer-OTP Provisioning Data contains data that identifies the Customer product, Keys etc. and in general all data that shall not be updated.

Bootloader: Holds bootloader code/data.

Firmware: Application/services/drivers/Zephyr OS code and data (both for Intel® Quark™ and ARC).

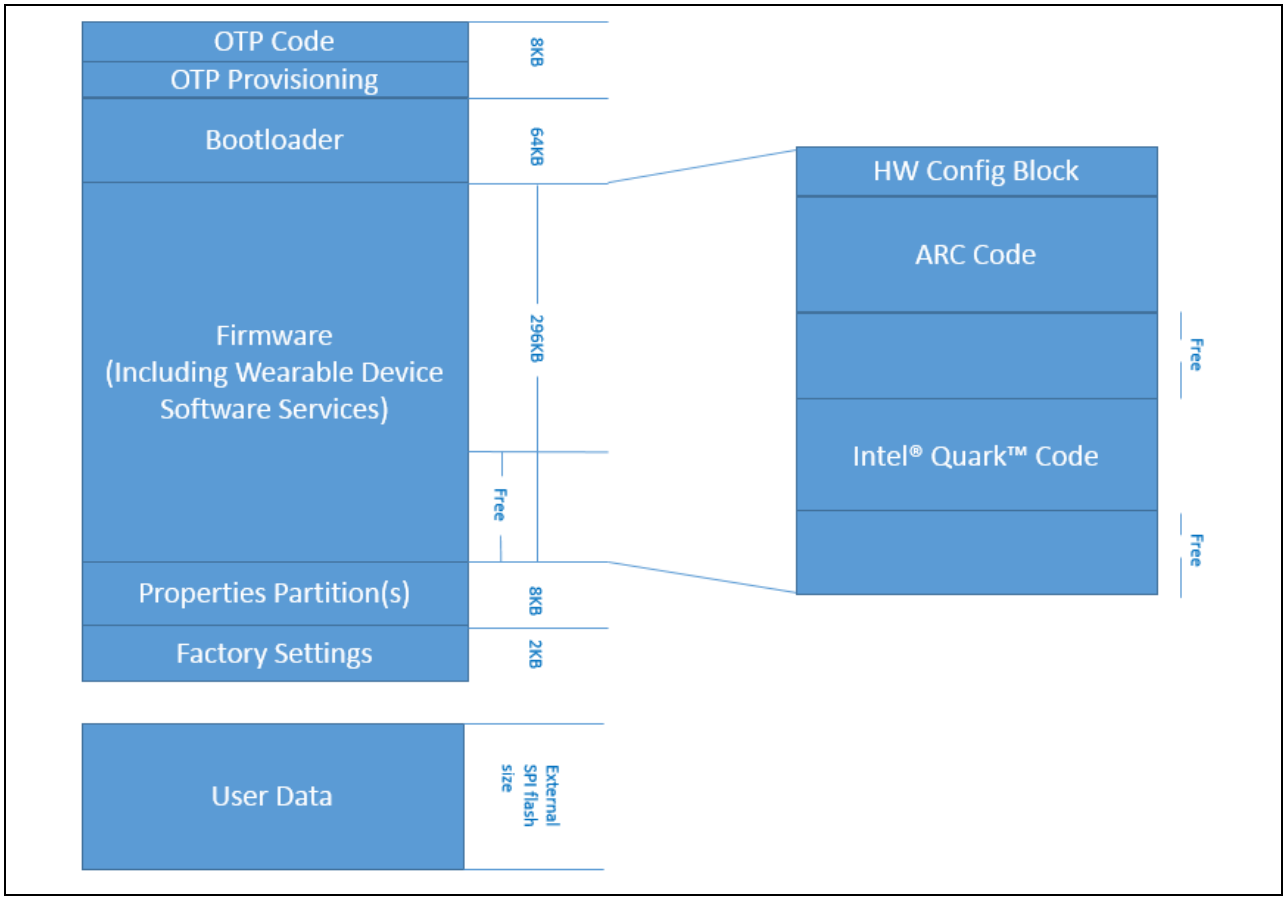
Properties partition is divided in two parts:

- Factory Reset Persistent Runtime Data: Partition to store data that are updated at runtime by the device, accessible over Properties Storage Service, but not reset during Factory Reset.
- Factory Reset non-persistent Runtime Data: Partition to store data that are updated at runtime by the device, accessible over Properties Storage Service, but reset during Factory Reset.

Factory Settings: Contains factory default settings.

User Data (0...n): User-Data partition (multiple partitions can be supported).

Figure 8 Memory layout with external SPI flash



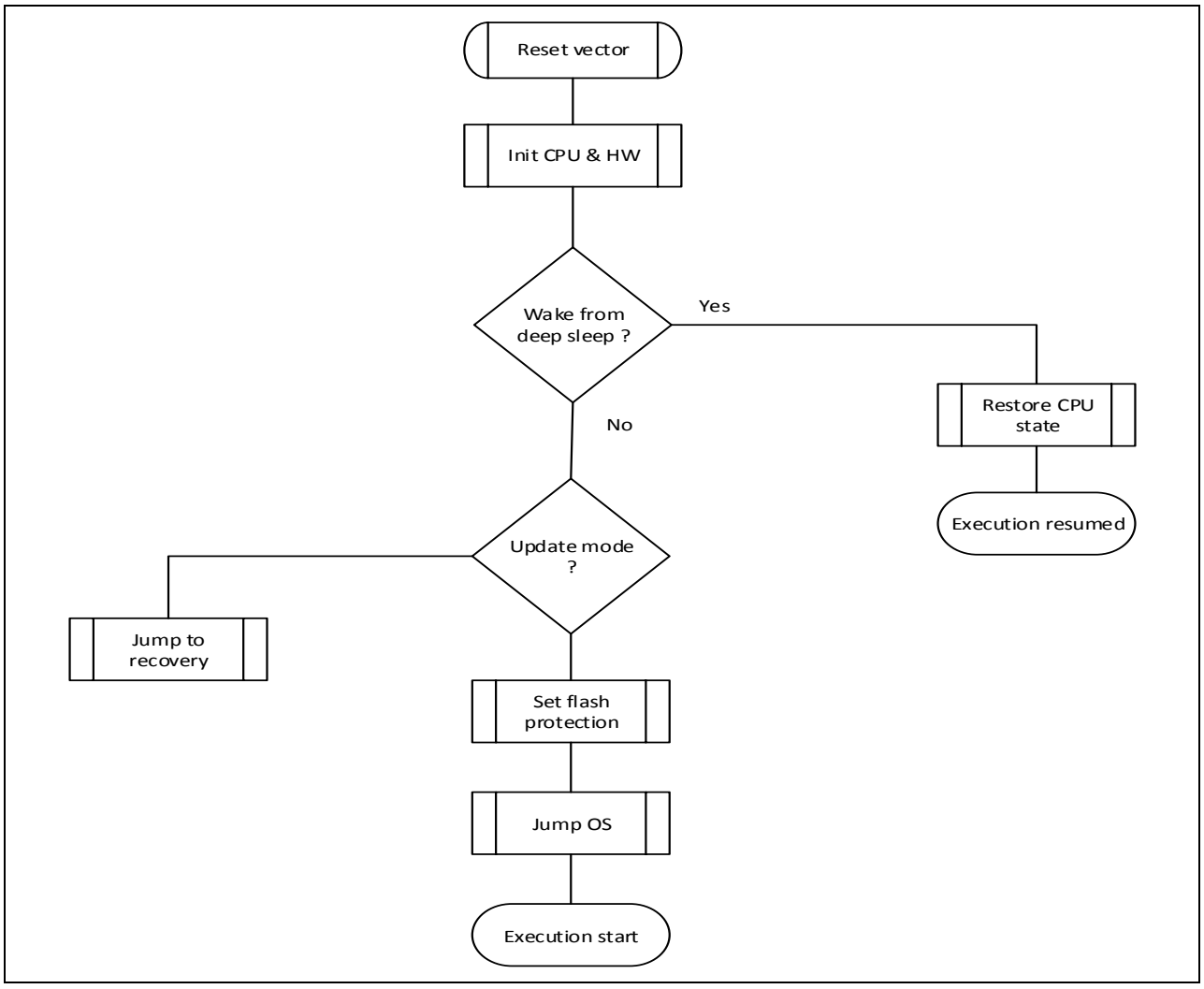
HW Config block (first part of the firmware flash block) consists of:

- Start address of ARC code.
- Start address of the context restoration procedure (for waking from deep sleep) on the ARC subsystem.
- Start address of Intel® Quark™ code.
- Start address of the context restoration procedure (for wake from deep sleep) for the Intel® Quark™ processor.

More details regarding the memory mapping for Intel® Quark™ SE can be found in this file: *"wearable_device_sw projects/curie_common/common/include/quark_se_mapping.h"*

4.2 Boot flow

Figure 5 Early boot flow



The boot flow is the following:

1. Reset vector jumps to the OTP section.
2. OTP section configures CPU and required hardware.
3. OTP section checks if the wake from deep sleep flag is set.
   a. If wake from deep sleep flag is set, jump to the defined address.
4. Check if update mode is requested:
   a. If update mode is set, jump to recovery mode. (Recovery mode is not supported at this time.)
   b. Or flash downloaded update packet if valid.
5. Protects the bootloader's flash region.
6. Verify the main image signature validity.
7. Protect flash memory regions.
8. Jump to main image.

4.3 Debug features

4.3.1 Logging

- Logging goes to a circular buffer in RAM. The buffer size is defined by the *CONFIG_LOG_CBUFFER_SIZE* parameter.
- There is one buffer per CPU.
- Time stamping of the messages uses a common timer for both CPUs in order to be able to make the log file consistent with regard to timestamps.
- Circular buffers flushed when USB connected.

A log message is characterized by:

- A time stamp in milliseconds.
- A CPU ID in case of multi-CPU log.
- A module from which it originates, e.g. USB, LED, UTIL.
- A log level: debug, info, warning or error.
- The actual ASCII string content.

A typical output looks like this:

```
2502|QRK|    FG_S| INFO| fg_timer started
2502|QRK|    FG_S|ERROR| terminate voltage too low 0 now is 3200
2220|ARC|  SS_SVC| INFO| [arc_svc_client_connected]18
2430|ARC|     LOG| WARN| -- log saturation --
2430|ARC|  SS_SVC| INFO| [ss_svc_msg_handler]is subscribing
2675|QRK|    CHGR| INFO| CHARGER State : CHARGE COMPLETE
2430|ARC|PSH_CORE|ERROR| ([sensor_register_evt]ms=2430,sid=0): WARNING ASSERT
sensor_register_evt:73
```

There are four log levels:

```
enum {
    LOG_LEVEL_ERROR,          /*!< Error log level */
    LOG_LEVEL_WARNING,        /*!< Warning log level */
    LOG_LEVEL_INFO,           /*!< Info log level */
    LOG_LEVEL_DEBUG,          /*!< Debug log level */
    LOG_LEVEL_NUM
};
```

Once the log system init is done, messages can be logged using the following helper functions:

- pr_error(module, format,...) log_printk(LOG_LEVEL_ERROR, module, format,##__VA_ARGS__)
- pr_warning(module, format,...) log_printk(LOG_LEVEL_WARNING, module, format,##__VA_ARGS__)
- pr_Info(module, format,...) log_printk(LOG_LEVEL_INFO, module, format,##__VA_ARGS__)
- pr_debug(module, format,...) log_printk(LOG_LEVEL_DEBUG, module, format,##__VA_ARGS__)

For example: pr_debug(LOG_MODULE_BLE, "%s: level: %d",__func__, level);

4.3.2 Test commands

Test commands are used to configure the debugging features, and offers a standardized transport for debug features. Test commands are ASCII-encoded messages allowing a laptop or desktop (host) to perform specific tasks on an embedded device (target) using a communication interface such as UART, USB, or BLE.

- On the host, test commands are either typed in by a user in a console application on a per command basis or run automatically in batch scripts.
- On the target, test commands are captured by a test command adapter built on top of a dedicated interface such as UART, parsed by the test command engine, and routed towards appropriate handlers.
- Responses are sent back to the host through the adapter using the interface they were received.

Wearable Device software has a command-line interface for testing various platform hardware and software features over UART. These test commands can be used to validate modules on the reference board.

After the reference board is powered on, open a serial console using either screen (115200-8N1-HW-None) or minicom. In the serial console, type *help* and then press *Enter*. Table 2 lists the commands supported.

Table 2 Test commands supported

Group	Name
adc	get
battery	level, period, status, temperature, vbatt
ble	info, version
charger	status, type
gpio	conf, get, set
i2c	probe, read, rx, write
log	Set, setbackend
mem	read, write
spi	rx, trx, tx
system	power_off, reset, shutdown
version	get

The test commands are identified by their Group and Name.

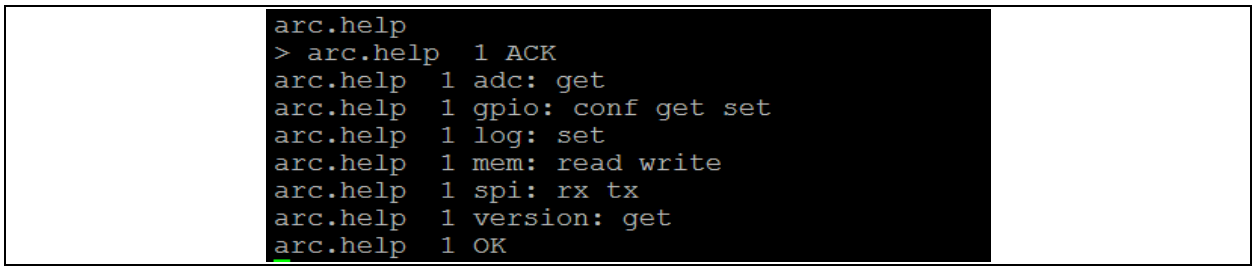
- *Group*: name for the feature to which the command belongs. Examples: *battery, ble, i2c*, etc.
- *Name*: Unique action to perform on the target for the specified feature. Examples: version, probe, etc.

Syntax of the command: *<group> <name> (<param1>…<paramN>)*:

```
version get
```

By default, the test commands execute on the Intel® Quark™ core. To check the test commands, supported by ARC core, enter `arc.help` on the serial console.

Figure 6 ARC help



To run the test commands in the ARC core, prepend `arc.` to each test command:

```
arc.version get
```

4.3.3 Panics

Panics can be solicited or unsolicited:

- Any unexpected exceptions should generate a panic.
- Any unexpected interrupts should generate a panic.

Exceptions in the code itself can be caught by explicitly generating a panic:

- ***assert***(int expression) is to capture programming errors during development phase. It is disabled as soon as NDEBUG is defined and is consequently only enabled for debug builds. It is used widely throughout the code.
- ***panic***(int error) is to reset the device in case of unrecoverable errors, dumping minimal information for reporting/debugging. It is enabled for both debug and release builds and is used sparingly.

4.3.3.1 Calling a panic

```
void panic (int error);
```

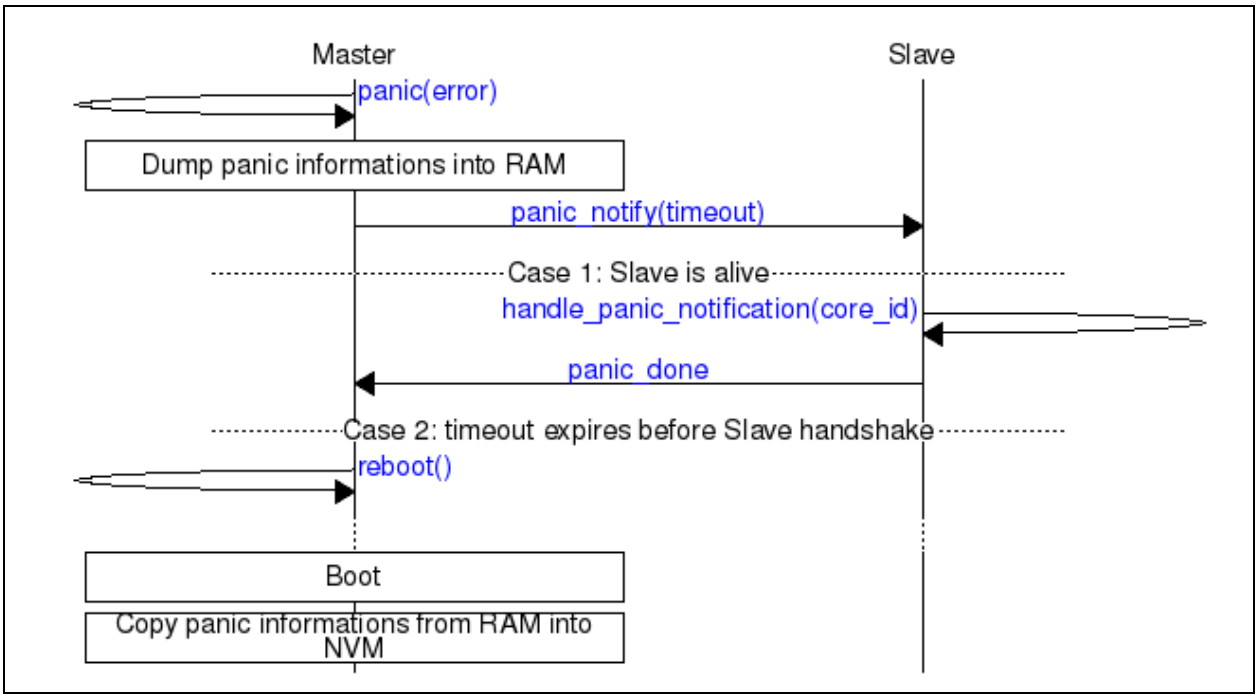
Error code definition is the developer's responsibility. It is local to a function. We recommend using an enum. Panic postprocessing uses *PC+error code* combo as a unique identifier (panic ID) to ease correlation from build to build:

- PC post processing is providing the function name, the file name and the line number,
- Line number varying from build to build, error code is used.

4.3.3.2 Panic mechanism

Processing is kept minimal during panic handling and platform reset is done as soon as possible. Once reset, additional processing is performed.

Figure 7 Panic operational flow



§

5 Power Management

5.1 Power management policy

The power management main principle for the platform is to go to lowest power consumption state as soon as possible. Each CPU in the system is responsible to manage its own state and inform the master CPU of the possibility to go for deep sleep mode. As an example, the BLE chip will be going to its lowest power consumption mode as soon as possible, but will always be able to execute requests from the main SoC by a control line that will make the BLE chip to wake up prior to the request to be sent. For the Intel® Quark™ SE SoC, deep sleep is the lowest power consumption state. In this state only the RAM content is preserved. Both the ARC and the Intel® Quark™ CPUs are sharing the same system constraints, and both will be shut down in deep sleep cycle. This implies the need to synchronize both the Intel® Quark™ and the ARC for transitioning to the deep sleep state.

In the Intel® Curie™ module, the Intel® Quark™ SE SoC is the master and ARC is the slave. The BLE chip is managing its own power states as no power resources are shared between the Intel® Quark™ SE SoC and BLE.

5.2 Power Management Tools

5.2.1 Wakelocks

Wakelocks are software objects and interfaces that allow a platform component to prevent deep sleep or shutdown in case it has some important things to do in an “atomic” way. This is useful for critical tasks, like flash memory operations. They need to be used carefully, as wakelocks that would be held too long or never released will prevent the platform to go to deep sleep state and tremendously decrease battery life. Wakelocks implementation needs strong instrumentation support in order to detect code that would hold a wakelock for too long.

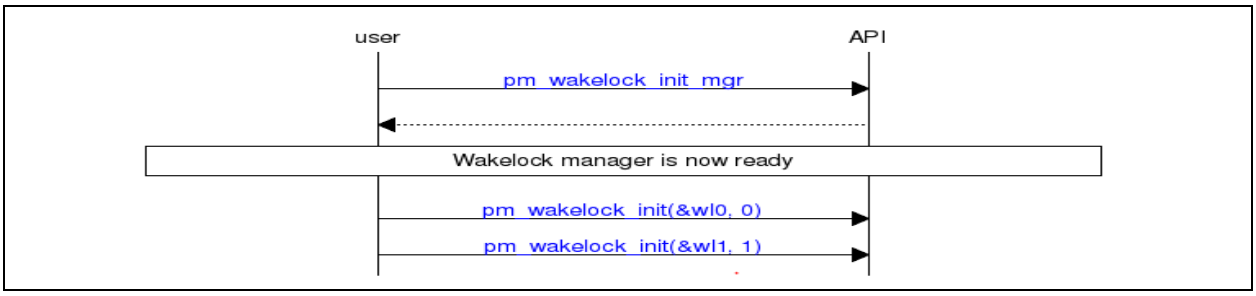
The user can register a unique callback which is called when last wakelock is released. The power manager is currently using this callback, so a user callback can be overwritten at any time. So a better advice to the user is to not use this callback

There is no timeout/expiry of the wakelock , user/developer implementation has to check whether the wakelocks are released succefully

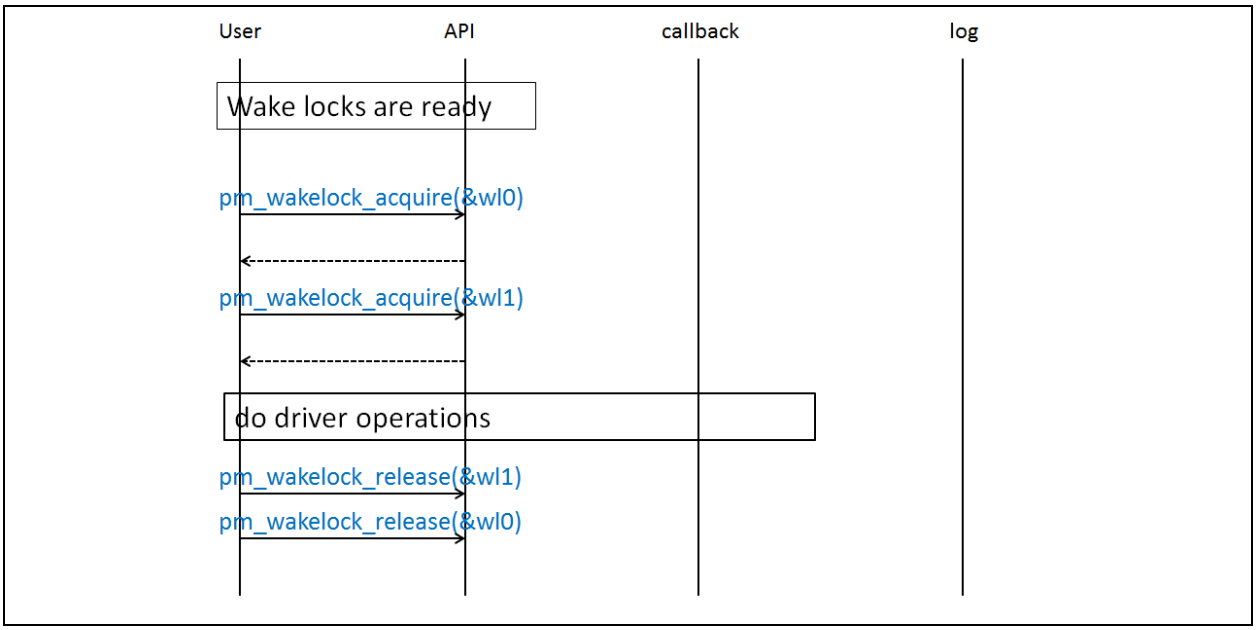
Each wakelock has a unique ID on a specific core, which is used for debug purposes (task/driver failed to release it). When a suspend request is received by a core, it will check that all wakelocks are released before processing this request:

- For deep sleep requests, core will send an error code and cancel this request.
- For shutdown requests, core will block and wait for all wakelocks to be released.

Initialize Wakelocks



Example of Wakelock Usage



5.2.2 Suspend blockers

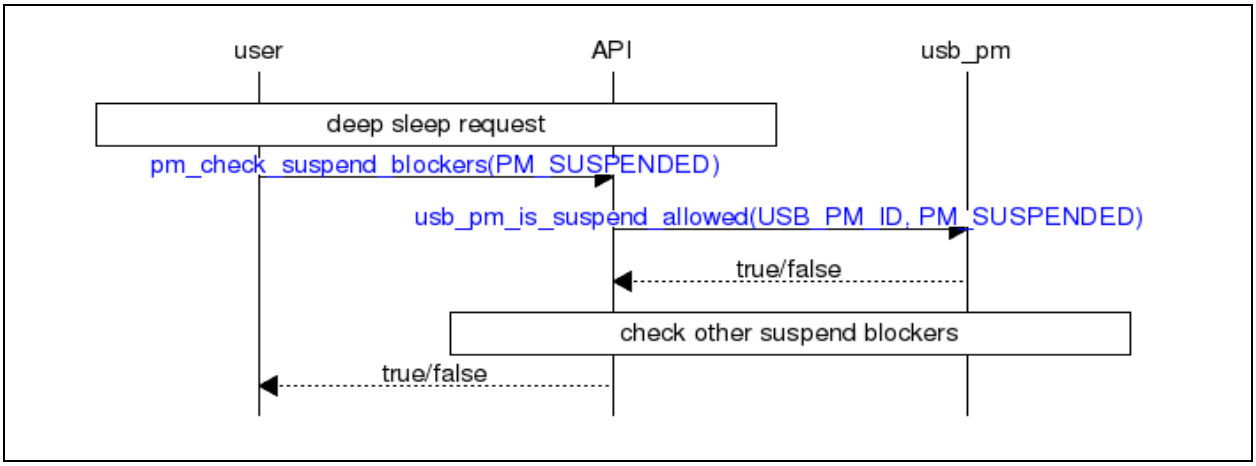
Suspend blockers are another mechanism to prevent deep sleep/shutdown modes. This consists of an array of callbacks that check if deep sleep/shutdown is allowed.

A suspend blocker callback takes as parameter a device and to power state to go to (deep sleep or shutdown). If a callback returns “false”, then deep sleep/shutdown is not allowed. This is used, for example, to prevent platform to go to deep sleep mode if USB is plugged, but to allow shutdown.

Suspend blockers are defined in *soc_config.c* file:

```
DECLARE_SUSPEND_BLOCKERS(
    {
        .cb = usb_pm_is_suspend_allowed,
        .dev_id = USB_PM_ID
    },
    /* etc ...
);
```

Suspend Blockers Use Case



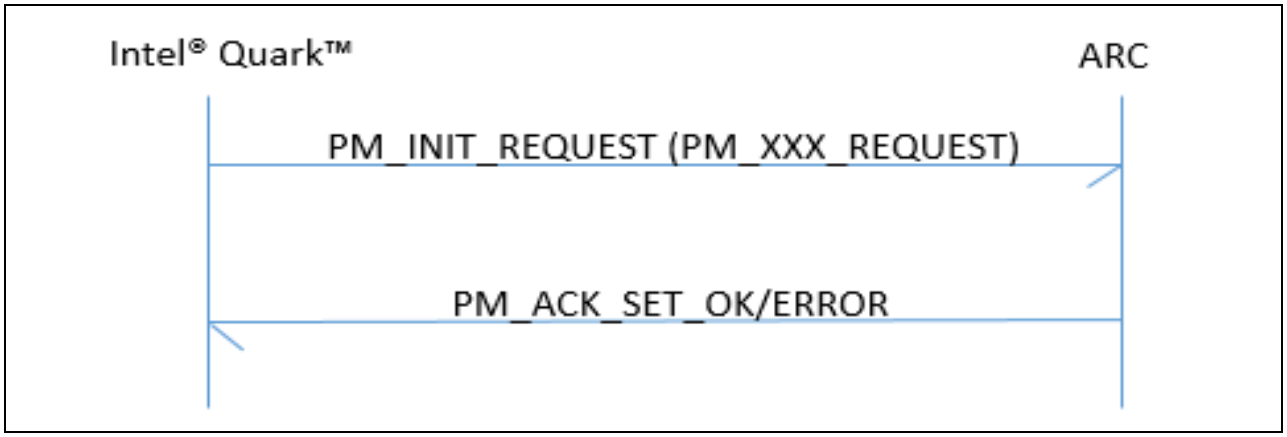
5.3 Core synchronization

Two synchronizations are needed between ARC and Intel® Quark™ cores:

- When Intel® Quark™ wants to transition to deep sleep mode
- When Intel® Quark™ resumes and restart ARC core

This synchronization is based on a soft IPC that uses shared memory. Requests are always sent by the master. Slaves are only allowed to acknowledge master core requests.

Figure 8 Core synchronization flow



5.4 Power management framework

Power state modes

Platform supports several power states, defined in enum PM_POWERSTATE.

Table 3 Device power states (from shutdown to running)

Enumerator	Description
PM_NOT_INIT	Device not initialized.
PM_SHUTDOWN	Device stopped.
PM_SUSPENDED	Device suspended.

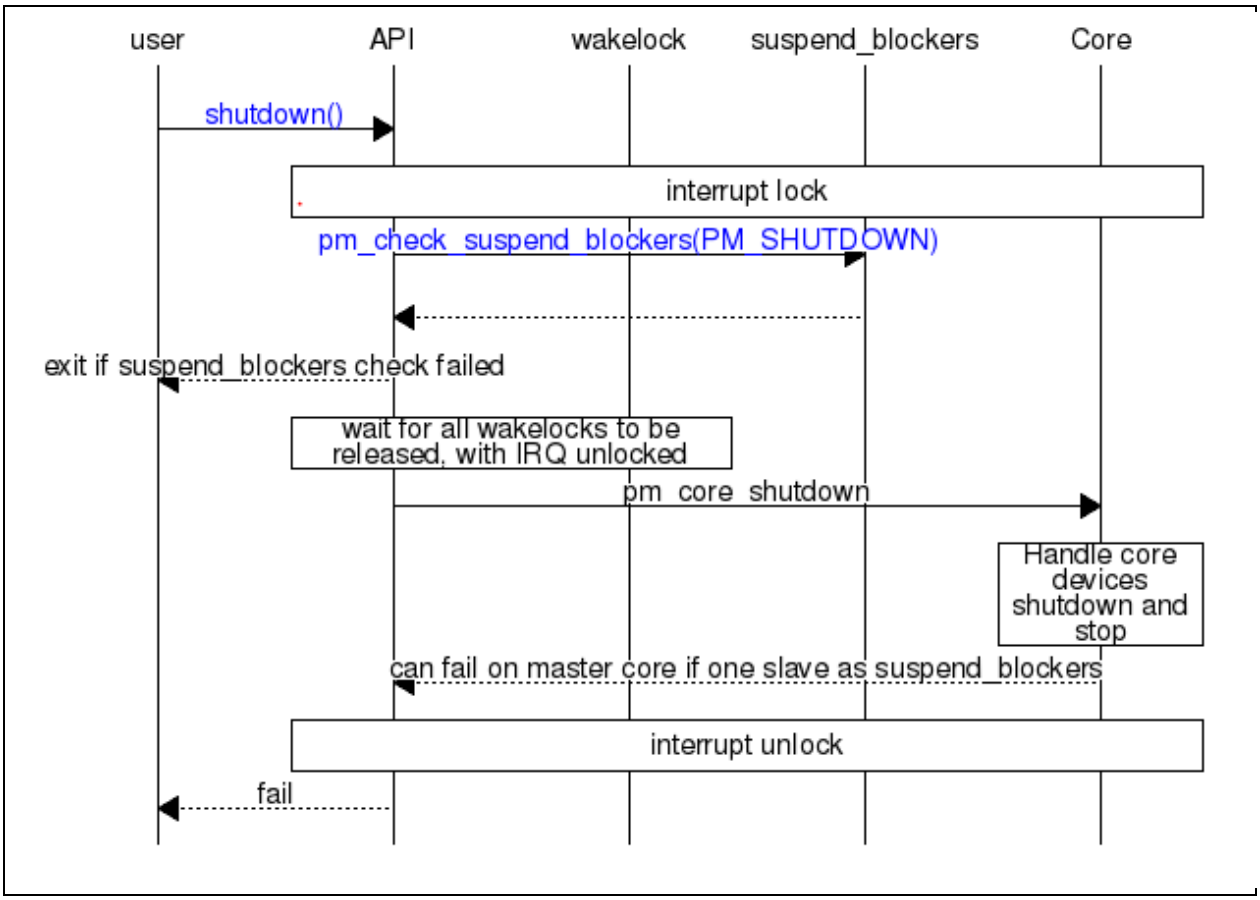
PM_RUNNING	Device working properly.

Intel® Quark™ doesn't have any deepsleep interfaces exposed to the user, but when deepsleep is activated through KConfig, the platform will automatically enter into deepsleep when possible. There is no way for the user to force the platform to go in deepsleep.

Shutdown transition

Shutdown transition is close to deep sleep. There is one generic function *shutdown* and a specific function for each cores *pm_core_shutdown*. You may call this function if a shutdown is required. The *shutdown* function can only fail because of a suspend blocker on a master/slave core.

Figure 9 Shutdown transition flow



Reboot

The *reboot* command currently reboots the platform, like *reset* does. Devices are not properly stopped and *wakelock/suspend_blockers* are not checked (not safe to use). There are several reboot modes, defined in enum *boot_targets.*

5.5 Drivers Responsibility

As in Intel® Quark™ SE the peripherals blocks are also shut down in deep sleep state, it is required that the drivers save and restore their state in the transitions in and out of deep sleep.

All drivers must then implement two callbacks: suspend and resume. The system will call all the suspend functions of the drivers prior to going to deep sleep, and will call all resume functions upon return of deep sleep.

The drivers should also configure whatever is needed to allow wake up from deep sleep if they are connected to a device that is supposed to wake the platform up on certain events. Each driver is responsible for handling clock gating for its devices when they are not used. It allows lower power consumption.

5.6 Wakeup source events

Platform can be woken from deep sleep mode by several sources:

- Comparator interrupt
- AON (Always ON) GPIO interrupt
- AON (Always ON) timer interrupt
- RTC timer interrupt

Table 4 Power management service APIs

void board_shutdown_hook (void)	Board specific hook to provide shutdown.
void board_poweroff_hook (void)	Board specific hook to provide poweroff.
void pm_set_shutdown_event_cb (void(*shutdown_event_cb)(enum shutdown_types))	Set the callback for shutdown events.
void pm_set_reboot_event_cb (void(*reboot_event_cb)(enum boot_targets))	Set the callback for reboot events.
int pm_wakelock_init_mgr()	Initialize wakelock management structure.
void pm_wakelock_init(struct pm_wakelock *wli, int id)	Initialize a wakelock structure.
int pm_wakelock_acquire(struct pm_wakelock *wl)	To acquire the wakelock and forcing the device to stay on.
int pm_wakelock_release(struct pm_wakelock *wl)	Release a wakelock.
bool pm_wakelock_is_list_empty()	Check if wakelock list is empty.
bool pm_check_suspend_blockers(PM_POWERSTATE state)	Check that all deep sleep blockers are released.
void pm_wakelock_set_list_empty_cb(void (*cb)(void*), void* priv)	Set callback to call when wakelock list is empty. It will replace the previous callback.
int pm_notification_cb(uint8_t cpu_id, int pm_req, int param)	This defines the function that pm slaves nodes needs to implement in order to handle power management IPC requests

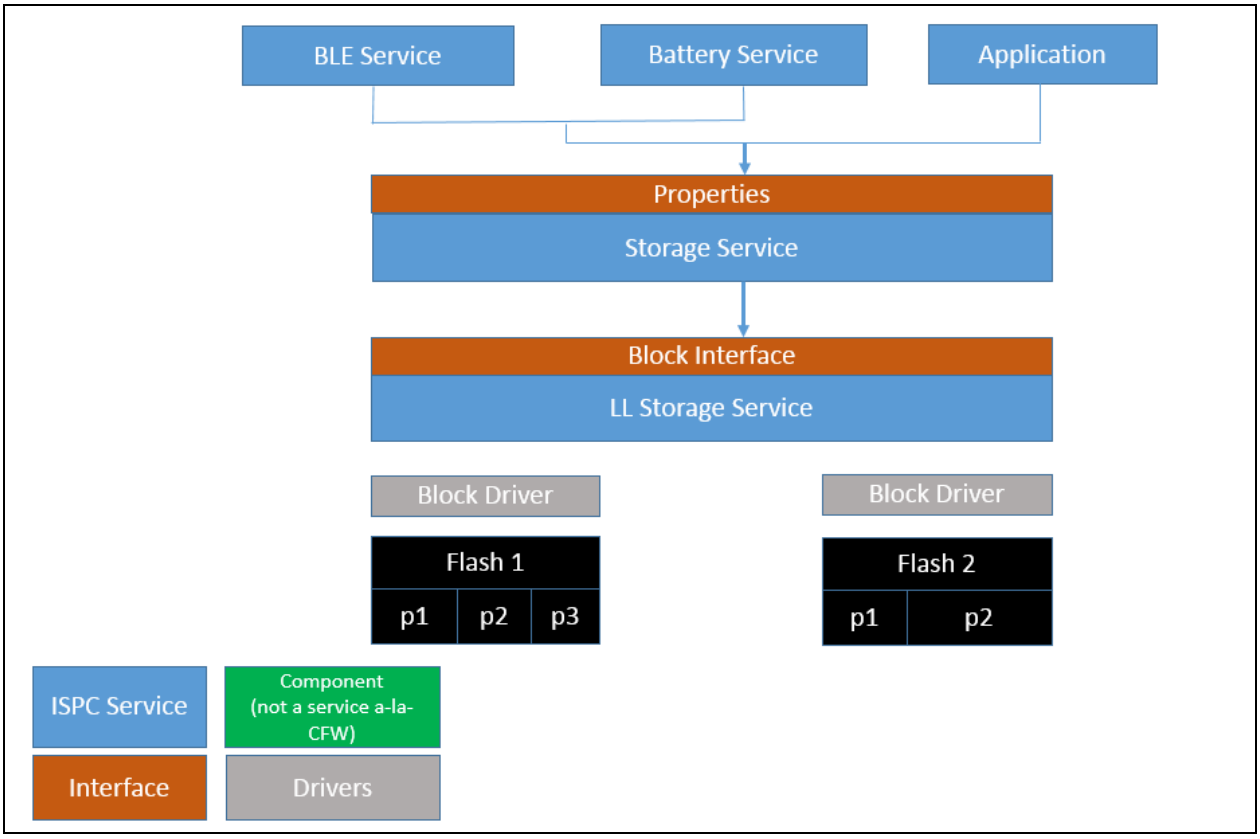
§

6 Storage Service

6.1 Description

The storage service manages the nonvolatile memories, allowing read, write, and erase accesses. The list of nonvolatile memory partitions managed by the storage service is provided at boot time. The number of partitions for a platform and their properties are fixed at firmware build time. As all partitions are not meant to be managed by storage service, the list of partitions it manages is configured at boot time by the board configurable code (using structure *storage_configuration*). The storage service does not support any wear leveling (bad block management).

Figure 10 Storage systems architecture



Two levels of APIs are provided for full flexibility to access each partitions. A partition shall be accessed only using one and only one API.

High level API should be used by applications and other services:

- It provides file system-like operations where a property (buffer) is defined with two IDs: The first ID mimics a directory name, and the second ID mimics a file name.
- Property (buffer) max size and its storage class (factory erasable or not) can't be changed after being added.
- It shall be used only for data that are updated at low rate because no real support of wear-leveling.
- The max size of a property depends on the underlying flash memory (and is linked to the block size).

Low level API with block level read/write/erase operations:

- Given that the block size is large and the amount of internal RAM is small, we provide a read/write API that works at a lower level than block level to avoid caching a large amount of data.

- The minimum size / alignment of an access depends on the underlying nonvolatile memory, and shall be enforced by the application/service when Low Level API is used directly.
- The Intel® Curie™ on-die flash requires accesses to be four bytes aligned and the size being a multiple of four bytes.

Clients of this interface can:

- **read:** Read data from a partition; return size that has been read and data or error code in case of issues.
- **write**: Write to a partition; return size that has been written or error code in case of issues.
- **read_block:** Read a full block from a partition; return number of blocks that has been read and data or error code in case of issues.
- **write_block:** Write a full block to a partition; return number of blocks that has been written or error code in case of issues.
- **erase_block:** Erase blocks in the requested partition; returns status of erase.
- **erase_partition:** Erase an entire partition; returns status of erase.

Table 5 Low level storage service APIs

void ll_storage_service_init (void *queue, flash_partition_t *storage_partitions, uint8_t number_of_partitions)	Init and configure partitions seen by the Storage Service.
int ll_storage_service_erase_partition (cfw_service_conn_t *conn, uint16_t partition_id, void *priv)	Low level partition erase.
int ll_storage_service_erase_block (cfw_service_conn_t *conn, uint16_t partition_id, uint16_t start_block, uint16_t number_of_blocks, void *priv)	Low level erase blocks
int ll_storage_service_read (cfw_service_conn_t *conn, uint16_t partition_id, uint32_t start_offset, uint32_t size, void *priv)	Low level data read.
int ll_storage_service_write (cfw_service_conn_t *conn, uint16_t partition_id, uint32_t start_offset, void *buffer, uint32_t size, void *priv)	Low level data write.

§

7 Battery Service

7.1 Features

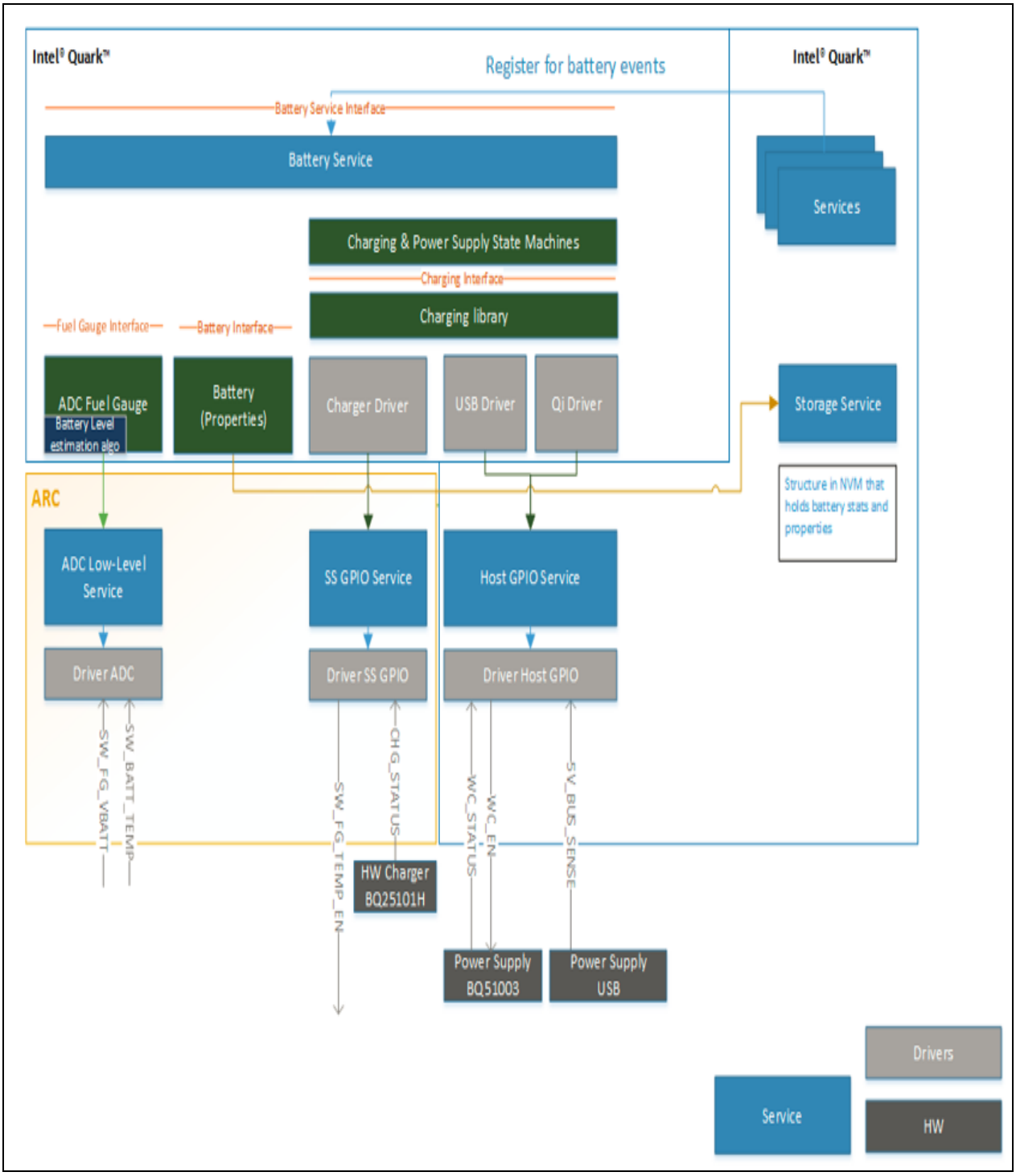
Application, and (more generally) all services are able to register against battery service to get battery/charger events. For example, BLE BAS service registers for battery state of charge changes for implementing battery state of charge reporting over BLE to the companion device. Applications can register for battery state of charge changes, battery low/critical, or charger connect/disconnect events for notifying to the user through display or LEDs the charger/battery state.

The battery service has multiple functionalities:

- Get battery voltage.
- Get battery temperature.
- Get battery capacity level (called State of Charge).
- Get the charging source (USB, QI).
- Get if the charger is connected or not.
- Get the battery status (CHARGING, DISCHARGING, MAINTENANCE).
- Get number of charging cycle
- Set the interval between 2 ADC measurement
- Event messages are sent to notify system when following events appear:
  - MSG_ID_BATT_SVC_LEVEL_UPDATED_EVT: When Battery level changed.
  - MSG_ID_BATT_SVC_LEVEL_LOW_EVT: When Battery level reached Low Level Alarm. This parameter is configurable.
  - MSG_ID_BATT_SVC_LEVEL_CRITICAL_EVT: When Battery level reached Critical Level Alarm. This parameter is configurable.
  - MSG_ID_BATT_SVC_LEVEL_SHUTDOWN_EVT: When Battery level reached Shutdown Level Alarm - A shutdown is recommended.
- By default below events are sent when battery voltage is below 3200 mV (defined as FG_DFLT_SHUTDOWN_ALARM_THRESHOLD in *adc_fuel_gauge_api.c*).
  - MSG_ID_BATT_SVC_FULLY_CHARGED_EVT: When Battery is fully charged.
  - MSG_ID_BATT_SVC_CHARGER_CONNECTED_EVT: When Charger is connected.
  - MSG_ID_BATT_SVC_CHARGER_DISCONNECTED_EVT: When Charger is disconnected.

7.2 Architecture

Figure 11 Battery service block description



7.3 Developer manual

Initialization

To interact with every service implemented within the platform, the service has to be opened using *cfw_open_service* with the corresponding ID. The ID of the Battery service is BATTERY_SERVICE_ID.

```
cfw_client_t *client = cfw_client_init(queue, your_handle_function, NULL);
cfw_open_service_conn(client, BATTERY_SERVICE_ID, NULL);
```

Attached to the event

In order to catch an event sent by Battery Service, first you have to be registered. This could be done like that:

```
int events[] = {MSG_ID_BATT_SVC_LEVEL_UPDATED_EVT,
MSG_ID_BATT_SVC_CHARGER_CONNECTED_EVT};
cfw_register_events(client, events, sizeof(event)/sizeof(int),
CFW_MESSAGE_PRIV(msg));
```

Received an event or response message

To receive a message from the Battery Service, within *your_handle_function*, the message has to be parsed. For example:

```
static void your_handle_function(struct cfw_message * msg, void *param)
{
    switch(CFW_MESSAGE_ID(msg)) {
    case MSG_ID_BATT_SVC_GET_BATTERY_INFO_RSP:
    switch(((bs_get_status_batt_rsp_msg_t *)msg)->batt_info_id) {
    case BS_CMD_BATT_VBATT:
            bat_status = ((bs_get_status_batt_rsp_msg_t *) msg)-
>rsp_header.status;
            if (BATT_STATUS_SUCCESS == bat_status) {
             pr_info(LOG_MODULE_MAIN, "Battery
Voltage:\t%d[mV]\n",((bs_get_status_batt_rsp_msg_t *)msg)-
>bs_get_voltage.bat_vol);
             }
            break;
     case BS_CMD_CHG_STATUS:
            bat_status = ((bs_get_status_batt_rsp_msg_t *) msg)-
>rsp_header.status;
            if (BATT_STATUS_SUCCESS == bat_status) {
            pr_info(LOG_MODULE_MAIN,"the charger have been connected");
        }
            break;
        default:
            break;
    }
    default:
        break;
    }
    cfw_msg_free(msg);
}
```

Battery lookup tables

Default Battery Lookup Tables are located in *Battery_LUT directory* (array *dflt_lookup_tables*). This could be changed according to your battery properties.

Shutdown, Low and critical level alarm

Default Shutdown, low and critical Level Alarm are located in file *adc_fuel_gauge_api.c* (FG_DFLT_SHUTDOWN_ALARM_THRESHOLD, FG_DFLT_LOW_ALARM_THRESHOLD, FG_DFLT_CRITICAL_ALARM_THRESHOLD). This could be changed according to your battery properties.

Table 6 Battery service APIs

batt_status_t bs_init (void *batt_svc_queue, int service_id)	Initialize and register the Battery Service.
batt_status_t bs_get_battery_info (cfw_service_conn_t *c, bs_data_info_t batt_info_id, void *priv)	Get the latest battery Information.
batt_status_t bs_set_level_alarm_thr (cfw_service_conn_t *c, uint8_t level_alarm, int message_id, void *priv)	Set the battery level of alarm.
batt_status_t bs_set_measure_interval (cfw_service_conn_t *c, struct period_cfg_t *period_cfg, void *priv)	Set ADC measure interval.

§

8 UI Service (Buttons/LED/Vibra/Touch)

The *UI* (user interaction) service is an event/notification aggregator designed to enable all user events an application will subscribe to and handle all requests for user notifications. The application manager will subscribe to and handle all requests for user notifications. It is in charge of managing LEDs, buttons, and haptics. UI service supports the following features:

- Handle power button event.
- Handle single button press event.
- Handle double button press event.
- Handle long button press event.
- Handle double tap event.
- Handle triple tap event.
- Expose supported and active events.
- Enable/disable events.
- Handle LEDs notifications, with patterns, color, and intensity support.
- Handle vibrator notification, with pattern.
- Expose supported and active notifications.
- Enable/disable notifications.
- Handle notifications requests.

The user interaction service relies on another service or a driver to catch events.

Table 7 UI service APIs

uint32_t ui_get_events_list(void)	Get available events.
int8_t ui_get_available_features (cfw_service_conn_t *service_conn, void *priv)	Retrieve all supported user events and user notifications.
int8_t ui_get_enabled_events (cfw_service_conn_t *service_conn, void *priv)	Retrieve all user events enabled.
int8_t ui_set_enabled_events (cfw_service_conn_t *service_conn, uint32_t mask, uint32_t enable, void *priv)	Enable one or several events.
int8_t ui_play_led_pattern (cfw_service_conn_t *service_conn, uint8_t led_id, enum led_type type, led_s *pattern, void *priv)	Play a led pattern on LED1 or LED2.
int8_t ui_play_vibr_pattern (cfw_service_conn_t *service_conn, vibration_type type, vibration_u *pattern, void *priv)	Play a vibration pattern.
int8_t ui_audio_response (cfw_service_conn_t *service_conn, uint32_t audio_resp_id, void *priv)	API function called when an audio tone or sentence has to be played.
void ui_service_init (void *ui_svc_queue, struct ui_config *config)	Initialize and register the UI Service.
int8_t ui_service_stop (void)	Stop the UI Service.
void ui_handle_audio_req (message_t *msg)	Handle audio requests to play tone and/or sentence.
void ui_audio_svc_open(void *queue, service_t *service)	Open connection with the audio service.

§

9 Sensors Service

9.1 Features

The *sensors service* manages all sensors that are available on the platforms and is providing the framework to integrate sensor-fusion algorithms. By default the below 5 algorithms are integrated in sensor service.

- Activity detection: walking/running/biking.
- Step counting.
- User tapping detection: double tap/triple tap.
- Simple user gesture detection: single-flick/shaking.
- User-defined gestures: up-down, down-up, left-right, right-left.

A generic sensor API is provided by the sensor service. The “generic” means that it provides the same set of API for:

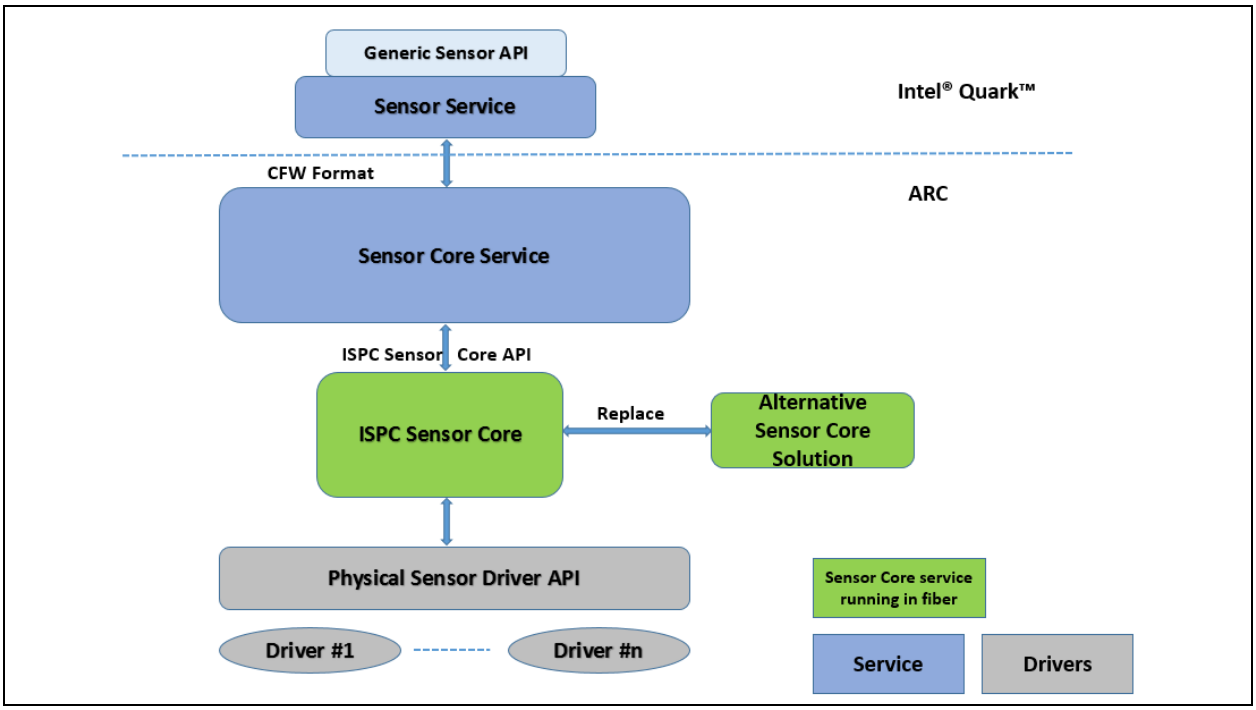
- Physical sensors (for example, accelerometer, gyroscope, and magnetic sensor) and
- Abstract sensors (for example, activity sensor to report the detected walking/running/biking state).

It is composed of the following features:

- Sensor enumeration: Enumerates the sensor devices of a certain type and inquiry sensor device’s properties.
- Sensor control: Discovers, starts/stops sensor, and subscribes/unsubscribes sensor data, calibration sensor etc.
- Sensor data: Defines the data format that is reported to application.
- Sensor status: Retrieves sensor status, such as sensor device battery level, connection with sensor device lost, etc.

9.2 Architecture

Figure 12 Sensor architecture



Physical sensor driver API is the well-defined layer, which hides the HW differences of various sensor hardware. This way it provides two benefits:

- The sensor drivers running under the physical sensor driver API can be replaced seamlessly without affecting the SW running on top of physical driver API.
- Software running on top of the physical sensor driver API do not care about the specific sensor hardware on the platform.

Wearable Device Software sensor core is designed with flexible architecture, so that new sensor algorithms can be integrated in modular way. It's a closed-source solution and can be replaced by an open-source sensor core thanks to:

- Wearable Device Software sensor core API between sensor core service <-> Wearable Device Software sensor core
- And physical sensor driver API between sensor core <-> physical sensor driver are well-defined standard interface.

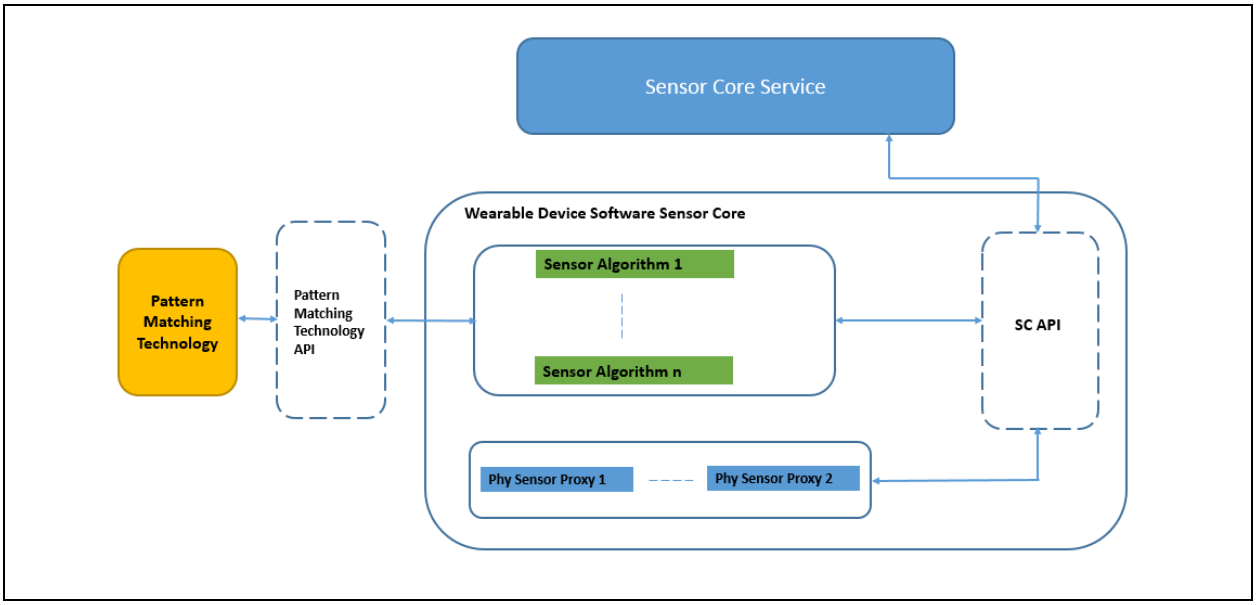
Sensor service software stack is designed with high scalability and provides two configuration options to achieve the best power and performance for different usage scenarios.

- If there are multiple client applications running on Intel® Quark™ to request the same sensor device with different sampling/reporting data rate, the sensor service on Intel® Quark™ side should be enabled. The sensor service on Intel® Quark™ side can do the arbitration and dispatch data to multiple client applications. This way, the traffic between Intel® Quark™ and ARC is reduced, and more power is saved.
- If there are no multiple client applications running on Intel® Quark™ to request the same sensor device with different sampling/reporting data rate, the sensor service on the Intel® Quark™ processor side can be disabled. This way, more flash/RAM space is saved for Intel® Quark™, and the data transmission latency is reduced.

9.3 Sensor core architecture and workflow

Figure 13 shows the top view of how the sensor core works.

Figure 13 Sensor core detailed structure



In Figure 13, there are two sets of main APIs:

- Sensor core API, i.e. algorithm integration API. In these APIs, the algorithms will communicate with upper level applications. Algorithms and sensors can be configured by these APIs, and all the output of algorithms can be passed to applications.
- Pattern-Matching Technology API algorithms use Pattern-Matching Technology as classifier and can call these APIs to call Pattern-Matching Technology.

9.3.1 How to start new algorithms on sensor core

The detailed steps are as follows:

- Define new sensor data structure definition in *sensor_data_format.h* for the new algorithm.
- Add new sensor type definition in enum of *sensor_data_format.h*.
- Add algorithm type definition in enum of *opencore_algo_common.h*.
- Generate call back function file.
- Add new sensor scanning/sensor data subscribing/sensor data processing code in Intel® Quark™ *sensing.c*.
- Add related configuration in configuration files.

Example

The following things need to be clarified before the example:

1. Only file names are referred to, but no absolute file path is given in this example as the code base keeps on changing. To find the absolute paths of those files, search their file names in root folder of code base.
2. Sensor core runs in the ARC, and all algorithms are called from the Intel® Quark™ processor. So we need to add code in corresponding location in Intel® Quark™ processor.
3. This example only introduces how to have sensor raw data fed into exec callback function. How to use those raw data is beyond the range of this example.

9.3.2 How to add new algorithms on Wearable Device Software sensor core

To add an algorithm, do the following:

1. Add a new sensor data structure definition in *sensor_data_format.h* file.

```
struct recognition_udg
{
    u16 size;
    s16 nClassLabel;
    s16 reserved;
}__attribute__ ((packed));
```

2. Add new sensor type definition in enum of *sensor_data_format.h* file:

```
SENSOR_ABS_ACTIVITY,
SENSOR_ALGO_DEMO,
SENSOR_ALGO_UDG,          // new sensor type
SENSOR_MOTION_DETECTOR,
......
#define ACTIVITY_TYPE_MASK          (1 << SENSOR_ABS_ACTIVITY)
#define HUMI_TYPE_MASK              (1 << SENSOR_HUMIDITY)
#define ALGO_DEMO_MASK              (1 << SENSOR_ALGO_DEMO)
#define ALGO_UDG_MASK               (1 << SENSOR_ALGO_UDG)
......
ACTIVITY_TYPE_MASK  |  \
TEMP_TYPE_MAS       |  \
ALGO_DEMO_MASK      |  \
ALGO_UDG_MASK       |  \
```

3. Add algorithm type definition in enum of *opencore_algo_common.h* file:

```
typedef enum
{
    BASIC_ALGO_GESTURE = 0,
    BASIC_ALGO_STEPCOUNTER,
    BASIC_ALGO_TAPPING,
    BASIC_ALGO_SIMPLEGES,
    BASIC_ALGO_RAWDATA,
    BASIC_ALGO_DEMO,
    BASIC_ALGO_UDG,    //new algo type
}basic_algo_type_t;
```

4. Generate call back function file as follows. The variables and function names should be customized according your algorithms. You can also refer to other existing algorithms for this file.

```
#include "opencore_algo_support.h"
static struct recognition_udg gs_rpt_buf;
static int udg_algo_exec(void** sensor_data, feed_general_t* feed)
{
    int ret = 0;
    int idx = GetDemandIdx(feed, SENSOR_ACCELEROMETER);
    common_data_buf_t* pdata;
    if(idx >= 0 && idx < PHY_SENSOR_SUM)
    {
        pdata = (common_data_buf_t *)sensor_data[idx];
        printk("%d,%d,%d\n",pdata->x, pdata->y, pdata->z);
    }
    return ret;
}
static int udg_algo_init(feed_general_t* feed_ptr)
```

```
{
    return 0;
}
static int udg_algo_goto_idle(feed_general_t* feed_ptr)
{
    return 0;
}
static int udg_algo_out_idle(feed_general_t* feed_ptr)
{
    return 0;
}
sensor_data_demand_t    atlsp_algoF_sensor_demand[]=
{
    {
        .type = SENSOR_ACCELEROMETER,
        .freq = 100,
        .rt = 1000,
    }
};
static feed_general_t atlsp_algoF=
{
    .type = BASIC_ALGO_UDG,
    .demand = atlsp_algoF_sensor_demand,
    .demand_length =
sizeof(atlsp_algoF_sensor_demand)/sizeof(sensor_data_demand_t),
    .ctl_api = {
        .init = &udg_algo_init,
        .exec = &udg_algo_exec,
        .goto_idle = &udg_algo_goto_idle,
        .out_idle = &udg_algo_out_idle,
    },
};
define_feedinit(atlsp_algoF);
static exposed_sensor_t udg_sensor=
{
    .depend_flag = 1 << BASIC_ALGO_UDG,
    .type = SENSOR_ALGO_UDG,
    .rpt_data_buf = (void*)&gs_rpt_buf,
    .rpt_data_buf_len = sizeof(struct recognition_udg),
};
define_exposedinit(udg_sensor);
```

5. Add new sensor scanning/sensor data subscribing/sensor data processing code in Intel® Quark™ *sensing.c.* Receive sensor data:

```
......
case SENSOR_PATTERN_MATCHING_GESTURE: {
    struct gs_personalize {
        short size;
        short nClassLabel;
        short reserved;
    }__packed;
    struct gs_personalize *p = (struct gs_personalize *) p_data_header-
>data;
    pr_info(LOG_MODULE_MAIN, "GESTURE=%d,size=%d", p->nClassLabel, p-
>size);
    sensing_callback(0, p->nClassLabel);
```

```
}
break;
case SENSOR_ALGO_UDG: {
    struct recognition_udg *p = (struct recognition_udg *)
p_data_header->data;
    pr_info(LOG_MODULE_MAIN, "UDG GESTURE=%d,size=%d", p->nClassLabel,
p->size);
    sensing_callback(0, p->nClassLabel);
}
break;
......
```

Start sensor data subscribing:

```
......
case SENSOR_PATTERN_MATCHING_GESTURE:
    ss_sensor_subscribe_data(sensor_handle, NULL, p_evt->handle,
                             ACCEL_DATA, 100, 10);
    break;
case SENSOR_ALGO_UDG:
    ss_sensor_subscribe_data(sensor_handle, NULL, p_evt->handle,
                             ACCEL_DATA, 100, 10);
    break;
......
```

Start sensor scanning:

```
......
case ARC_SC_SVC_ID:
    sensor_handle = req->client_handle;
    ss_start_sensor_scanning(sensor_handle, NULL,
                            PATTERN_MATCHING_TYPE_MASK
                            | TAPPING_TYPE_MASK
                            | STEPCOUNTER_TYPE_MASK
                            | ALGO_UDG_MASK);
    break;
......
```

6. Update Kconfig and Kbuild.mk
   Once the callback function file is added in *algo_support_src* folder, the Kbuild.mk file should be updated

```
obj-$(CONFIG_SENSOR_CORE_ALGO_TAPPING) += opencore_tapping.o
obj-$(CONFIG_SENSOR_CORE_ALGO_SIMPLEGES) += opencore_simpleges.o
obj-$(CONFIG_SENSOR_CORE_ALGO_DEMO) += opencore_demo.o
obj-$(CONFIG_SENSOR_CORE_ALGO_OPENCOREUDG) += opencore_udg.o
```

The configuration flag should be added in *defconfig_crb*:

("*wearable_device_sw/projects/curie_reference/arc/ defconfig_crb*")

```
CONFIG_SERVICES_SENSOR_IMPL=y
CONFIG_SENSOR_CORE_ALGO_UDG=y
CONFIG_SENSOR_CORE_ALGO_OPENCOREUDG=y
```

The configuration flag should also be added in *Kconfig*:

```
......
config SENSOR_CORE_ALGO_UDG
    bool "User Defined Gesture"
    depends on HAS_PATTERN_MATCHING
```

```
    select SENSOR_CORE_ALGO_COMMON
    select PATTERN_MATCHING_DRV
config SENSOR_CORE_ALGO_OPENCOREUDG
    bool "UDG Non Pattern Matching"
    select SENSOR_CORE_ALGO_COMMON
......
```

9.4 Sensor service APIs

Table 8 Sensor service APIs

void ss_service_init (void *p_queue)	Initialize the sensor framework client to connect to sensor framework and allocate resources
int ss_start_sensor_scanning (cfw_service_conn_t *p_service_conn, void *p_priv, uint32_t sensor_type_bit_map)	Used by application to start the sensor scanning.
int ss_stop_sensor_scanning (cfw_service_conn_t *p_service_conn, void *p_priv, uint32_t sensor_type_bit_map)	Used by application to stop the sensor scanning.
int ss_sensor_subscribe_data (cfw_service_conn_t *p_service_conn, void *p_priv, ss_sensor_t sensor, uint8_t *data_type, uint8_t data_type_nr, uint16_t sampling_interval, uint16_t reporting_interval)	Set the parameter of sensor subscribing data.
int ss_sensor_unsubscribe_data (cfw_service_conn_t *p_service_conn, void *p_priv, ss_sensor_t sensor, uint8_t *data_type, uint8_t data_type_nr)	Unset the parameter of sensor subscribing data.
int ss_sensor_set_property (cfw_service_conn_t *p_service_conn, void *p_priv, ss_sensor_t sensor, uint8_t len, uint8_t *value)	Set the sensor property.
int ss_sensor_get_property (cfw_service_conn_t *p_service_conn, void *p_priv, ss_sensor_t sensor)	Get the sensor property.
int ss_sensor_set_calibration (cfw_service_conn_t *p_service_conn, void *p_priv, ss_sensor_t sensor, uint8_t calibration_type, uint8_t len, uint8_t *value)	Set the calibration
int ss_sensor_opt_calibration (cfw_service_conn_t *p_service_conn, void *p_priv, ss_sensor_t sensor, uint8_t clb_cmd, uint8_t calibration_type)	Operation the calibration parameter.

§

10 Intel Topic Manager (ITM)

This defines and describes the Intel Topic Manager. This component provides companion applications with services to interact with Intel wearable devices over BLE. The service is made of a single manager to which several applications can register to offer for subscription. These topics can then be subscribed by remotely connected applications. Some of the Topic Manager's capabilities are subscribing/unsubscribing to device events and data like

- battery monitor
- User events (user tap, button press...)
- system(panic, low battery)
- User activity (Walk, Run...)
- Step count, calculates distance (calories are not provided only activity type and step count info)

10.1 Software API Definations

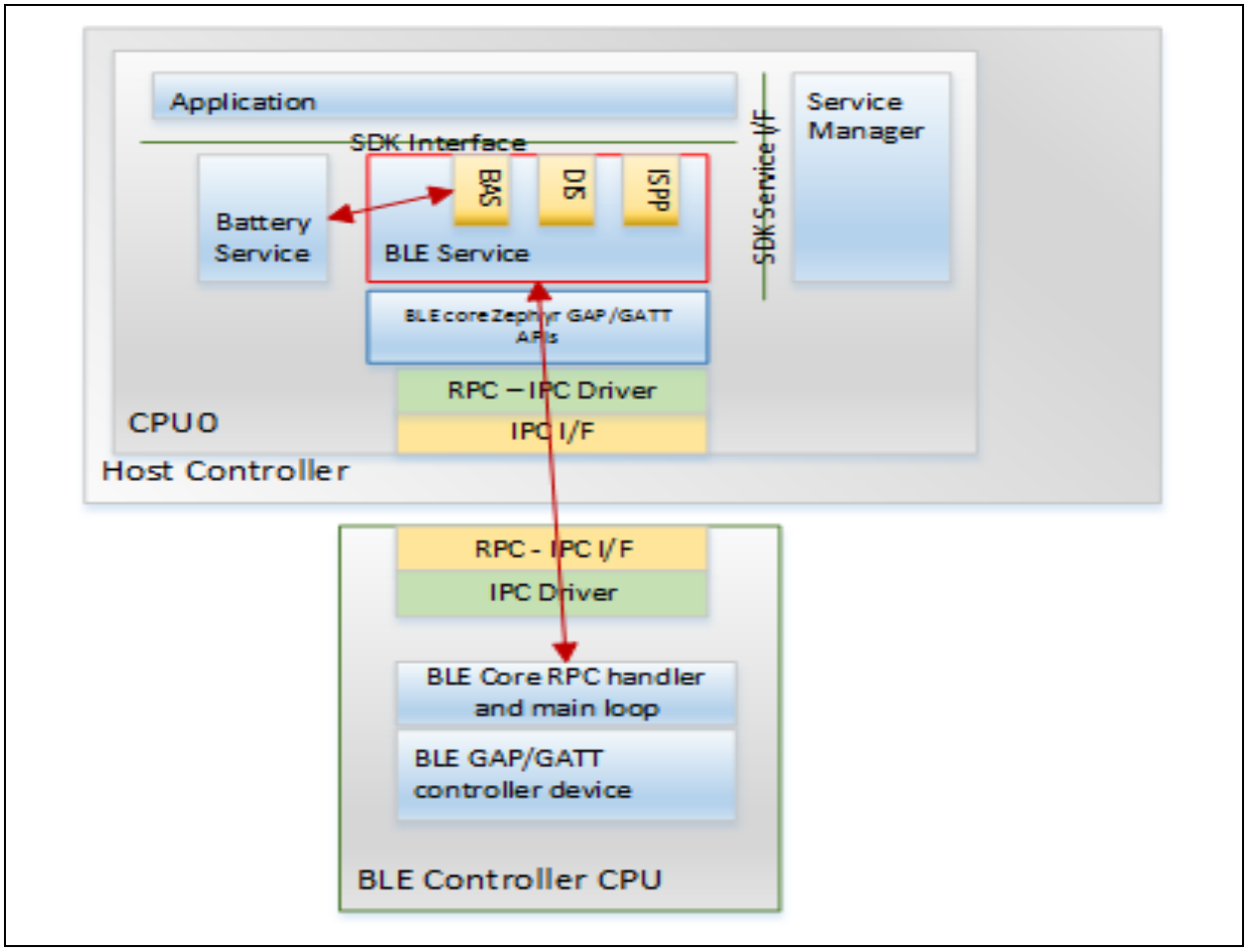
uint8_t topic_index(uint8_t const * const * topic_array, uint8_t const * const * topic) __attribute_const	Compute the index of a topic in a topic array. It returns the topic index
int8_t topic_size(const uint8_t * topic) __attribute__ ((pure))	Retrieves the size of the topic. This returns size in bytes.
int8_t topic_depth(const uint8_t * topic) __attribute__ ((pure));	This interfaces computes the hierarchical depth of a topic
int8_t topic_list_size(const uint8_t * list) __attribute__ ((pure))	This interface computes the total size of the topic list. Size is in bytes
int8_t topic_find(uint8_t const * const * topic_array, const uint8_t * topic) __attribute__ ((pure))	Interfaces return the index of the topic in the topic array else 0XFF if the topic is not in topic array.
bool topic_match(const uint8_t * topic, const uint8_t * pattern) __attribute__ ((pure))	Verifies whether a particular topic matches the pattern with the reference pattern
int topic_array_2_list(uint8_t * topic_list, uint8_t size, uint8_t const * const * topic_array)	Interface to convert a specified topic array to topic list.
int itm_add_application(const struct topic_application * app)	Add the desired application to the Topic Manager.
int itm_start_application(uint8_t app_id):	This interfaces starts the application that is added to the Topic Manager.
int itm_list_req(uint8_t app_id, uint8_t con_id, const uint8_t * pattern, uint8_t depth)	List topics offered by the remote topic manager on a connection. This request will receive a response in the application-callback.
int itm_subscribe_req(uint8_t app_id, uint8_t con_id, uint8_t const * const * topic_array)	Subscribe to topics coming from this connection. Response is received in the application-callback
int itm_unsubscribe_req(uint8_t app_id, uint8_t con_id, uint8_t const * const * topic_array)	This interface unsubscribe from topics coming from this connection. This request will receive a response in the application-callback
int itm_publish(uint8_t app_id, uint8_t con_id, uint8_t topic_ix, const uint8_t * data, uint16_t len)	Publish a new value on a topic. This procedure can be unicast or broadcast
int itm_request(uint8_t app_id, uint8_t con_id, const uint8_t * topic, const uint8_t * data, uint16_t len):	Request more data from topic on a remote topic_manager. This request will receive a response in the application callback.
int itm_response(uint8_t con_id, uint8_t req_id, uint8_t status, const uint8_t * data, uint16_t len)	This interfaces sends the response to a topic request. Returns 0 in case of success, a negative value is an error.

11 BLE Service

This service provides *BLE service* (in the sense of Bluetooth Low Energy) abstracting most of the complexity of the underlying BLE services/profiles. It provides APIs covering peripheral role and will support central role in the future. BLE services can be extended by using the BLE core APIs and implement a new BLE service/profile. This is achieved by using the *ble_core* service APIs.

11.1 BLE architecture

Figure 14 BLE service architecture



This architecture splits BLE support into two modules:

1. An application oriented CFW oriented BLE services. This provides more easy to use APIs.
2. The other is the BLE core API that offers smp/gap/gatt APIs, which allows abstracting from the actual BLE stack or chip.

Currently, the following BLE services are implemented and supported:

- Simplified security handling
- **BLE DIS service** (UUID: 0x180a): Different device info's (SW version etc) are displayed as per BT spec.
- **BLE BAS service** (UUID: 0x180f): BLE Battery level service as per BT spec. It uses the battery service to update automatically the battery level characteristic

- **BLE ISPP service** (UUID: dd97c415-fed9-4766-b18f-ba690d24a06a): Intel serial port protocol: A serial port emulation running over BLE
- **BLE DTM**: via *ble_test* DTM mode is supported

11.2 Application development

An application developer should simply use the BLE App APIs. Here is a typical sequence:

- In user application the api to be called to use BLE are.
    - Call ble_start_app() to start the Bluetooth stack, BLE service and register/start and restart the BLE controller.
    - ble_app_start_advertisement(): This interfaces to be called when user wants to start ISPC to start advertisement. App need to pass the advertisement options.
    - ble_app_stop_advertisement(): User can stop advertisement once the pairing is successful.
    - ble_app_clear_bonds(): Interface to clear the bonds(unpairing the ISPC device with Host).
- If user has custom service the initialization service interface should be called in _ble_register_services(/framework/src/lib/ble/ble_app.c) interface. This would enable the service and its characteristics of IPSC device to be visible in the Host(Android/IOS) devices.

Table 9 BLE service APIs

int ble_set_name (cfw_service_conn_t *p_service_conn, const uint8_t *p_name, void *p_priv)	Set/change local BLE device name.
int ble_connect (cfw_service_conn_t *p_service_conn, const ble_addr_t *p_addr, uint32_t interval, void *p_priv)	Connect to a remote sensor/device and optionally browse remote services.
int ble_disconnect (cfw_service_conn_t *p_service_conn, uint16_t conn_handle, void *p_priv)	Disconnect remote device/sensor (peripheral & central role).
int ble_test (cfw_service_conn_t *p_service_conn, const struct ble_test_cmd *p_cmd, void *p_priv)	Enable/disable BLE test mode.
int ble_start_advertisement (cfw_service_conn_t *p_service_conn, uint32_t options, const struct ble_adv_data_params *p_adv_params, void *p_priv)	Start advertisement of initialized services.
int ble_stop_advertisement (cfw_service_conn_t *p_service_conn, void *p_priv)	Stop advertisement of initialized services.
int ble_clear_bonds (cfw_service_conn_t *p_service_conn, void *p_priv)	Request to clear BLE bonds.
int ble_get_security_status (cfw_service_conn_t *p_service_conn, uint8_t op, const union ble_get_security_params *p_params, void *p_priv)	Get Current security status.
int ble_set_security_status (cfw_service_conn_t *p_service_conn, uint8_t op, const union ble_set_sec_params *p_params, void *p_priv)	Set current security status.
int ble_update_service_data (cfw_service_conn_t *p_service_conn, uint16_t conn_handle, const struct ble_char_data *p_params, void *p_priv)	Update GATT server characteristic value data
int ble_conn_update (cfw_service_conn_t *p_service_conn, uint16_t conn_handle, const struct ble_gap_connection_params *p_params, void *p_priv)	Request to update the connection.
int ble_send_passkey (cfw_service_conn_t *p_service_conn, uint16_t conn_handle, const struct ble_gap_sm_passkey *p_params, void *p_priv)	Send authentication passkey.
int ble_set_rssi_report (cfw_service_conn_t *p_service_conn, const struct rssi_report_params *params, void *p_priv)	Request to send RSSI report request.
int ble_service_get_version (cfw_service_conn_t *p_service_conn, void *p_priv)	Request to read ble_core version.
int ble_service_get_info (cfw_service_conn_t *p_service_conn, uint8_t info_type, void *p_priv)	Request to read device information (current BD address and GAP name)

11.3 How to write a customized BLE Service

BLE sample services running on device with ISPC (Intel Software Platform for Curie) are BAS, DIS, ISPP services. If the user wants to write a custom service then BAS or ISPP services can be used as references.

There are three major tasks that need to be implemented in the Service.

- **Initialization of the service:** BLE GATT service initialization should be done with the **bt_gatt_register()** interface. This interface takes **struct bt_gatt_attr** as input parameters. This structure should be initialized with service UUID, Characteristics UUID, Flags (Read, write, write without response, write encrypt..),read/write callbacks, client characteristic configurations, Client User format. All these information can be easily initialized with Macro
    - **BT_GATT_PRIMARY_SERVICE:** This is used to initialize the Service UUID.
    - **BT_GATT_CHARACTERISTIC:** Used for initializing Service Characteristic UUID, flags(write, read..), read/write callback.
    - **BT_GATT_DESCRIPTOR:** Initializes the read/write/encrypt write flags, read, callback, write callback, data
    - **BT_GATT_CCC:** Used to initialize Client Characteristic Configurations.
    - **BT_GATT_CUD:** Used to initialize User format Configurations.

      Note: BT_GATT_CUD is optional.

    BT_GATT_CHARACTERISTIC, BT_GATT_DESCRIPTOR, BT_GATT_CCC, BT_GATT_CUD should be done for each and every characteristics. Once the **struct bt_gatt_attr** is initialized this is passed to **bt_gatt_register().**The service is registered to the GATT server is complete. When a device is paired to the Phone/Host device and host sends data using this service to the BLE ISPC device. Data is send to the write callback registered (passed in **BT_GATT_DESCRIPTOR**) for that characteristics.
- **Sending the data to the Host (Android/IOS BLE devices):** Paired and Connected devices can start sending and receiving data with the registered service. The interfaces which should be used are:
    - **ble_gatt_notify():**This interface is used to send the data to the Host device(Android/IOS).This interface required the service handle, characteristics handle, data and data Len to be passed as input parameters. The service handle is returned in svc registration complete callback when the ble service is initialized from app using **MSG_ID_BLE_INIT_SVC_REQ** event, characteristic handle can be retrieved using **ble_attr_idx_to_handle()** interface.
    - **ble_attr_idx_to_handle()** requires 2 input parameters, **struct bt_gatt_attr** and the index of **BT_GATT_CHARACTERISTIC()** in struct bt_gatt_attr array.

    User should be able to send data successfully.
- **Reading the Data received from the Host (Android/IOS BLE device):** The data sent from Host to the device running **ISPC** is received to the service by a write callback that is registered to the Zephyr interface using **BT_GATT_DESCRIPTOR.** This callback can further process the received data**.**

Note: For references you can look into the sample service like BAS (battery services). For Zephyr interfaces please look into **framework/include/zephyr/bluetooth/gatt.h** file. If user wants to create more services then user need to configure the number of max services using **CONFIG_BT_GATT_BLE_MAX_SERVICES.**

12 ADC Service

This service is managing ADC (analog-to-digital converter) and is providing following features:

- Get ADC value for a channel

Figure 15 ADC service architecture

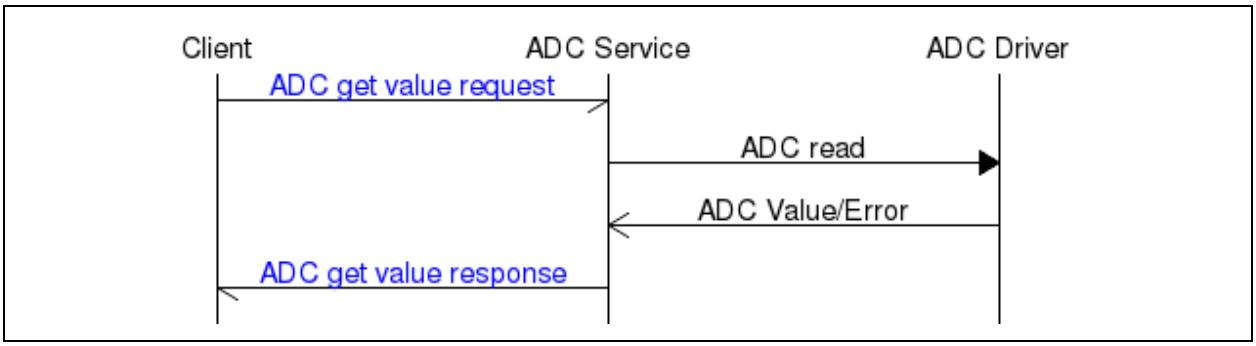


Table 10 ADC service APIs

void adc_service_init (T_QUEUE queue, int service_id)	Initialize/register the ADC service
uint32_t adc_get_value (cfw_service_conn_t * service_conn, uint32_t channel, void *priv)	Get ADC sample value for a channel.

§

13 Properties Service

The Properties Service allows the feature where any client can back up a property (service_id/property_id) into the nonvolatile memory. It provides file system-like operations where a Property (buffer) is defined with two IDs (the first id that can mimic a directory name, and the second id that can mimic a file name). Property (buffer) characteristics like its max size and its storage class can't be changed after being created. Properties can be spread across multiple blocks, but one property can't be stored across two blocks. The max size of all properties inside a property block is the block size minus the header minus "offset header". The "offset header" holds the service_id/property_id/offset to locate the property inside the block.

Properties service is suitable for storing data with a low update rate:

- Erased during factory reset:
  - product/application specific data
  - user settings, paired devices
  - Those settings are managed in at least two flash erase blocks to avoid losing any data (for example, battery removal use case).
- Never erased during factory reset:
  - System properties that hold for example battery charging cycle count.
  - Those settings are also managed in at least two flash erase blocks to avoid losing any data (for example, battery removal use case)
- Clients of this interface can be:
  - **add -** add a service/property if not already existing.
  - **read** - read the latest version of a property.
  - **write** - update the data for a given property.
  - **remove** - remove a service/property - NOT SUPPORTED

Table 11 Properties service APIs

void property_service_init (void *queue)	Initialize properties service.
int properties_service_read (cfw_service_conn_t *conn, uint16_t service_id, uint16_t property_id, void *priv)	Read property value.
int properties_service_write (cfw_service_conn_t *conn, uint16_t service_id, uint16_t property_id, void *buffer, uint16_t size, void *priv)	Change the value of a property.
int properties_service_add (cfw_service_conn_t *conn, uint16_t service_id, uint16_t property_id, bool factory_rest_persistent, void *buffer, uint16_t size, void *priv)	Add property structure to nonvolatile memory.
int properties_service_remove (cfw_service_conn_t *conn, uint16_t service_id, uint16_t property_id, void *priv)	Remove property from nonvolatile memory

§

14 GPIO Service

The *GPIO service* manages GPIOs and provides the following features:

- Configure GPIO pin
- Set/get GPIO pin state
- Listen/unlisten GPIO pin state changes

There are two instances of the GPIO service:

- Host GPIO which in in charge of GPIO pins controlled by the Host
- Sensor subsystem which is in charge of the GPIO pins controlled by sensor subsystem

Note: Sensor subsystem GPIOs are fully accessible from the Host side, and vice versa.

Three kinds of GPIO are reachable:

- The ARC specific GPIO (two blocks of 8 pin), reachable via the service **SS_GPIO_SERVICE**
- The common GPIO for QRK and ARC (pin [0..31]), reachable via the service **SOC_GPIO_SERVICE**
- The AON GPIO (pin [0..5]), reachable via the service **AON_GPIO_SERVICE**

14.1 How to use GPIO pin

According to the GPIO you want to reach, you may first open a connection (*cfw_open_service_conn*) to the right GPIO service (SS_GPIO_SERVICE, SOC_GPIO_SERVICE or AON_GPIO_SERVICE).
You are now a client of GPIO service.
Next you can configure the pin in input mode or output mode using *gpio_configure* :

- Output mode: you can change the pin state using *gpio_set_state.*
- Input mode: you can retrieve the pin state using *gpio_get_state* or you can "listen" the pin using *gpio_listen ;* this way, when the gpio state changes, the GPIO service automatically sends a message to the client.

Table 12 GPIO service APIs

void gpio_service_init (void *queue, int service_id)	Intialize/register the GPIO service.
int gpio_configure (cfw_service_conn_t *service_conn, uint8_t index, uint8_t mode, void *priv)	Configure a GPIO line.
int gpio_set_state (cfw_service_conn_t *service_conn, uint8_t index, uint8_t value, void *priv)	Set the state of the GPIO line.
int gpio_get_state (cfw_service_conn_t *service_conn, void *priv)	Get state of a GPIO line.
int gpio_listen (cfw_service_conn_t *c, uint8_t pin, gpio_service_isr_mode_t mode, uint8_t debounce, void *priv)	Register to gpio state change.
int gpio_unlisten (cfw_service_conn_t *c, uint8_t pin, void *priv)	Unregister to gpio state change.

§