System Controller Firmware Porting Guide

(B0)

Generated by Doxygen 1.8.15

Mon Nov 19 2018 13:16:52

Chapter 1

Overview

The System Controller (SC) provides an abstraction to many of the underlying features of the hardware. This function runs on a Cortex-M processor which executes SC firmware (FW). This overview describes the features of the SCFW and the APIs exposed to other software components.

Features include:

- System Initialization and Boot
- System Controller Communication
- Power Management
- Resource Management
- Pad Configuration
- Timers
- Interrupts
- Miscellaneous

Due to this abstraction, some HW describbed in the SoC RM that is used by the SCFW is not directly accessible to other cores. This includes:

- All resources in the SCU subsystem (SCU M4, SCU LPUART, SCU LPI2C, etc.).
- All resource accessed via MSI links from the SCU subsystem (inc. pads, DSC, XRDC2, eCSR)
- OCRAM controller, CAAM MP, eDMA MP & LPCG
- DB STC & LPCG, IMG GPR
- GIC/IRQSTR LPCG, IRQSTR.SCU and IRQSTR.CTI
- Any other resources reserved by the port of the SCFW to the board

1.1 System Initialization and Boot

The SC firmware runs on the SCU immediately after the SCU Read-only-memory (ROM) finishes loading code/data images from the first container. It is responsible for initializing many aspects of the system. This includes additional power and clock configuration and resource isolation hardware configuration. By default, the SC firmware configures the primary boot core to own most of the resources and launches the boot core. Additional configuration can be done by boot code.

1.2 System Controller Communication

Other software components in the system communicate to the SC via an exposed API library. This library is implemented to make Remote Procedure Calls (RPC) via an underlying Inter-Processor Communication (IPC) mechanism. The IPC is facilitated by a hardware-based mailbox system.

Software components (Linux, QNX, FreeRTOS, KSDK) delivered for i.MX8 already include ports of the client API. Other 3rd parties will need to first port the API to their environment before the API can be used. The porking kit release includes archives of the client API for existing SW. These can be used as reference for porting the client API. All that needs to be implemented is the IPC layer which will utilize messaging units (MU) to communicate with the SCFW.

1.3 System Controller Services

The SCFW provides API access to all the system controller functionality exported to other software systems. This includes:

1.3.1 Power Management Service

All aspects of power management including power control, bias control, clock control, reset control, and wake-up event monitoring are grouped within the SC Power Management service.

- **Power Control** - The SC firmware is responsible for centralized management of power controls and external power management devices. It manages the power state and voltage of power domains as well as bias control. It also resets peripherals as required due to power state transitions. This is all done via the API by communicating power state needs for individual resources.
- **Clock Control** - The SC firmware is responsible for centralized management of clock controls. This includes clock sources such as oscillators and PLLs as well as clock dividers, muxes, and gates. This is all done via the API by communicating clocking needs for individual resources.
- **Reset Control** - The SC firmware is responsible for reset control. This includes booting/rebooting a partition, obtaining reset reasons, and starting/stopping of CPUs.

Before any hardware in the SoC can be used, SW must first power up the resource and enable any clocks that it requires, otherwise access will generate a bus error. The Power Management (PM) API is documented here.

1.3.2 Resource Management Service

SC firmware is responsible for managing ownership and access permissions to system resources. The features of the resource management service supported by SC firmware include:

- Management of system resources such as SoC peripherals, memory regions, and pads
- Allows resources to be partitioned into different ownership groupings that are associated with different execution environments including multiple operating systems executing on different cores, TrustZone, and hypervisor
- Associates ownership with requests from messaging units within a resource partition
- Allows memory to be divided into memory regions that are then managed like other resources
- Allows owners to configure access permissions to resources
- Configures hardware components to provide hardware enforced isolation
- Configures hardware components to directly control secure/nonsecure attribute driven on bus fabric
- Provides ownership and access permission information to other system controller functions (e.g. pad ownership information to the pad muxing functions)

Protection of resources is provided in two ways. First, the SCFW itself checks resource access rights when API calls are made that affect a specific resource. Depending on the API call, this may require that the caller be the owner, parent of the owner, or an ancestor of the owner. Second, any hardware available to enforce access controls is configured based on the RM state. This includes the configuration of IP such as XRDC2, XRDC, or RDC, as well as management pages of IP like CAAM.

The Resource Management (RM) API is documented here.

1.3.3 Pad Configuration Service

Pad configuration is managed by SC firmware. The pad configuration features supported by the SC firmware include:

- Configuring the mux, input/output connection, and low-power isolation mode.
- Configuring the technology-specific pad setting such as drive strength, pullup/pulldown, etc.
- Configuring compensation for pad groups with dual voltage capability.

The Pad (PAD) API is documented here.

1.3.4 Timer Service

Many timer oriented services are grouped within the SC Timer service. This includes watchdogs, RTC, and system counter.

- **Watchdog** - The SC firmware provides "virtual" watchdogs for all execution environments. Features include update of the watchdog timeout, start/stop of the watchdog, refresh of the watchdog, return of the watchdog status such as maximum watchdog timeout that can be set, watchdog timeout interval, and watchdog timeout interval remaining.
- **Real-Time-Clock** - The SC firmware is responsible for providing access to the RTC. Features include setting the time, getting the time, and setting alarms.
- **System Counter** - The SC firmware is responsible for providing access to the SYSCTR. Features incude setting an absolute alarm or a relative, periodic alarm. Reading is done directly via local hardware interfaces available for each CPU.

The Timer API is documented here.

1.3.5 Interrupt Service

The System Controller needs a method to inform users about asynchronous notification events. This is done via the Interrupt service. The service provides APIs to enable/disable interrupts to the user and to read the status of pending interrupts. Reading the status automatically clears any pending state. The Interrupt (IRQ) API is documented here.

1.3.6 Miscellaneous Service

On previous i.MX devices, miscellaneous features were controlled using IOMUX GPR registers with signals connected to configurable hardware. This functionality is being replaced with DSC GPR signals. SC firmware is responsible for programming the GPR signals to configure these subsystem features. The SC firmware also responsible for monitoring various temperature, voltage, and clock sensors.

- **Controls** - The SC firmware provides access to miscellaneous controls. Features include software request to set (write) miscellaneous controls and software request to get (read) miscellaneous controls.
- **Security** - The SC firmware provides access to several security functions including image loading and authentication.
- **DMA** - The SC firmware provides access to DMA channel grouping and priority functions.
- **Temp** - The SC firmware provides access to temperature sensors.

The Miscellaneous (MISC) API is documented here.

Chapter 2

Disclaimer

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein. NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.↩ com/SalesTermsandConditions. While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREE↩ NCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE P↩ LUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, C 5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobile↩ GT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS are trademarks of NXP B.V. All other product or service names are the property of their respective owners. Arm, AMBA, Arm Powered, Artisan, Cortex, Jazelle, Keil, SecurCore, Thumb, TrustZone, and ?Vision are registered trademarks of Arm Limited (or its subsidiaries) in the EU and/or elsewhere. Arm7, Arm9, Arm11, big.LITTLE, CoreLink, CoreSight, DesignStart, Mali, Mbed, NEON, POP, Sensinode, Socrates, ULINK and Versatile are trademarks of Arm Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

Chapter 3

Porting Guide

The System Controller (SC) provides an abstraction to many of the underlying features of the hardware. This function runs on a Cortex-M processor which executes SC firmware (SCFW). The SCFW is responsible for some aspects of system boot, power/clock management, and resource management. As a result, the SCFW implementation is specific to the board. Board specific implementation includes mandatory PMIC support and optional DDR configuration. It also includes optional resource partition configuration.

The board support is contained in a board module consisting of a board.c implementation file and the board.h interface file. The board.c file has to be ported to any new board.

Functionality in the board module includes:

- **Initialization** - used to initialize board data structures and sometimes HW.
- **Configuration** - used to configure the SCFW, including which resources it keeps.
- **DDR Configuration** - used to configure DDR; This function is called indirectly by the ROM with all true parameters.
- **Resource Config** - used to optionally defined and configure resource partitions required before starting other CPUs
- **PMIC Support** - initialization, power supply control, temp sensor configuration and reporting, etc.
- **Board Reset** - generate board reset; usually standard but could require something unique on some boards
- **Board Control** - interface that can be exposed to SCFW clients handing things like PMIC temp sensor access and user-controlled voltages

The board.h interface is documented in the Board Interface Module section.

3.1 Release

The SCFW is released for porting as a collection of object files, header files, make files, build scripts, and source code for the board module. Using this package, one can port board.c for a new board, compile it, and link with the delivered object files to create an SCFW binary.

SCFW is released and tested as part of other SW releases such as Linux and QNX OS releases. As a result, the SCFW port to a board must be based on the version of SCFW supplied with an OS release. SCFW object/porting packages should be supplied with OS releases.

The porting kit contains separate tar files for each SoC/version combination. These can be combined using the following command:

```
find scfw_export_mx8*.gz -exec tar --strip-components 1 --one-top-level=scfw_export_mx8 -xzvf {} \;
```

3.2 Build Environment

SCFW builds are compiled with a cross compiler and will only run on i.MX8 hardware. The SC firmware is a 32-bit program compiled using the GNU C compiler (gcc), debugged using the GNU debugger (gdb/ddd), and documented using doxygen. As a result, a GNU-compliant build environment is required. Linux must be used to compile target builds as that requires a cross compiler.

3.3 Tool Chain

This SW package builds on a Linux host machine. The toolchain for comping should be obtained from the ARM download site. The version used is the GNU Arm Embedded Toolchain: 6-2017-q2-update June 28, 2017. Download the toolchain source and follow ARM's instructions for compiling on the host Linux machine.

Install the cross-compile toolchain on the Linux host. The TOOLS environment variable then needs to be set to the directory containing the compiler directory. For example, if using the GCC 4.9 cross-compile tools chain and installing to /home/example/toolsgcc-arm-none-eabi-4_9-2015q3 then TOOLS would be set to /home/example. Building also requires srec_cat which is usually found in the Linux srecord package. Optionally the cppcheck package is also useful.

If using bash, then set the TOOLS environment variable as follows:

```
export TOOLS=<your path to dir holding the toolchian>
```

This can be added to .bash_profile.user or .bash_profile depending on the environment.

If using tcsh, then set the TOOLS environment variable as follows:

```
setenv TOOLS <your path to dir holding the toolchian>
```

This can be added to .tcshrc.user or .tcshrc depending on the environment.

3.4 Compiling the Code

The SC firmware can be fully compiled using the Makefile. The command format is:

Usage: make TARGET OPTIONS

There are two primary SoC targets:

- **qm** : i.MX8QM
- **qx** : i.MX8QX

These generate the image (scfw_tcm.bin) in their respective build directory.

There are several options that can be specified on the make command line:

Option	Action
V=0	quite output (default)
V=1	verbose output
D=0	configure for no debug
D=1	configure for debug (default)
DL=<level>	configure debug level (0-5)
M=0	no debug monitor (default)
M=1	include debug monitor
B=<board>	configure board (default=val)
DDR_CON=<file>	specify DDR config file w/o extension
R=<srev>	silicon revision
T=<test>	run tests rather than boot next core

Note the debug level will only affect the code being compiled. It will not affect the code delivered in binary (i.e. object) form.

i.MX8QM Build

The i.MX8QM targets are as follows:

Target	Action
qm	build for i.MX8QM, output in build_mx8qm
clean-qm	remove all build files for the i.MX8QM build

The i.MX8DM targets are as follows:

Target	Action
dm	build for i.MX8DM, output in build_mx8dm
clean-dm	remove all build files for the i.MX8DM build

i.MX8QX (QXP, DX) Build

The i.MX8QX targets are as follows:

Target	Action
qx	build for i.MX8QX, output in build_mx8qx
clean-qx	remove all build files for the i.MX8QX build

Generic Targets

The Makefile supports some generic targets:

Target	Action
help	display help test
clean	delete all build directories

3.5 Production

When the SCFW is compiled for release into production devices, it is critical that this is done without debug (default is debug enabled, D=1) and without the debug monitor (default is no monitor, M=0). For example:

```
make qm R=B0 D=0 M=0
```

Turning off debug will eliminate the linking of the standard C library.

3.6 Porting

Porting starts with creating a copy of a NXP-supplied board port such as that for the MEK board (e.g. mx8qx_mek). These are found in the platform/board directory. Note the validation board ports dynamically detect the PMIC which affects boot time. Production ports should not do this.

- Create a copy and name it soc_board where soc is the name of the Soc and board is the name of the board
- The Makefile should be modified to compile all the board module files. Usually this is just board.c but it could also include other source files in the board directory such as DDR configuration code, etc.
- The board.bom file should be modified to include drivers required by the board file that are not part of the standard build. This is usually just PMIC drivers. LPI2C, GPIO, etc. drivers are built in already.
- Modify board.c to provide support for the new board.
- Build the SCFW as described in the next section (e.g. make qm B=newboard).

The resulting binary can be found in the output directory (e.g. build_mx8qx) as scfw_tcm.bin.

Note the board.c file can call any of the internal functions within the SCFW including driver functions and SCFW API functions. When calling SCFW API functions, use the internal version (those without the sc_ prefix). For example, call pm_get_clock_rate() instead of sc_pm_get_clock_rate(). The internal functions don't take an IPC channel as the first parameter and instead take a calling partition number. Use SC_PT for calls intended to come from the SCU and the partition number for calls intended to come from other partitions. The API description for the latter is included here for reference.

The SCFW is built on top of the Kinetis SDK for L5K. The CMSIS, C startup, and many of the drivers come from the KSDK. KSDK drivers include GPIO, LPI2C, LPIT, LPUART, MU, and WDOG32. The GPIO an LPI2C can be used in the board.c port to interface with the SCU RGPIO and SCU LPI2C.

3.7 Porting Notes

Below are some suggestions for board porting:

- Don't modify the section marked DO NOT CHANGE.
- Do not probe for a PMIC like the NXP validation board reference does (this consumes boot time).
- Don't communicate to the PMIC until absolutely necessary (I2C is slow).
- Using the rom_caller variable in your DDR config file to avoid running from ROM is faster and easier to debug but it is too late when booting the AP cluster from DDR.
- If M4 boot time is important and if the M4 code doesn't use DDR then don't configure DDR early. Wait until after the M4 has started. An 'early' parameter is passed to board_init_ddr() to indicate which phase the call is occurring in.
- Configure any supplies always needed in board_init(). Keep in mind this is still pretty late in the boot process (only after the SCFW is loaded and run). Any supplies needed to load the SCFW from the boot device must be configured using the OTP in the PMIC.
- board_set_power_mode() and board_set_voltage() are where mapping of SoC power inputs to PMIC supplies should be done. These functions may be called as a result of a client calling sc_pm_set_resource_power_mode() on a SoC resource or when changing the frequency for some resources that then need a different voltage.
- There are eight resources defined for board-level components (board resources). These are SC_R_BOARD_R0 through SC_R_BOARD_R7.
- board_trans_resource_power() is where mapping of board resource power inputs to PMIC supplies should be done. This function is called as a result of a client calling sc_pm_set_resource_power_mode() on a board resource.
- board_set_control() should be used if SCFW clients need to be able to set the voltage for PMIC supplies to board resources. The control would be SC_C_VOLTAGE. This function is called as a result of a client calling sc_misc_set_control().
- If power supplies are shared, usage counters will need to be used to keep track of how many domains or resources are using a supply and only change its state when the counter transitions from 0 to 1 or 1 to 0.
- Drivers are provided for the NXP PF100 and PF8100/8200 PMICs. If a new PMIC needs to be supported then the code should just be part of the board code base (not a new driver).
- board_parameter() is used to return various board design option info. For example, is the PCIe driven from an internal or external clock.
- board_rsrc_avail() returns SC_TRUE if a resource exists. The default is to return that all exist. Some boards will remove power for some resource (DRC 1, or ADC). This function then has to tell the SCFW that is the case else the SCFW will atempt to access, hang, and the SCFW WDOG reset the system. This is also the case for some i.MX8 packages.
- Isolate HW for booting cores by defining resource partitions in board_system_config(). Note these partitions should then be specified in the boot container using the mkimage -p option.

3.8 Boot Flow

The i.MX8 boot architecture is described in a document by that name. By the time the SCFW runs, most of the initial images have been loaded (SCFW, SECO FW, AP IPL, M4 images, etc.). Loading of AP images is done later when the AP cores are started and run the ROM and/or AP IPL. All the loading is done by the SCU ROM and once it is done it jumps to the SCFW which has no ability to load anything further. The ROM does not start any of the CPUs. The SCFW does this only after it configures the isolation HW and is ready to field API requests from new running cores. The SCFW is also responsible for configuring the DRC if not already done by the SCU ROM via DCD. So, the basic boot flow of the SCFW is to configure the isolation HW, power up various parts of the SoC, configure the DDR, and start CPUs as requested by the ROM. It then enters a WFI loop executing API requests.

There are two primary boot modes supported by the SCFW: standard and early. In standard, no CPU is started until all the other steps are completed. At the end, all CPUs are started in the order they are requested by the ROM (which is based on the order they occur in the boot image). In early, some of the CPUs are started early before DDR is configured and before later CPUs are powered up or started. This mode is used to minimize the time from POR to M4 execution for some specific fast-boot use-cases. In this mode, early CPUs report they are done using sc_misc_boot_done().

The boot mode is configured using one of the flags in the boot container. Bit 22 of the flags (SC_BD_FLAGS_EARL↩ Y_CPU_START) must be set for early boot mode. This is done when the image is created using mkimage. The -flags argument passes a flag as a hex value. See Boot Flags for more info.

The boot flow diagrams below show the flow, including call-outs to the porting layer (board.c). Typical boot times are listed. These are measured values on existing i.MX8 reference boards. These times can change significantly as a result of board design, PMIC configuration, DDR configuration, and implementation of the porting layer. Compile is with DL=0 option to minimize debug output. The DRC is configured by the SCFW rather than by the ROM via DCD. The system config is an example that configures for all M4's to run in an isolated partition.

Standard Boot

The standard SCFW boot flow is show in the diagram below:

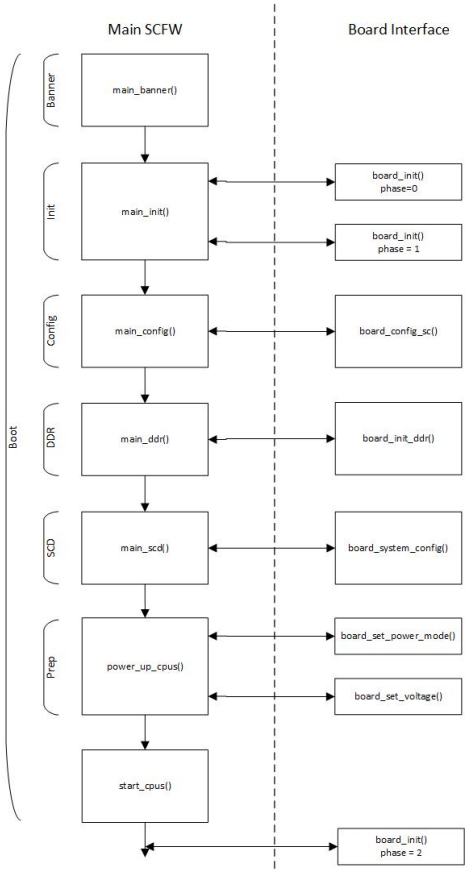

Figure 3.1 Standard SCFW Boot Flow

Details on the board.h interface are documented in the Board Interface Module section.

The boot times (measured on NXP reference boards) are as follows:

Phase	Typical Execution Time (ms)
Banner	0.1
Init	1.8
Config	3.2
DDR	5.3
SConfig	2.3
Prep	8.2
Boot	20.9

The Boot time represents the time at which all the CPUs requested to be started by the ROM will be started (relative to the time SCFW runs).

Early Boot

The early SCFW boot flow is show in the diagram below:

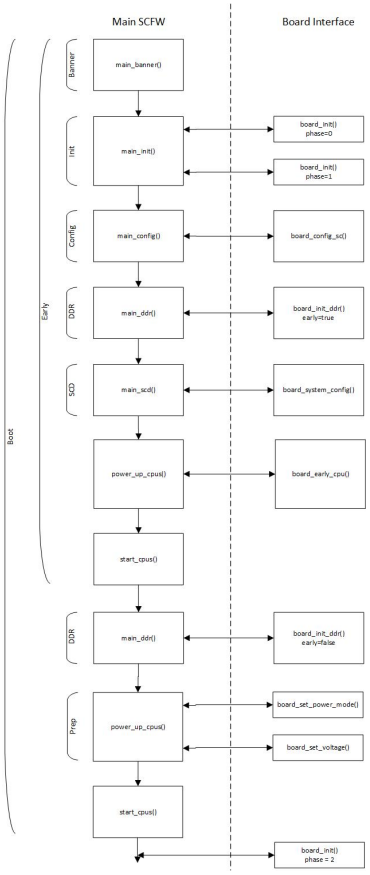

Figure 3.2 Early SCFW Boot Flow

Details on the board.h interface are documented in the Board Interface Module section.

The boot times (measured on NXP reference boards, DCD run by ROM) are as follows:

Phase	Typical Execution Time (ms)
Banner	0.1
Init	1.8
Config	2.9
DDR	0.0
SConfig	2.3
Prep	8.2
Early	7.9
Boot	16.2

The Early time represents the relative time at which the early cores will be started. The Boot time represents the relative time at which the remaining CPUs requested to be started by the ROM will be started (usually the AP core). Early cores are usually M4 cores and determined by calling board_early_cpu().

Note that starting cores early is not sufficient to guarentee they can run to the point of handling critical tasks. SCFW API calls could block if the SCFW is configuring DDR, powering up other subsystems, or starting other CPUs. As a result, when in the early boot mode, the SCFW will wait for all cores started early to make an API call to inform the SCFW that they are past the critical boot stage and okay with SCFW API delays. The API call is sc_misc_boot_done(). While waiting on these calls, the SCFW is not proceeding to boot the non-early cores so this feature, while accelerating the boot of early cores, comes as the expense of boot time for non-early cores (usually the AP cores).

PMIC fuses are normally set to insure all power supplies are enabled and the voltage is correct for starting subsystems required for initial boot. This includes supplies for all early CPUs. The overhead for communicating to the PMIC (usually via I2C) is too high in a critical boot time case. Dynamic PMIC programming should be restricted to the power up of non-early subsystems cand CPUs (typically AP cores, GPU, etc.).

3.9 Testing

When bringing up a new board, SCFW tests should be run before attempting to boot Linux. The tests are primarily used internally to NXP and not documented in detail beyond this section. In general, any hangs when running the tests indicate lack of power from the PMIC, incorrect board configuration, or improper DDR configuration. Tests are compiled in using the T=<test> option.

Test	Description
a5x	Tests Cortex-A subsystem(s) power up and access
all	Runs all automatic test
audio	Tests audio subsystem power up and access
cci	Tests CCI subsystem power up and access
conn	Tests connectivity subsystem power up and access
db	Tests DB subsystem power up and access
dblogic	Tests DBLOGIC subsystem power up and access
dc	Tests display subsystem(s) power up and access
ddr	Basic DDR test
ddr_stress	SCU-based DDR stress test
dma	Tests DMA subsystem power up and access
drc	Tests that the DRC(s) can be powered, configured, and DDR accessed
dsc	Test that all subsystems can be powered and each DSC accessed
gpu	Tests GPU subsystem(s) power up and access
hsio	Tests HSIO subsystem power up and access
img	Tests imaging subsystem power up and access
lsio	Tests LSIO subsystem power up and access
m4	Tests M4 subsystem(s) power up and access
pm	Tests power management service
pmic	Tests PMIC temp sensor(s)
rm	Tests resource management service
temp	Tests all SoC temp sensors
timer	Tests timer service
vpu	Tests VPU subsystem power up and access

The first test that should be run on a new board is the dsc test. This will determine if all resources (not removed via board_rsrc_avail()) can be powered and accessed. The drc, ddr, and ddr_stress tests should then be run to insure the DDR passes basic test. In the end, the all test should be run without hangs. Then the DDR stress test application should be run. Then OS booting can be tested.

3.10 Debug

Logging

The SCFW can log to a UART. The board.c file determines which UART will be used. Ideally, this should be the SCU UART as use of any other UART will impact power as additional subsystems have to be powered up for the SCU to access other UARTs. Default baud rate is 115,200bps.

By default, the SCFW has debug enabled (D=1), with a debug level of 2 (DL=2). The default debug categories can be found in the debug.h file. The object files in the porting package are compiled with the default debug.h settings but with DL=0. The compile of the board port files can be controlled by modifying debug.h or overriding the debug level (DL option). This will only affect compilation so only the files being compiled (usually board.c) and not the files delivered as object files.

Debugging board port files should be done using the BOARD category which is enabled by default. Use the board↩ _print() macro. The format of this macro is the same as standard printf() but with a debug level parameter as the first argument. Output will occur if the dl argument is less than or equal to the debug level. The debug level can be set at compile via the debug.h or the DL=<debug level>=""> make option.

board_print(dl, format string, optional args ...);

Output will occur if dl <= DEBUG_LEVEL.

Production SCFW should always be compiled with DL=0 to avoid extending boot time.

Debug Monitor

A debug monitor can be compiled into the SCFW using the M=1 make option. This can be used to R/W memory or registers, R/W power state, and dump some resource manager state. See Debug Monitor for more info.

Production SCFW should never have the monitor enabled (M=0, the default)!

JTAG

A JTAG debugger can also be used to debug the SCFW. Source-level debug is only available for the board port source files. Symbols are available in the object files as they are compiled with the -g -Os options. Note symbols should be stripped from final object files for production SCFW.

Chapter 4

Usage

4.1 SCFW API

Calling the functions in the SCFW requires a client API which utilizes an RPC/IPC layer to communicate to the SC↩ FW binary running on the SCU. Application environments (Linux, QNX, FreeRTOS, KSDK) delivered for i.MX8 already include ports of the client API. Other 3rd parties will need to first port the API to their environment before the API can be used. The porking kit release includes archives of the client API for existing SW. These can be used as reference for porting the client API. All that needs to be implemented is the IPC layer which will utilize messaging units (MU) to communicate with the SCFW.

The SCFW API is documented in the individual API sections. There are examples in the Details section for each service.

- Resource Management
- Power Management
- Pad Configuration
- Timers
- Interrupts
- Miscellaneous

4.2 Loading

The SCFW is loaded by including the binary (scfw_tcm.bin) in the boot container.

4.3 Boot Flags

The image container holding the SCFW should also have the boot flags configured. This is a set of flags (32-bits) specified with the -flags option to mkimage.

These are defined as:

Flag	Bit	Meaning
SC_BD_FLAGS_NOT_SECURE	16	Initial boot partition is not secure
SC_BD_FLAGS_NOT_ISOLATED	17	Initial boot partition is not isolated
SC_BD_FLAGS_RESTRICTED	18	Initial boot partition is restricted
SC_BD_FLAGS_GRANT	19	Initial boot partition grants access to the SCFW
SC_BD_FLAGS_NOT_COHERENT	20	Initial boot partition is not coherent
SC_BD_FLAGS_ALT_CONFIG	21	Alternate SCFW config (passed to board.c)
SC_BD_FLAGS_EARLY_CPU_START	22	Start some CPUs early
SC_BD_FLAGS_DDRTEST	23	Config for DDR stress test
SC_BD_FLAGS_NO_AP	24	Don't boot AP even if requested by ROM

See the sc_rm_partition_alloc() function for more info. Note some of the flags are inverted before calling this function! This results in a default (all 0) which is appropriate for normal OS execution. That is a partition which is secure, isolated, not restricted, not grant, coherent, no early start, and no DDR test.

Chapter 5

Debug Monitor

If the SCFW is compiled using the M=1 option (default is M=0) then it will include a debug monitor. The debug monitor allows command-line interaction via the SCU UART. Inclusion of the debug monitor affects SCFW timing and therefore should never be deployed in a product!

Note the terminal needs to be in a mode that sends CR or LF for a new line (not CR+LF).

The following commands are supported:

Command	Description
exit	exit the debug monitor
quit	exit the debug monitor
reset [mode]	request reset with mode (default = board)
reboot partition [type]	request partition reboot with type (default = cold)
md.b address [count]	display count bytes at address
md.w address [count]	display count words at address
md[.l] address [count]	display count long-words at address
mm.b address value	modify byte at address
mm.w address value	modify word at address
mm[.l] address value	modify long-word at address
ai.r ss sel addr	read analog interface (AI) register
ai.w ss sel addr data	write analog interface (AI) register
fuse.r word	read OTP fuse word
fuse.w word value	write value to OTP fuse word
power.r [resource]	read/get power mode of resource (default = all)
power.w resource mode	write/set power mode of resource to mode (off, stby, lp, on)
info	display SCFW/SoC info like unique ID, etc.
seco lifecycle change	send SECO lifecycle update command (change) to SECO
seco info	display SECO info like Lifecycle, SNVS state, etc.
seco debug	dump SECO debug log
seco events	dump SECO event log
seco commit	commit SRK and/or SECO FW version update
pmic.r id reg	read pmic register
pmic.w id reg val	write pmic register
pmic.l id	list pmic info (rail voltages, etc)

Resource and subsystem (ss) arguments are specified by name. All numeric arguments are decimal unless prefixed with 0x (for hex) or 0 (for octal).

Additional commands are available when the debug monitor is built from full source. See the Debug Monitor section.

Chapter 6

Module Index

6.1 Modules

Here is a list of all modules:

Chapter 7

Data Structure Index

7.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 8

File Index

8.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 9

Module Documentation

9.1 (DRV) General-Purpose Input/Output

Module for the GPIO and FPGIO drivers.

Modules

- GPIO Driver
- FGPIO Driver

Files

- file fsl_gpio.h

Data Structures

- struct gpio_pin_config_t

  The GPIO pin configuration structure.

Enumerations

- enum gpio_pin_direction_t { kGPIO_DigitalInput = 0U, kGPIO_DigitalOutput = 1U }

  GPIO direction definition.

Driver version

- #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

  GPIO driver version 2.1.0.

9.1.1 Detailed Description

Module for the GPIO and FPGIO drivers.

9.1.2 Macro Definition Documentation

9.1.2.1 FSL_GPIO_DRIVER_VERSION

```
#define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
```

GPIO driver version 2.1.0.

9.1.3 Enumeration Type Documentation

9.1.3.1 gpio_pin_direction_t

```
enum gpio_pin_direction_t
```

GPIO direction definition.

Enumerator

kGPIO_DigitalInput	Set current pin as digital input.
kGPIO_DigitalOutput	Set current pin as digital output.

9.2 GPIO Driver

GPIO Configuration

- void GPIO_PinInit (GPIO_Type *base, uint32_t pin, const gpio_pin_config_t *config)

  Initializes a GPIO pin used by the board.

GPIO Output Operations

- static void GPIO_WritePinOutput (GPIO_Type *base, uint32_t pin, uint8_t output)

  Sets the output level of the multiple GPIO pins to the logic 1 or 0.
- static void GPIO_SetPinsOutput (GPIO_Type *base, uint32_t mask)

  Sets the output level of the multiple GPIO pins to the logic 1.
- static void GPIO_ClearPinsOutput (GPIO_Type *base, uint32_t mask)

  Sets the output level of the multiple GPIO pins to the logic 0.
- static void GPIO_TogglePinsOutput (GPIO_Type *base, uint32_t mask)

  Reverses current output logic of the multiple GPIO pins.

GPIO Input Operations

- static uint32_t GPIO_ReadPinInput (GPIO_Type *base, uint32_t pin)

  Reads the current input value of the whole GPIO port.

GPIO Interrupt

- uint32_t GPIO_GetPinsInterruptFlags (GPIO_Type *base)

  Reads whole GPIO port interrupt status flag.
- void GPIO_ClearPinsInterruptFlags (GPIO_Type *base, uint32_t mask)

  Clears multiple GPIO pin interrupt status flag.

9.2.1 Detailed Description

The KSDK provides a peripheral driver for the General-Purpose Input/Output (GPIO) module of Kinetis devices.

9.2.2 Typical use case

9.2.2.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};
/*  Sets the configuration */
GPIO_PinInit(GPIO_LED, LED_PINNUM, &led_config);
```

9.2.2.2 Input Operation

```
/*  Input pin configuration */
PORT_SetPinInterruptConfig(BOARD_SW2_PORT, BOARD_SW2_GPIO_PIN, kPORT_InterruptFallingEdge);
NVIC_EnableIRQ(BOARD_SW2_IRQ);
gpio_pin_config_t sw1_config =
{
    kGpioDigitalInput,
    0,
};
/* Sets the input pin configuration */
GPIO_PinInit(GPIO_SW1, SW1_PINNUM, &sw1_config);
```

9.2.3 Function Documentation

9.2.3.1 GPIO_PinInit()

```
void GPIO_PinInit (
             GPIO_Type * base,
             uint32_t pin,
             const gpio_pin_config_t * config )
```

Initializes a GPIO pin used by the board.

To initialize the GPIO, define a pin configuration, either input or output, in the user file. Then, call the GPIO_PinInit() function.

This is an example to define an input pin or output pin configuration:

```
// Define a digital input pin configuration,
gpio_pin_config_t config =
{
  kGPIO_DigitalInput,
  0,
}
//Define a digital output pin configuration,
gpio_pin_config_t config =
{
  kGPIO_DigitalOutput,
  0,
}
```

Parameters

base	GPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
pin	GPIO port pin number
config	GPIO pin configuration pointer

9.2.3.2 GPIO_WritePinOutput()

```
static void GPIO_WritePinOutput (
             GPIO_Type * base,
```

```
            uint32_t pin,
            uint8_t output )  [inline], [static]
```

Sets the output level of the multiple GPIO pins to the logic 1 or 0.

Parameters

base	GPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
pin	GPIO pin number
output	GPIO pin output logic level. • 0: corresponding pin output low-logic level. • 1: corresponding pin output high-logic level.

9.2.3.3 GPIO_SetPinsOutput()

```
static void GPIO_SetPinsOutput (
            GPIO_Type * base,
            uint32_t mask )  [inline], [static]
```

Sets the output level of the multiple GPIO pins to the logic 1.

Parameters

base	GPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
mask	GPIO pin number macro

9.2.3.4 GPIO_ClearPinsOutput()

```
static void GPIO_ClearPinsOutput (
            GPIO_Type * base,
            uint32_t mask )  [inline], [static]
```

Sets the output level of the multiple GPIO pins to the logic 0.

Parameters

base	GPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
mask	GPIO pin number macro

9.2.3.5 GPIO_TogglePinsOutput()

```
static void GPIO_TogglePinsOutput (
             GPIO_Type * base,
             uint32_t mask )  [inline], [static]
```

Reverses current output logic of the multiple GPIO pins.

Parameters

base	GPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
mask	GPIO pin number macro

9.2.3.6 GPIO_ReadPinInput()

```
static uint32_t GPIO_ReadPinInput (
             GPIO_Type * base,
             uint32_t pin )  [inline], [static]
```

Reads the current input value of the whole GPIO port.

Parameters

base	GPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
pin	GPIO pin number

Return values

GPIO	port input value • 0: corresponding pin input low-logic level. • 1: corresponding pin input high-logic level.

9.2.3.7 GPIO_GetPinsInterruptFlags()

```
uint32_t GPIO_GetPinsInterruptFlags (
             GPIO_Type * base )
```

Reads whole GPIO port interrupt status flag.

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

base	GPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)

Return values

Current	GPIO port interrupt status flag, for example, 0x00010001 means the pin 0 and 17 have the interrupt.

9.2.3.8 GPIO_ClearPinsInterruptFlags()

```
void GPIO_ClearPinsInterruptFlags (
            GPIO_Type * base,
            uint32_t mask )
```

Clears multiple GPIO pin interrupt status flag.

Parameters

base	GPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
mask	GPIO pin number macro

9.3 FGPIO Driver

FGPIO Configuration

- void FGPIO_PinInit (FGPIO_Type *base, uint32_t pin, const gpio_pin_config_t *config)

  Initializes a FGPIO pin used by the board.

FGPIO Output Operations

- static void FGPIO_WritePinOutput (FGPIO_Type *base, uint32_t pin, uint8_t output)

  Sets the output level of the multiple FGPIO pins to the logic 1 or 0.
- static void FGPIO_SetPinsOutput (FGPIO_Type *base, uint32_t mask)

  Sets the output level of the multiple FGPIO pins to the logic 1.
- static void FGPIO_ClearPinsOutput (FGPIO_Type *base, uint32_t mask)

  Sets the output level of the multiple FGPIO pins to the logic 0.
- static void FGPIO_TogglePinsOutput (FGPIO_Type *base, uint32_t mask)

  Reverses current output logic of the multiple FGPIO pins.

FGPIO Input Operations

- static uint32_t FGPIO_ReadPinInput (FGPIO_Type *base, uint32_t pin)

  Reads the current input value of the whole FGPIO port.

FGPIO Interrupt

- uint32_t FGPIO_GetPinsInterruptFlags (FGPIO_Type *base)

  Reads the whole FGPIO port interrupt status flag.
- void FGPIO_ClearPinsInterruptFlags (FGPIO_Type *base, uint32_t mask)

  Clears the multiple FGPIO pin interrupt status flag.

9.3.1 Detailed Description

This section describes the programming interface of the FGPIO driver. The FGPIO driver configures the FGPIO module and provides a functional interface to build the GPIO application.

Note

FGPIO (Fast GPIO) is only available in a few MCUs. FGPIO and GPIO share the same peripheral but use different registers. FGPIO is closer to the core than the regular GPIO and it's faster to read and write.

9.3.2 Typical use case

9.3.2.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};
/*  Sets the configuration */
FGPIO_PinInit(FGPIO_LED, LED_PINNUM, &led_config);
```

9.3.2.2 Input Operation

```
/*  Input pin configuration */
PORT_SetPinInterruptConfig(BOARD_SW2_PORT, BOARD_SW2_FGPIO_PIN, kPORT_InterruptFallingEdge);
NVIC_EnableIRQ(BOARD_SW2_IRQ);
gpio_pin_config_t sw1_config =
{
    kGpioDigitalInput,
    0,
};
/* Sets the input pin configuration */
FGPIO_PinInit(FGPIO_SW1, SW1_PINNUM, &sw1_config);
```

9.3.3 Function Documentation

9.3.3.1 FGPIO_PinInit()

```
void FGPIO_PinInit (
             FGPIO_Type * base,
             uint32_t pin,
             const gpio_pin_config_t * config )
```

Initializes a FGPIO pin used by the board.

To initialize the FGPIO driver, define a pin configuration, either input or output, in the user file. Then, call the FGPIO_PinInit() function.

This is an example to define an input pin or output pin configuration:

```
// Define a digital input pin configuration,
gpio_pin_config_t config =
{
  kGPIO_DigitalInput,
  0,
}
//Define a digital output pin configuration,
gpio_pin_config_t config =
{
  kGPIO_DigitalOutput,
  0,
}
```

Parameters

base	FGPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
pin	FGPIO port pin number
config	FGPIO pin configuration pointer

9.3.3.2 FGPIO_WritePinOutput()

```
static void FGPIO_WritePinOutput (
            FGPIO_Type * base,
            uint32_t pin,
            uint8_t output )  [inline], [static]
```

Sets the output level of the multiple FGPIO pins to the logic 1 or 0.

Parameters

base	FGPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
pin	FGPIO pin number
output	FGPIOpin output logic level. • 0: corresponding pin output low-logic level. • 1: corresponding pin output high-logic level.

9.3.3.3 FGPIO_SetPinsOutput()

```
static void FGPIO_SetPinsOutput (
            FGPIO_Type * base,
            uint32_t mask )  [inline], [static]
```

Sets the output level of the multiple FGPIO pins to the logic 1.

Parameters

base	FGPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
mask	FGPIO pin number macro

9.3.3.4 FGPIO_ClearPinsOutput()

```
static void FGPIO_ClearPinsOutput (
            FGPIO_Type * base,
            uint32_t mask )  [inline], [static]
```

Sets the output level of the multiple FGPIO pins to the logic 0.

Parameters

base	FGPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
mask	FGPIO pin number macro

9.3.3.5 FGPIO_TogglePinsOutput()

```
static void FGPIO_TogglePinsOutput (
             FGPIO_Type * base,
             uint32_t mask )  [inline], [static]
```

Reverses current output logic of the multiple FGPIO pins.

Parameters

base	FGPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
mask	FGPIO pin number macro

9.3.3.6 FGPIO_ReadPinInput()

```
static uint32_t FGPIO_ReadPinInput (
             FGPIO_Type * base,
             uint32_t pin )  [inline], [static]
```

Reads the current input value of the whole FGPIO port.

Parameters

base	FGPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
pin	FGPIO pin number

Return values

FGPIO	port input value • 0: corresponding pin input low-logic level. • 1: corresponding pin input high-logic level.

9.3.3.7 FGPIO_GetPinsInterruptFlags()

```
uint32_t FGPIO_GetPinsInterruptFlags (
             FGPIO_Type * base )
```

Reads the whole FGPIO port interrupt status flag.

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

base	FGPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)

Return values

Current	FGPIO port interrupt status flags, for example, 0x00010001 means the pin 0 and 17 have the interrupt.

9.3.3.8 FGPIO_ClearPinsInterruptFlags()

```
void FGPIO_ClearPinsInterruptFlags (
             FGPIO_Type * base,
             uint32_t mask )
```

Clears the multiple FGPIO pin interrupt status flag.

Parameters

base	FGPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
mask	FGPIO pin number macro

9.4 (DRV) Low Power I2C Driver

Module for the LPI2C driver.

Modules

- LPI2C Master Driver
- LPI2C Slave Driver
- LPI2C Master DMA Driver
- LPI2C Slave DMA Driver
- LPI2C FreeRTOS Driver
- LPI2C µCOS/II Driver
- LPI2C µCOS/III Driver

Files

- file fsl_lpi2c.h

Enumerations

- enum _lpi2c_status {
  kStatus_LPI2C_Busy = MAKE_STATUS(kStatusGroup_LPI2C, 0), kStatus_LPI2C_Idle = MAKE_STATUS(k↩
  StatusGroup_LPI2C, 1), kStatus_LPI2C_Nak = MAKE_STATUS(kStatusGroup_LPI2C, 2), kStatus_LPI2C_FifoError
  = MAKE_STATUS(kStatusGroup_LPI2C, 3),
  kStatus_LPI2C_BitError = MAKE_STATUS(kStatusGroup_LPI2C, 4), kStatus_LPI2C_ArbitrationLost = MAKE↩
  _STATUS(kStatusGroup_LPI2C, 5), kStatus_LPI2C_PinLowTimeout, kStatus_LPI2C_NoTransferInProgress,
  kStatus_LPI2C_DmaRequestFail = MAKE_STATUS(kStatusGroup_LPI2C, 7) }

  LPI2C status return codes.

Driver version

- #define FSL_LPI2C_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

  LPI2C driver version 2.1.0.

9.4.1 Detailed Description

Module for the LPI2C driver.

9.4.2 Macro Definition Documentation

9.4.2.1 FSL_LPI2C_DRIVER_VERSION

```
#define FSL_LPI2C_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
```

LPI2C driver version 2.1.0.

9.4.3 Enumeration Type Documentation

9.4.3.1 _lpi2c_status

```
enum _lpi2c_status
```

LPI2C status return codes.

Enumerator

kStatus_LPI2C_Busy	The master is already performing a transfer.
kStatus_LPI2C_Idle	The slave driver is idle.
kStatus_LPI2C_Nak	The slave device sent a NAK in response to a byte.
kStatus_LPI2C_FifoError	FIFO under run or overrun.
kStatus_LPI2C_BitError	Transferred bit was not seen on the bus.
kStatus_LPI2C_ArbitrationLost	Arbitration lost error.
kStatus_LPI2C_PinLowTimeout	SCL or SDA were held low longer than the timeout.
kStatus_LPI2C_NoTransferInProgress	Attempt to abort a transfer when one is not in progress.
kStatus_LPI2C_DmaRequestFail	DMA request failed.

9.5 LPI2C Master Driver

Data Structures

- struct lpi2c_master_config_t

  Structure with settings to initialize the LPI2C master module.
- struct lpi2c_data_match_config_t

  LPI2C master data match configuration structure.
- struct lpi2c_master_transfer_t

  Non-blocking transfer descriptor structure.
- struct lpi2c_master_handle_t

  Driver handle for master non-blocking APIs.

Typedefs

- typedef void(* lpi2c_master_transfer_callback_t) (LPI2C_Type *base, lpi2c_master_handle_t *handle, status_t completionStatus, void *userData)

  Master completion callback function pointer type.

Enumerations

- enum _lpi2c_master_flags {
  kLPI2C_MasterTxReadyFlag = LPI2C_MSR_TDF_MASK, kLPI2C_MasterRxReadyFlag = LPI2C_MSR_RDF↩
  _MASK, kLPI2C_MasterEndOfPacketFlag = LPI2C_MSR_EPF_MASK, kLPI2C_MasterStopDetectFlag = LPI2↩
  C_MSR_SDF_MASK,
  kLPI2C_MasterNackDetectFlag = LPI2C_MSR_NDF_MASK, kLPI2C_MasterArbitrationLostFlag = LPI2C_M↩
  SR_ALF_MASK, kLPI2C_MasterFifoErrFlag = LPI2C_MSR_FEF_MASK, kLPI2C_MasterPinLowTimeoutFlag =
  LPI2C_MSR_PLTF_MASK,
  kLPI2C_MasterDataMatchFlag = LPI2C_MSR_DMF_MASK, kLPI2C_MasterBusyFlag = LPI2C_MSR_MBF_M↩
  ASK, kLPI2C_MasterBusBusyFlag = LPI2C_MSR_BBF_MASK }

  LPI2C master peripheral flags.
- enum lpi2c_direction_t { kLPI2C_Write = 0U, kLPI2C_Read = 1U }

  Direction of master and slave transfers.
- enum lpi2c_master_pin_config_t {
  kLPI2C_2PinOpenDrain = 0x0U, kLPI2C_2PinOutputOnly = 0x1U, kLPI2C_2PinPushPull = 0x2U, kLPI2C_4PinPushPull
  = 0x3U,
  kLPI2C_2PinOpenDrainWithSeparateSlave, kLPI2C_2PinOutputOnlyWithSeparateSlave, kLPI2C_2PinPushPullWithSeparateSlave
  kLPI2C_4PinPushPullWithInvertedOutput = 0x7U }

  LPI2C pin configuration.
- enum lpi2c_host_request_source_t { kLPI2C_HostRequestExternalPin = 0x0U, kLPI2C_HostRequestInputTrigger
  = 0x1U }

  LPI2C master host request selection.
- enum lpi2c_host_request_polarity_t { kLPI2C_HostRequestPinActiveLow = 0x0U, kLPI2C_HostRequestPinActiveHigh
  = 0x1U }

  LPI2C master host request pin polarity configuration.

- enum lpi2c_data_match_config_mode_t { kLPI2C_MatchDisabled = 0x0U, kLPI2C_1stWordEqualsM0OrM1 = 0x2U, kLPI2C_AnyWordEqualsM0OrM1 = 0x3U, kLPI2C_1stWordEqualsM0And2ndWordEqualsM1, kLPI2C_AnyWordEqualsM0AndNextWordEqualsM1, kLPI2C_1stWordAndM1EqualsM0AndM1, kLPI2C_AnyWordAndM1EqualsM0 }

  LPI2C master data match configuration modes.
- enum _lpi2c_master_transfer_flags { kLPI2C_TransferDefaultFlag = 0x00U, kLPI2C_TransferNoStartFlag = 0x01U, kLPI2C_TransferRepeatedStartFlag = 0x02U, kLPI2C_TransferNoStopFlag = 0x04U }

  Transfer option flags.

Initialization and deinitialization

- void LPI2C_MasterGetDefaultConfig (lpi2c_master_config_t *masterConfig)

  Provides a default configuration for the LPI2C master peripheral.
- void LPI2C_MasterInit (LPI2C_Type *base, const lpi2c_master_config_t *masterConfig, uint32_t sourceClock↩_Hz)

  Initializes the LPI2C master peripheral.
- void LPI2C_MasterDeinit (LPI2C_Type *base)

  Deinitializes the LPI2C master peripheral.
- void LPI2C_MasterConfigureDataMatch (LPI2C_Type *base, const lpi2c_data_match_config_t *config)

  Configures LPI2C master data match feature.
- static void LPI2C_MasterReset (LPI2C_Type *base)

  Performs a software reset.
- static void LPI2C_MasterEnable (LPI2C_Type *base, bool enable)

  Enables or disables the LPI2C module as master.

Status

- static uint32_t LPI2C_MasterGetStatusFlags (LPI2C_Type *base)

  Gets the LPI2C master status flags.
- static void LPI2C_MasterClearStatusFlags (LPI2C_Type *base, uint32_t statusMask)

  Clears the LPI2C master status flag state.

Interrupts

- static void LPI2C_MasterEnableInterrupts (LPI2C_Type *base, uint32_t interruptMask)

  Enables the LPI2C master interrupt requests.
- static void LPI2C_MasterDisableInterrupts (LPI2C_Type *base, uint32_t interruptMask)

  Disables the LPI2C master interrupt requests.
- static uint32_t LPI2C_MasterGetEnabledInterrupts (LPI2C_Type *base)

  Returns the set of currently enabled LPI2C master interrupt requests.

DMA control

- static void LPI2C_MasterEnableDMA (LPI2C_Type *base, bool enableTx, bool enableRx)

  Enables or disables LPI2C master DMA requests.
- static uint32_t LPI2C_MasterGetTxFifoAddress (LPI2C_Type *base)

  Gets LPI2C master transmit data register address for DMA transfer.
- static uint32_t LPI2C_MasterGetRxFifoAddress (LPI2C_Type *base)

  Gets LPI2C master receive data register address for DMA transfer.

FIFO control

- static void LPI2C_MasterSetWatermarks (LPI2C_Type *base, size_t txWords, size_t rxWords)

  Sets the watermarks for LPI2C master FIFOs.
- static void LPI2C_MasterGetFifoCounts (LPI2C_Type *base, size_t *rxCount, size_t *txCount)

  Gets the current number of words in the LPI2C master FIFOs.

Bus operations

- void LPI2C_MasterSetBaudRate (LPI2C_Type *base, uint32_t sourceClock_Hz, uint32_t baudRate_Hz)

  Sets the I2C bus frequency for master transactions.
- static bool LPI2C_MasterGetBusIdleState (LPI2C_Type *base)

  Returns whether the bus is idle.
- status_t LPI2C_MasterStart (LPI2C_Type *base, uint8_t address, lpi2c_direction_t dir)

  Sends a START signal and slave address on the I2C bus.
- static status_t LPI2C_MasterRepeatedStart (LPI2C_Type *base, uint8_t address, lpi2c_direction_t dir)

  Sends a repeated START signal and slave address on the I2C bus.
- status_t LPI2C_MasterSend (LPI2C_Type *base, const void *txBuff, size_t txSize)

  Performs a polling send transfer on the I2C bus.
- status_t LPI2C_MasterReceive (LPI2C_Type *base, void *rxBuff, size_t rxSize)

  Performs a polling receive transfer on the I2C bus.
- status_t LPI2C_MasterStop (LPI2C_Type *base)

  Sends a STOP signal on the I2C bus.

Non-blocking

- void LPI2C_MasterTransferCreateHandle (LPI2C_Type *base, lpi2c_master_handle_t *handle, lpi2c_master_transfer_callback_t callback, void *userData)

  Creates a new handle for the LPI2C master non-blocking APIs.
- status_t LPI2C_MasterTransferNonBlocking (LPI2C_Type *base, lpi2c_master_handle_t *handle, lpi2c_master↩ _transfer_t *transfer)

  Performs a non-blocking transaction on the I2C bus.
- status_t LPI2C_MasterTransferGetCount (LPI2C_Type *base, lpi2c_master_handle_t *handle, size_t *count)

  Returns number of bytes transferred so far.
- void LPI2C_MasterTransferAbort (LPI2C_Type *base, lpi2c_master_handle_t *handle)

  Terminates a non-blocking LPI2C master transmission early.

IRQ handler

- void LPI2C_MasterTransferHandleIRQ (LPI2C_Type *base, lpi2c_master_handle_t *handle)

  Reusable routine to handle master interrupts.

9.5.1 Detailed Description

9.5.2 Typedef Documentation

9.5.2.1 lpi2c_master_transfer_callback_t

```
typedef void(* lpi2c_master_transfer_callback_t) (LPI2C_Type *base, lpi2c_master_handle_t *handle,
status_t completionStatus, void *userData)
```

Master completion callback function pointer type.

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to LPI2C_MasterTransferCreateHandle().

Parameters

base	The LPI2C peripheral base address.
completionStatus	Either #kStatus_Success or an error code describing how the transfer completed.
userData	Arbitrary pointer-sized value passed from the application.

9.5.3 Enumeration Type Documentation

9.5.3.1 _lpi2c_master_flags

```
enum _lpi2c_master_flags
```

LPI2C master peripheral flags.

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag

- kLPI2C_MasterNackDetectFlag
- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Note

These enums are meant to be OR'd together to form a bit mask.

Enumerator

kLPI2C_MasterTxReadyFlag	Transmit data flag.
kLPI2C_MasterRxReadyFlag	Receive data flag.
kLPI2C_MasterEndOfPacketFlag	End Packet flag.
kLPI2C_MasterStopDetectFlag	Stop detect flag.
kLPI2C_MasterNackDetectFlag	NACK detect flag.
kLPI2C_MasterArbitrationLostFlag	Arbitration lost flag.
kLPI2C_MasterFifoErrFlag	FIFO error flag.
kLPI2C_MasterPinLowTimeoutFlag	Pin low timeout flag.
kLPI2C_MasterDataMatchFlag	Data match flag.
kLPI2C_MasterBusyFlag	Master busy flag.
kLPI2C_MasterBusBusyFlag	Bus busy flag.

9.5.3.2 lpi2c_direction_t

```
enum lpi2c_direction_t
```

Direction of master and slave transfers.

Enumerator

kLPI2C_Write	Master transmit.
kLPI2C_Read	Master receive.

9.5.3.3 lpi2c_master_pin_config_t

enum lpi2c_master_pin_config_t

LPI2C pin configuration.

Enumerator

kLPI2C_2PinOpenDrain	LPI2C Configured for 2-pin open drain mode.
kLPI2C_2PinOutputOnly	LPI2C Configured for 2-pin output only mode (ultra-fast mode)
kLPI2C_2PinPushPull	LPI2C Configured for 2-pin push-pull mode.
kLPI2C_4PinPushPull	LPI2C Configured for 4-pin push-pull mode.
kLPI2C_2PinOpenDrainWithSeparateSlave	LPI2C Configured for 2-pin open drain mode with separate LPI2C slave.
kLPI2C_2PinOutputOnlyWithSeparateSlave	LPI2C Configured for 2-pin output only mode(ultra-fast mode) with separate LPI2C slave.
kLPI2C_2PinPushPullWithSeparateSlave	LPI2C Configured for 2-pin push-pull mode with separate LPI2C slave.
kLPI2C_4PinPushPullWithInvertedOutput	LPI2C Configured for 4-pin push-pull mode(inverted outputs)

9.5.3.4 lpi2c_host_request_source_t

enum lpi2c_host_request_source_t

LPI2C master host request selection.

Enumerator

kLPI2C_HostRequestExternalPin	Select the LPI2C_HREQ pin as the host request input.
kLPI2C_HostRequestInputTrigger	Select the input trigger as the host request input.

9.5.3.5 lpi2c_host_request_polarity_t

enum lpi2c_host_request_polarity_t

LPI2C master host request pin polarity configuration.

Enumerator

kLPI2C_HostRequestPinActiveLow	Configure the LPI2C_HREQ pin active low.
kLPI2C_HostRequestPinActiveHigh	Configure the LPI2C_HREQ pin active high.

9.5.3.6 lpi2c_data_match_config_mode_t

enum lpi2c_data_match_config_mode_t

LPI2C master data match configuration modes.

Enumerator

kLPI2C_MatchDisabled	LPI2C Match Disabled.
kLPI2C_1stWordEqualsM0OrM1	LPI2C Match Enabled and 1st data word equals MATCH0 OR MATCH1.
kLPI2C_AnyWordEqualsM0OrM1	LPI2C Match Enabled and any data word equals MATCH0 OR MATCH1.
kLPI2C_1stWordEqualsM0And2ndWordEqualsM1	LPI2C Match Enabled and 1st data word equals MATCH0, 2nd data equals MATCH1.
kLPI2C_AnyWordEqualsM0AndNextWordEqualsM1	LPI2C Match Enabled and any data word equals MATCH0, next data equals MATCH1.
kLPI2C_1stWordAndM1EqualsM0AndM1	LPI2C Match Enabled and 1st data word and MATCH0 equals MATCH0 and MATCH1.
kLPI2C_AnyWordAndM1EqualsM0AndM1	LPI2C Match Enabled and any data word and MATCH0 equals MATCH0 and MATCH1.

9.5.3.7 _lpi2c_master_transfer_flags

enum _lpi2c_master_transfer_flags

Transfer option flags.

Note

These enumerations are intended to be OR'd together to form a bit mask of options for the _lpi2c_master_transfer::flags field.

Enumerator

kLPI2C_TransferDefaultFlag	Transfer starts with a start signal, stops with a stop signal.
kLPI2C_TransferNoStartFlag	Don't send a start condition, address, and sub address.
kLPI2C_TransferRepeatedStartFlag	Send a repeated start condition.
kLPI2C_TransferNoStopFlag	Don't send a stop condition.

9.5.4 Function Documentation

9.5.4.1 LPI2C_MasterGetDefaultConfig()

```
void LPI2C_MasterGetDefaultConfig (
            lpi2c_master_config_t * masterConfig )
```

Provides a default configuration for the LPI2C master peripheral.

This function provides the following default configuration for the LPI2C master peripheral:

```
masterConfig->enableMaster            = true;
masterConfig->debugEnable             = false;
masterConfig->ignoreAck               = false;
masterConfig->pinConfig               = kLPI2C_2PinOpenDrain;
masterConfig->baudRate_Hz             = 100000U;
masterConfig->busIdleTimeout_ns       = 0;
masterConfig->pinLowTimeout_ns        = 0;
masterConfig->sdaGlitchFilterWidth_ns = 0;
masterConfig->sclGlitchFilterWidth_ns = 0;
masterConfig->hostRequest.enable      = false;
masterConfig->hostRequest.source      = kLPI2C_HostRequestExternalPin;
masterConfig->hostRequest.polarity    = kLPI2C_HostRequestPinActiveHigh;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with LPI2C_MasterInit().

Parameters

out	*masterConfig*	User provided configuration structure for default values. Refer to lpi2c_master_config_t.

9.5.4.2 LPI2C_MasterInit()

```
void LPI2C_MasterInit (
            LPI2C_Type * base,
            const lpi2c_master_config_t * masterConfig,
            uint32_t sourceClock_Hz )
```

Initializes the LPI2C master peripheral.

This function enables the peripheral clock and initializes the LPI2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

base	The LPI2C peripheral base address.
masterConfig	User provided peripheral configuration. Use LPI2C_MasterGetDefaultConfig() to get a set of defaults that you can override.
sourceClock_Hz	Frequency in Hertz of the LPI2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

9.5.4.3 LPI2C_MasterDeinit()

```
void LPI2C_MasterDeinit (
            LPI2C_Type * base )
```

Deinitializes the LPI2C master peripheral.

This function disables the LPI2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

base	The LPI2C peripheral base address.

9.5.4.4 LPI2C_MasterConfigureDataMatch()

```
void LPI2C_MasterConfigureDataMatch (
            LPI2C_Type * base,
            const lpi2c_data_match_config_t * config )
```

Configures LPI2C master data match feature.

Parameters

base	The LPI2C peripheral base address.
config	Settings for the data match feature.

9.5.4.5 LPI2C_MasterReset()

```
static void LPI2C_MasterReset (
            LPI2C_Type * base )  [inline], [static]
```

Performs a software reset.

Restores the LPI2C master peripheral to reset conditions.

Parameters

base	The LPI2C peripheral base address.

9.5.4.6 LPI2C_MasterEnable()

```
static void LPI2C_MasterEnable (
            LPI2C_Type * base,
            bool enable )  [inline], [static]
```

Enables or disables the LPI2C module as master.

Parameters

base	The LPI2C peripheral base address.
enable	Pass true to enable or false to disable the specified LPI2C as master.

9.5.4.7 LPI2C_MasterGetStatusFlags()

```
static uint32_t LPI2C_MasterGetStatusFlags (
            LPI2C_Type * base )  [inline], [static]
```

Gets the LPI2C master status flags.

A bit mask with the state of all LPI2C master status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

Parameters

base	The LPI2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

See also

_lpi2c_master_flags

9.5.4.8 LPI2C_MasterClearStatusFlags()

```
static void LPI2C_MasterClearStatusFlags (
            LPI2C_Type * base,
            uint32_t statusMask )  [inline], [static]
```

Clears the LPI2C master status flag state.

The following status register flags can be cleared:

- kLPI2C_MasterEndOfPacketFlag
- kLPI2C_MasterStopDetectFlag
- kLPI2C_MasterNackDetectFlag
- kLPI2C_MasterArbitrationLostFlag
- kLPI2C_MasterFifoErrFlag
- kLPI2C_MasterPinLowTimeoutFlag
- kLPI2C_MasterDataMatchFlag

Attempts to clear other flags has no effect.

Parameters

base	The LPI2C peripheral base address.
statusMask	A bitmask of status flags that are to be cleared. The mask is composed of _lpi2c_master_flags enumerators OR'd together. You may pass the result of a previous call to LPI2C_MasterGetStatusFlags().

See also

_lpi2c_master_flags.

9.5.4.9 LPI2C_MasterEnableInterrupts()

```
static void LPI2C_MasterEnableInterrupts (
            LPI2C_Type * base,
            uint32_t interruptMask )  [inline], [static]
```

Enables the LPI2C master interrupt requests.

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Parameters

base	The LPI2C peripheral base address.
interruptMask	Bit mask of interrupts to enable. See _lpi2c_master_flags for the set of constants that should be OR'd together to form the bit mask.

9.5.4.10 LPI2C_MasterDisableInterrupts()

```
static void LPI2C_MasterDisableInterrupts (
             LPI2C_Type * base,
             uint32_t interruptMask )  [inline], [static]
```

Disables the LPI2C master interrupt requests.

All flags except kLPI2C_MasterBusyFlag and kLPI2C_MasterBusBusyFlag can be enabled as interrupts.

Parameters

base	The LPI2C peripheral base address.
interruptMask	Bit mask of interrupts to disable. See _lpi2c_master_flags for the set of constants that should be OR'd together to form the bit mask.

9.5.4.11 LPI2C_MasterGetEnabledInterrupts()

```
static uint32_t LPI2C_MasterGetEnabledInterrupts (
             LPI2C_Type * base )  [inline], [static]
```

Returns the set of currently enabled LPI2C master interrupt requests.

Parameters

base	The LPI2C peripheral base address.

Returns

A bitmask composed of _lpi2c_master_flags enumerators OR'd together to indicate the set of enabled interrupts.

9.5.4.12 LPI2C_MasterEnableDMA()

```
static void LPI2C_MasterEnableDMA (
            LPI2C_Type * base,
            bool enableTx,
            bool enableRx )  [inline], [static]
```

Enables or disables LPI2C master DMA requests.

Parameters

base	The LPI2C peripheral base address.
enableTx	Enable flag for transmit DMA request. Pass true for enable, false for disable.
enableRx	Enable flag for receive DMA request. Pass true for enable, false for disable.

9.5.4.13 LPI2C_MasterGetTxFifoAddress()

```
static uint32_t LPI2C_MasterGetTxFifoAddress (
            LPI2C_Type * base )  [inline], [static]
```

Gets LPI2C master transmit data register address for DMA transfer.

Parameters

base	The LPI2C peripheral base address.

Returns

The LPI2C Master Transmit Data Register address.

9.5.4.14 LPI2C_MasterGetRxFifoAddress()

```
static uint32_t LPI2C_MasterGetRxFifoAddress (
            LPI2C_Type * base )  [inline], [static]
```

Gets LPI2C master receive data register address for DMA transfer.

Parameters

base	The LPI2C peripheral base address.

Returns

The LPI2C Master Receive Data Register address.

9.5.4.15 LPI2C_MasterSetWatermarks()

```
static void LPI2C_MasterSetWatermarks (
            LPI2C_Type * base,
            size_t txWords,
            size_t rxWords )  [inline], [static]
```

Sets the watermarks for LPI2C master FIFOs.

Parameters

base	The LPI2C peripheral base address.
txWords	Transmit FIFO watermark value in words. The kLPI2C_MasterTxReadyFlag flag is set whenever the number of words in the transmit FIFO is equal or less than *txWords*. Writing a value equal or greater than the FIFO size is truncated.
rxWords	Receive FIFO watermark value in words. The kLPI2C_MasterRxReadyFlag flag is set whenever the number of words in the receive FIFO is greater than *rxWords*. Writing a value equal or greater than the FIFO size is truncated.

9.5.4.16 LPI2C_MasterGetFifoCounts()

```
static void LPI2C_MasterGetFifoCounts (
            LPI2C_Type * base,
            size_t * rxCount,
            size_t * txCount )  [inline], [static]
```

Gets the current number of words in the LPI2C master FIFOs.

Parameters

	base	The LPI2C peripheral base address.
out	*txCount*	Pointer through which the current number of words in the transmit FIFO is returned. Pass NULL if this value is not required.
out	*rxCount*	Pointer through which the current number of words in the receive FIFO is returned. Pass NULL if this value is not required.

9.5.4.17 LPI2C_MasterSetBaudRate()

```
void LPI2C_MasterSetBaudRate (
            LPI2C_Type * base,
            uint32_t sourceClock_Hz,
            uint32_t baudRate_Hz )
```

Sets the I2C bus frequency for master transactions.

The LPI2C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Parameters

base	The LPI2C peripheral base address.
sourceClock_Hz	LPI2C functional clock frequency in Hertz.
baudRate_Hz	Requested bus frequency in Hertz.

9.5.4.18 LPI2C_MasterGetBusIdleState()

```
static bool LPI2C_MasterGetBusIdleState (
            LPI2C_Type * base )  [inline], [static]
```

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

base	The LPI2C peripheral base address.

Return values

true	Bus is busy.
false	Bus is idle.

9.5.4.19 LPI2C_MasterStart()

```
status_t LPI2C_MasterStart (
            LPI2C_Type * base,
```

```
            uint8_t address,
            lpi2c_direction_t dir )
```

Sends a START signal and slave address on the I2C bus.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

base	The LPI2C peripheral base address.
address	7-bit slave device address, in bits [6:0].
dir	Master transfer direction, either kLPI2C_Read or kLPI2C_Write. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

#kStatus_Success	START signal and address were successfully enqueued in the transmit FIFO.
kStatus_LPI2C_Busy	Another master is currently utilizing the bus.

9.5.4.20 LPI2C_MasterRepeatedStart()

```
static status_t LPI2C_MasterRepeatedStart (
            LPI2C_Type * base,
            uint8_t address,
            lpi2c_direction_t dir )  [inline], [static]
```

Sends a repeated START signal and slave address on the I2C bus.

This function is used to send a Repeated START signal when a transfer is already in progress. Like LPI2C_MasterStart(), it also sends the specified 7-bit address.

Note

This function exists primarily to maintain compatible APIs between LPI2C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

base	The LPI2C peripheral base address.
address	7-bit slave device address, in bits [6:0].
dir	Master transfer direction, either kLPI2C_Read or kLPI2C_Write. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

#kStatus_Success	Repeated START signal and address were successfully enqueued in the transmit FIFO.
kStatus_LPI2C_Busy	Another master is currently utilizing the bus.

References LPI2C_MasterStart().

9.5.4.21 LPI2C_MasterSend()

```
status_t LPI2C_MasterSend (
            LPI2C_Type * base,
            const void * txBuff,
            size_t txSize )
```

Performs a polling send transfer on the I2C bus.

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns kStatus_LPI2C_Nak.

Parameters

base	The LPI2C peripheral base address.
txBuff	The pointer to the data to be transferred.
txSize	The length in bytes of the data to be transferred.

Return values

#kStatus_Success	Data was sent successfully.
kStatus_LPI2C_Busy	Another master is currently utilizing the bus.
kStatus_LPI2C_Nak	The slave device sent a NAK in response to a byte.
kStatus_LPI2C_FifoError	FIFO under run or over run.
kStatus_LPI2C_ArbitrationLost	Arbitration lost error.
kStatus_LPI2C_PinLowTimeout	SCL or SDA were held low longer than the timeout.

9.5.4.22 LPI2C_MasterReceive()

```
status_t LPI2C_MasterReceive (
            LPI2C_Type * base,
            void * rxBuff,
            size_t rxSize )
```

Performs a polling receive transfer on the I2C bus.

Parameters

base	The LPI2C peripheral base address.
rxBuff	The pointer to the data to be transferred.
rxSize	The length in bytes of the data to be transferred.

Return values

#kStatus_Success	Data was received successfully.
kStatus_LPI2C_Busy	Another master is currently utilizing the bus.
kStatus_LPI2C_Nak	The slave device sent a NAK in response to a byte.
kStatus_LPI2C_FifoError	FIFO under run or overrun.
kStatus_LPI2C_ArbitrationLost	Arbitration lost error.
kStatus_LPI2C_PinLowTimeout	SCL or SDA were held low longer than the timeout.

9.5.4.23 LPI2C_MasterStop()

```
status_t LPI2C_MasterStop (
             LPI2C_Type * base )
```

Sends a STOP signal on the I2C bus.

This function does not return until the STOP signal is seen on the bus, or an error occurs.

Parameters

base	The LPI2C peripheral base address.

Return values

#kStatus_Success	The STOP signal was successfully sent on the bus and the transaction terminated.
kStatus_LPI2C_Busy	Another master is currently utilizing the bus.
kStatus_LPI2C_Nak	The slave device sent a NAK in response to a byte.
kStatus_LPI2C_FifoError	FIFO under run or overrun.
kStatus_LPI2C_ArbitrationLost	Arbitration lost error.
kStatus_LPI2C_PinLowTimeout	SCL or SDA were held low longer than the timeout.

9.5.4.24 LPI2C_MasterTransferCreateHandle()

```
void LPI2C_MasterTransferCreateHandle (
             LPI2C_Type * base,
```

```
            lpi2c_master_handle_t * handle,
            lpi2c_master_transfer_callback_t callback,
            void * userData )
```

Creates a new handle for the LPI2C master non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the LPI2C_MasterTransferAbort() API shall be called.

Parameters

	base	The LPI2C peripheral base address.
out	*handle*	Pointer to the LPI2C master driver handle.
	callback	User provided pointer to the asynchronous callback function.
	userData	User provided pointer to the application callback data.

9.5.4.25 LPI2C_MasterTransferNonBlocking()

```
status_t LPI2C_MasterTransferNonBlocking (
            LPI2C_Type * base,
            lpi2c_master_handle_t * handle,
            lpi2c_master_transfer_t * transfer )
```

Performs a non-blocking transaction on the I2C bus.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to the LPI2C master driver handle.
transfer	The pointer to the transfer descriptor.

Return values

#kStatus_Success	The transaction was started successfully.
kStatus_LPI2C_Busy	Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

9.5.4.26 LPI2C_MasterTransferGetCount()

```
status_t LPI2C_MasterTransferGetCount (
            LPI2C_Type * base,
```

```
            lpi2c_master_handle_t * handle,
            size_t * count )
```

Returns number of bytes transferred so far.

Parameters

	base	The LPI2C peripheral base address.
	handle	Pointer to the LPI2C master driver handle.
out	*count*	Number of bytes transferred so far by the non-blocking transaction.

Return values

#kStatus_Success	
#kStatus_NoTransferInProgress	There is not a non-blocking transaction currently in progress.

9.5.4.27 LPI2C_MasterTransferAbort()

```
void LPI2C_MasterTransferAbort (
            LPI2C_Type * base,
            lpi2c_master_handle_t * handle )
```

Terminates a non-blocking LPI2C master transmission early.

Note

It is not safe to call this function from an IRQ handler that has a higher priority than the LPI2C peripheral's IRQ priority.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to the LPI2C master driver handle.

Return values

#kStatus_Success	A transaction was successfully aborted.
kStatus_LPI2C_Idle	There is not a non-blocking transaction currently in progress.

9.5.4.28 LPI2C_MasterTransferHandleIRQ()

```
void LPI2C_MasterTransferHandleIRQ (
            LPI2C_Type * base,
            lpi2c_master_handle_t * handle )
```

Reusable routine to handle master interrupts.

Note

This function does not need to be called unless you are reimplementing the nonblocking API's interrupt handler routines to add special functionality.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to the LPI2C master driver handle.

9.6 LPI2C Slave Driver

Data Structures

- struct lpi2c_slave_config_t

  Structure with settings to initialize the LPI2C slave module.
- struct lpi2c_slave_transfer_t

  LPI2C slave transfer structure.
- struct lpi2c_slave_handle_t

  LPI2C slave handle structure.

Typedefs

- typedef void(∗ lpi2c_slave_transfer_callback_t) (LPI2C_Type ∗base, lpi2c_slave_transfer_t ∗transfer, void ∗user↩Data)

  Slave event callback function pointer type.

Enumerations

- enum _lpi2c_slave_flags {
  kLPI2C_SlaveTxReadyFlag = LPI2C_SSR_TDF_MASK, kLPI2C_SlaveRxReadyFlag = LPI2C_SSR_RDF_MA↩SK, kLPI2C_SlaveAddressValidFlag = LPI2C_SSR_AVF_MASK, kLPI2C_SlaveTransmitAckFlag = LPI2C_SS↩R_TAF_MASK,
  kLPI2C_SlaveRepeatedStartDetectFlag = LPI2C_SSR_RSF_MASK, kLPI2C_SlaveStopDetectFlag = LPI2C_↩SSR_SDF_MASK, kLPI2C_SlaveBitErrFlag = LPI2C_SSR_BEF_MASK, kLPI2C_SlaveFifoErrFlag = LPI2C_S↩SR_FEF_MASK,
  kLPI2C_SlaveAddressMatch0Flag = LPI2C_SSR_AM0F_MASK, kLPI2C_SlaveAddressMatch1Flag = LPI2C_↩SSR_AM1F_MASK, kLPI2C_SlaveGeneralCallFlag = LPI2C_SSR_GCF_MASK, kLPI2C_SlaveBusyFlag = LP↩I2C_SSR_SBF_MASK,
  kLPI2C_SlaveBusBusyFlag = LPI2C_SSR_BBF_MASK }

  LPI2C slave peripheral flags.
- enum lpi2c_slave_address_match_t { kLPI2C_MatchAddress0 = 0U, kLPI2C_MatchAddress0OrAddress1 = 2U, kLPI2C_MatchAddress0ThroughAddress1 = 6U }

  LPI2C slave address match options.
- enum lpi2c_slave_transfer_event_t {
  kLPI2C_SlaveAddressMatchEvent = 0x01U, kLPI2C_SlaveTransmitEvent = 0x02U, kLPI2C_SlaveReceiveEvent = 0x04U, kLPI2C_SlaveTransmitAckEvent = 0x08U,
  kLPI2C_SlaveRepeatedStartEvent = 0x10U, kLPI2C_SlaveCompletionEvent = 0x20U, kLPI2C_SlaveAllEvents }

  Set of events sent to the callback for non blocking slave transfers.

Slave initialization and deinitialization

- void LPI2C_SlaveGetDefaultConfig (lpi2c_slave_config_t ∗slaveConfig)

  Provides a default configuration for the LPI2C slave peripheral.
- void LPI2C_SlaveInit (LPI2C_Type ∗base, const lpi2c_slave_config_t ∗slaveConfig, uint32_t sourceClock_Hz)

  Initializes the LPI2C slave peripheral.
- void LPI2C_SlaveDeinit (LPI2C_Type ∗base)

  Deinitializes the LPI2C slave peripheral.
- static void LPI2C_SlaveReset (LPI2C_Type ∗base)

  Performs a software reset of the LPI2C slave peripheral.
- static void LPI2C_SlaveEnable (LPI2C_Type ∗base, bool enable)

  Enables or disables the LPI2C module as slave.

Slave status

- static uint32_t LPI2C_SlaveGetStatusFlags (LPI2C_Type *base)

  Gets the LPI2C slave status flags.
- static void LPI2C_SlaveClearStatusFlags (LPI2C_Type *base, uint32_t statusMask)

  Clears the LPI2C status flag state.

Slave interrupts

- static void LPI2C_SlaveEnableInterrupts (LPI2C_Type *base, uint32_t interruptMask)

  Enables the LPI2C slave interrupt requests.
- static void LPI2C_SlaveDisableInterrupts (LPI2C_Type *base, uint32_t interruptMask)

  Disables the LPI2C slave interrupt requests.
- static uint32_t LPI2C_SlaveGetEnabledInterrupts (LPI2C_Type *base)

  Returns the set of currently enabled LPI2C slave interrupt requests.

Slave DMA control

- static void LPI2C_SlaveEnableDMA (LPI2C_Type *base, bool enableAddressValid, bool enableRx, bool enable↩ Tx)

  Enables or disables the LPI2C slave peripheral DMA requests.

Slave bus operations

- static bool LPI2C_SlaveGetBusIdleState (LPI2C_Type *base)

  Returns whether the bus is idle.
- static void LPI2C_SlaveTransmitAck (LPI2C_Type *base, bool ackOrNack)

  Transmits either an ACK or NAK on the I2C bus in response to a byte from the master.
- static uint32_t LPI2C_SlaveGetReceivedAddress (LPI2C_Type *base)

  Returns the slave address sent by the I2C master.
- status_t LPI2C_SlaveSend (LPI2C_Type *base, const void *txBuff, size_t txSize, size_t *actualTxSize)

  Performs a polling send transfer on the I2C bus.
- status_t LPI2C_SlaveReceive (LPI2C_Type *base, void *rxBuff, size_t rxSize, size_t *actualRxSize)

  Performs a polling receive transfer on the I2C bus.

Slave non-blocking

- void LPI2C_SlaveTransferCreateHandle (LPI2C_Type *base, lpi2c_slave_handle_t *handle, lpi2c_slave_transfer_callback_t callback, void *userData)

  Creates a new handle for the LPI2C slave non-blocking APIs.
- status_t LPI2C_SlaveTransferNonBlocking (LPI2C_Type *base, lpi2c_slave_handle_t *handle, uint32_t event↩ Mask)

  Starts accepting slave transfers.
- status_t LPI2C_SlaveTransferGetCount (LPI2C_Type *base, lpi2c_slave_handle_t *handle, size_t *count)

  Gets the slave transfer status during a non-blocking transfer.
- void LPI2C_SlaveTransferAbort (LPI2C_Type *base, lpi2c_slave_handle_t *handle)

  Aborts the slave non-blocking transfers.

Slave IRQ handler

- void LPI2C_SlaveTransferHandleIRQ (LPI2C_Type *base, lpi2c_slave_handle_t *handle)

  Reusable routine to handle slave interrupts.

9.6.1 Detailed Description

9.6.2 Typedef Documentation

9.6.2.1 lpi2c_slave_transfer_callback_t

```
typedef void(* lpi2c_slave_transfer_callback_t) (LPI2C_Type *base, lpi2c_slave_transfer_t *transfer,
void *userData)
```

Slave event callback function pointer type.

This callback is used only for the slave non-blocking transfer API. To install a callback, use the LPI2C_SlaveSetCallback() function after you have created a handle.

Parameters

base	Base address for the LPI2C instance on which the event occurred.
transfer	Pointer to transfer descriptor containing values passed to and/or from the callback.
userData	Arbitrary pointer-sized value passed from the application.

9.6.3 Enumeration Type Documentation

9.6.3.1 _lpi2c_slave_flags

```
enum _lpi2c_slave_flags
```

LPI2C slave peripheral flags.

The following status register flags can be cleared:

- kLPI2C_SlaveRepeatedStartDetectFlag
- kLPI2C_SlaveStopDetectFlag
- kLPI2C_SlaveBitErrFlag
- kLPI2C_SlaveFifoErrFlag

All flags except kLPI2C_SlaveBusyFlag and kLPI2C_SlaveBusBusyFlag can be enabled as interrupts.

Note

These enumerations are meant to be OR'd together to form a bit mask.

Enumerator

kLPI2C_SlaveTxReadyFlag	Transmit data flag.
kLPI2C_SlaveRxReadyFlag	Receive data flag.
kLPI2C_SlaveAddressValidFlag	Address valid flag.
kLPI2C_SlaveTransmitAckFlag	Transmit ACK flag.
kLPI2C_SlaveRepeatedStartDetectFlag	Repeated start detect flag.
kLPI2C_SlaveStopDetectFlag	Stop detect flag.
kLPI2C_SlaveBitErrFlag	Bit error flag.
kLPI2C_SlaveFifoErrFlag	FIFO error flag.
kLPI2C_SlaveAddressMatch0Flag	Address match 0 flag.
kLPI2C_SlaveAddressMatch1Flag	Address match 1 flag.
kLPI2C_SlaveGeneralCallFlag	General call flag.
kLPI2C_SlaveBusyFlag	Master busy flag.
kLPI2C_SlaveBusBusyFlag	Bus busy flag.

9.6.3.2 lpi2c_slave_address_match_t

`enum lpi2c_slave_address_match_t`

LPI2C slave address match options.

Enumerator

kLPI2C_MatchAddress0	Match only address 0.
kLPI2C_MatchAddress0OrAddress1	Match either address 0 or address 1.
kLPI2C_MatchAddress0ThroughAddress1	Match a range of slave addresses from address 0 through address 1.

9.6.3.3 lpi2c_slave_transfer_event_t

`enum lpi2c_slave_transfer_event_t`

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to LPI2C_SlaveTransferNonBlocking() in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note

These enumerations are meant to be OR'd together to form a bit mask of events.

Enumerator

kLPI2C_SlaveAddressMatchEvent	Received the slave address after a start or repeated start.
kLPI2C_SlaveTransmitEvent	Callback is requested to provide data to transmit (slave-transmitter role).
kLPI2C_SlaveReceiveEvent	Callback is requested to provide a buffer in which to place received data (slave-receiver role).
kLPI2C_SlaveTransmitAckEvent	Callback needs to either transmit an ACK or NACK.
kLPI2C_SlaveRepeatedStartEvent	A repeated start was detected.
kLPI2C_SlaveCompletionEvent	A stop was detected, completing the transfer.
kLPI2C_SlaveAllEvents	Bit mask of all available events.

9.6.4 Function Documentation

9.6.4.1 LPI2C_SlaveGetDefaultConfig()

```
void LPI2C_SlaveGetDefaultConfig (
            lpi2c_slave_config_t * slaveConfig )
```

Provides a default configuration for the LPI2C slave peripheral.

This function provides the following default configuration for the LPI2C slave peripheral:

```
slaveConfig->enableSlave               = true;
slaveConfig->address0                  = 0U;
slaveConfig->address1                  = 0U;
slaveConfig->addressMatchMode          = kLPI2C_MatchAddress0;
slaveConfig->filterDozeEnable          = true;
slaveConfig->filterEnable              = true;
slaveConfig->enableGeneralCall         = false;
slaveConfig->sclStall.enableAck        = false;
slaveConfig->sclStall.enableTx         = true;
slaveConfig->sclStall.enableRx         = true;
slaveConfig->sclStall.enableAddress    = true;
slaveConfig->ignoreAck                 = false;
slaveConfig->enableReceivedAddressRead = false;
slaveConfig->sdaGlitchFilterWidth_ns   = 0; // TODO determine default width values
slaveConfig->sclGlitchFilterWidth_ns   = 0;
slaveConfig->dataValidDelay_ns         = 0;
slaveConfig->clockHoldTime_ns          = 0;
```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with LPI2C_SlaveInit(). Be sure to override at least the *address0* member of the configuration structure with the desired slave address.

Parameters

out	*slaveConfig*	User provided configuration structure that is set to default values. Refer to lpi2c_slave_config_t.

9.6.4.2 LPI2C_SlaveInit()

```
void LPI2C_SlaveInit (
            LPI2C_Type * base,
            const lpi2c_slave_config_t * slaveConfig,
            uint32_t sourceClock_Hz )
```

Initializes the LPI2C slave peripheral.

This function enables the peripheral clock and initializes the LPI2C slave peripheral as described by the user provided configuration.

Parameters

base	The LPI2C peripheral base address.
slaveConfig	User provided peripheral configuration. Use LPI2C_SlaveGetDefaultConfig() to get a set of defaults that you can override.
sourceClock_Hz	Frequency in Hertz of the LPI2C functional clock. Used to calculate the filter widths, data valid delay, and clock hold time.

9.6.4.3 LPI2C_SlaveDeinit()

```
void LPI2C_SlaveDeinit (
            LPI2C_Type * base )
```

Deinitializes the LPI2C slave peripheral.

This function disables the LPI2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

base	The LPI2C peripheral base address.

9.6.4.4 LPI2C_SlaveReset()

```
static void LPI2C_SlaveReset (
            LPI2C_Type * base )  [inline], [static]
```

Performs a software reset of the LPI2C slave peripheral.

Parameters

base	The LPI2C peripheral base address.

9.6.4.5 LPI2C_SlaveEnable()

```
static void LPI2C_SlaveEnable (
            LPI2C_Type * base,
            bool enable )  [inline], [static]
```

Enables or disables the LPI2C module as slave.

Parameters

base	The LPI2C peripheral base address.
enable	Pass true to enable or false to disable the specified LPI2C as slave.

9.6.4.6 LPI2C_SlaveGetStatusFlags()

```
static uint32_t LPI2C_SlaveGetStatusFlags (
            LPI2C_Type * base )  [inline], [static]
```

Gets the LPI2C slave status flags.

A bit mask with the state of all LPI2C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

Parameters

base	The LPI2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

See also

_lpi2c_slave_flags

9.6.4.7 LPI2C_SlaveClearStatusFlags()

```
static void LPI2C_SlaveClearStatusFlags (
            LPI2C_Type * base,
            uint32_t statusMask )  [inline], [static]
```

Clears the LPI2C status flag state.

The following status register flags can be cleared:

- kLPI2C_SlaveRepeatedStartDetectFlag
- kLPI2C_SlaveStopDetectFlag
- kLPI2C_SlaveBitErrFlag
- kLPI2C_SlaveFifoErrFlag

Attempts to clear other flags has no effect.

Parameters

base	The LPI2C peripheral base address.
statusMask	A bitmask of status flags that are to be cleared. The mask is composed of _lpi2c_slave_flags enumerators OR'd together. You may pass the result of a previous call to LPI2C_SlaveGetStatusFlags().

See also

_lpi2c_slave_flags.

9.6.4.8 LPI2C_SlaveEnableInterrupts()

```
static void LPI2C_SlaveEnableInterrupts (
            LPI2C_Type * base,
            uint32_t interruptMask )  [inline], [static]
```

Enables the LPI2C slave interrupt requests.

All flags except kLPI2C_SlaveBusyFlag and kLPI2C_SlaveBusBusyFlag can be enabled as interrupts.

Parameters

base	The LPI2C peripheral base address.
interruptMask	Bit mask of interrupts to enable. See _lpi2c_slave_flags for the set of constants that should be OR'd together to form the bit mask.

9.6.4.9 LPI2C_SlaveDisableInterrupts()

```
static void LPI2C_SlaveDisableInterrupts (
            LPI2C_Type * base,
            uint32_t interruptMask )  [inline], [static]
```

Disables the LPI2C slave interrupt requests.

All flags except kLPI2C_SlaveBusyFlag and kLPI2C_SlaveBusBusyFlag can be enabled as interrupts.

Parameters

base	The LPI2C peripheral base address.
interruptMask	Bit mask of interrupts to disable. See _lpi2c_slave_flags for the set of constants that should be OR'd together to form the bit mask.

9.6.4.10 LPI2C_SlaveGetEnabledInterrupts()

```
static uint32_t LPI2C_SlaveGetEnabledInterrupts (
            LPI2C_Type * base )  [inline], [static]
```

Returns the set of currently enabled LPI2C slave interrupt requests.

Parameters

base	The LPI2C peripheral base address.

Returns

A bitmask composed of _lpi2c_slave_flags enumerators OR'd together to indicate the set of enabled interrupts.

9.6.4.11 LPI2C_SlaveEnableDMA()

```
static void LPI2C_SlaveEnableDMA (
            LPI2C_Type * base,
            bool enableAddressValid,
            bool enableRx,
            bool enableTx )  [inline], [static]
```

Enables or disables the LPI2C slave peripheral DMA requests.

Parameters

base	The LPI2C peripheral base address.
enableAddressValid	Enable flag for the address valid DMA request. Pass true for enable, false for disable. The address valid DMA request is shared with the receive data DMA request.
enableRx	Enable flag for the receive data DMA request. Pass true for enable, false for disable.
enableTx	Enable flag for the transmit data DMA request. Pass true for enable, false for disable.

9.6.4.12 LPI2C_SlaveGetBusIdleState()

```
static bool LPI2C_SlaveGetBusIdleState (
            LPI2C_Type * base )  [inline], [static]
```

Returns whether the bus is idle.

Requires the slave mode to be enabled.

Parameters

base	The LPI2C peripheral base address.

Return values

true	Bus is busy.
false	Bus is idle.

9.6.4.13 LPI2C_SlaveTransmitAck()

```
static void LPI2C_SlaveTransmitAck (
            LPI2C_Type * base,
            bool ackOrNack )  [inline], [static]
```

Transmits either an ACK or NAK on the I2C bus in response to a byte from the master.

Use this function to send an ACK or NAK when the kLPI2C_SlaveTransmitAckFlag is asserted. This only happens if you enable the sclStall.enableAck field of the lpi2c_slave_config_t configuration structure used to initialize the slave peripheral.

Parameters

base	The LPI2C peripheral base address.
ackOrNack	Pass true for an ACK or false for a NAK.

9.6.4.14 LPI2C_SlaveGetReceivedAddress()

```
static uint32_t LPI2C_SlaveGetReceivedAddress (
            LPI2C_Type * base )  [inline], [static]
```

Returns the slave address sent by the I2C master.

This function should only be called if the kLPI2C_SlaveAddressValidFlag is asserted.

Parameters

base	The LPI2C peripheral base address.

Returns

The 8-bit address matched by the LPI2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

9.6.4.15 LPI2C_SlaveSend()

```
status_t LPI2C_SlaveSend (
            LPI2C_Type * base,
            const void * txBuff,
            size_t txSize,
            size_t * actualTxSize )
```

Performs a polling send transfer on the I2C bus.

Parameters

	base	The LPI2C peripheral base address.
	txBuff	The pointer to the data to be transferred.
	txSize	The length in bytes of the data to be transferred.
out	*actualTxSize*	

Returns

Error or success status returned by API.

9.6.4.16 LPI2C_SlaveReceive()

```
status_t LPI2C_SlaveReceive (
             LPI2C_Type * base,
             void * rxBuff,
             size_t rxSize,
             size_t * actualRxSize )
```

Performs a polling receive transfer on the I2C bus.

Parameters

	base	The LPI2C peripheral base address.
	rxBuff	The pointer to the data to be transferred.
	rxSize	The length in bytes of the data to be transferred.
out	*actualRxSize*	

Returns

Error or success status returned by API.

9.6.4.17 LPI2C_SlaveTransferCreateHandle()

```
void LPI2C_SlaveTransferCreateHandle (
             LPI2C_Type * base,
             lpi2c_slave_handle_t * handle,
             lpi2c_slave_transfer_callback_t callback,
             void * userData )
```

Creates a new handle for the LPI2C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the LPI2C_SlaveTransferAbort() API shall be called.

Parameters

	base	The LPI2C peripheral base address.
out	*handle*	Pointer to the LPI2C slave driver handle.
	callback	User provided pointer to the asynchronous callback function.
	userData	User provided pointer to the application callback data.

9.6.4.18 LPI2C_SlaveTransferNonBlocking()

```
status_t LPI2C_SlaveTransferNonBlocking (
            LPI2C_Type * base,
            lpi2c_slave_handle_t * handle,
            uint32_t eventMask )
```

Starts accepting slave transfers.

Call this API after calling I2C_SlaveInit() and LPI2C_SlaveTransferCreateHandle() to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to LPI2C_SlaveTransferCreateHandle(). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of lpi2c_slave_transfer_event_t enumerators for the events you wish to receive. The kLPI2C_SlaveTransmitEvent and kLPI2C_SlaveReceiveEvent events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the kLPI2C_SlaveAllEvents constant is provided as a convenient way to enable all events.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to #lpi2c_slave_handle_t structure which stores the transfer state.
eventMask	Bit mask formed by OR'ing together lpi2c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kLPI2C_SlaveAllEvents to enable all events.

Return values

#kStatus_Success	Slave transfers were successfully started.
kStatus_LPI2C_Busy	Slave transfers have already been started on this handle.

9.6.4.19 LPI2C_SlaveTransferGetCount()

```
status_t LPI2C_SlaveTransferGetCount (
            LPI2C_Type * base,
            lpi2c_slave_handle_t * handle,
            size_t * count )
```

Gets the slave transfer status during a non-blocking transfer.

Parameters

	base	The LPI2C peripheral base address.
	handle	Pointer to i2c_slave_handle_t structure.
out	*count*	Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required.

Return values

#kStatus_Success	
#kStatus_NoTransferInProgress	

9.6.4.20 LPI2C_SlaveTransferAbort()

```
void LPI2C_SlaveTransferAbort (
             LPI2C_Type * base,
             lpi2c_slave_handle_t * handle )
```

Aborts the slave non-blocking transfers.

Note

This API could be called at any time to stop slave for handling the bus events.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to #lpi2c_slave_handle_t structure which stores the transfer state.

Return values

#kStatus_Success	
kStatus_LPI2C_Idle	

9.6.4.21 LPI2C_SlaveTransferHandleIRQ()

```
void LPI2C_SlaveTransferHandleIRQ (
             LPI2C_Type * base,
             lpi2c_slave_handle_t * handle )
```

Reusable routine to handle slave interrupts.

Note

This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

base	The LPI2C peripheral base address.
handle	Pointer to #lpi2c_slave_handle_t structure which stores the transfer state.

9.7 LPI2C Master DMA Driver

9.8 LPI2C Slave DMA Driver

9.9 LPI2C FreeRTOS Driver

9.10 LPI2C µCOS/II Driver

9.11 LPI2C µCOS/III Driver

9.12 (DRV) Power Management IC Driver

Module for the PMIC driver.

Files

- file fsl_pmic.h

Data Structures

- struct pmic_version_t

  Structure for ID and Revision of PMIC.

Macros

- #define **PMIC_SET_VOLTAGE** dynamic_pmic_set_voltage
- #define **PMIC_GET_VOLTAGE** dynamic_pmic_get_voltage
- #define **PMIC_SET_MODE** dynamic_pmic_set_mode
- #define **PMIC_GET_MODE** dynamic_pmic_get_mode
- #define **PMIC_IRQ_SERVICE** dynamic_pmic_irq_service
- #define **PMIC_REGISTER_ACCESS** dynamic_pmic_register_access
- #define **GET_PMIC_VERSION** dynamic_get_pmic_version
- #define **GET_PMIC_TEMP** dynamic_get_pmic_temp
- #define **SET_PMIC_TEMP_ALARM** dynamic_set_pmic_temp_alarm
- #define **i2c_error_flags**

Typedefs

- typedef uint8_t pmic_id_t

  This type is used to declare which PMIC to address.

Functions

- sc_err_t dynamic_pmic_set_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t vol_mv, uint32_t mode_to_set)

  This function sets the voltage of a corresponding voltage regulator for the supported PMIC types.
- sc_err_t dynamic_pmic_get_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t *vol_mv, uint32_t mode_to_get)

  This function gets the voltage on a corresponding voltage regulator of the PMIC.
- sc_err_t dynamic_pmic_set_mode (pmic_id_t id, uint32_t pmic_reg, uint32_t mode)

  This function sets the mode of the specified regulator.
- sc_err_t dynamic_pmic_get_mode (pmic_id_t id, uint32_t pmic_reg, uint32_t *mode)

  This function gets the mode of the specified regulator.
- sc_bool_t dynamic_pmic_irq_service (pmic_id_t id)

  This function services the interrupt for the temp alarm.
- sc_err_t dynamic_pmic_register_access (pmic_id_t id, uint32_t address, sc_bool_t read_write, uint8_t *value)

This function allows access to individual registers of the PMIC.

- pmic_version_t dynamic_get_pmic_version (pmic_id_t id)

  This function returns the device ID and revision for the PMIC.
- uint32_t dynamic_get_pmic_temp (pmic_id_t id)

  This function gets the current PMIC temperature as sensed by the PMIC temperature sensor.
- uint32_t dynamic_set_pmic_temp_alarm (pmic_id_t id, uint32_t temp)

  This function sets the temp alarm for the PMIC in Celsius.
- status_t i2c_write_sub (uint8_t device_addr, uint8_t reg, void ∗data, uint32_t dataLength)

  This function is a simple write to an i2c register on the PMIC.
- status_t i2c_write (uint8_t device_addr, uint8_t reg, void ∗data, uint32_t dataLength)

  This function writes an i2c register on the PMIC device with clock management.
- status_t i2c_read_sub (uint8_t device_addr, uint8_t reg, void ∗data, uint32_t dataLength)

  This function is a simple read of an i2c register on the PMIC.
- status_t i2c_read (uint8_t device_addr, uint8_t reg, void ∗data, uint32_t dataLength)

  This function reads an i2c register on the PMIC device with clock management.
- status_t **i2c_j1850_write** (uint8_t device_addr, uint8_t reg, void ∗data, uint8_t dataLength)
- status_t **i2c_j1850_read** (uint8_t device_addr, uint8_t reg, void ∗data, uint8_t dataLength)
- uint8_t pmic_get_device_id (uint8_t address)

  This function reads the register at address 0x0 for the Device ID.

Variables

- uint8_t PMIC_TYPE

  Global PMIC type identifier.

Defines for supported PMIC devices

- #define **PMIC_NONE** 0U
- #define **PF100** 1U
- #define **PF8100** 2U
- #define **PF8200** 3U

Defines for PMIC configuration

- #define **PF100_DEV_ID** 0x10U
- #define **PF8100_DEV_ID** 0x40U
- #define **PF8200_DEV_ID** 0x48U
- #define **PF8X00_FAM_ID** 0x40U
- #define **PF8100_A0_REV** 0x10U
- #define **FAM_ID_MASK** 0xF0U

9.12.1 Detailed Description

Module for the PMIC driver.

It is an SDK driver for the PMIC module of i.MX devices. PMIC users shoud not call the PMIC driver functions directly. Instead, use the PMIC access macros. This allows for quick PMIC change and dynamic PMIC binding.

9.12.2 Macro Definition Documentation

9.12.2.1 i2c_error_flags

```
#define i2c_error_flags
```

Value:

```
(U32(kLPI2C_MasterArbitrationLostFlag) |        \
                                    U32(kLPI2C_MasterFifoErrFlag) |                \
                                    U32(kLPI2C_MasterPinLowTimeoutFlag) |          \
                                    U32(kLPI2C_MasterDataMatchFlag) |              \
                                    U32(kLPI2C_MasterBusyFlag) |                   \
                                    U32(kLPI2C_MasterBusBusyFlag))
```

9.12.3 Function Documentation

9.12.3.1 dynamic_pmic_set_voltage()

```
sc_err_t dynamic_pmic_set_voltage (
             pmic_id_t id,
             uint32_t pmic_reg,
             uint32_t vol_mv,
             uint32_t mode_to_set )
```

This function sets the voltage of a corresponding voltage regulator for the supported PMIC types.

Parameters

in	*id*	I2C address of PMIC device
in	*pmic_reg*	Register corresponding to regulator e.g pf8100_vregs_t
in	*vol_mv*	New voltage setpoint for the regulator in millivolts
in	*mode_to_set*	Which mode to change setpoint for. Refer to each PMIC for valid modes

Returns

Returns an error code (SC_ERR_NONE = success)

Return errors:

- SC_ERR_PARM if invalid parameters
- SC_ERR_FAIL if writing the register failed

9.12.3.2 dynamic_pmic_get_voltage()

```
sc_err_t dynamic_pmic_get_voltage (
            pmic_id_t id,
            uint32_t pmic_reg,
            uint32_t * vol_mv,
            uint32_t mode_to_get )
```

This function gets the voltage on a corresponding voltage regulator of the PMIC.

Parameters

in	*id*	I2C address of PMIC device
in	*pmic_reg*	Register corresponding to regulator e.g pf8100_vregs_t
out	*vol_mv*	pointer to return voltage in millivolts
in	*mode_to_get*	Mode for which to get the voltage. Refer to each PMIC for valid modes.

Returns

Returns an error code (SC_ERR_NONE = success)

Return errors:

- SC_ERR_PARM if invalid parameters

9.12.3.3 dynamic_pmic_set_mode()

```
sc_err_t dynamic_pmic_set_mode (
            pmic_id_t id,
            uint32_t pmic_reg,
            uint32_t mode )
```

This function sets the mode of the specified regulator.

Parameters

in	*id*	I2C address of PMIC device
in	*pmic_reg*	Register corresponding to regulator; e.g pf8100_vregs_t
in	*mode*	mode to set the regulator; Refer to each PMIC for valid modes.

Returns

Returns an error code (SC_ERR_NONE = success)

Return errors:

- SC_ERR_PARM if invalid parameters
- SC_ERR_FAIL if writing the register failed

9.12.3.4 dynamic_pmic_get_mode()

```
sc_err_t dynamic_pmic_get_mode (
            pmic_id_t id,
            uint32_t pmic_reg,
            uint32_t * mode )
```

This function gets the mode of the specified regulator.

Parameters

in	*id*	I2C address of PMIC device
in	*pmic_reg*	Register corresponding to regulator; e.g pf8100_vregs_t
in	*mode*	pointer to return mode in raw hex form

Returns

Returns an error code (SC_ERR_NONE = success)

Return errors:

- SC_ERR_PARM if invalid parameters
- SC_ERR_FAIL if writing the register failed

9.12.3.5 dynamic_pmic_irq_service()

```
sc_bool_t dynamic_pmic_irq_service (
            pmic_id_t id )
```

This function services the interrupt for the temp alarm.

Parameters

in	*id*	I2C address of PMIC device

Returns

Returns SC_TRUE if there was a temperature interrupt to be cleared

9.12.3.6 dynamic_pmic_register_access()

```
sc_err_t dynamic_pmic_register_access (
             pmic_id_t id,
             uint32_t address,
             sc_bool_t read_write,
             uint8_t * value )
```

This function allows access to individual registers of the PMIC.

Parameters

in	*id*	I2C address of PMIC device
in	*address*	register address to access
in	*read_write*	bool indicating read(SC_FALSE/0) or write(SC_TRUE/1)
in,out	*value*	value to read or to set

Returns

Returns ar error code (SC_ERR_NONE = success)

Return errors:

- SC_ERR_PARM if invalid parameters

9.12.3.7 dynamic_get_pmic_version()

```
pmic_version_t dynamic_get_pmic_version (
             pmic_id_t id )
```

This function returns the device ID and revision for the PMIC.

Parameters

in	*id*	I2C address of PMIC device

Returns

Returns a structure with the device ID and revision.

9.12.3.8 dynamic_get_pmic_temp()

```
uint32_t dynamic_get_pmic_temp (
            pmic_id_t id )
```

This function gets the current PMIC temperature as sensed by the PMIC temperature sensor.

Parameters

in	*id*	I2C address of PMIC device

Returns

returns the temp sensed by the PMIC in a UINT32 in Celsius

Note: Refer to Refer to each PMIC for temperature details

Return errors:

- SC_ERR_CONFIG if temperature monitor is not enabled

9.12.3.9 dynamic_set_pmic_temp_alarm()

```
uint32_t dynamic_set_pmic_temp_alarm (
            pmic_id_t id,
            uint32_t temp )
```

This function sets the temp alarm for the PMIC in Celsius.

Parameters

in	*id*	I2C address of PMIC device
in	*temp*	Temperature to set the alarm

Note: Refer to Refer to each PMIC for temperature details

Returns

Returns the temperature that the alarm is set to in Celsius

9.12.3.10 i2c_write_sub()

```
status_t i2c_write_sub (
            uint8_t device_addr,
            uint8_t reg,
            void * data,
            uint32_t dataLength )
```

This function is a simple write to an i2c register on the PMIC.

Parameters

in	*device_addr*	I2C address of device
in	*reg*	address of register on device
in	*data*	data to be written
in	*dataLength*	length of data to be written

Returns

Returns the status of the write (success = kStatus_Success)

Return errors

- kStatus_Fail if any of the transactions failed

Note there is no clock management in this function

9.12.3.11 i2c_write()

```
status_t i2c_write (
            uint8_t device_addr,
            uint8_t reg,
            void * data,
            uint32_t dataLength )
```

This function writes an i2c register on the PMIC device with clock management.

Parameters

in	*device_addr*	I2C address of device
in	*reg*	address of register on device
in	*data*	data to be written
in	*dataLength*	length of data to be written

Returns

Returns the status of the write (success = kStatus_Success)

Return errors

- kStatus_Fail if any of the transactions failed

9.12.3.12 i2c_read_sub()

```
status_t i2c_read_sub (
            uint8_t device_addr,
            uint8_t reg,
            void * data,
            uint32_t dataLength )
```

This function is a simple read of an i2c register on the PMIC.

Parameters

in	*device_addr*	I2C address of device
in	*reg*	address of register on device
out	*data*	data to be read
in	*dataLength*	length of data to be read

Returns

Returns the status of the read (success = kStatus_Success)

Return errors

- kStatus_Fail if any of the transactions failed

9.12.3.13 i2c_read()

```
status_t i2c_read (
            uint8_t device_addr,
            uint8_t reg,
            void * data,
            uint32_t dataLength )
```

This function reads an i2c register on the PMIC device with clock management.

Parameters

in	*device_addr*	I2C address of device
in	*reg*	address of register on device
out	*data*	data to be read
in	*dataLength*	length of data to be read

Returns

Returns the status of the read (success = kStatus_Success)

Return errors

- kStatus_Fail if any of the transactions failed

9.12.3.14 pmic_get_device_id()

```
uint8_t pmic_get_device_id (
            uint8_t address )
```

This function reads the register at address 0x0 for the Device ID.

Parameters

in	*address*	I2C address of device

Returns

Returns the device ID

Return Errors

- 0 if any error in the function

9.13 (DRV) PF100 Power Management IC Driver

Module for the PF100 PMIC driver.

Files

- file fsl_pf100.h

Typedefs

- typedef uint8_t pf100_vol_regs_t

  This type is used to indicate which register to address.
- typedef uint8_t sw_pmic_mode_t

  This type is used to indicate a switching regulator mode.
- typedef uint8_t vgen_pmic_mode_t

  This type is used to indicate a VGEN (LDO) regulator mode.
- typedef uint8_t sw_vmode_reg_t

  This type encodes which voltage mode register to set when calling pf100_pmic_set_voltage().

Functions

- pmic_version_t pf100_get_pmic_version (pmic_id_t id)

  This function returns the device ID and revision for the PF100 PMIC.
- sc_err_t pf100_pmic_set_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t vol_mv, uint32_t mode_to_set)

  This function sets the voltage of a corresponding voltage regulator for the PF100 PMIC.
- sc_err_t pf100_pmic_get_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t *vol_mv, uint32_t mode_to_get)

  This function gets the voltage on a corresponding voltage regulator for the PF100 PMIC.
- sc_err_t pf100_pmic_set_mode (pmic_id_t id, uint32_t pmic_reg, uint32_t mode)

  This function sets the mode of the specified regulator.
- uint32_t pf100_get_pmic_temp (pmic_id_t id)

  This function gets the current PMIC temperature as sensed by the PMIC temperature sensor.
- uint32_t pf100_set_pmic_temp_alarm (pmic_id_t id, uint32_t temp)

  This function sets the temp alarm for the PMIC in Celsius.
- sc_err_t **pf100_pmic_register_access** (pmic_id_t id, uint32_t address, sc_bool_t read_write, uint8_t *value)
- sc_bool_t pf100_pmic_irq_service (pmic_id_t id)

  This function services the interrupt for the temp alarm.

Defines for pf100_vol_regs_t

- #define SW1AB 0x20U

  Base register for SW1AB control.
- #define SW1C 0x2EU

  Base register for SW1C control.
- #define SW2 0x35U

  Base register for SW2 control.
- #define SW3A 0x3cU

  Base register for SW3A control.
- #define SW3B 0x43U

  Base register for SW3B control.
- #define SW4 0x4AU

  Base register for SW4 control.
- #define VGEN1 0x6CU

  Base register for VGEN1 control.
- #define VGEN2 0x6DU

  Base register for VGEN2 control.
- #define VGEN3 0x6EU

  Base register for VGEN3 control.
- #define VGEN4 0x6FU

  Base register for VGEN4 control.
- #define VGEN5 0x70U

  Base register for VGEN5 control.
- #define VGEN6 0x71U

  Base register for VGEN6 control.

Defines for sw_pmic_mode_t

- #define SW_MODE_OFF_STBY_OFF 0x0U

  Normal Mode: OFF, Standby Mode: OFF.
- #define SW_MODE_PWM_STBY_OFF 0x1U

  Normal Mode: PWM, Standby Mode: OFF.
- #define SW_MODE_PFM_STBY_OFF 0x3U

  Normal Mode: PFM, Standby Mode: OFF.
- #define SW_MODE_APS_STBY_OFF 0x4U

  Normal Mode: APS, Standby Mode: OFF.
- #define SW_MODE_PWM_STBY_PWM 0x5U

  Normal Mode: PWM, Standby Mode: PWM.
- #define SW_MODE_PWM_STBY_APS 0x6U

  Normal Mode: PWM, Standby Mode: APS.
- #define SW_MODE_APS_STBY_APS 0x8U

  Normal Mode: APS, Standby Mode: APS.
- #define SW_MODE_APS_STBY_PFM 0xCU

  Normal Mode: APS, Standby Mode: PFM.
- #define SW_MODE_PWM_STBY_PFM 0xDU

  Normal Mode: PWM, Standby Mode: PFM.

Defines for vgen_pmic_mode_t

- #define VGEN_MODE_OFF (0x0U << 4U)

  VGEN always OFF.
- #define VGEN_MODE_ON (0x1U << 4U)

  VGEN always ON.
- #define VGEN_MODE_STBY_OFF (0x3U << 4U)

  VGEN Run: ON STBY: OFF.
- #define VGEN_MODE_LP (0x5U << 4U)

  VGEN Run: LPWR STBY: LPWR.
- #define VGEN_MODE_LP2 (0x7U << 4U)

  VGEN Run: LPWR STBY: LPWR.

Defines for sw_vmode_reg_t

- #define SW_RUN_MODE 0U

  SW run mode voltage.
- #define SW_STBY_MODE 1U

  SW standby mode voltage.
- #define SW_OFF_MODE 2U

  SW off/sleep mode voltage.

9.13.1 Detailed Description

Module for the PF100 PMIC driver.

This is an SDK driver for the NXP PF100 PMIC. For more information, see the PF100 Datasheet.

9.13.2 Typedef Documentation

9.13.2.1 pf100_vol_regs_t

```
typedef uint8_t pf100_vol_regs_t
```

This type is used to indicate which register to address.

Refer to the PF100 Datasheet for the description of regsiter.

9.13.2.2 sw_pmic_mode_t

```
typedef uint8_t sw_pmic_mode_t
```

This type is used to indicate a switching regulator mode.

Refer to the PF100 Datasheet for the description of each mode.

9.13.2.3 vgen_pmic_mode_t

```
typedef uint8_t vgen_pmic_mode_t
```

This type is used to indicate a VGEN (LDO) regulator mode.

Refer to the LDO control register description in the PF100 Datasheet for possible mode combinations.

9.13.2.4 sw_vmode_reg_t

```
typedef uint8_t sw_vmode_reg_t
```

This type encodes which voltage mode register to set when calling pf100_pmic_set_voltage().

Possible modes are Run, Standby and Off/Sleep.

9.13.3 Function Documentation

9.13.3.1 pf100_get_pmic_version()

```
pmic_version_t pf100_get_pmic_version (
            pmic_id_t id )
```

This function returns the device ID and revision for the PF100 PMIC.

Parameters

in	*id*	I2C address of PMIC device

Returns

Returns a structure with the device ID and revision.

9.13.3.2 pf100_pmic_set_voltage()

```
sc_err_t pf100_pmic_set_voltage (
            pmic_id_t id,
            uint32_t pmic_reg,
            uint32_t vol_mv,
            uint32_t mode_to_set )
```

This function sets the voltage of a corresponding voltage regulator for the PF100 PMIC.

Parameters

in	*id*	I2C address of PMIC device
in	*pmic_reg*	Register corresponding to regulator; see pf100_vol_regs_t
in	*vol_mv*	New voltage setpoint for the regulator in millivolts
in	*mode_to_set*	Mode to set the voltage for run, standby and off; only applicable to switching regulators, ignored otherwise; see sw_vmode_reg_t

Returns

Returns an error code (SC_ERR_NONE = success)

Return errors:

- SC_ERR_PARM if invalid parameters
- SC_ERR_FAIL if writing the register failed

9.13.3.3 pf100_pmic_get_voltage()

```
sc_err_t pf100_pmic_get_voltage (
            pmic_id_t id,
            uint32_t pmic_reg,
            uint32_t * vol_mv,
            uint32_t mode_to_get )
```

This function gets the voltage on a corresponding voltage regulator for the PF100 PMIC.

Parameters

in	*id*	I2C address of PMIC device
in	*pmic_reg*	Register corresponding to regulator; see pf8100_vregs_t
out	*vol_mv*	pointer to return voltage in millivolts
in	*mode_to_get*	Mode to get the voltage for run, standby and off; only applicable to switching regulators, ignored otherwise; see sw_vmode_reg_t

Returns

Returns an error code (SC_ERR_NONE = success)

Return errors:

- SC_ERR_PARM if invalid parameters

9.13.3.4 pf100_pmic_set_mode()

```
sc_err_t pf100_pmic_set_mode (
             pmic_id_t id,
             uint32_t pmic_reg,
             uint32_t mode )
```

This function sets the mode of the specified regulator.

Parameters

in	*id*	I2C address of PMIC device
in	*pmic_reg*	Register corresponding to regulator; see pf100_vol_regs_t
in	*mode*	mode to set the regulator; see vgen_pmic_mode_t and sw_pmic_mode_t

Returns

Returns an error code (SC_ERR_NONE = success)

Return errors:

- SC_ERR_PARM if invalid parameters
- SC_ERR_FAIL if writing the register failed

9.13.3.5 pf100_get_pmic_temp()

```
uint32_t pf100_get_pmic_temp (
             pmic_id_t id )
```

This function gets the current PMIC temperature as sensed by the PMIC temperature sensor.

Parameters

in	*id*	I2C address of PMIC device

Returns

returns the temp sensed by the PMIC in a UINT32 in Celsius

Return errors:

- SC_ERR_CONFIG if temperature monitor is not enabled

Note PMIC PF100 temp is returned as the highest temp sensor enabled.

9.13.3.6 pf100_set_pmic_temp_alarm()

```
uint32_t pf100_set_pmic_temp_alarm (
            pmic_id_t id,
            uint32_t temp )
```

This function sets the temp alarm for the PMIC in Celsius.

Parameters

in	*id*	I2C address of PMIC device
in	*temp*	Temperature to set the alarm

Note the granularity for PF100 PMIC only allows the following values: 110 120 125 130 135

Returns

Returns the temperature that the alarm is set to in Celsius

9.13.3.7 pf100_pmic_irq_service()

```
sc_bool_t pf100_pmic_irq_service (
            pmic_id_t id )
```

This function services the interrupt for the temp alarm.

Parameters

in	*id*	I2C address of PMIC device

Returns

Returns SC_TRUE if there was a temperature interrupt to be cleared

9.14 (DRV) PF8100 Power Management IC Driver

Module for the PF8100 PMIC driver.

Files

- file fsl_pf8100.h

Macros

- #define **I2C_WRITE** i2c_write
- #define **I2C_READ** i2c_read

Typedefs

- typedef uint8_t pf8100_vregs_t

  This type is used to indicate which register to address.
- typedef uint8_t sw_mode_t

  This type is used to indicate a switching regulator mode.
- typedef uint8_t ldo_mode_t

  This type is used to indicate an LDO regulator mode.
- typedef uint8_t vmode_reg_t

  This type is used to indicate a Switching regulator voltage setpoint.

Functions

- pmic_version_t pf8100_get_pmic_version (pmic_id_t id)

  This function returns the device ID and revision for the PF8100 PMIC.
- sc_err_t pf8100_pmic_set_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t vol_mv, uint32_t mode_to_set)

  This function sets the voltage of a corresponding voltage regulator for the PF8100 PMIC.
- sc_err_t pf8100_pmic_get_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t *vol_mv, uint32_t mode_to_get)

  This function gets the voltage on a corresponding voltage regulator for the PF8100 PMIC.
- sc_err_t pf8100_pmic_set_mode (pmic_id_t id, uint32_t pmic_reg, uint32_t mode)

  This function sets the mode of the specified regulator.
- sc_err_t pf8100_pmic_get_mode (pmic_id_t id, uint32_t pmic_reg, uint32_t *mode)

  This function gets the mode of the specified regulator.
- uint32_t pf8100_get_pmic_temp (pmic_id_t id)

  This function gets the current PMIC temperature as sensed by the PMIC temperature sensor.
- uint32_t pf8100_set_pmic_temp_alarm (pmic_id_t id, uint32_t temp)

  This function sets the temp alarm for the PMIC in Celsius.
- sc_err_t **pf8100_pmic_register_access** (pmic_id_t id, uint32_t address, sc_bool_t read_write, uint8_t *value)
- sc_bool_t pf8100_pmic_irq_service (pmic_id_t id)

  This function services the interrupt for the temp alarm.

Defines for pf8100_vregs_t

- #define PF8100_SW1 0x4DU

  Base register for SW1 regulator control.
- #define PF8100_SW2 0x55U

  Base register for SW2 regulator control.
- #define PF8100_SW3 0x5DU

  Base register for SW3 regulator control.
- #define PF8100_SW4 0x65U

  Base register for SW4 regulator control.
- #define PF8100_SW5 0x6DU

  Base register for SW5 regulator control.
- #define PF8100_SW6 0x75U

  Base register for SW6 regulator control.
- #define PF8100_SW7 0x7DU

  Base register for SW7 regulator control.
- #define PF8100_LDO1 0x85U

  Base register for LDO1 regulator control.
- #define PF8100_LDO2 0x8BU

  Base register for LDO2 regulator control.
- #define PF8100_LDO3 0x91U

  Base register for LDO3 regulator control.
- #define PF8100_LDO4 0x97U

  Base register for LDO4 regulator control.

Defines for sw_mode_t

- #define SW_RUN_OFF 0x0U

  Run mode: OFF.
- #define SW_RUN_PWM 0x1U

  Run mode: PWM.
- #define SW_RUN_PFM 0x2U

  Run mode: PFM.
- #define SW_RUN_ASM 0x3U

  Run mode: ASM.
- #define SW_STBY_OFF (0x0U << 2U)

  Standby mode: OFF.
- #define SW_STBY_PWM (0x1U << 2U)

  Standby mode: PWM.
- #define SW_STBY_PFM (0x2U << 2U)

  Standby mode: PFM.
- #define SW_STBY_ASM (0x3U << 2U)

  Standby mode: ASM.

Defines for ldo_mode_t

- #define RUN_OFF_STBY_OFF 0x0U

  Run mode: OFF, Standby mode: OFF.
- #define RUN_OFF_STBY_EN 0x1U

  Run mode: OFF, Standby mode: ON.
- #define RUN_EN_STBY_OFF 0x2U

  Run mode: ON, Standby mode: OFF.
- #define RUN_EN_STBY_EN 0x3U

  Run mode: ON, Standby mode: ON.

Defines for vmode_reg_t

- #define **REG_STBY_MODE** 0U
- #define **REG_RUN_MODE** 1U

9.14.1 Detailed Description

Module for the PF8100 PMIC driver.

This is an SDK driver for the NXP PF8100 PMIC. For more information, see the PF8100 Datasheet.

9.14.2 Typedef Documentation

9.14.2.1 pf8100_vregs_t

```
typedef uint8_t pf8100_vregs_t
```

This type is used to indicate which register to address.

Refer to the PF8100 Datasheet for the description of regsiter.

9.14.2.2 sw_mode_t

```
typedef uint8_t sw_mode_t
```

This type is used to indicate a switching regulator mode.

Refer to the PF8100 Datasheet for the description of each mode.

These modes are used in combination to designate a run and standby mode i.e. (SW_RUN_PWM | SW_STBY_OFF).

9.14.2.3 ldo_mode_t

```
typedef uint8_t ldo_mode_t
```

This type is used to indicate an LDO regulator mode.

Refer to the PF8100 Datasheet for the description of each mode.

9.14.2.4 vmode_reg_t

```
typedef uint8_t vmode_reg_t
```

This type is used to indicate a Switching regulator voltage setpoint.

Refer to the PF8100 Datasheet for the description of each mode.

9.14.3 Function Documentation

9.14.3.1 pf8100_get_pmic_version()

```
pmic_version_t pf8100_get_pmic_version (
            pmic_id_t id )
```

This function returns the device ID and revision for the PF8100 PMIC.

Parameters

in	*id*	I2C address of PMIC device

Returns

Returns a structure with the device ID and revision.

9.14.3.2 pf8100_pmic_set_voltage()

```
sc_err_t pf8100_pmic_set_voltage (
            pmic_id_t id,
            uint32_t pmic_reg,
            uint32_t vol_mv,
            uint32_t mode_to_set )
```

This function sets the voltage of a corresponding voltage regulator for the PF8100 PMIC.

Parameters

in	*id*	I2C address of PMIC device
in	*pmic_reg*	Register corresponding to regulator; see pf8100_vregs_t
in	*vol_mv*	New voltage setpoint for the regulator in millivolts
in	*mode_to_set*	Mode to set the voltage for run (RUN or STANDBY)

Returns

Returns an error code (SC_ERR_NONE = success)

Note *mode_to_set* is SC_TRUE for RUN and SC_FALSE for STANDBY.

Return errors:

- SC_ERR_PARM if invalid parameters
- SC_ERR_FAIL if writing the register failed

9.14.3.3 pf8100_pmic_get_voltage()

```
sc_err_t pf8100_pmic_get_voltage (
            pmic_id_t id,
            uint32_t pmic_reg,
            uint32_t * vol_mv,
            uint32_t mode_to_get )
```

This function gets the voltage on a corresponding voltage regulator for the PF8100 PMIC.

Parameters

in	*id*	I2C address of PMIC device
in	*pmic_reg*	Register corresponding to regulator; see pf8100_vregs_t
out	*vol_mv*	pointer to return voltage in millivolts
in	*mode_to_get*	Mode to get the voltage for (RUN or STANDBY)

Returns

Returns an error code (SC_ERR_NONE = success)

Note *mode_to_get* is SC_TRUE for RUN and SC_FALSE for STANDBY.

Return errors:

- SC_ERR_PARM if invalid parameters

9.14.3.4 pf8100_pmic_set_mode()

```
sc_err_t pf8100_pmic_set_mode (
            pmic_id_t id,
            uint32_t pmic_reg,
            uint32_t mode )
```

This function sets the mode of the specified regulator.

Parameters

in	*id*	I2C address of PMIC device
in	*pmic_reg*	Register corresponding to regulator; see pf8100_vregs_t
in	*mode*	mode to set the regulator; see sw_mode_t and ldo_mode_t

Note SW modes are used in combination to designate a run and standby mode i.e. (SW_RUN_PWM | SW_STBY_OFF).

Returns

Returns an error code (SC_ERR_NONE = success)

Return errors:

- SC_ERR_PARM if invalid parameters
- SC_ERR_FAIL if writing the register failed

9.14.3.5 pf8100_pmic_get_mode()

```
sc_err_t pf8100_pmic_get_mode (
            pmic_id_t id,
            uint32_t pmic_reg,
            uint32_t * mode )
```

This function gets the mode of the specified regulator.

Parameters

in	*id*	I2C address of PMIC device
in	*pmic_reg*	Register corresponding to regulator; see pf8100_vregs_t
out	*mode*	pointer to return mode in raw hex form

Note SW modes are used in combination to designate a run and standby mode i.e. (SW_RUN_PWM | SW_STBY_OFF).

Returns

Returns an error code (SC_ERR_NONE = success)

Return errors:

- SC_ERR_PARM if invalid parameters
- SC_ERR_FAIL if writing the register failed

9.14.3.6 pf8100_get_pmic_temp()

```
uint32_t pf8100_get_pmic_temp (
            pmic_id_t id )
```

This function gets the current PMIC temperature as sensed by the PMIC temperature sensor.

Parameters

in	*id*	I2C address of PMIC device

Returns

returns the temp sensed by the PMIC in a UINT32 in Celsius

Return errors:

- SC_ERR_CONFIG if temperature monitor is not enabled

Note PMIC PF100 temp is returned as the highest temp sensor enabled.

9.14.3.7 pf8100_set_pmic_temp_alarm()

```
uint32_t pf8100_set_pmic_temp_alarm (
            pmic_id_t id,
            uint32_t temp )
```

This function sets the temp alarm for the PMIC in Celsius.

Parameters

in	*id*	I2C address of PMIC device
in	*temp*	Temperature to set the alarm

Note the granularity for PF100 PMIC only allows the following values: 80 95 110 125 140 155

Returns

Returns the temperature that the alarm is set to in Celsius

9.14.3.8 pf8100_pmic_irq_service()

```
sc_bool_t pf8100_pmic_irq_service (
            pmic_id_t id )
```

This function services the interrupt for the temp alarm.

Parameters

in	*id*	I2C address of PMIC device

Returns

Returns SC_TRUE if the temperature interrupt was cleared

9.15 (SVC) Pad Service

Module for the Pad Control (PAD) service.

Typedefs

- typedef uint8_t sc_pad_config_t

  This type is used to declare a pad config.
- typedef uint8_t sc_pad_iso_t

  This type is used to declare a pad low-power isolation config.
- typedef uint8_t sc_pad_28fdsoi_dse_t

  This type is used to declare a drive strength.
- typedef uint8_t sc_pad_28fdsoi_ps_t

  This type is used to declare a pull select.
- typedef uint8_t sc_pad_28fdsoi_pus_t

  This type is used to declare a pull-up select.
- typedef uint8_t sc_pad_wakeup_t

  This type is used to declare a wakeup mode of a pad.

Defines for type widths

- #define SC_PAD_MUX_W 3U

  Width of mux parameter.

Defines for sc_pad_config_t

- #define SC_PAD_CONFIG_NORMAL 0U

  Normal.
- #define SC_PAD_CONFIG_OD 1U

  Open Drain.
- #define SC_PAD_CONFIG_OD_IN 2U

  Open Drain and input.
- #define SC_PAD_CONFIG_OUT_IN 3U

  Output and input.

Defines for sc_pad_iso_t

- #define SC_PAD_ISO_OFF 0U

  ISO latch is transparent.
- #define SC_PAD_ISO_EARLY 1U

  Follow EARLY_ISO.
- #define SC_PAD_ISO_LATE 2U

  Follow LATE_ISO.
- #define SC_PAD_ISO_ON 3U

  ISO latched data is held.

Defines for sc_pad_28fdsoi_dse_t

- #define SC_PAD_28FDSOI_DSE_18V_1MA 0U

  Drive strength of 1mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_2MA 1U

  Drive strength of 2mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_4MA 2U

  Drive strength of 4mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_6MA 3U

  Drive strength of 6mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_8MA 4U

  Drive strength of 8mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_10MA 5U

  Drive strength of 10mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_12MA 6U

  Drive strength of 12mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_HS 7U

  High-speed drive strength for 1.8v.
- #define SC_PAD_28FDSOI_DSE_33V_2MA 0U

  Drive strength of 2mA for 3.3v.
- #define SC_PAD_28FDSOI_DSE_33V_4MA 1U

  Drive strength of 4mA for 3.3v.
- #define SC_PAD_28FDSOI_DSE_33V_8MA 2U

  Drive strength of 8mA for 3.3v.
- #define SC_PAD_28FDSOI_DSE_33V_12MA 3U

  Drive strength of 12mA for 3.3v.
- #define SC_PAD_28FDSOI_DSE_DV_HIGH 0U

  High drive strength for dual volt.
- #define SC_PAD_28FDSOI_DSE_DV_LOW 1U

  Low drive strength for dual volt.

Defines for sc_pad_28fdsoi_ps_t

- #define SC_PAD_28FDSOI_PS_KEEPER 0U

  Bus-keeper (only valid for 1.8v)
- #define SC_PAD_28FDSOI_PS_PU 1U

  Pull-up.
- #define SC_PAD_28FDSOI_PS_PD 2U

  Pull-down.
- #define SC_PAD_28FDSOI_PS_NONE 3U

  No pull (disabled)

Defines for sc_pad_28fdsoi_pus_t

- #define SC_PAD_28FDSOI_PUS_30K_PD 0U

  30K pull-down
- #define SC_PAD_28FDSOI_PUS_100K_PU 1U

  100K pull-up
- #define SC_PAD_28FDSOI_PUS_3K_PU 2U

  3K pull-up
- #define SC_PAD_28FDSOI_PUS_30K_PU 3U

  30K pull-up

Defines for sc_pad_wakeup_t

- #define SC_PAD_WAKEUP_OFF 0U

  Off.
- #define SC_PAD_WAKEUP_CLEAR 1U

  Clears pending flag.
- #define SC_PAD_WAKEUP_LOW_LVL 4U

  Low level.
- #define SC_PAD_WAKEUP_FALL_EDGE 5U

  Falling edge.
- #define SC_PAD_WAKEUP_RISE_EDGE 6U

  Rising edge.
- #define SC_PAD_WAKEUP_HIGH_LVL 7U

  High-level.

Generic Functions

- sc_err_t sc_pad_set_mux (sc_ipc_t ipc, sc_pad_t pad, uint8_t mux, sc_pad_config_t config, sc_pad_iso_t iso)

  This function configures the mux settings for a pad.
- sc_err_t sc_pad_get_mux (sc_ipc_t ipc, sc_pad_t pad, uint8_t *mux, sc_pad_config_t *config, sc_pad_iso_t *iso)

  This function gets the mux settings for a pad.
- sc_err_t sc_pad_set_gp (sc_ipc_t ipc, sc_pad_t pad, uint32_t ctrl)

  This function configures the general purpose pad control.
- sc_err_t sc_pad_get_gp (sc_ipc_t ipc, sc_pad_t pad, uint32_t *ctrl)

  This function gets the general purpose pad control.
- sc_err_t sc_pad_set_wakeup (sc_ipc_t ipc, sc_pad_t pad, sc_pad_wakeup_t wakeup)

  This function configures the wakeup mode of the pad.
- sc_err_t sc_pad_get_wakeup (sc_ipc_t ipc, sc_pad_t pad, sc_pad_wakeup_t *wakeup)

  This function gets the wakeup mode of a pad.
- sc_err_t sc_pad_set_all (sc_ipc_t ipc, sc_pad_t pad, uint8_t mux, sc_pad_config_t config, sc_pad_iso_t iso, uint32_t ctrl, sc_pad_wakeup_t wakeup)

  This function configures a pad.
- sc_err_t sc_pad_get_all (sc_ipc_t ipc, sc_pad_t pad, uint8_t *mux, sc_pad_config_t *config, sc_pad_iso_t *iso, uint32_t *ctrl, sc_pad_wakeup_t *wakeup)

  This function gets a pad's config.

SoC Specific Functions

- sc_err_t sc_pad_set (sc_ipc_t ipc, sc_pad_t pad, uint32_t val)

  This function configures the settings for a pad.
- sc_err_t sc_pad_get (sc_ipc_t ipc, sc_pad_t pad, uint32_t *val)

  This function gets the settings for a pad.

Technology Specific Functions

- sc_err_t sc_pad_set_gp_28fdsoi (sc_ipc_t ipc, sc_pad_t pad, sc_pad_28fdsoi_dse_t dse, sc_pad_28fdsoi_ps_t ps)

  This function configures the pad control specific to 28FDSOI.
- sc_err_t sc_pad_get_gp_28fdsoi (sc_ipc_t ipc, sc_pad_t pad, sc_pad_28fdsoi_dse_t *dse, sc_pad_28fdsoi_ps_t *ps)

  This function gets the pad control specific to 28FDSOI.
- sc_err_t sc_pad_set_gp_28fdsoi_hsic (sc_ipc_t ipc, sc_pad_t pad, sc_pad_28fdsoi_dse_t dse, sc_bool_t hys, sc_pad_28fdsoi_pus_t pus, sc_bool_t pke, sc_bool_t pue)

  This function configures the pad control specific to 28FDSOI.
- sc_err_t sc_pad_get_gp_28fdsoi_hsic (sc_ipc_t ipc, sc_pad_t pad, sc_pad_28fdsoi_dse_t *dse, sc_bool_t *hys, sc_pad_28fdsoi_pus_t *pus, sc_bool_t *pke, sc_bool_t *pue)

  This function gets the pad control specific to 28FDSOI.
- sc_err_t sc_pad_set_gp_28fdsoi_comp (sc_ipc_t ipc, sc_pad_t pad, uint8_t compen, sc_bool_t fastfrz, uint8_t rasrcp, uint8_t rasrcn, sc_bool_t nasrc_sel, sc_bool_t psw_ovr)

  This function configures the compensation control specific to 28FDSOI.
- sc_err_t sc_pad_get_gp_28fdsoi_comp (sc_ipc_t ipc, sc_pad_t pad, uint8_t *compen, sc_bool_t *fastfrz, uint8_t *rasrcp, uint8_t *rasrcn, sc_bool_t *nasrc_sel, sc_bool_t *compok, uint8_t *nasrc, sc_bool_t *psw_ovr)

  This function gets the compensation control specific to 28FDSOI.

9.15.1 Detailed Description

Module for the Pad Control (PAD) service.

Pad configuration is managed by SC firmware. The pad configuration features supported by the SC firmware include:

- Configuring the mux, input/output connection, and low-power isolation mode.
- Configuring the technology-specific pad setting such as drive strength, pullup/pulldown, etc.
- Configuring compensation for pad groups with dual voltage capability.

Pad functions fall into one of three categories. Generic functions are common to all SoCs and all process technologies. SoC functions are raw low-level functions. Technology-specific functions are specific to the process technology.

The list of pads is SoC specific. Refer to the SoC Pad List for valid pad values. Note that all pads exist on a die but may or may not be brought out by the specific package. Mapping of pads to package pins/balls is documented in the associated Data Sheet. Some pads may not be brought out because the part (die+package) is defeatured and some pads may connect to the substrate in the package.

Some pads (SC_P_COMP_∗) that can be specified are not individual pads but are in fact pad groups. These groups have additional configuration that can be done using the sc_pad_set_gp_28fdsoi_comp() function. More info on these can be found in the associated Reference Manual.

Pads are managed as a resource by the Resource Manager (RM). They have assigned owners and only the owners can configure the pads. Some of the pads are reserved for use by the SCFW itself and this can be overriden with the implementation of board_config_sc(). Additionally, pads may be assigned to various other partitions via the implementation of board_system_config().

Note muxing two input pads to the same IP functional signal will result in undefined behavior.

The following SCFW pad code is an example of how to configure pads. In this example, two pads are configured for use by the i.MX8QXP I2C_0 (ADMA.I2C0). Another dual-voltge pad is configured as SPI_0 SCK (ADMA.SPI0.SCK).

The ipc parameter most functions take is a handle to the IPC channel opened to communicate to the SC. It is implementation defined. Most API ports include an sc_ipc_open() and sc_ipc_close() function to manage this. The sc_ipc_open() takes an argument to identify the communication channel (usually the MU address) and returns the IPC handle that all API calls should then use.

```
1 /* Configure I2C_0 SCL pad */
2 sc_pad_set_mux(ipc, SC_P_MIPI_CSI0_GPIO0_00, 1,       SC_PAD_CONFIG_OD_IN, SC_PAD_ISO_OFF);
3 sc_pad_set_gp_28fdsoi(ipc, SC_P_MIPI_CSI0_GPIO0_00,       SC_PAD_28FDSOI_DSE_18V_1MA, SC_PAD_28FDSOI_PS_PU);
4
5 /* Configure I2C_0 SDA pad */
6 sc_pad_set_mux(ipc, SC_P_MIPI_CSI0_GPIO0_01, 1,       SC_PAD_CONFIG_OD_IN, SC_PAD_ISO_OFF);
7 sc_pad_set_gp_28fdsoi(ipc, SC_P_MIPI_CSI0_GPIO0_01,       SC_PAD_28FDSOI_DSE_18V_1MA, SC_PAD_28FDSOI_PS_PU);
8
9 /* Configure SPI0 SCK pad (dual-voltage) */
10 sc_pad_set_mux(ipc, SC_P_SPI0_SCK, 0, SC_PAD_CONFIG_NORMAL,       SC_PAD_ISO_OFF);
11 sc_pad_set_gp_28fdsoi(ipc, SC_P_SPI0_SCK,       SC_PAD_28FDSOI_DSE_DV_LOW, SC_PAD_28FDSOI_PS_NONE);
```

The first pair of pads in question are MIPI_CSI0_GPIO0_00 (used for SCL) and MIPI_CSI0_GPIO0_01 (used for SDA). I2C_0 is mux select 1 for both pads.

The first two lines configure the SCL pad. The first configures the SCL pad for mux select 1, and as open-drain with with input. The second configures the drive strength and enables the pull-up.

The last two lines do the same for the SDA pad.

For 28FDSIO single voltage pads, SC_PAD_28FDSOI_DSE_DV_HIGH and SC_PAD_28FDSOI_DSE_DV_LOW are not valid drive strenths.

```
/* Configure I2C_0 SCL pad */
sc_pad_set_mux(ipc, SC_P_MIPI_CSI0_GPIO0_00, 1,       SC_PAD_CONFIG_OD_IN, SC_PAD_ISO_OFF);
sc_pad_set_gp_28fdsoi(ipc, SC_P_MIPI_CSI0_GPIO0_00,       SC_PAD_28FDSOI_DSE_18V_1MA, SC_PAD_28FDSOI_PS_PU);
/* Configure I2C_0 SDA pad */
sc_pad_set_mux(ipc, SC_P_MIPI_CSI0_GPIO0_01, 1,       SC_PAD_CONFIG_OD_IN, SC_PAD_ISO_OFF);
sc_pad_set_gp_28fdsoi(ipc, SC_P_MIPI_CSI0_GPIO0_01,       SC_PAD_28FDSOI_DSE_18V_1MA, SC_PAD_28FDSOI_PS_PU);
```

The next pad configured is SPI0_SCK. It is configured as mux select 0. the first line configures the mux select as 0 and normal push-pull. The second line configures the drive strength and no pull-up. Note the drive strength setting is different for this dual voltage pad.

```
/* Configure SPI0 SCK pad (dual-voltage) */
sc_pad_set_mux(ipc, SC_P_SPI0_SCK, 0, SC_PAD_CONFIG_NORMAL,       SC_PAD_ISO_OFF);
sc_pad_set_gp_28fdsoi(ipc, SC_P_SPI0_SCK,       SC_PAD_28FDSOI_DSE_DV_LOW, SC_PAD_28FDSOI_PS_NONE);
```

For 28FDSIO dual voltage pads, only SC_PAD_28FDSOI_DSE_DV_HIGH and SC_PAD_28FDSOI_DSE_DV_LOW are valid drive strenths.

The voltage of the pad is determined by the supply for the pad group (the VDD_SPI_SAI_1P8_3P3 pad in this case).

9.15.2 Typedef Documentation

9.15.2.1 sc_pad_config_t

```
typedef uint8_t sc_pad_config_t
```

This type is used to declare a pad config.

It determines how the output data is driven, pull-up is controlled, and input signal is connected. Normal and OD are typical and only connect the input when the output is not driven. The IN options are less common and force an input connection even when driving the output.

9.15.2.2 sc_pad_iso_t

```
typedef uint8_t sc_pad_iso_t
```

This type is used to declare a pad low-power isolation config.

ISO_LATE is the most common setting. ISO_EARLY is only used when an output pad is directly determined by another input pad. The other two are only used when SW wants to directly contol isolation.

9.15.2.3 sc_pad_28fdsoi_dse_t

```
typedef uint8_t sc_pad_28fdsoi_dse_t
```

This type is used to declare a drive strength.

Note it is specific to 28FDSOI. Also note that valid values depend on the pad type.

9.15.2.4 sc_pad_28fdsoi_ps_t

```
typedef uint8_t sc_pad_28fdsoi_ps_t
```

This type is used to declare a pull select.

Note it is specific to 28FDSOI.

9.15.2.5 sc_pad_28fdsoi_pus_t

```
typedef uint8_t sc_pad_28fdsoi_pus_t
```

This type is used to declare a pull-up select.

Note it is specific to 28FDSOI HSIC pads.

9.15.3 Function Documentation

9.15.3.1 sc_pad_set_mux()

```
sc_err_t sc_pad_set_mux (
             sc_ipc_t ipc,
             sc_pad_t pad,
             uint8_t mux,
             sc_pad_config_t config,
             sc_pad_iso_t iso )
```

This function configures the mux settings for a pad.

This includes the signal mux, pad config, and low-power isolation mode.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to configure
in	*mux*	mux setting
in	*config*	pad config
in	*iso*	low-power isolation mode

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner

Note muxing two input pads to the same IP functional signal will result in undefined behavior.

Refer to the SoC Pad List for valid pad values.

9.15.3.2 sc_pad_get_mux()

```
sc_err_t sc_pad_get_mux (
             sc_ipc_t ipc,
             sc_pad_t pad,
             uint8_t * mux,
             sc_pad_config_t * config,
             sc_pad_iso_t * iso )
```

This function gets the mux settings for a pad.

This includes the signal mux, pad config, and low-power isolation mode.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to query
out	*mux*	pointer to return mux setting
out	*config*	pointer to return pad config
out	*iso*	pointer to return low-power isolation mode

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner

Refer to the SoC Pad List for valid pad values.

9.15.3.3 sc_pad_set_gp()

```
sc_err_t sc_pad_set_gp (
             sc_ipc_t ipc,
             sc_pad_t pad,
             uint32_t ctrl )
```

This function configures the general purpose pad control.

This is technology dependent and includes things like drive strength, slew rate, pull up/down, etc. Refer to the SoC Reference Manual for bit field details.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to configure
in	*ctrl*	control value to set

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner

Refer to the SoC Pad List for valid pad values.

9.15.3.4 sc_pad_get_gp()

```
sc_err_t sc_pad_get_gp (
             sc_ipc_t ipc,
             sc_pad_t pad,
             uint32_t * ctrl )
```

This function gets the general purpose pad control.

This is technology dependent and includes things like drive strength, slew rate, pull up/down, etc. Refer to the SoC Reference Manual for bit field details.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to query
out	*ctrl*	pointer to return control value

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner

Refer to the SoC Pad List for valid pad values.

9.15.3.5 sc_pad_set_wakeup()

```
sc_err_t sc_pad_set_wakeup (
             sc_ipc_t ipc,
             sc_pad_t pad,
             sc_pad_wakeup_t wakeup )
```

This function configures the wakeup mode of the pad.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to configure
in	*wakeup*	wakeup to set

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner

Refer to the SoC Pad List for valid pad values.

9.15.3.6 sc_pad_get_wakeup()

```
sc_err_t sc_pad_get_wakeup (
             sc_ipc_t ipc,
             sc_pad_t pad,
             sc_pad_wakeup_t * wakeup )
```

This function gets the wakeup mode of a pad.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to query
out	*wakeup*	pointer to return wakeup

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner

Refer to the SoC Pad List for valid pad values.

9.15.3.7 sc_pad_set_all()

```
sc_err_t sc_pad_set_all (
             sc_ipc_t ipc,
             sc_pad_t pad,
             uint8_t mux,
             sc_pad_config_t config,
             sc_pad_iso_t iso,
             uint32_t ctrl,
             sc_pad_wakeup_t wakeup )
```

This function configures a pad.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to configure
in	*mux*	mux setting
in	*config*	pad config
in	*iso*	low-power isolation mode
in	*ctrl*	control value
in	*wakeup*	wakeup to set

See also

sc_pad_set_mux().
sc_pad_set_gp().

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner

Returns

Returns an error code (SC_ERR_NONE = success).

Note muxing two input pads to the same IP functional signal will result in undefined behavior.

Refer to the SoC Pad List for valid pad values.

9.15.3.8 sc_pad_get_all()

```
sc_err_t sc_pad_get_all (
             sc_ipc_t ipc,
             sc_pad_t pad,
             uint8_t * mux,
             sc_pad_config_t * config,
             sc_pad_iso_t * iso,
             uint32_t * ctrl,
             sc_pad_wakeup_t * wakeup )
```

This function gets a pad's config.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to query
out	*mux*	pointer to return mux setting
out	*config*	pointer to return pad config
out	*iso*	pointer to return low-power isolation mode
out	*ctrl*	pointer to return control value
out	*wakeup*	pointer to return wakeup to set

See also

sc_pad_set_mux().
sc_pad_set_gp().

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner

Returns

Returns an error code (SC_ERR_NONE = success).

Refer to the SoC Pad List for valid pad values.

9.15.3.9 sc_pad_set()

```
sc_err_t sc_pad_set (
             sc_ipc_t ipc,
             sc_pad_t pad,
             uint32_t val )
```

This function configures the settings for a pad.

This setting is SoC specific.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to configure
in	*val*	value to set

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner

Refer to the SoC Pad List for valid pad values.

9.15.3.10 sc_pad_get()

```
sc_err_t sc_pad_get (
             sc_ipc_t ipc,
             sc_pad_t pad,
             uint32_t * val )
```

This function gets the settings for a pad.

This setting is SoC specific.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to query
out	*val*	pointer to return setting

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner

Refer to the SoC Pad List for valid pad values.

9.15.3.11 sc_pad_set_gp_28fdsoi()

```
sc_err_t sc_pad_set_gp_28fdsoi (
             sc_ipc_t ipc,
             sc_pad_t pad,
             sc_pad_28fdsoi_dse_t dse,
             sc_pad_28fdsoi_ps_t ps )
```

This function configures the pad control specific to 28FDSOI.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to configure
in	*dse*	drive strength
in	*ps*	pull select

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner,
- SC_ERR_UNAVAILABLE if process not applicable

Refer to the SoC Pad List for valid pad values.

9.15.3.12 sc_pad_get_gp_28fdsoi()

```
sc_err_t sc_pad_get_gp_28fdsoi (
            sc_ipc_t ipc,
            sc_pad_t pad,
            sc_pad_28fdsoi_dse_t * dse,
            sc_pad_28fdsoi_ps_t * ps )
```

This function gets the pad control specific to 28FDSOI.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to query
out	*dse*	pointer to return drive strength
out	*ps*	pointer to return pull select

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner,
- SC_ERR_UNAVAILABLE if process not applicable

Refer to the SoC Pad List for valid pad values.

9.15.3.13 sc_pad_set_gp_28fdsoi_hsic()

```
sc_err_t sc_pad_set_gp_28fdsoi_hsic (
             sc_ipc_t ipc,
             sc_pad_t pad,
             sc_pad_28fdsoi_dse_t dse,
             sc_bool_t hys,
             sc_pad_28fdsoi_pus_t pus,
             sc_bool_t pke,
             sc_bool_t pue )
```

This function configures the pad control specific to 28FDSOI.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to configure
in	*dse*	drive strength
in	*hys*	hysteresis
in	*pus*	pull-up select
in	*pke*	pull keeper enable
in	*pue*	pull-up enable

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner,
- SC_ERR_UNAVAILABLE if process not applicable

Refer to the SoC Pad List for valid pad values.

9.15.3.14 sc_pad_get_gp_28fdsoi_hsic()

```
sc_err_t sc_pad_get_gp_28fdsoi_hsic (
             sc_ipc_t ipc,
             sc_pad_t pad,
             sc_pad_28fdsoi_dse_t * dse,
             sc_bool_t * hys,
             sc_pad_28fdsoi_pus_t * pus,
             sc_bool_t * pke,
             sc_bool_t * pue )
```

This function gets the pad control specific to 28FDSOI.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to query
out	*dse*	pointer to return drive strength
out	*hys*	pointer to return hysteresis
out	*pus*	pointer to return pull-up select
out	*pke*	pointer to return pull keeper enable
out	*pue*	pointer to return pull-up enable

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner,
- SC_ERR_UNAVAILABLE if process not applicable

Refer to the SoC Pad List for valid pad values.

9.15.3.15 sc_pad_set_gp_28fdsoi_comp()

```
sc_err_t sc_pad_set_gp_28fdsoi_comp (
             sc_ipc_t ipc,
             sc_pad_t pad,
             uint8_t compen,
             sc_bool_t fastfrz,
             uint8_t rasrcp,
             uint8_t rasrcn,
             sc_bool_t nasrc_sel,
             sc_bool_t psw_ovr )
```

This function configures the compensation control specific to 28FDSOI.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to configure
in	*compen*	compensation/freeze mode
in	*fastfrz*	fast freeze
in	*rasrcp*	compensation code for PMOS
in	*rasrcn*	compensation code for NMOS
in	*nasrc_sel*	NASRC read select
in	*psw_ovr*	2.5v override

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner,
- SC_ERR_UNAVAILABLE if process not applicable

Refer to the SoC Pad List for valid pad values.

Note *psw_ovr* is only applicable to pads supporting 2.5 volt operation (e.g. some Ethernet pads).

9.15.3.16 sc_pad_get_gp_28fdsoi_comp()

```
sc_err_t sc_pad_get_gp_28fdsoi_comp (
            sc_ipc_t ipc,
            sc_pad_t pad,
            uint8_t * compen,
            sc_bool_t * fastfrz,
            uint8_t * rasrcp,
            uint8_t * rasrcn,
            sc_bool_t * nasrc_sel,
            sc_bool_t * compok,
            uint8_t * nasrc,
            sc_bool_t * psw_ovr )
```

This function gets the compensation control specific to 28FDSOI.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to query
out	*compen*	pointer to return compensation/freeze mode
out	*fastfrz*	pointer to return fast freeze
out	*rasrcp*	pointer to return compensation code for PMOS
out	*rasrcn*	pointer to return compensation code for NMOS
out	*nasrc_sel*	pointer to return NASRC read select
out	*compok*	pointer to return compensation status
out	*nasrc*	pointer to return NASRCP/NASRCN
out	*psw_ovr*	pointer to return the 2.5v override

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner,
- SC_ERR_UNAVAILABLE if process not applicable

Refer to the SoC Pad List for valid pad values.

9.16 (SVC) Timer Service

Module for the Timer service.

Typedefs

- typedef uint8_t sc_timer_wdog_action_t

  This type is used to configure the watchdog action.
- typedef uint32_t sc_timer_wdog_time_t

  This type is used to declare a watchdog time value in milliseconds.

Defines for type widths

- #define SC_TIMER_ACTION_W 3U

  Width of sc_timer_wdog_action_t.

Defines for sc_timer_wdog_action_t

- #define SC_TIMER_WDOG_ACTION_PARTITION 0U

  Reset partition.
- #define SC_TIMER_WDOG_ACTION_WARM 1U

  Warm reset system.
- #define SC_TIMER_WDOG_ACTION_COLD 2U

  Cold reset system.
- #define SC_TIMER_WDOG_ACTION_BOARD 3U

  Reset board.
- #define SC_TIMER_WDOG_ACTION_IRQ 4U

  Only generate IRQs.

Wathdog Functions

- sc_err_t sc_timer_set_wdog_timeout (sc_ipc_t ipc, sc_timer_wdog_time_t timeout)

  This function sets the watchdog timeout in milliseconds.
- sc_err_t sc_timer_set_wdog_pre_timeout (sc_ipc_t ipc, sc_timer_wdog_time_t pre_timeout)

  This function sets the watchdog pre-timeout in milliseconds.
- sc_err_t sc_timer_start_wdog (sc_ipc_t ipc, sc_bool_t lock)

  This function starts the watchdog.
- sc_err_t sc_timer_stop_wdog (sc_ipc_t ipc)

  This function stops the watchdog if it is not locked.
- sc_err_t sc_timer_ping_wdog (sc_ipc_t ipc)

  This function pings (services, kicks) the watchdog resetting the time before expiration back to the timeout.
- sc_err_t sc_timer_get_wdog_status (sc_ipc_t ipc, sc_timer_wdog_time_t *timeout, sc_timer_wdog_time_t *max_timeout, sc_timer_wdog_time_t *remaining_time)

  This function gets the status of the watchdog.
- sc_err_t sc_timer_pt_get_wdog_status (sc_ipc_t ipc, sc_rm_pt_t pt, sc_bool_t *enb, sc_timer_wdog_time_t *timeout, sc_timer_wdog_time_t *remaining_time)

  This function gets the status of the watchdog of a partition.
- sc_err_t sc_timer_set_wdog_action (sc_ipc_t ipc, sc_rm_pt_t pt, sc_timer_wdog_action_t action)

  This function configures the action to be taken when a watchdog expires.

Real-Time Clock (RTC) Functions

- sc_err_t sc_timer_set_rtc_time (sc_ipc_t ipc, uint16_t year, uint8_t mon, uint8_t day, uint8_t hour, uint8_t min, uint8_t sec)

  This function sets the RTC time.
- sc_err_t sc_timer_get_rtc_time (sc_ipc_t ipc, uint16_t *year, uint8_t *mon, uint8_t *day, uint8_t *hour, uint8_t *min, uint8_t *sec)

  This function gets the RTC time.
- sc_err_t sc_timer_get_rtc_sec1970 (sc_ipc_t ipc, uint32_t *sec)

  This function gets the RTC time in seconds since 1/1/1970.
- sc_err_t sc_timer_set_rtc_alarm (sc_ipc_t ipc, uint16_t year, uint8_t mon, uint8_t day, uint8_t hour, uint8_t min, uint8_t sec)

  This function sets the RTC alarm.
- sc_err_t sc_timer_set_rtc_periodic_alarm (sc_ipc_t ipc, uint32_t sec)

  This function sets the RTC alarm (periodic mode).
- sc_err_t sc_timer_cancel_rtc_alarm (sc_ipc_t ipc)

  This function cancels the RTC alarm.
- sc_err_t sc_timer_set_rtc_calb (sc_ipc_t ipc, int8_t count)

  This function sets the RTC calibration value.

System Counter (SYSCTR) Functions

- sc_err_t sc_timer_set_sysctr_alarm (sc_ipc_t ipc, uint64_t ticks)

  This function sets the SYSCTR alarm.
- sc_err_t sc_timer_set_sysctr_periodic_alarm (sc_ipc_t ipc, uint64_t ticks)

  This function sets the SYSCTR alarm (periodic mode).
- sc_err_t sc_timer_cancel_sysctr_alarm (sc_ipc_t ipc)

  This function cancels the SYSCTR alarm.

9.16.1 Detailed Description

Module for the Timer service.

This includes support for the watchdog, RTC, and system counter. Note every resource partition has a watchdog it can use.

9.16.2 Function Documentation

9.16.2.1 sc_timer_set_wdog_timeout()

```
sc_err_t sc_timer_set_wdog_timeout (
             sc_ipc_t ipc,
             sc_timer_wdog_time_t timeout )
```

This function sets the watchdog timeout in milliseconds.

If not set then the timeout defaults to the max. Once locked this value cannot be changed.

Parameters

in	*ipc*	IPC handle
in	*timeout*	timeout period for the watchdog

Returns

Returns an error code (SC_ERR_NONE = success, SC_ERR_LOCKED = locked).

9.16.2.2 sc_timer_set_wdog_pre_timeout()

```
sc_err_t sc_timer_set_wdog_pre_timeout (
             sc_ipc_t ipc,
             sc_timer_wdog_time_t pre_timeout )
```

This function sets the watchdog pre-timeout in milliseconds.

If not set then the pre-timeout defaults to the max. Once locked this value cannot be changed.

Parameters

in	*ipc*	IPC handle
in	*pre_timeout*	pre-timeout period for the watchdog

When the pre-timout expires an IRQ will be generated. Note this timeout clears when the IRQ is triggered. An IRQ is generated for the failing partition and all of its child partitions.

Returns

Returns an error code (SC_ERR_NONE = success).

9.16.2.3 sc_timer_start_wdog()

```
sc_err_t sc_timer_start_wdog (
             sc_ipc_t ipc,
             sc_bool_t lock )
```

This function starts the watchdog.

Parameters

in	*ipc*	IPC handle
in	*lock*	boolean indicating the lock status

Returns

Returns an error code (SC_ERR_NONE = success).

If *lock* is set then the watchdog cannot be stopped or the timeout period changed.

9.16.2.4 sc_timer_stop_wdog()

```
sc_err_t sc_timer_stop_wdog (
            sc_ipc_t ipc )
```

This function stops the watchdog if it is not locked.

Parameters

in	*ipc*	IPC handle

Returns

Returns an error code (SC_ERR_NONE = success, SC_ERR_LOCKED = locked).

9.16.2.5 sc_timer_ping_wdog()

```
sc_err_t sc_timer_ping_wdog (
            sc_ipc_t ipc )
```

This function pings (services, kicks) the watchdog resetting the time before expiration back to the timeout.

Parameters

in	*ipc*	IPC handle

Returns

Returns an error code (SC_ERR_NONE = success).

9.16.2.6 sc_timer_get_wdog_status()

```
sc_err_t sc_timer_get_wdog_status (
            sc_ipc_t ipc,
            sc_timer_wdog_time_t * timeout,
            sc_timer_wdog_time_t * max_timeout,
            sc_timer_wdog_time_t * remaining_time )
```

This function gets the status of the watchdog.

All arguments are in milliseconds.

Parameters

in	*ipc*	IPC handle
out	*timeout*	pointer to return the timeout
out	*max_timeout*	pointer to return the max timeout
out	*remaining_time*	pointer to return the time remaining until trigger

Returns

Returns an error code (SC_ERR_NONE = success).

9.16.2.7 sc_timer_pt_get_wdog_status()

```
sc_err_t sc_timer_pt_get_wdog_status (
            sc_ipc_t ipc,
            sc_rm_pt_t pt,
            sc_bool_t * enb,
            sc_timer_wdog_time_t * timeout,
            sc_timer_wdog_time_t * remaining_time )
```

This function gets the status of the watchdog of a partition.

All arguments are in milliseconds.

Parameters

in	*ipc*	IPC handle
in	*pt*	partition to query
out	*enb*	pointer to return enable status
out	*timeout*	pointer to return the timeout
out	*remaining_time*	pointer to return the time remaining until trigger

Returns

Returns an error code (SC_ERR_NONE = success).

9.16.2.8 sc_timer_set_wdog_action()

```
sc_err_t sc_timer_set_wdog_action (
             sc_ipc_t ipc,
             sc_rm_pt_t pt,
             sc_timer_wdog_action_t action )
```

This function configures the action to be taken when a watchdog expires.

Parameters

in	*ipc*	IPC handle
in	*pt*	partition to affect
in	*action*	action to take

Default action is inherited from the parent.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid parameters,
- SC_ERR_NOACCESS if caller's partition is not the SYSTEM owner,
- SC_ERR_LOCKED if the watchdog is locked

9.16.2.9 sc_timer_set_rtc_time()

```
sc_err_t sc_timer_set_rtc_time (
             sc_ipc_t ipc,
             uint16_t year,
             uint8_t mon,
             uint8_t day,
             uint8_t hour,
             uint8_t min,
             uint8_t sec )
```

This function sets the RTC time.

Only the owner of the SC_R_SYSTEM resource can set the time.

Parameters

in	*ipc*	IPC handle
in	*year*	year (min 1970)
in	*mon*	month (1-12)
in	*day*	day of the month (1-31)
in	*hour*	hour (0-23)
in	*min*	minute (0-59)
in	*sec*	second (0-59)

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid time/date parameters,
- SC_ERR_NOACCESS if caller's partition is not the SYSTEM owner

9.16.2.10 sc_timer_get_rtc_time()

```
sc_err_t sc_timer_get_rtc_time (
             sc_ipc_t ipc,
             uint16_t * year,
             uint8_t * mon,
             uint8_t * day,
             uint8_t * hour,
             uint8_t * min,
             uint8_t * sec )
```

This function gets the RTC time.

Parameters

in	*ipc*	IPC handle
out	*year*	pointer to return year (min 1970)
out	*mon*	pointer to return month (1-12)
out	*day*	pointer to return day of the month (1-31)
out	*hour*	pointer to return hour (0-23)
out	*min*	pointer to return minute (0-59)
out	*sec*	pointer to return second (0-59)

Returns

Returns an error code (SC_ERR_NONE = success).

9.16.2.11 sc_timer_get_rtc_sec1970()

```
sc_err_t sc_timer_get_rtc_sec1970 (
            sc_ipc_t ipc,
            uint32_t * sec )
```

This function gets the RTC time in seconds since 1/1/1970.

Parameters

in	*ipc*	IPC handle
out	*sec*	pointer to return second

Returns

Returns an error code (SC_ERR_NONE = success).

9.16.2.12 sc_timer_set_rtc_alarm()

```
sc_err_t sc_timer_set_rtc_alarm (
            sc_ipc_t ipc,
            uint16_t year,
            uint8_t mon,
            uint8_t day,
            uint8_t hour,
            uint8_t min,
            uint8_t sec )
```

This function sets the RTC alarm.

Parameters

in	*ipc*	IPC handle
in	*year*	year (min 1970)
in	*mon*	month (1-12)
in	*day*	day of the month (1-31)
in	*hour*	hour (0-23)
in	*min*	minute (0-59)
in	*sec*	second (0-59)

Note this alarm setting clears when the alarm is triggered.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid time/date parameters

9.16.2.13 sc_timer_set_rtc_periodic_alarm()

```
sc_err_t sc_timer_set_rtc_periodic_alarm (
            sc_ipc_t ipc,
            uint32_t sec )
```

This function sets the RTC alarm (periodic mode).

Parameters

in	*ipc*	IPC handle
in	*sec*	period in seconds

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid time/date parameters

9.16.2.14 sc_timer_cancel_rtc_alarm()

```
sc_err_t sc_timer_cancel_rtc_alarm (
            sc_ipc_t ipc )
```

This function cancels the RTC alarm.

Parameters

in	*ipc*	IPC handle

Note this alarm setting clears when the alarm is triggered.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid time/date parameters

9.16.2.15 sc_timer_set_rtc_calb()

```
sc_err_t sc_timer_set_rtc_calb (
             sc_ipc_t ipc,
             int8_t count )
```

This function sets the RTC calibration value.

Only the owner of the SC_R_SYSTEM resource can set the calibration.

Parameters

in	*ipc*	IPC handle
in	*count*	calbration count (-16 to 15)

The calibration value is a 5-bit value including the sign bit, which is implemented in 2's complement. It is added or subtracted from the RTC on a perdiodic basis, once per 32768 cycles of the RTC clock.

Returns

Returns an error code (SC_ERR_NONE = success).

9.16.2.16 sc_timer_set_sysctr_alarm()

```
sc_err_t sc_timer_set_sysctr_alarm (
             sc_ipc_t ipc,
             uint64_t ticks )
```

This function sets the SYSCTR alarm.

Parameters

in	*ipc*	IPC handle
in	*ticks*	number of 8MHz cycles

Note this alarm setting clears when the alarm is triggered.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid time/date parameters

9.16.2.17 sc_timer_set_sysctr_periodic_alarm()

```
sc_err_t sc_timer_set_sysctr_periodic_alarm (
            sc_ipc_t ipc,
            uint64_t ticks )
```

This function sets the SYSCTR alarm (periodic mode).

Parameters

in	*ipc*	IPC handle
in	*ticks*	number of 8MHz cycles

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid time/date parameters

9.16.2.18 sc_timer_cancel_sysctr_alarm()

```
sc_err_t sc_timer_cancel_sysctr_alarm (
            sc_ipc_t ipc )
```

This function cancels the SYSCTR alarm.

Parameters

in	*ipc*	IPC handle

Note this alarm setting clears when the alarm is triggered.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid time/date parameters

9.17 (SVC) Power Management Service

Module for the Power Management (PM) service.

Typedefs

- typedef uint8_t sc_pm_power_mode_t

  This type is used to declare a power mode.
- typedef uint8_t sc_pm_clk_t

  This type is used to declare a clock.
- typedef uint8_t sc_pm_clk_mode_t

  This type is used to declare a clock mode.
- typedef uint8_t sc_pm_clk_parent_t

  This type is used to declare the clock parent.
- typedef uint32_t sc_pm_clock_rate_t

  This type is used to declare clock rates.
- typedef uint8_t sc_pm_reset_type_t

  This type is used to declare a desired reset type.
- typedef uint8_t sc_pm_reset_reason_t

  This type is used to declare a reason for a reset.
- typedef uint8_t sc_pm_sys_if_t

  This type is used to specify a system-level interface to be power managed.
- typedef uint8_t sc_pm_wake_src_t

  This type is used to specify a wake source for CPU resources.

Defines for type widths

- #define SC_PM_POWER_MODE_W 2U

  Width of sc_pm_power_mode_t.
- #define SC_PM_CLOCK_MODE_W 3U

  Width of sc_pm_clock_mode_t.
- #define SC_PM_RESET_TYPE_W 2U

  Width of sc_pm_reset_type_t.
- #define SC_PM_RESET_REASON_W 4U

  Width of sc_pm_reset_reason_t.

Defines for ALL parameters

- #define SC_PM_CLK_ALL ((sc_pm_clk_t) UINT8_MAX)

  All clocks.

Defines for sc_pm_power_mode_t

- #define SC_PM_PW_MODE_OFF 0U

  Power off.
- #define SC_PM_PW_MODE_STBY 1U

  Power in standby.
- #define SC_PM_PW_MODE_LP 2U

  Power in low-power.
- #define SC_PM_PW_MODE_ON 3U

  Power on.

Defines for sc_pm_clk_t

- #define SC_PM_CLK_SLV_BUS 0U

  Slave bus clock.
- #define SC_PM_CLK_MST_BUS 1U

  Master bus clock.
- #define SC_PM_CLK_PER 2U

  Peripheral clock.
- #define SC_PM_CLK_PHY 3U

  Phy clock.
- #define SC_PM_CLK_MISC 4U

  Misc clock.
- #define SC_PM_CLK_MISC0 0U

  Misc 0 clock.
- #define SC_PM_CLK_MISC1 1U

  Misc 1 clock.
- #define SC_PM_CLK_MISC2 2U

  Misc 2 clock.
- #define SC_PM_CLK_MISC3 3U

  Misc 3 clock.
- #define SC_PM_CLK_MISC4 4U

  Misc 4 clock.
- #define SC_PM_CLK_CPU 2U

  CPU clock.
- #define SC_PM_CLK_PLL 4U

  PLL.
- #define SC_PM_CLK_BYPASS 4U

  Bypass clock.

Defines for sc_pm_clk_mode_t

- #define SC_PM_CLK_MODE_ROM_INIT 0U

  Clock is initialized by ROM.
- #define SC_PM_CLK_MODE_OFF 1U

  Clock is disabled.
- #define SC_PM_CLK_MODE_ON 2U

  Clock is enabled.
- #define SC_PM_CLK_MODE_AUTOGATE_SW 3U

  Clock is in SW autogate mode.
- #define SC_PM_CLK_MODE_AUTOGATE_HW 4U

  Clock is in HW autogate mode.
- #define SC_PM_CLK_MODE_AUTOGATE_SW_HW 5U

  Clock is in SW-HW autogate mode.

Defines for sc_pm_clk_parent_t

- #define SC_PM_PARENT_XTAL 0U

  Parent is XTAL.
- #define SC_PM_PARENT_PLL0 1U

  Parent is PLL0.
- #define SC_PM_PARENT_PLL1 2U

  Parent is PLL1 or PLL0/2.
- #define SC_PM_PARENT_PLL2 3U

  Parent in PLL2 or PLL0/4.
- #define SC_PM_PARENT_BYPS 4U

  Parent is a bypass clock.

Defines for sc_pm_reset_type_t

- #define SC_PM_RESET_TYPE_COLD 0U

  Cold reset.
- #define SC_PM_RESET_TYPE_WARM 1U

  Warm reset.
- #define SC_PM_RESET_TYPE_BOARD 2U

  Board reset.

Defines for sc_pm_reset_reason_t

- #define SC_PM_RESET_REASON_POR 0U

  Power on reset.
- #define SC_PM_RESET_REASON_JTAG 1U

  JTAG reset.
- #define SC_PM_RESET_REASON_SW 2U

  Software reset.
- #define SC_PM_RESET_REASON_WDOG 3U

  Partition watchdog reset.
- #define SC_PM_RESET_REASON_LOCKUP 4U

  SCU lockup reset.
- #define SC_PM_RESET_REASON_SNVS 5U

  SNVS reset.
- #define SC_PM_RESET_REASON_TEMP 6U

  Temp panic reset.
- #define SC_PM_RESET_REASON_MSI 7U

  MSI reset.
- #define SC_PM_RESET_REASON_UECC 8U

  ECC reset.
- #define SC_PM_RESET_REASON_SCFW_WDOG 9U

  SCFW watchdog reset.
- #define SC_PM_RESET_REASON_ROM_WDOG 10U

  SCU ROM watchdog reset.
- #define SC_PM_RESET_REASON_SECO 11U

  SECO reset.
- #define SC_PM_RESET_REASON_SCFW_FAULT 12U

  SCFW fault reset.

Defines for sc_pm_sys_if_t

- #define SC_PM_SYS_IF_INTERCONNECT 0U

  System interconnect.
- #define SC_PM_SYS_IF_MU 1U

  AP -> SCU message units.
- #define SC_PM_SYS_IF_OCMEM 2U

  On-chip memory (ROM/OCRAM)
- #define SC_PM_SYS_IF_DDR 3U

  DDR memory.

Defines for sc_pm_wake_src_t

- #define SC_PM_WAKE_SRC_NONE 0U

  No wake source, used for self-kill.
- #define SC_PM_WAKE_SRC_SCU 1U

  Wakeup from SCU to resume CPU (IRQSTEER & GIC powered down)
- #define SC_PM_WAKE_SRC_IRQSTEER 2U

  Wakeup from IRQSTEER to resume CPU (GIC powered down)
- #define SC_PM_WAKE_SRC_IRQSTEER_GIC 3U

  Wakeup from IRQSTEER+GIC to wake CPU (GIC clock gated)
- #define SC_PM_WAKE_SRC_GIC 4U

  Wakeup from GIC to wake CPU.

Power Functions

- sc_err_t sc_pm_set_sys_power_mode (sc_ipc_t ipc, sc_pm_power_mode_t mode)

  This function sets the system power mode.
- sc_err_t sc_pm_set_partition_power_mode (sc_ipc_t ipc, sc_rm_pt_t pt, sc_pm_power_mode_t mode)

  This function sets the power mode of a partition.
- sc_err_t sc_pm_get_sys_power_mode (sc_ipc_t ipc, sc_rm_pt_t pt, sc_pm_power_mode_t *mode)

  This function gets the power mode of a partition.
- sc_err_t sc_pm_set_resource_power_mode (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_power_mode_t mode)

  This function sets the power mode of a resource.
- sc_err_t sc_pm_set_resource_power_mode_all (sc_ipc_t ipc, sc_rm_pt_t pt, sc_pm_power_mode_t mode, sc_rsrc_t exclude)

  This function sets the power mode for all the resources owned by a child partition.
- sc_err_t sc_pm_get_resource_power_mode (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_power_mode_t *mode)

  This function gets the power mode of a resource.
- sc_err_t sc_pm_req_low_power_mode (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_power_mode_t mode)

  This function requests the low power mode some of the resources can enter based on their state.
- sc_err_t sc_pm_req_cpu_low_power_mode (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_power_mode_t mode, sc_pm_wake_src_t wake_src)

  This function requests low-power mode entry for CPU/cluster resources.
- sc_err_t sc_pm_set_cpu_resume_addr (sc_ipc_t ipc, sc_rsrc_t resource, sc_faddr_t address)

  This function is used to set the resume address of a CPU.
- sc_err_t sc_pm_set_cpu_resume (sc_ipc_t ipc, sc_rsrc_t resource, sc_bool_t isPrimary, sc_faddr_t address)

  This function is used to set parameters for CPU resume from low-power mode.
- sc_err_t sc_pm_req_sys_if_power_mode (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_sys_if_t sys_if, sc_pm_power_mode_t hpm, sc_pm_power_mode_t lpm)

  This function requests the power mode configuration for system-level interfaces including messaging units, interconnect, and memories.

Clock/PLL Functions

- sc_err_t sc_pm_set_clock_rate (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_clk_t clk, sc_pm_clock_rate_t *rate)

  This function sets the rate of a resource's clock/PLL.
- sc_err_t sc_pm_get_clock_rate (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_clk_t clk, sc_pm_clock_rate_t *rate)

  This function gets the rate of a resource's clock/PLL.
- sc_err_t sc_pm_clock_enable (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_clk_t clk, sc_bool_t enable, sc_bool_t autog)

  This function enables/disables a resource's clock.
- sc_err_t sc_pm_set_clock_parent (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_clk_t clk, sc_pm_clk_parent_t parent)

  This function sets the parent of a resource's clock.
- sc_err_t sc_pm_get_clock_parent (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_clk_t clk, sc_pm_clk_parent_t *parent)

  This function gets the parent of a resource's clock.

Reset Functions

- sc_err_t sc_pm_reset (sc_ipc_t ipc, sc_pm_reset_type_t type)

  This function is used to reset the system.
- sc_err_t sc_pm_reset_reason (sc_ipc_t ipc, sc_pm_reset_reason_t *reason)

  This function gets a caller's reset reason.
- sc_err_t sc_pm_boot (sc_ipc_t ipc, sc_rm_pt_t pt, sc_rsrc_t resource_cpu, sc_faddr_t boot_addr, sc_rsrc_t resource_mu, sc_rsrc_t resource_dev)

  This function is used to boot a partition.
- void sc_pm_reboot (sc_ipc_t ipc, sc_pm_reset_type_t type)

  This function is used to reboot the caller's partition.
- sc_err_t sc_pm_reboot_partition (sc_ipc_t ipc, sc_rm_pt_t pt, sc_pm_reset_type_t type)

  This function is used to reboot a partition.
- sc_err_t sc_pm_cpu_start (sc_ipc_t ipc, sc_rsrc_t resource, sc_bool_t enable, sc_faddr_t address)

  This function is used to start/stop a CPU.

9.17.1 Detailed Description

Module for the Power Management (PM) service.

The following SCFW PM code is an example of how to configure the power and clocking of a UART. All resources MUST be powered on before accessing.

The ipc parameter most functions take is a handle to the IPC channel opened to communicate to the SC. It is implementation defined. Most API ports include an sc_ipc_open() and sc_ipc_close() function to manage this. The sc_ipc_open() takes an argument to identify the communication channel (usually the MU address) and returns the IPC handle that all API calls should then use.

Refer to the SoC-specific RESOURCES for a list of resources. Refer to the SoC-specific CLOCKS for a list of clocks.

```
1 sc_pm_clock_rate_t rate = SC_160MHZ;
```

```
2
3 /* Powerup UART 0 */
4 sc_pm_set_resource_power_mode(ipc, SC_R_UART_0,       SC_PM_PW_MODE_ON);
5
6 /* Configure UART 0 baud clock */
7 sc_pm_set_clock_rate(ipc, SC_R_UART_0, SC_PM_CLK_PER, &rate);
8
9 /* Enable UART 0 clock */
10 sc_pm_clock_enable(ipc, SC_R_UART_0, SC_PM_CLK_PER,       SC_TRUE, SC_FALSE);
```

First, a variable is declared to hold the rate to request and return for the UART peripheral clock. Note this is the baud clock going into the UART which is then further divided within the UART itself.

```
sc_pm_clock_rate_t rate = SC_160MHZ;
```

Then change the power state of the UART to the ON state.

```
/* Powerup UART 0 */
sc_pm_set_resource_power_mode(ipc, SC_R_UART_0,       SC_PM_PW_MODE_ON);
```

Then configure the UART peripheral clock. Note that due to hardware limitation, the exact rate may not be what is requested. The rate is guarenteed to not be greater than the requested rate. The actual rate is returned in the varaible. The actual rate should be used when configuring the UART IP. Note that 160MHz is used as that can be divided by the UART to hit all the common UART rates within required error. Other frequencies may have issues and the caller needs to calculate the baud clock error rate. See the UART section of the SoC RM.

```
/* Configure UART 0 baud clock */
sc_pm_set_clock_rate(ipc, SC_R_UART_0, SC_PM_CLK_PER, &rate);
```

Then enable the clock.

```
/* Enable UART 0 clock */
sc_pm_clock_enable(ipc, SC_R_UART_0, SC_PM_CLK_PER,       SC_TRUE, SC_FALSE);
```

At this point, the UART IP can be configured and used.

9.17.2 Macro Definition Documentation

9.17.2.1 SC_PM_CLK_MODE_ROM_INIT

```
#define SC_PM_CLK_MODE_ROM_INIT 0U
```

Clock is initialized by ROM.

9.17.2.2 SC_PM_CLK_MODE_ON

```
#define SC_PM_CLK_MODE_ON 2U
```

Clock is enabled.

9.17.2.3 SC_PM_PARENT_XTAL

```
#define SC_PM_PARENT_XTAL 0U
```

Parent is XTAL.

9.17.2.4 SC_PM_PARENT_BYPS

```
#define SC_PM_PARENT_BYPS 4U
```

Parent is a bypass clock.

9.17.3 Typedef Documentation

9.17.3.1 sc_pm_power_mode_t

```
typedef uint8_t sc_pm_power_mode_t
```

This type is used to declare a power mode.

Note resources only use SC_PM_PW_MODE_OFF and SC_PM_PW_MODE_ON. The other modes are used only as system power modes.

9.17.4 Function Documentation

9.17.4.1 sc_pm_set_sys_power_mode()

```
sc_err_t sc_pm_set_sys_power_mode (
             sc_ipc_t ipc,
             sc_pm_power_mode_t mode )
```

This function sets the system power mode.

Only the owner of the SC_R_SYSTEM resource can do this.

Parameters

in	*ipc*	IPC handle
in	*mode*	power mode to apply

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid mode,
- SC_ERR_NOACCESS if caller not the owner of SC_R_SYSTEM

See also

sc_pm_set_sys_power_mode().

9.17.4.2 sc_pm_set_partition_power_mode()

```
sc_err_t sc_pm_set_partition_power_mode (
             sc_ipc_t ipc,
             sc_rm_pt_t pt,
             sc_pm_power_mode_t mode )
```

This function sets the power mode of a partition.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition
in	*mode*	power mode to apply

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid partition or mode,
- SC_ERR_NOACCESS if caller's partition is not the owner or parent of *pt*

The power mode of the partitions is a max power any resource will be set to. Calling this will result in all resources owned by *pt* to have their power changed to the lower of *mode* or the individual resource mode set using sc_pm_set_resource_power_mode().

9.17.4.3 sc_pm_get_sys_power_mode()

```
sc_err_t sc_pm_get_sys_power_mode (
             sc_ipc_t ipc,
             sc_rm_pt_t pt,
             sc_pm_power_mode_t * mode )
```

This function gets the power mode of a partition.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition
out	*mode*	pointer to return power mode

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid partition

9.17.4.4 sc_pm_set_resource_power_mode()

```
sc_err_t sc_pm_set_resource_power_mode (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_pm_power_mode_t mode )
```

This function sets the power mode of a resource.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the resource
in	*mode*	power mode to apply

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid resource or mode,
- SC_ERR_NOACCESS if caller's partition is not the resource owner or parent of the owner

Resources must be at SC_PM_PW_MODE_LP mode or higher to access them, otherwise the master will get a bus error or hang.

This function will record the individual resource power mode and change it if the requested mode is lower than or equal to the partition power mode set with sc_pm_set_partition_power_mode(). In other words, the power mode of the resource will be the minimum of the resource power mode and the partition power mode.

Note some resources are still not accessible even when powered up if bus transactions go through a fabric not powered up. Examples of this are resources in display and capture subsystems which require the display controller or the imaging subsytem to be powered up first.

Not that resources are grouped into power domains by the underlying hardware. If any resource in the domain is on, the entire power domain will be on. Other power domains required to access the resource will also be turned on. Clocks required to access the peripheral will be turned on. Refer to the SoC RM for more info on power domains and access infrastructure (bus fabrics, clock domains, etc.).

9.17.4.5 sc_pm_set_resource_power_mode_all()

```
sc_err_t sc_pm_set_resource_power_mode_all (
             sc_ipc_t ipc,
             sc_rm_pt_t pt,
             sc_pm_power_mode_t mode,
             sc_rsrc_t exclude )
```

This function sets the power mode for all the resources owned by a child partition.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of child partition
in	*mode*	power mode to apply
in	*exclude*	resource to exclude

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid partition or mode,
- SC_ERR_NOACCESS if caller's partition is not the parent of *pt*

This functions loops through all the resources owned by *pt* and sets the power mode to *mode*. It will skip setting *exclude* (SC_R_LAST to skip none).

This function can only be called by the parent. It is used to implement some aspects of virtualization.

9.17.4.6 sc_pm_get_resource_power_mode()

```
sc_err_t sc_pm_get_resource_power_mode (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_pm_power_mode_t * mode )
```

This function gets the power mode of a resource.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the resource
out	*mode*	pointer to return power mode

Returns

Returns an error code (SC_ERR_NONE = success).

Note only SC_PM_PW_MODE_OFF and SC_PM_PW_MODE_ON are valid. The value returned does not reflect the power mode of the partition..

9.17.4.7 sc_pm_req_low_power_mode()

```
sc_err_t sc_pm_req_low_power_mode (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_pm_power_mode_t mode )
```

This function requests the low power mode some of the resources can enter based on their state.

This API is only valid for the following resources : SC_R_A53, SC_R_A53_0, SC_R_A53_1, SC_A53_2, SC_A53_3, SC_R_A72, SC_R_A72_0, SC_R_A72_1, SC_R_CC1, SC_R_A35, SC_R_A35_0, SC_R_A35_1, SC_R_A35_2, SC↩_R_A35_3. For all other resources it will return SC_ERR_PARAM. This function will set the low power mode the cores, cluster and cluster associated resources will enter when all the cores in a given cluster execute WFI

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the resource
in	*mode*	power mode to apply

Returns

Returns an error code (SC_ERR_NONE = success).

9.17.4.8 sc_pm_req_cpu_low_power_mode()

```
sc_err_t sc_pm_req_cpu_low_power_mode (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_pm_power_mode_t mode,
             sc_pm_wake_src_t wake_src )
```

This function requests low-power mode entry for CPU/cluster resources.

This API is only valid for the following resources: SC_R_A53, SC_R_A53_x, SC_R_A72, SC_R_A72_x, SC_R_A35, SC_R_A35_x, SC_R_CCI. For all other resources it will return SC_ERR_PARAM. For individual core resources, the specified power mode and wake source will be applied after the core has entered WFI. For cluster resources, the specified power mode is applied after all cores in the cluster have entered low-power mode.
For multicluster resources, the specified power mode is applied after all clusters have reached low-power mode.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the resource
in	*mode*	power mode to apply
in	*wake_src*	wake source for low-power exit

Returns

Returns an error code (SC_ERR_NONE = success).

9.17.4.9 sc_pm_set_cpu_resume_addr()

```
sc_err_t sc_pm_set_cpu_resume_addr (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_faddr_t address )
```

This function is used to set the resume address of a CPU.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the CPU resource
in	*address*	64-bit resume address

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid resource or address,
- SC_ERR_NOACCESS if caller's partition is not the parent of the resource (CPU) owner

9.17.4.10 sc_pm_set_cpu_resume()

```
sc_err_t sc_pm_set_cpu_resume (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_bool_t isPrimary,
             sc_faddr_t address )
```

This function is used to set parameters for CPU resume from low-power mode.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the CPU resource
in	*isPrimary*	set SC_TRUE if primary wake CPU
in	*address*	64-bit resume address

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid resource or address,
- SC_ERR_NOACCESS if caller's partition is not the parent of the resource (CPU) owner

9.17.4.11 sc_pm_req_sys_if_power_mode()

```
sc_err_t sc_pm_req_sys_if_power_mode (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
```

```
            sc_pm_sys_if_t sys_if,
            sc_pm_power_mode_t hpm,
            sc_pm_power_mode_t lpm )
```

This function requests the power mode configuration for system-level interfaces including messaging units, interconnect, and memories.

This API is only valid for the following resources : SC_R_A53, SC_R_A72, and SC_R_M4_x_PID_y. For all other resources, it will return SC_ERR_PARAM. The requested power mode will be captured and applied to system-level resources as system conditions allow.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the resource
in	*sys_if*	system-level interface to be configured
in	*hpm*	high-power mode for the system interface
in	*lpm*	low-power mode for the system interface

Returns

Returns an error code (SC_ERR_NONE = success).

9.17.4.12 sc_pm_set_clock_rate()

```
sc_err_t sc_pm_set_clock_rate (
            sc_ipc_t ipc,
            sc_rsrc_t resource,
            sc_pm_clk_t clk,
            sc_pm_clock_rate_t * rate )
```

This function sets the rate of a resource's clock/PLL.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the resource
in	*clk*	clock/PLL to affect
in,out	*rate*	pointer to rate to set, return actual rate

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid resource or clock/PLL,
- SC_ERR_NOACCESS if caller's partition is not the resource owner or parent of the owner,
- SC_ERR_UNAVAILABLE if clock/PLL not applicable to this resource,
- SC_ERR_LOCKED if rate locked (usually because shared clock/PLL)

Refer to the Clock List for valid clock/PLL values.

9.17.4.13 sc_pm_get_clock_rate()

```
sc_err_t sc_pm_get_clock_rate (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_pm_clk_t clk,
             sc_pm_clock_rate_t * rate )
```

This function gets the rate of a resource's clock/PLL.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the resource
in	*clk*	clock/PLL to affect
out	*rate*	pointer to return rate

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid resource or clock/PLL,
- SC_ERR_NOACCESS if caller's partition is not the resource owner or parent of the owner,
- SC_ERR_UNAVAILABLE if clock/PLL not applicable to this resource

Refer to the Clock List for valid clock/PLL values.

9.17.4.14 sc_pm_clock_enable()

```
sc_err_t sc_pm_clock_enable (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_pm_clk_t clk,
             sc_bool_t enable,
             sc_bool_t autog )
```

This function enables/disables a resource's clock.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the resource
in	*clk*	clock to affect
in	*enable*	enable if SC_TRUE; otherwise disabled
in	*autog*	HW auto clock gating

If *resource* is SC_R_ALL then all resources owned will be affected. No error will be returned.

If *clk* is SC_PM_CLK_ALL, then an error will be returned if any of the available clocks returns an error.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid resource or clock,
- SC_ERR_NOACCESS if caller's partition is not the resource owner or parent of the owner,
- SC_ERR_UNAVAILABLE if clock not applicable to this resource

Refer to the Clock List for valid clock values.

9.17.4.15 sc_pm_set_clock_parent()

```
sc_err_t sc_pm_set_clock_parent (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_pm_clk_t clk,
             sc_pm_clk_parent_t parent )
```

This function sets the parent of a resource's clock.

This function should only be called when the clock is disabled.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the resource
in	*clk*	clock to affect
in	*parent*	New parent of the clock.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid resource or clock,
- SC_ERR_NOACCESS if caller's partition is not the resource owner or parent of the owner,
- SC_ERR_UNAVAILABLE if clock not applicable to this resource
- SC_ERR_BUSY if clock is currently enabled.
- SC_ERR_NOPOWER if resource not powered

Refer to the Clock List for valid clock values.

9.17.4.16 sc_pm_get_clock_parent()

```
sc_err_t sc_pm_get_clock_parent (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_pm_clk_t clk,
             sc_pm_clk_parent_t * parent )
```

This function gets the parent of a resource's clock.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the resource
in	*clk*	clock to affect
out	*parent*	pointer to return parent of clock.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid resource or clock,
- SC_ERR_NOACCESS if caller's partition is not the resource owner or parent of the owner,
- SC_ERR_UNAVAILABLE if clock not applicable to this resource

Refer to the Clock List for valid clock values.

9.17.4.17 sc_pm_reset()

```
sc_err_t sc_pm_reset (
             sc_ipc_t ipc,
             sc_pm_reset_type_t type )
```

This function is used to reset the system.

Only the owner of the SC_R_SYSTEM resource can do this.

Parameters

in	*ipc*	IPC handle
in	*type*	reset type

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid type,
- SC_ERR_NOACCESS if caller not the owner of SC_R_SYSTEM

If this function returns, then the reset did not occur due to an invalid parameter.

9.17.4.18 sc_pm_reset_reason()

```
sc_err_t sc_pm_reset_reason (
             sc_ipc_t ipc,
             sc_pm_reset_reason_t * reason )
```

This function gets a caller's reset reason.

Parameters

in	*ipc*	IPC handle
out	*reason*	pointer to return reset reason

This function returns the reason a partition was reset. If the reason is POR, then the system reset reason will be returned.

Note depending on the connection of the WDOG_OUT signal and the OTP programming of the PMIC, some reset reasons my trigger a system POR and the original reason will be lost.

Returns

Returns an error code (SC_ERR_NONE = success).

9.17.4.19 sc_pm_boot()

```
sc_err_t sc_pm_boot (
             sc_ipc_t ipc,
             sc_rm_pt_t pt,
             sc_rsrc_t resource_cpu,
             sc_faddr_t boot_addr,
             sc_rsrc_t resource_mu,
             sc_rsrc_t resource_dev )
```

This function is used to boot a partition.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition to boot
in	*resource_cpu*	ID of the CPU resource to start
in	*boot_addr*	64-bit boot address
in	*resource_mu*	ID of the MU that must be powered
in	*resource_dev*	ID of the boot device that must be powered

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid partition, resource, or addr,
- SC_ERR_NOACCESS if caller's partition is not the parent of the partition to boot

9.17.4.20 sc_pm_reboot()

```
void sc_pm_reboot (
             sc_ipc_t ipc,
             sc_pm_reset_type_t type )
```

This function is used to reboot the caller's partition.

Parameters

in	*ipc*	IPC handle
in	*type*	reset type

If *type* is SC_PM_RESET_TYPE_COLD, then most peripherals owned by the calling partition will be reset if possible. SC state (partitions, power, clocks, etc.) is reset. The boot SW of the booting CPU must be able to handle peripherals that that are not reset.

If *type* is SC_PM_RESET_TYPE_WARM, then only the boot CPU is reset. SC state (partitions, power, clocks, etc.) are NOT reset. The boot SW of the booting CPU must be able to handle peripherals and SC state that that are not reset.

If *type* is SC_PM_RESET_TYPE_BOARD, then return with no action.

If this function returns, then the reset did not occur due to an invalid parameter.

9.17.4.21 sc_pm_reboot_partition()

```
sc_err_t sc_pm_reboot_partition (
             sc_ipc_t ipc,
             sc_rm_pt_t pt,
             sc_pm_reset_type_t type )
```

This function is used to reboot a partition.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition to reboot
in	*type*	reset type

If *type* is SC_PM_RESET_TYPE_COLD, then most peripherals owned by the calling partition will be reset if possible. SC state (partitions, power, clocks, etc.) is reset. The boot SW of the booting CPU must be able to handle peripherals that that are not reset.

If *type* is SC_PM_RESET_TYPE_WARM, then only the boot CPU is reset. SC state (partitions, power, clocks, etc.) are NOT reset. The boot SW of the booting CPU must be able to handle peripherals and SC state that that are not reset.

If *type* is SC_PM_RESET_TYPE_BOARD, then return with no action.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid partition or type
- SC_ERR_NOACCESS if caller's partition is not the parent of *pt*,

Most peripherals owned by the partition will be reset if possible. SC state (partitions, power, clocks, etc.) is reset. The boot SW of the booting CPU must be able to handle peripherals that that are not reset.

9.17.4.22 sc_pm_cpu_start()

```
sc_err_t sc_pm_cpu_start (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_bool_t enable,
             sc_faddr_t address )
```

This function is used to start/stop a CPU.

Parameters

in	*ipc*	IPC handle
in	*resource*	ID of the CPU resource
in	*enable*	start if SC_TRUE; otherwise stop
in	*address*	64-bit boot address

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid resource or address,
- SC_ERR_NOACCESS if caller's partition is not the parent of the resource (CPU) owner

9.18 (SVC) Interrupt Service

Module for the Interrupt (IRQ) service.

Macros

- #define SC_IRQ_NUM_GROUP 5U

  Number of groups.

Typedefs

- typedef uint8_t sc_irq_group_t

  This type is used to declare an interrupt group.
- typedef uint8_t sc_irq_temp_t

  This type is used to declare a bit mask of temp interrupts.
- typedef uint8_t sc_irq_wdog_t

  This type is used to declare a bit mask of watchdog interrupts.
- typedef uint8_t sc_irq_rtc_t

  This type is used to declare a bit mask of RTC interrupts.
- typedef uint8_t sc_irq_wake_t

  This type is used to declare a bit mask of wakeup interrupts.

Functions

- sc_err_t sc_irq_enable (sc_ipc_t ipc, sc_rsrc_t resource, sc_irq_group_t group, uint32_t mask, sc_bool_t enable)

  This function enables/disables interrupts.
- sc_err_t sc_irq_status (sc_ipc_t ipc, sc_rsrc_t resource, sc_irq_group_t group, uint32_t *status)

  This function returns the current interrupt status (regardless if masked).

Defines for sc_irq_group_t

- #define SC_IRQ_GROUP_TEMP 0U

  Temp interrupts.
- #define SC_IRQ_GROUP_WDOG 1U

  Watchdog interrupts.
- #define SC_IRQ_GROUP_RTC 2U

  RTC interrupts.
- #define SC_IRQ_GROUP_WAKE 3U

  Wakeup interrupts.
- #define SC_IRQ_GROUP_SYSCTR 4U

  System counter interrupts.

Defines for sc_irq_temp_t

- #define SC_IRQ_TEMP_HIGH (1UL << 0U)

  Temp alarm interrupt.
- #define SC_IRQ_TEMP_CPU0_HIGH (1UL << 1U)

  CPU0 temp alarm interrupt.
- #define SC_IRQ_TEMP_CPU1_HIGH (1UL << 2U)

  CPU1 temp alarm interrupt.
- #define SC_IRQ_TEMP_GPU0_HIGH (1UL << 3U)

  GPU0 temp alarm interrupt.
- #define SC_IRQ_TEMP_GPU1_HIGH (1UL << 4U)

  GPU1 temp alarm interrupt.
- #define SC_IRQ_TEMP_DRC0_HIGH (1UL << 5U)

  DRC0 temp alarm interrupt.
- #define SC_IRQ_TEMP_DRC1_HIGH (1UL << 6U)

  DRC1 temp alarm interrupt.
- #define SC_IRQ_TEMP_VPU_HIGH (1UL << 7U)

  DRC1 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC0_HIGH (1UL << 8U)

  PMIC0 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC1_HIGH (1UL << 9U)

  PMIC1 temp alarm interrupt.
- #define SC_IRQ_TEMP_LOW (1UL << 10U)

  Temp alarm interrupt.
- #define SC_IRQ_TEMP_CPU0_LOW (1UL << 11U)

  CPU0 temp alarm interrupt.
- #define SC_IRQ_TEMP_CPU1_LOW (1UL << 12U)

  CPU1 temp alarm interrupt.
- #define SC_IRQ_TEMP_GPU0_LOW (1UL << 13U)

  GPU0 temp alarm interrupt.
- #define SC_IRQ_TEMP_GPU1_LOW (1UL << 14U)

  GPU1 temp alarm interrupt.
- #define SC_IRQ_TEMP_DRC0_LOW (1UL << 15U)

  DRC0 temp alarm interrupt.
- #define SC_IRQ_TEMP_DRC1_LOW (1UL << 16U)

  DRC1 temp alarm interrupt.
- #define SC_IRQ_TEMP_VPU_LOW (1UL << 17U)

  DRC1 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC0_LOW (1UL << 18U)

  PMIC0 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC1_LOW (1UL << 19U)

  PMIC1 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC2_HIGH (1UL << 20U)

  PMIC2 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC2_LOW (1UL << 21U)

  PMIC2 temp alarm interrupt.

Defines for sc_irq_wdog_t

- #define SC_IRQ_WDOG (1U << 0U)
  Watchdog interrupt.

Defines for sc_irq_rtc_t

- #define SC_IRQ_RTC (1U << 0U)
  RTC interrupt.

Defines for sc_irq_wake_t

- #define SC_IRQ_BUTTON (1U << 0U)
  Button interrupt.
- #define SC_IRQ_PAD (1U << 1U)
  Pad wakeup.

Defines for sc_irq_sysctr_t

- #define SC_IRQ_SYSCTR (1U << 0U)
  SYSCTR interrupt.

9.18.1 Detailed Description

Module for the Interrupt (IRQ) service.

9.18.2 Function Documentation

9.18.2.1 sc_irq_enable()

```
sc_err_t sc_irq_enable (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_irq_group_t group,
             uint32_t mask,
             sc_bool_t enable )
```

This function enables/disables interrupts.

If pending interrupts are unmasked, an interrupt will be triggered.

Parameters

in	*ipc*	IPC handle
in	*resource*	MU channel
in	*group*	group the interrupts are in
in	*mask*	mask of interrupts to affect
in	*enable*	state to change interrupts to

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if group invalid

9.18.2.2 sc_irq_status()

```
sc_err_t sc_irq_status (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_irq_group_t group,
             uint32_t * status )
```

This function returns the current interrupt status (regardless if masked).

Automatically clears pending interrupts.

Parameters

in	*ipc*	IPC handle
in	*resource*	MU channel
in	*group*	groups the interrupts are in
in	*status*	status of interrupts

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if group invalid

The returned *status* may show interrupts pending that are currently masked.

9.19 (SVC) Miscellaneous Service

Module for the Miscellaneous (MISC) service.

Macros

- #define SC_MISC_DMA_GRP_MAX 31U

  Max DMA channel priority group.

Typedefs

- typedef uint8_t sc_misc_dma_group_t

  This type is used to store a DMA channel priority group.
- typedef uint8_t sc_misc_boot_status_t

  This type is used report boot status.
- typedef uint8_t sc_misc_seco_auth_cmd_t

  This type is used to issue SECO authenticate commands.
- typedef uint8_t sc_misc_temp_t

  This type is used report boot status.
- typedef uint8_t sc_misc_bt_t

  This type is used report the boot type.

Defines for type widths

- #define SC_MISC_DMA_GRP_W 5U

  Width of sc_misc_dma_group_t.

Defines for sc_misc_boot_status_t

- #define SC_MISC_BOOT_STATUS_SUCCESS 0U

  Success.
- #define SC_MISC_BOOT_STATUS_SECURITY 1U

  Security violation.

Defines for sc_misc_temp_t

- #define SC_MISC_TEMP 0U

  Temp sensor.
- #define SC_MISC_TEMP_HIGH 1U

  Temp high alarm.
- #define SC_MISC_TEMP_LOW 2U

  Temp low alarm.

Defines for sc_misc_seco_auth_cmd_t

- #define SC_MISC_AUTH_CONTAINER 0U

  Authenticate container.
- #define SC_MISC_VERIFY_IMAGE 1U

  Verify image.
- #define SC_MISC_REL_CONTAINER 2U

  Release container.
- #define SC_MISC_SECO_AUTH_SECO_FW 3U

  SECO Firmware.
- #define SC_MISC_SECO_AUTH_HDMI_TX_FW 4U

  HDMI TX Firmware.
- #define SC_MISC_SECO_AUTH_HDMI_RX_FW 5U

  HDMI RX Firmware.

Defines for sc_misc_bt_t

- #define **SC_MISC_BT_PRIMARY** 0U
- #define **SC_MISC_BT_SECONDARY** 1U
- #define **SC_MISC_BT_RECOVERY** 2U
- #define **SC_MISC_BT_MANUFACTURE** 3U
- #define **SC_MISC_BT_SERIAL** 4U

Control Functions

- sc_err_t sc_misc_set_control (sc_ipc_t ipc, sc_rsrc_t resource, sc_ctrl_t ctrl, uint32_t val)

  This function sets a miscellaneous control value.
- sc_err_t sc_misc_get_control (sc_ipc_t ipc, sc_rsrc_t resource, sc_ctrl_t ctrl, uint32_t *val)

  This function gets a miscellaneous control value.

DMA Functions

- sc_err_t sc_misc_set_max_dma_group (sc_ipc_t ipc, sc_rm_pt_t pt, sc_misc_dma_group_t max)

  This function configures the max DMA channel priority group for a partition.
- sc_err_t sc_misc_set_dma_group (sc_ipc_t ipc, sc_rsrc_t resource, sc_misc_dma_group_t group)

  This function configures the priority group for a DMA channel.

Security Functions

- sc_err_t sc_misc_seco_image_load (sc_ipc_t ipc, sc_faddr_t addr_src, sc_faddr_t addr_dst, uint32_t len, sc_bool_t fw)

  This function loads a SECO image.
- sc_err_t sc_misc_seco_authenticate (sc_ipc_t ipc, sc_misc_seco_auth_cmd_t cmd, sc_faddr_t addr)

  This function is used to authenticate a SECO image or command.
- sc_err_t sc_misc_seco_fuse_write (sc_ipc_t ipc, sc_faddr_t addr)

  This function securely writes a group of fuse words.
- sc_err_t sc_misc_seco_enable_debug (sc_ipc_t ipc, sc_faddr_t addr)

  This function securely enables debug.
- sc_err_t sc_misc_seco_forward_lifecycle (sc_ipc_t ipc, uint32_t change)

  This function updates the lifecycle of the device.
- sc_err_t sc_misc_seco_return_lifecycle (sc_ipc_t ipc, sc_faddr_t addr)

  This function updates the lifecycle to one of the return lifecycles.
- void sc_misc_seco_build_info (sc_ipc_t ipc, uint32_t *version, uint32_t *commit)

  This function is used to return the SECO FW build info.
- sc_err_t sc_misc_seco_chip_info (sc_ipc_t ipc, uint16_t *lc, uint16_t *monotonic, uint32_t *uid_l, uint32_t *uid_h)

  This function is used to return SECO chip info.
- sc_err_t sc_misc_seco_attest_mode (sc_ipc_t ipc, uint32_t mode)

  This function is used to set the attestation mode.
- sc_err_t sc_misc_seco_attest (sc_ipc_t ipc, uint64_t nonce)

  This function is used to request atestation.
- sc_err_t sc_misc_seco_get_attest_pkey (sc_ipc_t ipc, sc_faddr_t addr)

  This function is used to retrieve the attestation public key.
- sc_err_t sc_misc_seco_get_attest_sign (sc_ipc_t ipc, sc_faddr_t addr)

  This function is used to retrieve attestation signature and parameters.
- sc_err_t sc_misc_seco_attest_verify (sc_ipc_t ipc, sc_faddr_t addr)

  This function is used to verify attestation.
- sc_err_t sc_misc_seco_commit (sc_ipc_t ipc, uint32_t *info)

  This function is used to commit into the fuses any new SRK revocation and FW version information that have been found in the primary and secondary containers.

Debug Functions

- void sc_misc_debug_out (sc_ipc_t ipc, uint8_t ch)

  This function is used output a debug character from the SCU UART.
- sc_err_t sc_misc_waveform_capture (sc_ipc_t ipc, sc_bool_t enable)

  This function starts/stops emulation waveform capture.
- void sc_misc_build_info (sc_ipc_t ipc, uint32_t *build, uint32_t *commit)

  This function is used to return the SCFW build info.
- void sc_misc_unique_id (sc_ipc_t ipc, uint32_t *id_l, uint32_t *id_h)

  This function is used to return the device's unique ID.

Other Functions

- sc_err_t sc_misc_set_ari (sc_ipc_t ipc, sc_rsrc_t resource, sc_rsrc_t resource_mst, uint16_t ari, sc_bool_t enable)

  This function configures the ARI match value for PCIe/SATA resources.
- void sc_misc_boot_status (sc_ipc_t ipc, sc_misc_boot_status_t status)

  This function reports boot status.
- sc_err_t sc_misc_boot_done (sc_ipc_t ipc, sc_rsrc_t cpu)

  This function tells the SCFW that a CPU is done booting.
- sc_err_t sc_misc_otp_fuse_read (sc_ipc_t ipc, uint32_t word, uint32_t *val)

  This function reads a given fuse word index.
- sc_err_t sc_misc_otp_fuse_write (sc_ipc_t ipc, uint32_t word, uint32_t val)

  This function writes a given fuse word index.
- sc_err_t sc_misc_set_temp (sc_ipc_t ipc, sc_rsrc_t resource, sc_misc_temp_t temp, int16_t celsius, int8_t tenths)

  This function sets a temp sensor alarm.
- sc_err_t sc_misc_get_temp (sc_ipc_t ipc, sc_rsrc_t resource, sc_misc_temp_t temp, int16_t *celsius, int8_t *tenths)

  This function gets a temp sensor value.
- void sc_misc_get_boot_dev (sc_ipc_t ipc, sc_rsrc_t *dev)

  This function returns the boot device.
- sc_err_t sc_misc_get_boot_type (sc_ipc_t ipc, sc_misc_bt_t *type)

  This function returns the boot type.
- void sc_misc_get_button_status (sc_ipc_t ipc, sc_bool_t *status)

  This function returns the current status of the ON/OFF button.
- sc_err_t sc_misc_rompatch_checksum (sc_ipc_t ipc, uint32_t *checksum)

  This function returns the ROM patch checksum.

9.19.1 Detailed Description

Module for the Miscellaneous (MISC) service.

9.19.2 Function Documentation

9.19.2.1 sc_misc_set_control()

```
sc_err_t sc_misc_set_control (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_ctrl_t ctrl,
             uint32_t val )
```

This function sets a miscellaneous control value.

Parameters

in	*ipc*	IPC handle
in	*resource*	resource the control is associated with
in	*ctrl*	control to change
in	*val*	value to apply to the control

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the resource owner or parent of the owner

Refer to the Control List for valid control values.

9.19.2.2 sc_misc_get_control()

```
sc_err_t sc_misc_get_control (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_ctrl_t ctrl,
             uint32_t * val )
```

This function gets a miscellaneous control value.

Parameters

in	*ipc*	IPC handle
in	*resource*	resource the control is associated with
in	*ctrl*	control to get
out	*val*	pointer to return the control value

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the resource owner or parent of the owner

Refer to the Control List for valid control values.

9.19.2.3 sc_misc_set_max_dma_group()

```
sc_err_t sc_misc_set_max_dma_group (
             sc_ipc_t ipc,
             sc_rm_pt_t pt,
             sc_misc_dma_group_t max )
```

This function configures the max DMA channel priority group for a partition.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition to assign *max*
in	*max*	max priority group (0-31)

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the parent of the affected partition

Valid *max* range is 0-31 with 0 being the lowest and 31 the highest. Default is the max priority group for the parent partition of *pt*.

9.19.2.4 sc_misc_set_dma_group()

```
sc_err_t sc_misc_set_dma_group (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_misc_dma_group_t group )
```

This function configures the priority group for a DMA channel.

Parameters

in	*ipc*	IPC handle
in	*resource*	DMA channel resource
in	*group*	priority group (0-31)

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the owner or parent of the owner of the DMA channel

Valid *group* range is 0-31 with 0 being the lowest and 31 the highest. The max value of *group* is limited by the partition max set using sc_misc_set_max_dma_group().

9.19.2.5 sc_misc_seco_image_load()

```
sc_err_t sc_misc_seco_image_load (
             sc_ipc_t ipc,
             sc_faddr_t addr_src,
             sc_faddr_t addr_dst,
             uint32_t len,
             sc_bool_t fw )
```

This function loads a SECO image.

Parameters

in	*ipc*	IPC handle
in	*addr_src*	address of image source
in	*addr_dst*	address of image destination
in	*len*	lenth of image to load
in	*fw*	SC_TRUE = firmware load

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_PARM if word fuse index param out of range or invalid
- SC_ERR_UNAVAILABLE if SECO not available

This is used to load images via the SECO. Examples include SECO Firmware and IVT/CSF data used for authentication. These are usually loaded into SECO TCM. *addr_src* is in secure memory.

See the Security Reference Manual (SRM) for more info.

9.19.2.6 sc_misc_seco_authenticate()

```
sc_err_t sc_misc_seco_authenticate (
            sc_ipc_t ipc,
            sc_misc_seco_auth_cmd_t cmd,
            sc_faddr_t addr )
```

This function is used to authenticate a SECO image or command.

Parameters

in	*ipc*	IPC handle
in	*cmd*	authenticate command
in	*addr*	address of/or metadata

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_PARM if word fuse index param out of range or invalid
- SC_ERR_UNAVAILABLE if SECO not available

This is used to authenticate a SECO image or issue a security command. *addr* often points to an container. It is also just data (or even unused) for some commands.

See the Security Reference Manual (SRM) for more info.

9.19.2.7 sc_misc_seco_fuse_write()

```
sc_err_t sc_misc_seco_fuse_write (
            sc_ipc_t ipc,
            sc_faddr_t addr )
```

This function securely writes a group of fuse words.

Parameters

in	*ipc*	IPC handle
in	*addr*	address of message block

Returns

Returns and error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_UNAVAILABLE if SECO not available

Note *addr* must be a pointer to a signed message block.

See the Security Reference Manual (SRM) for more info.

9.19.2.8 sc_misc_seco_enable_debug()

```
sc_err_t sc_misc_seco_enable_debug (
            sc_ipc_t ipc,
            sc_faddr_t addr )
```

This function securely enables debug.

Parameters

in	*ipc*	IPC handle
in	*addr*	address of message block

Returns

Returns and error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_UNAVAILABLE if SECO not available

Note *addr* must be a pointer to a signed message block.

See the Security Reference Manual (SRM) for more info.

9.19.2.9 sc_misc_seco_forward_lifecycle()

```
sc_err_t sc_misc_seco_forward_lifecycle (
            sc_ipc_t ipc,
            uint32_t change )
```

This function updates the lifecycle of the device.

Parameters

in	*ipc*	IPC handle
in	*change*	desired lifecycle transistion

Returns

Returns and error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_UNAVAILABLE if SECO not available

This message is used for going from Open to NXP Closed to OEM Closed. Note *change* is NOT the new desired lifecycle. It is a lifecycle transition as documented in the Security Reference Manual (SRM).

If any SECO request fails or only succeeds because the part is in an "OEM open" lifecycle, then a request to transition from "NXP closed" to "OEM closed" will also fail. For example, booting a signed container when the OEM SRK is not fused will succeed, but as it is an abnormal situation, a subsequent request to transition the lifecycle will return an error.

9.19.2.10 sc_misc_seco_return_lifecycle()

```
sc_err_t sc_misc_seco_return_lifecycle (
            sc_ipc_t ipc,
            sc_faddr_t addr )
```

This function updates the lifecycle to one of the return lifecycles.

Parameters

in	*ipc*	IPC handle
in	*addr*	address of message block

Returns

Returns and error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_UNAVAILABLE if SECO not available

Note *addr* must be a pointer to a signed message block.

To switch back to NXP states (Full Field Return), message must be signed by NXP SRK. For OEM States (Partial Field Return), must be signed by OEM SRK.

See the Security Reference Manual (SRM) for more info.

9.19.2.11 sc_misc_seco_build_info()

```
void sc_misc_seco_build_info (
            sc_ipc_t ipc,
            uint32_t * version,
            uint32_t * commit )
```

This function is used to return the SECO FW build info.

Parameters

in	*ipc*	IPC handle
out	*version*	pointer to return build number
out	*commit*	pointer to return commit ID (git SHA-1)

9.19.2.12 sc_misc_seco_chip_info()

```
sc_err_t sc_misc_seco_chip_info (
            sc_ipc_t ipc,
            uint16_t * lc,
            uint16_t * monotonic,
            uint32_t * uid_l,
            uint32_t * uid_h )
```

This function is used to return SECO chip info.

Parameters

in	*ipc*	IPC handle
out	*lc*	pointer to return lifecycle
out	*monotonic*	pointer to return monotonic counter
out	*uid_l*	pointer to return UID (lower 32 bits)
out	*uid_h*	pointer to return UID (upper 32 bits)

9.19.2.13 sc_misc_seco_attest_mode()

```
sc_err_t sc_misc_seco_attest_mode (
            sc_ipc_t ipc,
            uint32_t mode )
```

This function is used to set the attestation mode.

Only the owner of the SC_R_ATTESTATION resource may make this call.

Parameters

in	*ipc*	IPC handle
in	*mode*	mode

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_PARM if *mode* is invalid
- SC_ERR_NOACCESS if SC_R_ATTESTATON not owned by caller
- SC_ERR_UNAVAILABLE if SECO not available

This is used to set the SECO attestation mode. This can be prover or verfier. See the Security Reference Manual (SRM) for more on the suported modes, mode values, and mode behavior.

9.19.2.14 sc_misc_seco_attest()

```
sc_err_t sc_misc_seco_attest (
             sc_ipc_t ipc,
             uint64_t nonce )
```

This function is used to request atestation.

Only the owner of the SC_R_ATTESTATION resource may make this call.

Parameters

in	*ipc*	IPC handle
in	*nonce*	unique value

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_NOACCESS if SC_R_ATTESTATON not owned by caller
- SC_ERR_UNAVAILABLE if SECO not available

This is used to ask SECO to perform an attestation. The result depends on the attestation mode. After this call, the signature can be requested or a verify can be requested.

See the Security Reference Manual (SRM) for more info.

9.19.2.15 sc_misc_seco_get_attest_pkey()

```
sc_err_t sc_misc_seco_get_attest_pkey (
            sc_ipc_t ipc,
            sc_faddr_t addr )
```

This function is used to retrieve the attestation public key.

Mode must be verifier. Only the owner of the SC_R_ATTESTATION resource may make this call.

Parameters

in	*ipc*	IPC handle
in	*addr*	address to write response

Result will be written to *addr*. The *addr* parmater must point to an address SECO can access. It must be 64-bit aligned. There should be 96 bytes of space.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_PARM if *addr* bad or attestation has not been requested
- SC_ERR_NOACCESS if SC_R_ATTESTATON not owned by caller
- SC_ERR_UNAVAILABLE if SECO not available

See the Security Reference Manual (SRM) for more info.

9.19.2.16 sc_misc_seco_get_attest_sign()

```
sc_err_t sc_misc_seco_get_attest_sign (
            sc_ipc_t ipc,
            sc_faddr_t addr )
```

This function is used to retrieve attestation signature and parameters.

Mode must be provider. Only the owner of the SC_R_ATTESTATION resource may make this call.

Parameters

in	*ipc*	IPC handle
in	*addr*	address to write response

Result will be written to *addr*. The *addr* parmater must point to an address SECO can access. It must be 64-bit aligned. There should be 120 bytes of space.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_PARM if *addr* bad or attestation has not been requested
- SC_ERR_NOACCESS if SC_R_ATTESTATON not owned by caller
- SC_ERR_UNAVAILABLE if SECO not available

See the Security Reference Manual (SRM) for more info.

9.19.2.17 sc_misc_seco_attest_verify()

```
sc_err_t sc_misc_seco_attest_verify (
            sc_ipc_t ipc,
            sc_faddr_t addr )
```

This function is used to verify attestation.

Mode must be verifier. Only the owner of the SC_R_ATTESTATION resource may make this call.

Parameters

in	*ipc*	IPC handle
in	*addr*	address of signature

The *addr* parmater must point to an address SECO can access. It must be 64-bit aligned.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_PARM if *addr* bad or attestation has not been requested
- SC_ERR_NOACCESS if SC_R_ATTESTATON not owned by caller
- SC_ERR_UNAVAILABLE if SECO not available
- SC_ERR_FAIL if signature doesn't match

See the Security Reference Manual (SRM) for more info.

9.19.2.18 sc_misc_seco_commit()

```
sc_err_t sc_misc_seco_commit (
            sc_ipc_t ipc,
            uint32_t * info )
```

This function is used to commit into the fuses any new SRK revocation and FW version information that have been found in the primary and secondary containers.

Parameters

in	*ipc*	IPC handle
in,out	*info*	pointer to information type to be committed The return *info* will contain what was actually committed.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_PARM if *info* is invalid
- SC_ERR_UNAVAILABLE if SECO not available

9.19.2.19 sc_misc_debug_out()

```
void sc_misc_debug_out (
            sc_ipc_t ipc,
            uint8_t ch )
```

This function is used output a debug character from the SCU UART.

Parameters

in	*ipc*	IPC handle
in	*ch*	character to output

9.19.2.20 sc_misc_waveform_capture()

```
sc_err_t sc_misc_waveform_capture (
```

```
            sc_ipc_t ipc,
            sc_bool_t enable )
```

This function starts/stops emulation waveform capture.

Parameters

in	*ipc*	IPC handle
in	*enable*	flag to enable/disable capture

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_UNAVAILABLE if not running on emulation

9.19.2.21 sc_misc_build_info()

```
void sc_misc_build_info (
            sc_ipc_t ipc,
            uint32_t * build,
            uint32_t * commit )
```

This function is used to return the SCFW build info.

Parameters

in	*ipc*	IPC handle
out	*build*	pointer to return build number
out	*commit*	pointer to return commit ID (git SHA-1)

9.19.2.22 sc_misc_unique_id()

```
void sc_misc_unique_id (
            sc_ipc_t ipc,
            uint32_t * id_l,
            uint32_t * id_h )
```

This function is used to return the device's unique ID.

Parameters

in	*ipc*	IPC handle
out	*id↩ _l*	pointer to return lower 32-bit of ID [31:0]
out	*id↩ _h*	pointer to return upper 32-bits of ID [63:32]

9.19.2.23 sc_misc_set_ari()

```
sc_err_t sc_misc_set_ari (
            sc_ipc_t ipc,
            sc_rsrc_t resource,
            sc_rsrc_t resource_mst,
            uint16_t ari,
            sc_bool_t enable )
```

This function configures the ARI match value for PCIe/SATA resources.

Parameters

in	*ipc*	IPC handle
in	*resource*	match resource
in	*resource_mst*	PCIe/SATA master to match
in	*ari*	ARI to match
in	*enable*	enable match or not

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the owner or parent of the owner of the resource and translation

For PCIe, the ARI is the 16-bit value that includes the bus number, device number, and function number. For SATA, this value includes the FISType and PM_Port.

9.19.2.24 sc_misc_boot_status()

```
void sc_misc_boot_status (
            sc_ipc_t ipc,
            sc_misc_boot_status_t status )
```

This function reports boot status.

Parameters

in	*ipc*	IPC handle
in	*status*	boot status This is used by SW partitions to report status of boot. This is normally used to report a boot failure.

9.19.2.25 sc_misc_boot_done()

```
sc_err_t sc_misc_boot_done (
             sc_ipc_t ipc,
             sc_rsrc_t cpu )
```

This function tells the SCFW that a CPU is done booting.

Parameters

in	*ipc*	IPC handle
in	*cpu*	CPU that is done booting

This is called by early booting CPUs to report they are done with initialization. After starting early CPUs, the SCFW halts the booting process until they are done. During this time, early CPUs can call the SCFW with lower latency as the SCFW is idle.

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the CPU owner

9.19.2.26 sc_misc_otp_fuse_read()

```
sc_err_t sc_misc_otp_fuse_read (
             sc_ipc_t ipc,
             uint32_t word,
             uint32_t * val )
```

This function reads a given fuse word index.

Parameters

in	*ipc*	IPC handle
in	*word*	fuse word index
out	*val*	fuse read value

Returns

Returns and error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_PARM if word fuse index param out of range or invalid
- SC_ERR_NOACCESS if read operation failed
- SC_ERR_LOCKED if read operation is locked

9.19.2.27 sc_misc_otp_fuse_write()

```
sc_err_t sc_misc_otp_fuse_write (
             sc_ipc_t ipc,
             uint32_t word,
             uint32_t val )
```

This function writes a given fuse word index.

Parameters

in	*ipc*	IPC handle
in	*word*	fuse word index
in	*val*	fuse write value

The command is passed as is to SECO. SECO uses part of the *word* parameter to indicate if the fuse should be locked after programming. See the "Write common fuse" section of the Security Reference Manual (SRM) for more info.

Returns

Returns and error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_PARM if word fuse index param out of range or invalid
- SC_ERR_NOACCESS if write operation failed
- SC_ERR_LOCKED if write operation is locked

9.19.2.28 sc_misc_set_temp()

```
sc_err_t sc_misc_set_temp (
            sc_ipc_t ipc,
            sc_rsrc_t resource,
            sc_misc_temp_t temp,
            int16_t celsius,
            int8_t tenths )
```

This function sets a temp sensor alarm.

Parameters

in	*ipc*	IPC handle
in	*resource*	resource with sensor
in	*temp*	alarm to set
in	*celsius*	whole part of temp to set
in	*tenths*	fractional part of temp to set

Returns

Returns and error code (SC_ERR_NONE = success).

This function will enable the alarm interrupt if the temp requested is not the min/max temp. This enable automatically clears when the alarm occurs and this function has to be called again to re-enable.

Return errors codes:

- SC_ERR_PARM if parameters invalid

9.19.2.29 sc_misc_get_temp()

```
sc_err_t sc_misc_get_temp (
            sc_ipc_t ipc,
            sc_rsrc_t resource,
            sc_misc_temp_t temp,
            int16_t * celsius,
            int8_t * tenths )
```

This function gets a temp sensor value.

Parameters

in	*ipc*	IPC handle
in	*resource*	resource with sensor
in	*temp*	value to get (sensor or alarm)
out	*celsius*	whole part of temp to get
out	*tenths*	fractional part of temp to get

Returns

Returns and error code (SC_ERR_NONE = success).

Return errors codes:

- SC_ERR_PARM if parameters invalid
- SC_ERR_BUSY if temp not ready yet (time delay after power on)

9.19.2.30 sc_misc_get_boot_dev()

```
void sc_misc_get_boot_dev (
             sc_ipc_t ipc,
             sc_rsrc_t * dev )
```

This function returns the boot device.

Parameters

in	*ipc*	IPC handle
out	*dev*	pointer to return boot device

9.19.2.31 sc_misc_get_boot_type()

```
sc_err_t sc_misc_get_boot_type (
             sc_ipc_t ipc,
             sc_misc_bt_t * type )
```

This function returns the boot type.

Parameters

in	*ipc*	IPC handle
out	*type*	pointer to return boot type

Returns

Returns and error code (SC_ERR_NONE = success).

Return errors code:

- SC_ERR_UNAVAILABLE if type not passed by ROM

9.19.2.32 sc_misc_get_button_status()

```
void sc_misc_get_button_status (
            sc_ipc_t ipc,
            sc_bool_t * status )
```

This function returns the current status of the ON/OFF button.

Parameters

in	*ipc*	IPC handle
out	*status*	pointer to return button status

9.19.2.33 sc_misc_rompatch_checksum()

```
sc_err_t sc_misc_rompatch_checksum (
            sc_ipc_t ipc,
            uint32_t * checksum )
```

This function returns the ROM patch checksum.

Parameters

in	*ipc*	IPC handle
out	*checksum*	pointer to return checksum

Returns

Returns and error code (SC_ERR_NONE = success).

9.20 (SVC) Resource Management Service

Module for the Resource Management (RM) service.

Typedefs

- typedef uint8_t sc_rm_pt_t

  This type is used to declare a resource partition.
- typedef uint8_t sc_rm_mr_t

  This type is used to declare a memory region.
- typedef uint8_t sc_rm_did_t

  This type is used to declare a resource domain ID used by the isolation HW.
- typedef uint16_t sc_rm_sid_t

  This type is used to declare an SMMU StreamID.
- typedef uint8_t sc_rm_spa_t

  This type is a used to declare master transaction attributes.
- typedef uint8_t sc_rm_perm_t

  This type is used to declare a resource/memory region access permission.

Defines for type widths

- #define SC_RM_PARTITION_W 5U

  Width of sc_rm_pt_t.
- #define SC_RM_MEMREG_W 6U

  Width of sc_rm_mr_t.
- #define SC_RM_DID_W 4U

  Width of sc_rm_did_t.
- #define SC_RM_SID_W 6U

  Width of sc_rm_sid_t.
- #define SC_RM_SPA_W 2U

  Width of sc_rm_spa_t.
- #define SC_RM_PERM_W 3U

  Width of sc_rm_perm_t.

Defines for ALL parameters

- #define SC_RM_PT_ALL ((sc_rm_pt_t) UINT8_MAX)

  All partitions.
- #define SC_RM_MR_ALL ((sc_rm_mr_t) UINT8_MAX)

  All memory regions.

Defines for sc_rm_spa_t

- #define SC_RM_SPA_PASSTHRU 0U

  Pass through (attribute driven by master)
- #define SC_RM_SPA_PASSSID 1U

  Pass through and output on SID.
- #define SC_RM_SPA_ASSERT 2U

  Assert (force to be secure/privileged)
- #define SC_RM_SPA_NEGATE 3U

  Negate (force to be non-secure/user)

Defines for sc_rm_perm_t

- #define SC_RM_PERM_NONE 0U

  No access.
- #define SC_RM_PERM_SEC_R 1U

  Secure RO.
- #define SC_RM_PERM_SECPRIV_RW 2U

  Secure privilege R/W.
- #define SC_RM_PERM_SEC_RW 3U

  Secure R/W.
- #define SC_RM_PERM_NSPRIV_R 4U

  Secure R/W, non-secure privilege RO.
- #define SC_RM_PERM_NS_R 5U

  Secure R/W, non-secure RO.
- #define SC_RM_PERM_NSPRIV_RW 6U

  Secure R/W, non-secure privilege R/W.
- #define SC_RM_PERM_FULL 7U

  Full access.

Partition Functions

- sc_err_t sc_rm_partition_alloc (sc_ipc_t ipc, sc_rm_pt_t *pt, sc_bool_t secure, sc_bool_t isolated, sc_bool_t restricted, sc_bool_t grant, sc_bool_t coherent)

  This function requests that the SC create a new resource partition.
- sc_err_t sc_rm_set_confidential (sc_ipc_t ipc, sc_rm_pt_t pt, sc_bool_t retro)

  This function makes a partition confidential.
- sc_err_t sc_rm_partition_free (sc_ipc_t ipc, sc_rm_pt_t pt)

  This function frees a partition and assigns all resources to the caller.
- sc_rm_did_t sc_rm_get_did (sc_ipc_t ipc)

  This function returns the DID of a partition.
- sc_err_t sc_rm_partition_static (sc_ipc_t ipc, sc_rm_pt_t pt, sc_rm_did_t did)

  This function forces a partition to use a specific static DID.
- sc_err_t sc_rm_partition_lock (sc_ipc_t ipc, sc_rm_pt_t pt)

  This function locks a partition.
- sc_err_t sc_rm_get_partition (sc_ipc_t ipc, sc_rm_pt_t *pt)

This function gets the partition handle of the caller.

- sc_err_t sc_rm_set_parent (sc_ipc_t ipc, sc_rm_pt_t pt, sc_rm_pt_t pt_parent)

  This function sets a new parent for a partition.

- sc_err_t sc_rm_move_all (sc_ipc_t ipc, sc_rm_pt_t pt_src, sc_rm_pt_t pt_dst, sc_bool_t move_rsrc, sc_bool_t move_pads)

  This function moves all movable resources/pads owned by a source partition to a destination partition.

Resource Functions

- sc_err_t sc_rm_assign_resource (sc_ipc_t ipc, sc_rm_pt_t pt, sc_rsrc_t resource)

  This function assigns ownership of a resource to a partition.

- sc_err_t sc_rm_set_resource_movable (sc_ipc_t ipc, sc_rsrc_t resource_fst, sc_rsrc_t resource_lst, sc_bool_t movable)

  This function flags resources as movable or not.

- sc_err_t sc_rm_set_subsys_rsrc_movable (sc_ipc_t ipc, sc_rsrc_t resource, sc_bool_t movable)

  This function flags all of a subsystem's resources as movable or not.

- sc_err_t sc_rm_set_master_attributes (sc_ipc_t ipc, sc_rsrc_t resource, sc_rm_spa_t sa, sc_rm_spa_t pa, sc_bool_t smmu_bypass)

  This function sets attributes for a resource which is a bus master (i.e.

- sc_err_t sc_rm_set_master_sid (sc_ipc_t ipc, sc_rsrc_t resource, sc_rm_sid_t sid)

  This function sets the StreamID for a resource which is a bus master (i.e.

- sc_err_t sc_rm_set_peripheral_permissions (sc_ipc_t ipc, sc_rsrc_t resource, sc_rm_pt_t pt, sc_rm_perm_t perm)

  This function sets access permissions for a peripheral resource.

- sc_bool_t sc_rm_is_resource_owned (sc_ipc_t ipc, sc_rsrc_t resource)

  This function gets ownership status of a resource.

- sc_bool_t sc_rm_is_resource_master (sc_ipc_t ipc, sc_rsrc_t resource)

  This function is used to test if a resource is a bus master.

- sc_bool_t sc_rm_is_resource_peripheral (sc_ipc_t ipc, sc_rsrc_t resource)

  This function is used to test if a resource is a peripheral.

- sc_err_t sc_rm_get_resource_info (sc_ipc_t ipc, sc_rsrc_t resource, sc_rm_sid_t *sid)

  This function is used to obtain info about a resource.

Memory Region Functions

- sc_err_t sc_rm_memreg_alloc (sc_ipc_t ipc, sc_rm_mr_t *mr, sc_faddr_t addr_start, sc_faddr_t addr_end)

  This function requests that the SC create a new memory region.

- sc_err_t sc_rm_memreg_split (sc_ipc_t ipc, sc_rm_mr_t mr, sc_rm_mr_t *mr_ret, sc_faddr_t addr_start, sc_faddr_t addr_end)

  This function requests that the SC split a memory region.

- sc_err_t sc_rm_memreg_free (sc_ipc_t ipc, sc_rm_mr_t mr)

  This function frees a memory region.

- sc_err_t sc_rm_find_memreg (sc_ipc_t ipc, sc_rm_mr_t *mr, sc_faddr_t addr_start, sc_faddr_t addr_end)

  Internal SC function to find a memory region.

- sc_err_t sc_rm_assign_memreg (sc_ipc_t ipc, sc_rm_pt_t pt, sc_rm_mr_t mr)

  This function assigns ownership of a memory region.

- sc_err_t sc_rm_set_memreg_permissions (sc_ipc_t ipc, sc_rm_mr_t mr, sc_rm_pt_t pt, sc_rm_perm_t perm)

  This function sets access permissions for a memory region.
- sc_bool_t sc_rm_is_memreg_owned (sc_ipc_t ipc, sc_rm_mr_t mr)

  This function gets ownership status of a memory region.
- sc_err_t sc_rm_get_memreg_info (sc_ipc_t ipc, sc_rm_mr_t mr, sc_faddr_t *addr_start, sc_faddr_t *addr_end)

  This function is used to obtain info about a memory region.

Pad Functions

- sc_err_t sc_rm_assign_pad (sc_ipc_t ipc, sc_rm_pt_t pt, sc_pad_t pad)

  This function assigns ownership of a pad to a partition.
- sc_err_t sc_rm_set_pad_movable (sc_ipc_t ipc, sc_pad_t pad_fst, sc_pad_t pad_lst, sc_bool_t movable)

  This function flags pads as movable or not.
- sc_bool_t sc_rm_is_pad_owned (sc_ipc_t ipc, sc_pad_t pad)

  This function gets ownership status of a pad.

Debug Functions

- void sc_rm_dump (sc_ipc_t ipc)

  This function dumps the RM state for debug.

9.20.1 Detailed Description

Module for the Resource Management (RM) service.

The following SCFW resource manager (RM) code is an example of how to create a partition for an M4 core and its resources. This could be run from another core, an SCD, or be embedded into board.c.

The ipc parameter most functions take is a handle to the IPC channel opened to communicate to the SC. It is implementation defined. Most API ports include an sc_ipc_open() and sc_ipc_close() function to manage this. The sc_ipc_open() takes an argument to identify the communication channel (usually the MU address) and returns the IPC handle that all API calls should then use.

Note all this configuration can be done with the M4 subsystem powered off. It will be loaded when the M4 is powered on.

```
1 sc_rm_pt_t pt_m4_0;
2 sc_rm_mr_t mr_ddr1, mr_ddr2, mr_m4_0;
3
4 //sc_rm_dump(ipc);
5
6 /* Mark all resources as not movable */
7 sc_rm_set_resource_movable(ipc, SC_R_ALL,       SC_R_ALL, SC_FALSE);
8 sc_rm_set_pad_movable(ipc, SC_P_ALL, SC_P_ALL,       SC_FALSE);
9
10 /* Allocate M4_0 partition */
11 sc_rm_partition_alloc(ipc, &pt_m4_0, SC_FALSE,       SC_TRUE, SC_FALSE, SC_TRUE,
12     SC_FALSE);
13
14 /* Mark all M4_0 subsystem resources as movable */
15 sc_rm_set_subsys_rsrc_movable(ipc, SC_R_M4_0_PID0,       SC_TRUE);
16 sc_rm_set_pad_movable(ipc,SC_P_ADC_IN3,SC_P_ADC_IN2,       SC_TRUE);
```

```
17
18 /* Keep some resources in the parent partition */
19 sc_rm_set_resource_movable(ipc, SC_R_M4_0_PID1, SC_R_M4_0_PID4,
20     SC_FALSE);
21 sc_rm_set_resource_movable(ipc, SC_R_M4_0_MU_0A0, SC_R_M4_0_MU_0A3,
22     SC_FALSE);
23
24 /* Move some resource not in the M4_0 subsystem */
25 sc_rm_set_resource_movable(ipc, SC_R_IRQSTR_M4_0, SC_R_IRQSTR_M4_0,
26     SC_TRUE);
27 sc_rm_set_resource_movable(ipc, SC_R_M4_1_MU_0A0, SC_R_M4_1_MU_0A0,
28     SC_TRUE);
29
30 /* Move everything flagged as movable */
31 sc_rm_move_all(ipc, ipc, pt_m4_0, SC_TRUE, SC_TRUE);
32
33 /* Allow all to access the SEMA42 */
34 sc_rm_set_peripheral_permissions(pt_m4_0, SC_R_M4_0_SEMA42,
35     SC_RM_PT_ALL, SC_RM_PERM_FULL);
36
37 /* Move M4 0 TCM */
38 sc_rm_find_memreg(ipc, &mr_m4_0, 0x034FE0000, 0x034FE0000);
39 sc_rm_assign_memreg(ipc, pt_m4_0, mr_m4_0);
40
41 /* Split DDR space, assign 0x88000000-0x8FFFFFFF to CM4 */
42 sc_rm_find_memreg(ipc, &mr_ddr1, 0x080000000, 0x080000000);
43 sc_rm_memreg_split(ipc, mr_ddr1, &mr_ddr2, 0x090000000, 0x0FFFFFFFF);
44
45 /* Reserve DDR for M4_0 */
46 sc_rm_memreg_split(ipc, mr_ddr1, &mr_m4_0, 0x088000000, 0x08FFFFFFF);
47 sc_rm_assign_memreg(ipc, pt_m4_0, mr_m4_0);
48
49 //sc_rm_dump(ipc);
```

First, variables are declared to hold return partition and memory region handles.

```
sc_rm_pt_t pt_m4_0;
sc_rm_mr_t mr_ddr1, mr_ddr2, mr_m4_0;
```

Optionally, call sc_rm_dump() to dump the state of the RM to the SCFW debug UART.

```
//sc_rm_dump(ipc);
```

Mark resources and pins as movable or not movable to the new partition. By default, all resources are marked as movable. Marking all as movable or not movable first depends on how many resources are to be moved and which is the most efficient. Marking does not move the resource yet. Note, that it is also possible to assign resources individually using sc_rm_assign_resource().

```
/* Mark all resources as not movable */
sc_rm_set_resource_movable(ipc, SC_R_ALL,       SC_R_ALL, SC_FALSE);
sc_rm_set_pad_movable(ipc, SC_P_ALL, SC_P_ALL,       SC_FALSE);
```

The sc_rm_partition_alloc() function call requests that the SC create a new partition to contain the M4 system. This function does not access the hardware at all. It allocates a new partition and returns a partition handle (pt_m4_0). The partition is marked non-secure as secure=SC_FALSE. Marking as non-secure prevents subsequent functions from configuring masters in this partition to assert the TZPROT signal.

```
/* Allocate M4_0 partition */
sc_rm_partition_alloc(ipc, &pt_m4_0, SC_FALSE,       SC_TRUE, SC_FALSE, SC_TRUE,
    SC_FALSE);
```

Now mark some resources as movable. sc_rm_set_subsys_rsrc_movable() can be used to mark all resources in a HW subsystem. sc_rm_set_pad_movable() is used to mark some pads (i.e. pins) as movable.

```
/* Mark all M4_0 subsystem resources as movable */
sc_rm_set_subsys_rsrc_movable(ipc, SC_R_M4_0_PID0,       SC_TRUE);
sc_rm_set_pad_movable(ipc,SC_P_ADC_IN3,SC_P_ADC_IN2,       SC_TRUE);
```

Then mark some resources in the M4_0 subsystem (all marked movable above) as not movable using

sc_rm_set_resource_movable() In this case the process IDs used to access memory owned by other partitions as well as the MUs used for others to communicate with the M4 need to be left with the parent partition.

```
/* Keep some resources in the parent partition */
sc_rm_set_resource_movable(ipc, SC_R_M4_0_PID1, SC_R_M4_0_PID4,
    SC_FALSE);
sc_rm_set_resource_movable(ipc, SC_R_M4_0_MU_0A0, SC_R_M4_0_MU_0A3,
    SC_FALSE);
```

Move some resources in other subsystems. The new partition will require access to the IRQ Steer module which routes interrupts to this M4's NVIC. In this example, it also needs access to one of the M4_1 MUs.

```
/* Move some resource not in the M4_0 subsystem */
sc_rm_set_resource_movable(ipc, SC_R_IRQSTR_M4_0, SC_R_IRQSTR_M4_0,
```

Now assign (i.e. move) everything marked as movable. At this point, all these resources are in the new partition and HW will enforce isolation.

```
/* Move everything flagged as movable */
sc_rm_move_all(ipc, ipc, pt_m4_0, SC_TRUE, SC_TRUE);
```

Allow others to access some of the new partitions resources. In this case, the SEMA42 IP works by allowing multiple CPUs to access and acquire the semaphore.

```
/* Allow all to access the SEMA42 */
sc_rm_set_peripheral_permissions(pt_m4_0, SC_R_M4_0_SEMA42,
    SC_RM_PT_ALL, SC_RM_PERM_FULL);
```

Now assign the M4_0 TCM to the M4 partition. Note the M4 can always access its TCM. This action prevents the parent (current owner of the M4 TCM) from accessing. This should only be done after code for the M4 has been loaded into the TCM. Code loading will require the M4 subsystem already be powered on.

```
/* Move M4 0 TCM */
sc_rm_find_memreg(ipc, &mr_m4_0, 0x034FE0000, 0x034FE0000);
sc_rm_assign_memreg(ipc, pt_m4_0, mr_m4_0);
```

Next is to carve out some DDR for the M4. In this case, the memory is in the middle of the DDR so the DDR has to be split into three regions. First is to split off the end portion and keep this with the parent. Next is then to split off the end of the remaining part and assign this to the M4.

```
/* Split DDR space, assign 0x88000000-0x8FFFFFFF to CM4 */
sc_rm_find_memreg(ipc, &mr_ddr1, 0x080000000, 0x080000000);
sc_rm_memreg_split(ipc, mr_ddr1, &mr_ddr2, 0x090000000, 0x0FFFFFFFF);
/* Reserve DDR for M4_0 */
sc_rm_memreg_split(ipc, mr_ddr1, &mr_m4_0, 0x088000000, 0x08FFFFFFF);
sc_rm_assign_memreg(ipc, pt_m4_0, mr_m4_0);
```

Optionally, call sc_rm_dump() to dump the state of the RM to the SCFW debug UART.

```
//sc_rm_dump(ipc);
```

At this point, the M4 can be powered on (if not already) and the M4 can be started using sc_pm_boot(). *Do NOT start the CPU using sc_pm_cpu_start() as that function is for starting a secondary CPU in the calling core's partition.* In this case, the core is in another partition that needs to be booted.

Refer to the SoC-specific RESOURCES for a list of resources.

9.20.2 Typedef Documentation

9.20.2.1 sc_rm_perm_t

```
typedef uint8_t sc_rm_perm_t
```

This type is used to declare a resource/memory region access permission.

Refer to the XRDC2 Block Guide for more information.

9.20.3 Function Documentation

9.20.3.1 sc_rm_partition_alloc()

```
sc_err_t sc_rm_partition_alloc (
             sc_ipc_t ipc,
             sc_rm_pt_t * pt,
             sc_bool_t secure,
             sc_bool_t isolated,
             sc_bool_t restricted,
             sc_bool_t grant,
             sc_bool_t coherent )
```

This function requests that the SC create a new resource partition.

Parameters

in	*ipc*	IPC handle
out	*pt*	return handle for partition; used for subsequent function calls associated with this partition
in	*secure*	boolean indicating if this partition should be secure; only valid if caller is secure
in	*isolated*	boolean indicating if this partition should be HW isolated via XRDC; set SC_TRUE if new DID is desired
in	*restricted*	boolean indicating if this partition should be restricted; set SC_TRUE if masters in this partition cannot create new partitions
in	*grant*	boolean indicating if this partition should always grant access and control to the parent
in	*coherent*	boolean indicating if this partition is coherent; set SC_TRUE if only this partition will contain both AP clusters and they will be coherent via the CCI

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_NOACCESS if caller's partition is restricted,

- SC_ERR_PARM if caller's partition is not secure but a new secure partition is requested,
- SC_ERR_LOCKED if caller's partition is locked,
- SC_ERR_UNAVAILABLE if partition table is full (no more allocation space)

Marking as non-secure prevents subsequent functions from configuring masters in this partition to assert the secure signal. If restricted then the new partition is limited in what functions it can call, especially those associated with managing partitions.

The grant option is usually used to isolate a bus master's traffic to specific memory without isolating the peripheral interface of the master or the API controls of that master.

9.20.3.2 sc_rm_set_confidential()

```
sc_err_t sc_rm_set_confidential (
             sc_ipc_t ipc,
             sc_rm_pt_t pt,
             sc_bool_t retro )
```

This function makes a partition confidential.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition that is granting
in	*retro*	retroactive

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if *pt* out of range,
- SC_ERR_NOACCESS if caller's not allowed to change *pt*
- SC_ERR_LOCKED if partition *pt* is locked

Call to make a partition confidential. Confidential means only this partition should be able to grant access permissions to this partition.

If retroactive, then all resources owned by other partitions will have access rights for this partition removed, even if locked.

9.20.3.3 sc_rm_partition_free()

```
sc_err_t sc_rm_partition_free (
             sc_ipc_t ipc,
             sc_rm_pt_t pt )
```

This function frees a partition and assigns all resources to the caller.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition to free

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_NOACCESS if caller's partition is restricted,
- SC_PARM if *pt* out of range or invalid,
- SC_ERR_NOACCESS if *pt* is the SC partition,
- SC_ERR_NOACCESS if caller's partition is not the parent of *pt*,
- SC_ERR_LOCKED if *pt* or caller's partition is locked

All resources, memory regions, and pads are assigned to the caller/parent. The partition watchdog is disabled (even if locked). DID is freed.

9.20.3.4 sc_rm_get_did()

```
sc_rm_did_t sc_rm_get_did (
             sc_ipc_t ipc )
```

This function returns the DID of a partition.

Parameters

in	*ipc*	IPC handle

Returns

Returns the domain ID (DID) of the caller's partition.

The DID is a SoC-specific internal ID used by the HW resource protection mechanism. It is only required by clients when using the SEMA42 module as the DID is sometimes connected to the master ID.

9.20.3.5 sc_rm_partition_static()

```
sc_err_t sc_rm_partition_static (
             sc_ipc_t ipc,
```

```
sc_rm_pt_t pt,
sc_rm_did_t did )
```

This function forces a partition to use a specific static DID.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition to assign *did*
in	*did*	static DID to assign

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_NOACCESS if caller's partition is restricted,
- SC_PARM if *pt* or *did* out of range,
- SC_ERR_NOACCESS if caller's partition is not the parent of *pt*,
- SC_ERR_LOCKED if *pt* is locked

Assumes no assigned resources or memory regions yet! The number of static DID is fixed by the SC at boot.

9.20.3.6 sc_rm_partition_lock()

```
sc_err_t sc_rm_partition_lock (
             sc_ipc_t ipc,
             sc_rm_pt_t pt )
```

This function locks a partition.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition to lock

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if *pt* out of range,
- SC_ERR_NOACCESS if caller's partition is not the parent of *pt*

If a partition is locked it cannot be freed, have resources/pads assigned to/from it, memory regions created/assigned, DID changed, or parent changed.

9.20.3.7 sc_rm_get_partition()

```
sc_err_t sc_rm_get_partition (
             sc_ipc_t ipc,
             sc_rm_pt_t * pt )
```

This function gets the partition handle of the caller.

Parameters

in	*ipc*	IPC handle
out	*pt*	return handle for caller's partition

Returns

Returns an error code (SC_ERR_NONE = success).

9.20.3.8 sc_rm_set_parent()

```
sc_err_t sc_rm_set_parent (
             sc_ipc_t ipc,
             sc_rm_pt_t pt,
             sc_rm_pt_t pt_parent )
```

This function sets a new parent for a partition.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition for which parent is to be changed
in	*pt_parent*	handle of partition to set as parent

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_NOACCESS if caller's partition is restricted,
- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the parent of *pt*,
- SC_ERR_LOCKED if either partition is locked

9.20.3.9 sc_rm_move_all()

```
sc_err_t sc_rm_move_all (
            sc_ipc_t ipc,
            sc_rm_pt_t pt_src,
            sc_rm_pt_t pt_dst,
            sc_bool_t move_rsrc,
            sc_bool_t move_pads )
```

This function moves all movable resources/pads owned by a source partition to a destination partition.

It can be used to more quickly set up a new partition if a majority of the caller's resources are to be moved to a new partition.

Parameters

in	*ipc*	IPC handle
in	*pt_src*	handle of partition from which resources should be moved from
in	*pt_dst*	handle of partition to which resources should be moved to
in	*move_rsrc*	boolean to indicate if resources should be moved
in	*move_pads*	boolean to indicate if pads should be moved

Returns

Returns an error code (SC_ERR_NONE = success).

By default, all resources are movable. This can be changed using the sc_rm_set_resource_movable() function. Note all masters defaulted to SMMU bypass.

Return errors:

- SC_ERR_NOACCESS if caller's partition is restricted,
- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not *pt_src* or the parent of *pt_src*,
- SC_ERR_LOCKED if either partition is locked

9.20.3.10 sc_rm_assign_resource()

```
sc_err_t sc_rm_assign_resource (
            sc_ipc_t ipc,
            sc_rm_pt_t pt,
            sc_rsrc_t resource )
```

This function assigns ownership of a resource to a partition.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition to which resource should be assigned
in	*resource*	resource to assign

Returns

Returns an error code (SC_ERR_NONE = success).

This action resets the resource's master and peripheral attributes. Privilege attribute will be PASSTHRU, security attribute will be ASSERT if the partition si secure and NEGATE if it is not, and masters will defaulted to SMMU bypass. Access permissions will reset to SEC_RW for the owning partition only for secure partitions, FULL for non-secure. DEfault is no access by other partitions.

Return errors:

- SC_ERR_NOACCESS if caller's partition is restricted,
- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the resource owner or parent of the owner,
- SC_ERR_LOCKED if the owning partition or *pt* is locked

9.20.3.11 sc_rm_set_resource_movable()

```
sc_err_t sc_rm_set_resource_movable (
             sc_ipc_t ipc,
             sc_rsrc_t resource_fst,
             sc_rsrc_t resource_lst,
             sc_bool_t movable )
```

This function flags resources as movable or not.

Parameters

in	*ipc*	IPC handle
in	*resource_fst*	first resource for which flag should be set
in	*resource_lst*	last resource for which flag should be set
in	*movable*	movable flag (SC_TRUE is movable)

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if resources are out of range,
- SC_ERR_NOACCESS if caller's partition is not a parent of a resource owner,
- SC_ERR_LOCKED if the owning partition is locked

This function is used to determine the set of resources that will be moved using the sc_rm_move_all() function. All resources are movable by default so this function is normally used to prevent a set of resources from moving.

9.20.3.12 sc_rm_set_subsys_rsrc_movable()

```
sc_err_t sc_rm_set_subsys_rsrc_movable (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_bool_t movable )
```

This function flags all of a subsystem's resources as movable or not.

Parameters

in	*ipc*	IPC handle
in	*resource*	resource to use to identify subsystem
in	*movable*	movable flag (SC_TRUE is movable)

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if a function argument is out of range

Note *resource* is used to find the associated subsystem. Only resources owned by the caller are set.

9.20.3.13 sc_rm_set_master_attributes()

```
sc_err_t sc_rm_set_master_attributes (
             sc_ipc_t ipc,
             sc_rsrc_t resource,
             sc_rm_spa_t sa,
             sc_rm_spa_t pa,
             sc_bool_t smmu_bypass )
```

This function sets attributes for a resource which is a bus master (i.e.

capable of DMA).

Parameters

in	*ipc*	IPC handle
in	*resource*	master resource for which attributes should apply
in	*sa*	security attribute
in	*pa*	privilege attribute
in	*smmu_bypass*	SMMU bypass mode

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_NOACCESS if caller's partition is restricted,
- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not a parent of the resource owner,
- SC_ERR_LOCKED if the owning partition is locked

This function configures how the HW isolation will see bus transactions from the specified master. Note the security attribute will only be changed if the caller's partition is secure.

9.20.3.14 sc_rm_set_master_sid()

```
sc_err_t sc_rm_set_master_sid (
            sc_ipc_t ipc,
            sc_rsrc_t resource,
            sc_rm_sid_t sid )
```

This function sets the StreamID for a resource which is a bus master (i.e.

capable of DMA).

Parameters

in	*ipc*	IPC handle
in	*resource*	master resource for which attributes should apply
in	*sid*	StreamID

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_NOACCESS if caller's partition is restricted,
- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the resource owner or parent of the owner,
- SC_ERR_LOCKED if the owning partition is locked

This function configures the SID attribute associated with all bus transactions from this master. Note 0 is not a valid SID as it is reserved to indicate bypass.

9.20.3.15 sc_rm_set_peripheral_permissions()

```
sc_err_t sc_rm_set_peripheral_permissions (
            sc_ipc_t ipc,
            sc_rsrc_t resource,
            sc_rm_pt_t pt,
            sc_rm_perm_t perm )
```

This function sets access permissions for a peripheral resource.

Parameters

in	*ipc*	IPC handle
in	*resource*	peripheral resource for which permissions should apply
in	*pt*	handle of partition *perm* should by applied for
in	*perm*	permissions to apply to *resource* for *pt*

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the resource owner or parent of the owner,
- SC_ERR_LOCKED if the owning partition is locked
- SC_ERR_LOCKED if the *pt* is confidential and the caller isn't *pt*

This function configures how the HW isolation will restrict access to a peripheral based on the attributes of a transaction from bus master.

9.20.3.16 sc_rm_is_resource_owned()

```
sc_bool_t sc_rm_is_resource_owned (
            sc_ipc_t ipc,
            sc_rsrc_t resource )
```

This function gets ownership status of a resource.

Parameters

in	*ipc*	IPC handle
in	*resource*	resource to check

Returns

Returns a boolean (SC_TRUE if caller's partition owns the resource).

If *resource* is out of range then SC_FALSE is returned.

9.20.3.17 sc_rm_is_resource_master()

```
sc_bool_t sc_rm_is_resource_master (
            sc_ipc_t ipc,
            sc_rsrc_t resource )
```

This function is used to test if a resource is a bus master.

Parameters

in	*ipc*	IPC handle
in	*resource*	resource to check

Returns

Returns a boolean (SC_TRUE if the resource is a bus master).

If *resource* is out of range then SC_FALSE is returned.

9.20.3.18 sc_rm_is_resource_peripheral()

```
sc_bool_t sc_rm_is_resource_peripheral (
            sc_ipc_t ipc,
            sc_rsrc_t resource )
```

This function is used to test if a resource is a peripheral.

Parameters

in	*ipc*	IPC handle
in	*resource*	resource to check

Returns

Returns a boolean (SC_TRUE if the resource is a peripheral).

If *resource* is out of range then SC_FALSE is returned.

9.20.3.19 sc_rm_get_resource_info()

```
sc_err_t sc_rm_get_resource_info (
            sc_ipc_t ipc,
            sc_rsrc_t resource,
            sc_rm_sid_t * sid )
```

This function is used to obtain info about a resource.

Parameters

in	*ipc*	IPC handle
in	*resource*	resource to inquire about
out	*sid*	pointer to return StreamID

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if *resource* is out of range

9.20.3.20 sc_rm_memreg_alloc()

```
sc_err_t sc_rm_memreg_alloc (
            sc_ipc_t ipc,
            sc_rm_mr_t * mr,
            sc_faddr_t addr_start,
            sc_faddr_t addr_end )
```

This function requests that the SC create a new memory region.

Parameters

in	*ipc*	IPC handle
out	*mr*	return handle for region; used for subsequent function calls associated with this region
in	*addr_start*	start address of region (physical)
in	*addr_end*	end address of region (physical)

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if the new memory region is misaligned,
- SC_ERR_LOCKED if caller's partition is locked,
- SC_ERR_PARM if the new memory region spans multiple existing regions,
- SC_ERR_NOACCESS if caller's partition does not own the memory containing the new region,
- SC_ERR_UNAVAILABLE if memory region table is full (no more allocation space)

The area covered by the memory region must currently be owned by the caller. By default, the new region will have access permission set to allow the caller to access.

9.20.3.21 sc_rm_memreg_split()

```
sc_err_t sc_rm_memreg_split (
            sc_ipc_t ipc,
            sc_rm_mr_t mr,
            sc_rm_mr_t * mr_ret,
            sc_faddr_t addr_start,
            sc_faddr_t addr_end )
```

This function requests that the SC split a memory region.

Parameters

in	*ipc*	IPC handle
in	*mr*	handle of memory region to split
out	*mr_ret*	return handle for new region; used for subsequent function calls associated with this region
in	*addr_start*	start address of region (physical)
in	*addr_end*	end address of region (physical)

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if the new memory region is not start/end part of mr,
- SC_ERR_LOCKED if caller's partition is locked,
- SC_ERR_PARM if the new memory region spans multiple existing regions,

- SC_ERR_NOACCESS if caller's partition does not own the memory containing the new region,
- SC_ERR_UNAVAILABLE if memory region table is full (no more allocation space)

Note the new region must start or end on the split region.

9.20.3.22 sc_rm_memreg_free()

```
sc_err_t sc_rm_memreg_free (
             sc_ipc_t ipc,
             sc_rm_mr_t mr )
```

This function frees a memory region.

Parameters

in	*ipc*	IPC handle
in	*mr*	handle of memory region to free

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if *mr* out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not a parent of *mr*,
- SC_ERR_LOCKED if the owning partition of *mr* is locked

9.20.3.23 sc_rm_find_memreg()

```
sc_err_t sc_rm_find_memreg (
             sc_ipc_t ipc,
             sc_rm_mr_t * mr,
             sc_faddr_t addr_start,
             sc_faddr_t addr_end )
```

Internal SC function to find a memory region.

See also

sc_rm_find_memreg().

This function finds a memory region.

Parameters

in	*ipc*	IPC handle
out	*mr*	return handle for region; used for subsequent function calls associated with this region
in	*addr_start*	start address of region to search for
in	*addr_end*	end address of region to search for

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_NOTFOUND if region not found,

Searches only for regions owned by the caller. Finds first region containing the range specified.

9.20.3.24 sc_rm_assign_memreg()

```
sc_err_t sc_rm_assign_memreg (
             sc_ipc_t ipc,
             sc_rm_pt_t pt,
             sc_rm_mr_t mr )
```

This function assigns ownership of a memory region.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition to which memory region should be assigned
in	*mr*	handle of memory region to assign

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the *mr* owner or parent of the owner,
- SC_ERR_LOCKED if the owning partition or *pt* is locked

9.20.3.25 sc_rm_set_memreg_permissions()

```
sc_err_t sc_rm_set_memreg_permissions (
            sc_ipc_t ipc,
            sc_rm_mr_t mr,
            sc_rm_pt_t pt,
            sc_rm_perm_t perm )
```

This function sets access permissions for a memory region.

Parameters

in	*ipc*	IPC handle
in	*mr*	handle of memory region for which permissions should apply
in	*pt*	handle of partition *perm* should by applied for
in	*perm*	permissions to apply to *mr* for *pt*

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the region owner or parent of the owner,
- SC_ERR_LOCKED if the owning partition is locked
- SC_ERR_LOCKED if the *pt* is confidential and the caller isn't *pt*

This function configures how the HW isolation will restrict access to a memory region based on the attributes of a transaction from bus master.

9.20.3.26 sc_rm_is_memreg_owned()

```
sc_bool_t sc_rm_is_memreg_owned (
            sc_ipc_t ipc,
            sc_rm_mr_t mr )
```

This function gets ownership status of a memory region.

Parameters

in	*ipc*	IPC handle
in	*mr*	handle of memory region to check

Returns

Returns a boolean (SC_TRUE if caller's partition owns the memory region).

If *mr* is out of range then SC_FALSE is returned.

9.20.3.27 sc_rm_get_memreg_info()

```
sc_err_t sc_rm_get_memreg_info (
            sc_ipc_t ipc,
            sc_rm_mr_t mr,
            sc_faddr_t * addr_start,
            sc_faddr_t * addr_end )
```

This function is used to obtain info about a memory region.

Parameters

in	*ipc*	IPC handle
in	*mr*	handle of memory region to inquire about
out	*addr_start*	pointer to return start address
out	*addr_end*	pointer to return end address

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if *mr* is out of range

9.20.3.28 sc_rm_assign_pad()

```
sc_err_t sc_rm_assign_pad (
            sc_ipc_t ipc,
            sc_rm_pt_t pt,
            sc_pad_t pad )
```

This function assigns ownership of a pad to a partition.

Parameters

in	*ipc*	IPC handle
in	*pt*	handle of partition to which pad should be assigned
in	*pad*	pad to assign

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_NOACCESS if caller's partition is restricted,
- SC_PARM if arguments out of range or invalid,
- SC_ERR_NOACCESS if caller's partition is not the pad owner or parent of the owner,
- SC_ERR_LOCKED if the owning partition or *pt* is locked

9.20.3.29 sc_rm_set_pad_movable()

```
sc_err_t sc_rm_set_pad_movable (
            sc_ipc_t ipc,
            sc_pad_t pad_fst,
            sc_pad_t pad_lst,
            sc_bool_t movable )
```

This function flags pads as movable or not.

Parameters

in	*ipc*	IPC handle
in	*pad_fst*	first pad for which flag should be set
in	*pad_lst*	last pad for which flag should be set
in	*movable*	movable flag (SC_TRUE is movable)

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_PARM if pads are out of range,
- SC_ERR_NOACCESS if caller's partition is not a parent of a pad owner,
- SC_ERR_LOCKED if the owning partition is locked

This function is used to determine the set of pads that will be moved using the sc_rm_move_all() function. All pads are movable by default so this function is normally used to prevent a set of pads from moving.

9.20.3.30 sc_rm_is_pad_owned()

```
sc_bool_t sc_rm_is_pad_owned (
            sc_ipc_t ipc,
            sc_pad_t pad )
```

This function gets ownership status of a pad.

Parameters

in	*ipc*	IPC handle
in	*pad*	pad to check

Returns

Returns a boolean (SC_TRUE if caller's partition owns the pad).

If *pad* is out of range then SC_FALSE is returned.

9.20.3.31 sc_rm_dump()

```
void sc_rm_dump (
            sc_ipc_t ipc )
```

This function dumps the RM state for debug.

Parameters

in	*ipc*	IPC handle

9.21 (BRD) Board Interface

Module for the Board interface.

Macros

- #define **BOARD_PARM_RTN_NOT_USED** 0U
- #define **BOARD_PARM_RTN_USED** 1U
- #define **BOARD_PARM_RTN_EXTERNAL** 2U
- #define **BOARD_PARM_RTN_INTERNAL** 3U
- #define BOARD_PARM_KS1_RETENTION_DISABLE 0U

  Disable retention during KS1.
- #define BOARD_PARM_KS1_RETENTION_ENABLE 1U

  Enable retention during KS1.
- #define BOARD_PARM_KS1_ONOFF_WAKE_DISABLE 0U

  Disable ONOFF wakeup during KS1.
- #define BOARD_PARM_KS1_ONOFF_WAKE_ENABLE 1U

  Enable ONOFF wakeup during KS1.

Typedefs

- typedef uint32_t board_parm_rtn_t

  Board config parameter returns.

Enumerations

- enum board_parm_t { **BOARD_PARM_PCIE_PLL** = 0, BOARD_PARM_KS1_RESUME_USEC = 1, BOARD_PARM_KS1_RETENTION = 2, BOARD_PARM_KS1_ONOFF_WAKE = 3 }

  Board config parameter types.
- enum board_cpu_rst_ev_t { **BOARD_CPU_RESET_SELF** = 0, **BOARD_CPU_RESET_WDOG** = 1, **BOARD_CPU_RESET_LOCKUP** = 2, **BOARD_CPU_RESET_MEM_ERR** = 3 }

  Board reset event types for CPUs.
- enum board_ddr_action_t {
  BOARD_DDR_COLD_INIT = 0, BOARD_DDR_PERIODIC = 1, BOARD_DDR_SR_DRC_ON_ENTER = 2, BOARD_DDR_SR_DRC_ON_EXIT = 3,
  BOARD_DDR_SR_DRC_OFF_ENTER = 4, BOARD_DDR_SR_DRC_OFF_EXIT = 5 }

  DDR actions (power state transitions, etc.)

Variables

- const sc_rm_idx_t board_num_rsrc

  External variable for accessing the number of board resources.
- const sc_rsrc_map_t board_rsrc_map [BRD_NUM_RSRC_BRD]

  External variable for accessing the board resource map.
- const uint32_t board_ddr_period_ms

  External variable for specing DDR periodic training.

Macros for DCD processing

- #define **DATA4**(A, V) ∗((volatile uint32_t∗)(A)) = U32(V)
- #define **SET_BIT4**(A, V) ∗((volatile uint32_t∗)(A)) |= U32(V)
- #define **CLR_BIT4**(A, V) ∗((volatile uint32_t∗)(A)) &= ∼(U32(V))
- #define **CHECK_BITS_SET4**(A, M)
- #define **CHECK_BITS_CLR4**(A, M)
- #define **CHECK_ANY_BIT_SET4**(A, M)
- #define **CHECK_ANY_BIT_CLR4**(A, M)

Macro for debug of board calls

- #define **BRD_ERR**(X)

Initialization Functions

- void board_init (uint8_t phase)

  This function initalizes the board.
- LPUART_Type ∗ board_get_debug_uart (uint8_t ∗inst, uint32_t ∗baud)

  This function returns the debug UART info.
- void board_config_debug_uart (sc_bool_t early_phase)

  This function initalizes the debug UART.
- void board_config_sc (sc_rm_pt_t pt_sc)

  This function configures SCU resources.
- board_parm_rtn_t board_parameter (board_parm_t parm)

  This function returns board configuration info.
- sc_bool_t board_rsrc_avail (sc_rsrc_t rsrc)

  This function returns resource availability info.
- sc_err_t board_init_ddr (sc_bool_t early, sc_bool_t ddr_initialized)

  This function initalizes DDR.
- sc_err_t board_ddr_config (bool rom_caller, board_ddr_action_t action)

  This function cinfigures the DDR.
- sc_err_t board_system_config (sc_bool_t early, sc_rm_pt_t pt_boot)

  This function allows the board file to do SCFW configuration.
- sc_bool_t board_early_cpu (sc_rsrc_t cpu)

  This function returns SC_TRUE for early CPUs.

Power Functions

- void board_set_power_mode (sc_sub_t ss, uint8_t pd, sc_pm_power_mode_t from_mode, sc_pm_power_mode_t to_mode)

  This function transitions the power state for an external board- level supply which goes to the i.MX8.
- sc_err_t board_set_voltage (sc_sub_t ss, uint32_t new_volt, sc_bool_t wait)

  This function sets the voltage for a PMIC controlled SS.
- sc_err_t board_trans_resource_power (sc_rm_idx_t idx, sc_rm_idx_t rsrc_idx, sc_pm_power_mode_t from_↩ mode, sc_pm_power_mode_t to_mode)

  This function transitions the power state for an external board- level supply which goes to a board component.

Misc Functions

- sc_err_t board_power (sc_pm_power_mode_t mode)

  This function is used to set the board power.
- sc_err_t board_reset (sc_pm_reset_type_t type, sc_pm_reset_reason_t reason)

  This function is used to reset the system.
- void board_cpu_reset (sc_rsrc_t resource, board_cpu_rst_ev_t reset_event)

  This function is called when a CPU encounters a reset event.
- void board_panic (sc_dsc_t dsc)

  This function is called when a DSC reports a panic temp alarm.
- void board_fault (sc_bool_t restarted)

  This function is called when a fault is detected or the SCFW returns from main().
- void board_security_violation (void)

  This function is called when a security violation is reported by the SECO or SNVS.
- sc_bool_t board_get_button_status (void)

  This function is used to return the current status of the ON/OFF button.
- sc_err_t board_set_control (sc_rsrc_t resource, sc_rm_idx_t idx, sc_rm_idx_t rsrc_idx, uint32_t ctrl, uint32_t val)

  This function sets a miscellaneous control value.
- sc_err_t board_get_control (sc_rsrc_t resource, sc_rm_idx_t idx, sc_rm_idx_t rsrc_idx, uint32_t ctrl, uint32_t *val)

  This function gets a miscellaneous control value.
- void PMIC_IRQHandler (void)

  Interrupt handler for the PMIC.
- void SNVS_Button_IRQHandler (void)

  Interrupt handler for the SNVS button.

9.21.1 Detailed Description

Module for the Board interface.

9.21.2 Macro Definition Documentation

9.21.2.1 CHECK_BITS_SET4

```
#define CHECK_BITS_SET4(
              A,
              M )
```

Value:

```
while((*((volatile uint32_t*)(A))         \
    & U32(M)) != ((uint32_t)(M))){}
```

9.21.2.2 CHECK_BITS_CLR4

```
#define CHECK_BITS_CLR4(
                A,
                M )
```

Value:

```
while((*((volatile uint32_t*)(A))         \
    & U32(M)) != U32(0U)){}
```

9.21.2.3 CHECK_ANY_BIT_SET4

```
#define CHECK_ANY_BIT_SET4(
                A,
                M )
```

Value:

```
while((*((volatile uint32_t*)(A))       \
    & U32(M)) == U32(0U)){}
```

9.21.2.4 CHECK_ANY_BIT_CLR4

```
#define CHECK_ANY_BIT_CLR4(
                A,
                M )
```

Value:

```
while((*((volatile uint32_t*)(A))       \
    & U32(M)) == U32(M)){}
```

9.21.2.5 BRD_ERR

```
#define BRD_ERR(
                X )
```

Value:

```
{err = (X);                                 \
                    if (err != SC_ERR_NONE)                     \
                    {                                           \
                        board_print(2, "error @ line %d: %d\n", \
                            __LINE__, err);                     \
                        return err;                             \
                    }}
```

9.21.3 Enumeration Type Documentation

9.21.3.1 board_parm_t

```
enum board_parm_t
```

Board config parameter types.

Enumerator

BOARD_PARM_KS1_RESUME_USEC	Supply ramp delay in usec for KS1 exit.
BOARD_PARM_KS1_RETENTION	Controls if retention is applied during KS1.
BOARD_PARM_KS1_ONOFF_WAKE	Controls if ONOFF button can wake from KS1.

9.21.3.2 board_ddr_action_t

```
enum board_ddr_action_t
```

DDR actions (power state transitions, etc.)

Enumerator

BOARD_DDR_COLD_INIT	Init DDR from POR.
BOARD_DDR_PERIODIC	Run periodic training.
BOARD_DDR_SR_DRC_ON_ENTER	Enter self-refresh (leave DRC on)
BOARD_DDR_SR_DRC_ON_EXIT	Exit self-refresh (DRC was on)
BOARD_DDR_SR_DRC_OFF_ENTER	Enter self-refresh (turn off DRC)
BOARD_DDR_SR_DRC_OFF_EXIT	Exit self-refresh (DRC was off)

9.21.4 Function Documentation

9.21.4.1 board_init()

```
void board_init (
            uint8_t phase )
```

This function initalizes the board.

Parameters

in	*phase*	boot phase

There are four phases to board intialization. The fist phase is the API phase (*phase* = 0) and initializes all of the board interface data structures. The second phase (*phase* = 1) is the HW phase and this initializes the board hardware. The third phase (*phase* = 2) is the final boot phase and is used to wrap up any needed init. A test phase (*phase* = 3) is called only when an SCFW image is built with unit tests and is called just before any tests are run. All are called from main_init() only.

9.21.4.2 board_get_debug_uart()

```
LPUART_Type* board_get_debug_uart (
             uint8_t * inst,
             uint32_t * baud )
```

This function returns the debug UART info.

Parameters

in	*inst*	UART instance
in	*baud*	UART baud rate

Returns

Pointer to the debug UART type.

9.21.4.3 board_config_debug_uart()

```
void board_config_debug_uart (
             sc_bool_t early_phase )
```

This function initalizes the debug UART.

Parameters

in	*early_phase*	flag indicating phase

9.21.4.4 board_config_sc()

```
void board_config_sc (
             sc_rm_pt_t pt_sc )
```

This function configures SCU resources.

Parameters

in	*pt_sc*	SCU partition

By default, the SCFW keeps most of the resources found in the SCU subsystem. It also keeps the SCU/PMIC pads

required for the main code to function. Any additional resources or pads required for the board code to run should be kept here. This is done by marking them as not movable.

9.21.4.5 board_parameter()

```
board_parm_rtn_t board_parameter (
            board_parm_t parm )
```

This function returns board configuration info.

Parameters

in	*parm*	parameter to return

This function is used to return board configuration info. Parameters define if various how various SoC connections are made at the board-level. For example, the external PCIe clock input.

See example code (board.c) for parameter/returns options.

Returns

Returns the paramter value.

9.21.4.6 board_rsrc_avail()

```
sc_bool_t board_rsrc_avail (
            sc_rsrc_t rsrc )
```

This function returns resource availability info.

Parameters

in	*rsrc*	resource to check

This function is used to return board configuration info. It reports if resources are functional on this board. For example, which DDR controllers are used.

See example code (board.c) for more details.

Returns

Returns SC_TRUE if available.

9.21.4.7 board_init_ddr()

```
sc_err_t board_init_ddr (
            sc_bool_t early,
            sc_bool_t ddr_initialized )
```

This function initalizes DDR.

Parameters

in	*early*	phase of init
in	*ddr_initialized*	True if ROM initialized the DDR

This function may be called twice. The early parameter is SC_TRUE when called prior to M4 start and SC_FALSE when called after. This allows the implementation to decide when DDR init needs to be done.

Note the first call will not occur unless SC_BD_FLAGS_EARLY_CPU_START is set in bd_flags of the boot container.

Returns

Returns an error code (SC_ERR_NONE = success).

9.21.4.8 board_ddr_config()

```
sc_err_t board_ddr_config (
            bool rom_caller,
            board_ddr_action_t action )
```

This function cinfigures the DDR.

Because this may be called by the ROM before the SCFW init is run, this code cannot make any calls to any other SCFW APIs unless properly conditioned with rom_caller.

Parameters

in	*rom_caller*	is ROM the caller?
in	*action*	perform this action on DDR

Returns

Returns SC_ERR_NONE if successful.

9.21.4.9 board_system_config()

```
sc_err_t board_system_config (
             sc_bool_t early,
             sc_rm_pt_t pt_boot )
```

This function allows the board file to do SCFW configuration.

Parameters

in	*early*	phase of init
in	*pt_boot*	boot partition

This function may be called twice. The early parameter is SC_TRUE when called prior to M4 start and SC_FALSE when called after. This allows the implementation to decide when to do configuration processing.

Note the first call will not occur unless SC_BD_FLAGS_EARLY_CPU_START is set in bd_flags of the boot container.

Typical actions here include creating a resource partition for an M4, powering up a board component, or configuring a shared clock.

Returns

Returns an error code (SC_ERR_NONE = success).

9.21.4.10 board_early_cpu()

```
sc_bool_t board_early_cpu (
             sc_rsrc_t cpu )
```

This function returns SC_TRUE for early CPUs.

Parameters

in	*cpu*	CPU

This function should return SC_TRUE if the CPU in question may be started early. This early start is before power on of later CPU subsystems. It would normally return SC_TRUE for CM4 cores that need to run early. Only SC_R_M4_0_↩ PID0 and SC_R_M4_1_PID0 can return SC_TRUE.

Note CPUs will only get started early if SC_BD_FLAGS_EARLY_CPU_START is set in bd_flags of the boot container.

Returns

Returns SC_TRUE if CPU should start early.

9.21.4.11 board_set_power_mode()

```
void board_set_power_mode (
            sc_sub_t ss,
            uint8_t pd,
            sc_pm_power_mode_t from_mode,
            sc_pm_power_mode_t to_mode )
```

This function transitions the power state for an external board- level supply which goes to the i.MX8.

Parameters

in	*ss*	subsystem using supply
in	*pd*	power domain
in	*from_mode*	power mode transitioning from
in	*to_mode*	power mode transitioning to

This function is used to transition a board power supply that is used by the SoC. It allows mapping of subsystem power domains to board supplies.

Note that the base code will enable/disable isolators after changing the state of internal power domains. External supplies sometimes supply a domain connected via an isolator to a domain passed here. In this case, this function needs to also control the connected domain's supply. For example, when LVDS SS PD (PD1) is powered toggled, the external supply for the LVDS PHY must be toggled here. MIPI and CSI SS PD are similar.

9.21.4.12 board_set_voltage()

```
sc_err_t board_set_voltage (
            sc_sub_t ss,
            uint32_t new_volt,
            sc_bool_t wait )
```

This function sets the voltage for a PMIC controlled SS.

Parameters

in	*ss*	subsystem
in	*new_volt*	voltage value to be set
in	*wait*	if SC_TRUE, wait for the PMIC to change the voltage.

Returns

Returns an error code (SC_ERR_NONE = success).

9.21.4.13 board_trans_resource_power()

```
sc_err_t board_trans_resource_power (
            sc_rm_idx_t idx,
            sc_rm_idx_t rsrc_idx,
            sc_pm_power_mode_t from_mode,
            sc_pm_power_mode_t to_mode )
```

This function transitions the power state for an external board- level supply which goes to a board component.

Parameters

in	*idx*	board-relative resource index
in	*rsrc_idx*	unified resource index
in	*from_mode*	power mode transitioning from
in	*to_mode*	power mode transitioning to

This function is used to transition a board power supply that is used by a board component. It allows mapping of board resources (e.g. SC_R_BOARD_R0) to board supplies.

idx should be used to identify the resource. It is 0-n and is associated with the board reosurces PMIC_0 through BOARD_R7.

rsrc_idx is only useful for debug output of a resource name.

Returns

Returns an error code (SC_ERR_NONE = success).

9.21.4.14 board_power()

```
sc_err_t board_power (
            sc_pm_power_mode_t mode )
```

This function is used to set the board power.

Parameters

in	*mode*	power mode to apply

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid mode

9.21.4.15 board_reset()

```
sc_err_t board_reset (
            sc_pm_reset_type_t type,
            sc_pm_reset_reason_t reason )
```

This function is used to reset the system.

Parameters

in	*type*	reset type
in	*reason*	cause of reset

Returns

Returns an error code (SC_ERR_NONE = success).

Return errors:

- SC_ERR_PARM if invalid type

If this function returns, then the reset did not occur due to an invalid parameter.

9.21.4.16 board_cpu_reset()

```
void board_cpu_reset (
            sc_rsrc_t resource,
            board_cpu_rst_ev_t reset_event )
```

This function is called when a CPU encounters a reset event.

Parameters

in	*resource*	CPU resource
in	*reset_event*	CPU reset event

9.21.4.17 board_panic()

```
void board_panic (
            sc_dsc_t dsc )
```

This function is called when a DSC reports a panic temp alarm.

Parameters

in	*dsc*	dsc reporting alarm

Note this function would normally request a board reset.

9.21.4.18 board_fault()

```
void board_fault (
            sc_bool_t restarted )
```

This function is called when a fault is detected or the SCFW returns from main().

Parameters

in	*restarted*	SC_TRUE if called on restart

Note this function would normally request a board reset. For debug builds it is common to disable the watchdog and loop.

The *restarted* paramter is SC_TRUE if this error is pending from the last restart.

9.21.4.19 board_security_violation()

```
void board_security_violation (
            void  )
```

This function is called when a security violation is reported by the SECO or SNVS.

Note this function would normally request a board reset. For debug builds it is common to do nothing.

9.21.4.20 board_get_button_status()

```
sc_bool_t board_get_button_status (
            void  )
```

This function is used to return the current status of the ON/OFF button.

Returns

Returns the status

9.21.4.21 board_set_control()

```
sc_err_t board_set_control (
            sc_rsrc_t resource,
            sc_rm_idx_t idx,
            sc_rm_idx_t rsrc_idx,
            uint32_t ctrl,
            uint32_t val )
```

This function sets a miscellaneous control value.

Parameters

in	*resource*	resource
in	*idx*	board resource index
in	*rsrc_idx*	unified resource index
in	*ctrl*	control to write
in	*val*	value to write

Returns

Returns an error code (SC_ERR_NONE = success).

Note this function can be used to set voltages for both SoC resources and board resources (e.g. SC_R_BOARD_R0).

9.21.4.22 board_get_control()

```
sc_err_t board_get_control (
            sc_rsrc_t resource,
            sc_rm_idx_t idx,
            sc_rm_idx_t rsrc_idx,
            uint32_t ctrl,
            uint32_t * val )
```

This function gets a miscellaneous control value.

Parameters

in	*resource*	resource
in	*idx*	board resource index
in	*rsrc_idx*	unified resource index
in	*ctrl*	control to read
out	*val*	pointer to return value

Returns

Returns an error code (SC_ERR_NONE = success).

Chapter 10

Data Structure Documentation

10.1 gpio_pin_config_t Struct Reference

The GPIO pin configuration structure.

Data Fields

- gpio_pin_direction_t pinDirection

  GPIO direction, input or output.
- uint8_t outputLogic

  Set default output logic, no use in input.

10.1.1 Detailed Description

The GPIO pin configuration structure.

Every pin can only be configured as either output pin or input pin at a time. If configured as a input pin, then leave the outputConfig unused Note : In some use cases, the corresponding port property should be configured in advance with the PORT_SetPinConfig()

10.2 lpi2c_data_match_config_t Struct Reference

LPI2C master data match configuration structure.

Data Fields

- lpi2c_data_match_config_mode_t matchMode

  Data match configuration setting.
- bool rxDataMatchOnly

  When set to true, received data is ignored until a successful match.
- uint32_t match0

  Match value 0.
- uint32_t match1

  Match value 1.

10.2.1 Detailed Description

LPI2C master data match configuration structure.

10.2.2 Field Documentation

10.2.2.1 matchMode

```
lpi2c_data_match_config_mode_t lpi2c_data_match_config_t::matchMode
```

Data match configuration setting.

10.2.2.2 rxDataMatchOnly

```
bool lpi2c_data_match_config_t::rxDataMatchOnly
```

When set to true, received data is ignored until a successful match.

10.2.2.3 match0

```
uint32_t lpi2c_data_match_config_t::match0
```

Match value 0.

10.2.2.4 match1

```
uint32_t lpi2c_data_match_config_t::match1
```

Match value 1.

10.3 lpi2c_master_config_t Struct Reference

Structure with settings to initialize the LPI2C master module.

Data Fields

- bool enableMaster

  Whether to enable master mode.
- bool enableDoze

  Whether master is enabled in doze mode.
- bool debugEnable

  Enable transfers to continue when halted in debug mode.
- bool ignoreAck

  Whether to ignore ACK/NACK.
- lpi2c_master_pin_config_t pinConfig

  The pin configuration option.
- uint32_t baudRate_Hz

  Desired baud rate in Hertz.
- uint32_t busIdleTimeout_ns

  Bus idle timeout in nanoseconds.
- uint32_t pinLowTimeout_ns

  Pin low timeout in nanoseconds.
- uint8_t sdaGlitchFilterWidth_ns

  Width in nanoseconds of glitch filter on SDA pin.
- uint8_t sclGlitchFilterWidth_ns

  Width in nanoseconds of glitch filter on SCL pin.
- struct {
  bool enable
  Enable host request.
  lpi2c_host_request_source_t source
  Host request source.
  lpi2c_host_request_polarity_t polarity
  Host request pin polarity.
  } hostRequest

  Host request options.

10.3.1 Detailed Description

Structure with settings to initialize the LPI2C master module.

This structure holds configuration settings for the LPI2C peripheral. To initialize this structure to reasonable defaults, call the LPI2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

10.3.2 Field Documentation

10.3.2.1 enableMaster

```
bool lpi2c_master_config_t::enableMaster
```

Whether to enable master mode.

10.3.2.2 enableDoze

```
bool lpi2c_master_config_t::enableDoze
```

Whether master is enabled in doze mode.

10.3.2.3 debugEnable

```
bool lpi2c_master_config_t::debugEnable
```

Enable transfers to continue when halted in debug mode.

10.3.2.4 ignoreAck

```
bool lpi2c_master_config_t::ignoreAck
```

Whether to ignore ACK/NACK.

10.3.2.5 pinConfig

```
lpi2c_master_pin_config_t lpi2c_master_config_t::pinConfig
```

The pin configuration option.

10.3.2.6 baudRate_Hz

```
uint32_t lpi2c_master_config_t::baudRate_Hz
```

Desired baud rate in Hertz.

10.3.2.7 busIdleTimeout_ns

```
uint32_t lpi2c_master_config_t::busIdleTimeout_ns
```

Bus idle timeout in nanoseconds.

Set to 0 to disable.

10.3.2.8 pinLowTimeout_ns

```
uint32_t lpi2c_master_config_t::pinLowTimeout_ns
```

Pin low timeout in nanoseconds.

Set to 0 to disable.

10.3.2.9 sdaGlitchFilterWidth_ns

```
uint8_t lpi2c_master_config_t::sdaGlitchFilterWidth_ns
```

Width in nanoseconds of glitch filter on SDA pin.

Set to 0 to disable.

10.3.2.10 sclGlitchFilterWidth_ns

```
uint8_t lpi2c_master_config_t::sclGlitchFilterWidth_ns
```

Width in nanoseconds of glitch filter on SCL pin.

Set to 0 to disable.

10.3.2.11 enable

```
bool lpi2c_master_config_t::enable
```

Enable host request.

10.3.2.12 source

```
lpi2c_host_request_source_t lpi2c_master_config_t::source
```

Host request source.

10.3.2.13 polarity

```
lpi2c_host_request_polarity_t lpi2c_master_config_t::polarity
```

Host request pin polarity.

10.3.2.14 hostRequest

```
struct { ... } lpi2c_master_config_t::hostRequest
```

Host request options.

10.4 lpi2c_master_handle_t Struct Reference

Driver handle for master non-blocking APIs.

Data Fields

- uint8_t state
  Transfer state machine current state.
- uint16_t remainingBytes
  Remaining byte count in current state.
- uint16_t * buf
  Buffer pointer for current state.
- uint16_t commandBuffer [7]
  LPI2C command sequence.
- lpi2c_master_transfer_t transfer
  Copy of the current transfer info.
- lpi2c_master_transfer_callback_t completionCallback
  Callback function pointer.
- void * userData
  Application data passed to callback.

10.4.1 Detailed Description

Driver handle for master non-blocking APIs.

Note

The contents of this structure are private and subject to change.

10.4.2 Field Documentation

10.4.2.1 state

```
uint8_t lpi2c_master_handle_t::state
```

Transfer state machine current state.

10.4.2.2 remainingBytes

```
uint16_t lpi2c_master_handle_t::remainingBytes
```

Remaining byte count in current state.

10.4.2.3 buf

```
uint16_t* lpi2c_master_handle_t::buf
```

Buffer pointer for current state.

10.4.2.4 commandBuffer

```
uint16_t lpi2c_master_handle_t::commandBuffer[7]
```

LPI2C command sequence.

10.4.2.5 transfer

```
lpi2c_master_transfer_t lpi2c_master_handle_t::transfer
```

Copy of the current transfer info.

10.4.2.6 completionCallback

```
lpi2c_master_transfer_callback_t lpi2c_master_handle_t::completionCallback
```

Callback function pointer.

10.4.2.7 userData

```
void* lpi2c_master_handle_t::userData
```

Application data passed to callback.

10.5 lpi2c_master_transfer_t Struct Reference

Non-blocking transfer descriptor structure.

Data Fields

- uint32_t flags
  Bit mask of options for the transfer.
- uint16_t slaveAddress
  The 7-bit slave address.
- lpi2c_direction_t direction
  Either kLPI2C_Read or kLPI2C_Write.
- uint32_t subaddress
  Sub address.
- size_t subaddressSize
  Length of sub address to send in bytes.
- void * data
  Pointer to data to transfer.
- size_t dataSize
  Number of bytes to transfer.

10.5.1 Detailed Description

Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the LPI2C_MasterTransferNonBlocking() API.

10.5.2 Field Documentation

10.5.2.1 flags

```
uint32_t lpi2c_master_transfer_t::flags
```

Bit mask of options for the transfer.

See enumeration _lpi2c_master_transfer_flags for available options. Set to 0 or kLPI2C_TransferDefaultFlag for normal transfers.

10.5.2.2 slaveAddress

```
uint16_t lpi2c_master_transfer_t::slaveAddress
```

The 7-bit slave address.

10.5.2.3 direction

```
lpi2c_direction_t lpi2c_master_transfer_t::direction
```

Either kLPI2C_Read or kLPI2C_Write.

10.5.2.4 subaddress

```
uint32_t lpi2c_master_transfer_t::subaddress
```

Sub address.

Transferred MSB first.

10.5.2.5 subaddressSize

```
size_t lpi2c_master_transfer_t::subaddressSize
```

Length of sub address to send in bytes.

Maximum size is 4 bytes.

10.5.2.6 data

```
void* lpi2c_master_transfer_t::data
```

Pointer to data to transfer.

10.5.2.7 dataSize

```
size_t lpi2c_master_transfer_t::dataSize
```

Number of bytes to transfer.

10.6 lpi2c_slave_config_t Struct Reference

Structure with settings to initialize the LPI2C slave module.

Data Fields

- bool enableSlave
  Enable slave mode.
- uint8_t address0
  Slave's 7-bit address.
- uint8_t address1
  Alternate slave 7-bit address.
- lpi2c_slave_address_match_t addressMatchMode
  Address matching options.
- bool filterDozeEnable
  Enable digital glitch filter in doze mode.
- bool filterEnable
  Enable digital glitch filter.
- bool enableGeneralCall
  Enable general call address matching.
-

struct {
  bool enableAck
    Enables SCL clock stretching during slave-transmit address byte(s) and slave-receiver address and data byte(s) to allow softw
  bool enableTx
    Enables SCL clock stretching when the transmit data flag is set during a slave-transmit transfer.
  bool enableRx
    Enables SCL clock stretching when receive data flag is set during a slave-receive transfer.
  bool enableAddress
    Enables SCL clock stretching when the address valid flag is asserted.
} **sclStall**

- bool ignoreAck

  Continue transfers after a NACK is detected.
- bool enableReceivedAddressRead

  Enable reading the address received address as the first byte of data.
- uint32_t sdaGlitchFilterWidth_ns

  Width in nanoseconds of the digital filter on the SDA signal.
- uint32_t sclGlitchFilterWidth_ns

  Width in nanoseconds of the digital filter on the SCL signal.
- uint32_t dataValidDelay_ns

  Width in nanoseconds of the data valid delay.
- uint32_t clockHoldTime_ns

  Width in nanoseconds of the clock hold time.

10.6.1 Detailed Description

Structure with settings to initialize the LPI2C slave module.

This structure holds configuration settings for the LPI2C slave peripheral. To initialize this structure to reasonable defaults, call the LPI2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

10.6.2 Field Documentation

10.6.2.1 enableSlave

```
bool lpi2c_slave_config_t::enableSlave
```

Enable slave mode.

10.6.2.2 address0

```
uint8_t lpi2c_slave_config_t::address0
```

Slave's 7-bit address.

10.6.2.3 address1

```
uint8_t lpi2c_slave_config_t::address1
```

Alternate slave 7-bit address.

10.6.2.4 addressMatchMode

```
lpi2c_slave_address_match_t lpi2c_slave_config_t::addressMatchMode
```

Address matching options.

10.6.2.5 filterDozeEnable

```
bool lpi2c_slave_config_t::filterDozeEnable
```

Enable digital glitch filter in doze mode.

10.6.2.6 filterEnable

```
bool lpi2c_slave_config_t::filterEnable
```

Enable digital glitch filter.

10.6.2.7 enableGeneralCall

```
bool lpi2c_slave_config_t::enableGeneralCall
```

Enable general call address matching.

10.6.2.8 enableAck

```
bool lpi2c_slave_config_t::enableAck
```

Enables SCL clock stretching during slave-transmit address byte(s) and slave-receiver address and data byte(s) to allow software to write the Transmit ACK Register before the ACK or NACK is transmitted.

Clock stretching occurs when transmitting the 9th bit. When enableAckSCLStall is enabled, there is no need to set either enableRxDataSCLStall or enableAddressSCLStall.

10.6.2.9 enableTx

```
bool lpi2c_slave_config_t::enableTx
```

Enables SCL clock stretching when the transmit data flag is set during a slave-transmit transfer.

10.6.2.10 enableRx

```
bool lpi2c_slave_config_t::enableRx
```

Enables SCL clock stretching when receive data flag is set during a slave-receive transfer.

10.6.2.11 enableAddress

```
bool lpi2c_slave_config_t::enableAddress
```

Enables SCL clock stretching when the address valid flag is asserted.

10.6.2.12 ignoreAck

```
bool lpi2c_slave_config_t::ignoreAck
```

Continue transfers after a NACK is detected.

10.6.2.13 enableReceivedAddressRead

```
bool lpi2c_slave_config_t::enableReceivedAddressRead
```

Enable reading the address received address as the first byte of data.

10.6.2.14 sdaGlitchFilterWidth_ns

```
uint32_t lpi2c_slave_config_t::sdaGlitchFilterWidth_ns
```

Width in nanoseconds of the digital filter on the SDA signal.

10.6.2.15 sclGlitchFilterWidth_ns

```
uint32_t lpi2c_slave_config_t::sclGlitchFilterWidth_ns
```

Width in nanoseconds of the digital filter on the SCL signal.

10.6.2.16 dataValidDelay_ns

```
uint32_t lpi2c_slave_config_t::dataValidDelay_ns
```

Width in nanoseconds of the data valid delay.

10.6.2.17 clockHoldTime_ns

```
uint32_t lpi2c_slave_config_t::clockHoldTime_ns
```

Width in nanoseconds of the clock hold time.

10.7 lpi2c_slave_handle_t Struct Reference

LPI2C slave handle structure.

Data Fields

- lpi2c_slave_transfer_t transfer
  LPI2C slave transfer copy.
- bool isBusy
  Whether transfer is busy.
- bool wasTransmit
  Whether the last transfer was a transmit.
- uint32_t eventMask
  Mask of enabled events.
- uint32_t transferredCount
  Count of bytes transferred.
- lpi2c_slave_transfer_callback_t callback
  Callback function called at transfer event.
- void * userData
  Callback parameter passed to callback.

10.7.1 Detailed Description

LPI2C slave handle structure.

Note

The contents of this structure are private and subject to change.

10.7.2 Field Documentation

10.7.2.1 transfer

```
lpi2c_slave_transfer_t lpi2c_slave_handle_t::transfer
```

LPI2C slave transfer copy.

10.7.2.2 isBusy

```
bool lpi2c_slave_handle_t::isBusy
```

Whether transfer is busy.

10.7.2.3 wasTransmit

```
bool lpi2c_slave_handle_t::wasTransmit
```

Whether the last transfer was a transmit.

10.7.2.4 eventMask

```
uint32_t lpi2c_slave_handle_t::eventMask
```

Mask of enabled events.

10.7.2.5 transferredCount

```
uint32_t lpi2c_slave_handle_t::transferredCount
```

Count of bytes transferred.

10.7.2.6 callback

```
lpi2c_slave_transfer_callback_t lpi2c_slave_handle_t::callback
```

Callback function called at transfer event.

10.7.2.7 userData

```
void* lpi2c_slave_handle_t::userData
```

Callback parameter passed to callback.

10.8 lpi2c_slave_transfer_t Struct Reference

LPI2C slave transfer structure.

Data Fields

- lpi2c_slave_transfer_event_t event
  Reason the callback is being invoked.
- uint8_t receivedAddress
  Matching address send by master.
- uint8_t * data
  Transfer buffer.
- size_t dataSize
  Transfer size.
- status_t completionStatus
  Success or error code describing how the transfer completed.
- size_t transferredCount
  Number of bytes actually transferred since start or last repeated start.

10.8.1 Detailed Description

LPI2C slave transfer structure.

10.8.2 Field Documentation

10.8.2.1 event

```
lpi2c_slave_transfer_event_t lpi2c_slave_transfer_t::event
```

Reason the callback is being invoked.

10.8.2.2 receivedAddress

```
uint8_t lpi2c_slave_transfer_t::receivedAddress
```

Matching address send by master.

10.8.2.3 completionStatus

```
status_t lpi2c_slave_transfer_t::completionStatus
```

Success or error code describing how the transfer completed.

Only applies for kLPI2C_SlaveCompletionEvent.

10.8.2.4 transferredCount

```
size_t lpi2c_slave_transfer_t::transferredCount
```

Number of bytes actually transferred since start or last repeated start.

10.9 pmic_version_t Struct Reference

Structure for ID and Revision of PMIC.

Data Fields

- uint8_t device_id

  dev ID value (reg location may differ per device)
- uint8_t si_rev

  silicon revision value read from device

10.9.1 Detailed Description

Structure for ID and Revision of PMIC.

Chapter 11

File Documentation

11.1 platform/board/pmic.h File Reference

PMIC include for PMIC interface layer.

Macros

- #define **PMIC_SET_VOLTAGE** dynamic_pmic_set_voltage
- #define **PMIC_GET_VOLTAGE** dynamic_pmic_get_voltage
- #define **PMIC_SET_MODE** dynamic_pmic_set_mode
- #define **PMIC_GET_MODE** dynamic_pmic_get_mode
- #define **PMIC_IRQ_SERVICE** dynamic_pmic_irq_service
- #define **PMIC_REGISTER_ACCESS** dynamic_pmic_register_access
- #define **GET_PMIC_VERSION** dynamic_get_pmic_version
- #define **GET_PMIC_TEMP** dynamic_get_pmic_temp
- #define **SET_PMIC_TEMP_ALARM** dynamic_set_pmic_temp_alarm

Defines for supported PMIC devices

- #define **PMIC_NONE** 0U
- #define **PF100** 1U
- #define **PF8100** 2U
- #define **PF8200** 3U

Defines for PMIC configuration

- #define **PF100_DEV_ID** 0x10U
- #define **PF8100_DEV_ID** 0x40U
- #define **PF8200_DEV_ID** 0x48U
- #define **PF8X00_FAM_ID** 0x40U
- #define **PF8100_A0_REV** 0x10U
- #define **FAM_ID_MASK** 0xF0U

Functions

- sc_err_t dynamic_pmic_set_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t vol_mv, uint32_t mode_to_set)

  This function sets the voltage of a corresponding voltage regulator for the supported PMIC types.
- sc_err_t dynamic_pmic_get_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t *vol_mv, uint32_t mode_to_get)

  This function gets the voltage on a corresponding voltage regulator of the PMIC.
- sc_err_t dynamic_pmic_set_mode (pmic_id_t id, uint32_t pmic_reg, uint32_t mode)

  This function sets the mode of the specified regulator.
- sc_err_t dynamic_pmic_get_mode (pmic_id_t id, uint32_t pmic_reg, uint32_t *mode)

  This function gets the mode of the specified regulator.
- sc_bool_t dynamic_pmic_irq_service (pmic_id_t id)

  This function services the interrupt for the temp alarm.
- sc_err_t dynamic_pmic_register_access (pmic_id_t id, uint32_t address, sc_bool_t read_write, uint8_t *value)

  This function allows access to individual registers of the PMIC.
- pmic_version_t dynamic_get_pmic_version (pmic_id_t id)

  This function returns the device ID and revision for the PMIC.
- uint32_t dynamic_get_pmic_temp (pmic_id_t id)

  This function gets the current PMIC temperature as sensed by the PMIC temperature sensor.
- uint32_t dynamic_set_pmic_temp_alarm (pmic_id_t id, uint32_t temp)

  This function sets the temp alarm for the PMIC in Celsius.

Variables

- uint8_t PMIC_TYPE

  Global PMIC type identifier.

11.1.1 Detailed Description

PMIC include for PMIC interface layer.

This API is used to abstract the PMIC driver. It also supports dynamic PMIC identification and function binding (normally only used for dev boards).

11.2 platform/drivers/gpio/fsl_gpio.h File Reference

Data Structures

- struct gpio_pin_config_t

  The GPIO pin configuration structure.

Macros

Driver version

- #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))

  GPIO driver version 2.1.0.

Enumerations

- enum gpio_pin_direction_t { kGPIO_DigitalInput = 0U, kGPIO_DigitalOutput = 1U }
  GPIO direction definition.

Functions

GPIO Configuration

- void GPIO_PinInit (GPIO_Type *base, uint32_t pin, const gpio_pin_config_t *config)
  Initializes a GPIO pin used by the board.

GPIO Output Operations

- static void GPIO_WritePinOutput (GPIO_Type *base, uint32_t pin, uint8_t output)
  Sets the output level of the multiple GPIO pins to the logic 1 or 0.
- static void GPIO_SetPinsOutput (GPIO_Type *base, uint32_t mask)
  Sets the output level of the multiple GPIO pins to the logic 1.
- static void GPIO_ClearPinsOutput (GPIO_Type *base, uint32_t mask)
  Sets the output level of the multiple GPIO pins to the logic 0.
- static void GPIO_TogglePinsOutput (GPIO_Type *base, uint32_t mask)
  Reverses current output logic of the multiple GPIO pins.

GPIO Input Operations

- static uint32_t GPIO_ReadPinInput (GPIO_Type *base, uint32_t pin)
  Reads the current input value of the whole GPIO port.

GPIO Interrupt

- uint32_t GPIO_GetPinsInterruptFlags (GPIO_Type *base)
  Reads whole GPIO port interrupt status flag.
- void GPIO_ClearPinsInterruptFlags (GPIO_Type *base, uint32_t mask)
  Clears multiple GPIO pin interrupt status flag.

FGPIO Configuration

- void FGPIO_PinInit (FGPIO_Type *base, uint32_t pin, const gpio_pin_config_t *config)
  Initializes a FGPIO pin used by the board.

FGPIO Output Operations

- static void FGPIO_WritePinOutput (FGPIO_Type *base, uint32_t pin, uint8_t output)
  Sets the output level of the multiple FGPIO pins to the logic 1 or 0.
- static void FGPIO_SetPinsOutput (FGPIO_Type *base, uint32_t mask)
  Sets the output level of the multiple FGPIO pins to the logic 1.
- static void FGPIO_ClearPinsOutput (FGPIO_Type *base, uint32_t mask)
  Sets the output level of the multiple FGPIO pins to the logic 0.
- static void FGPIO_TogglePinsOutput (FGPIO_Type *base, uint32_t mask)
  Reverses current output logic of the multiple FGPIO pins.

FGPIO Input Operations

- static uint32_t FGPIO_ReadPinInput (FGPIO_Type *base, uint32_t pin)
  Reads the current input value of the whole FGPIO port.

FGPIO Interrupt

- uint32_t FGPIO_GetPinsInterruptFlags (FGPIO_Type *base)
  Reads the whole FGPIO port interrupt status flag.
- void FGPIO_ClearPinsInterruptFlags (FGPIO_Type *base, uint32_t mask)
  Clears the multiple FGPIO pin interrupt status flag.

11.3 platform/drivers/lpi2c/fsl_lpi2c.h File Reference

Data Structures

- struct lpi2c_master_config_t
  Structure with settings to initialize the LPI2C master module.
- struct lpi2c_data_match_config_t
  LPI2C master data match configuration structure.
- struct lpi2c_master_transfer_t
  Non-blocking transfer descriptor structure.
- struct lpi2c_master_handle_t
  Driver handle for master non-blocking APIs.
- struct lpi2c_slave_config_t
  Structure with settings to initialize the LPI2C slave module.
- struct lpi2c_slave_transfer_t
  LPI2C slave transfer structure.
- struct lpi2c_slave_handle_t
  LPI2C slave handle structure.

Macros

Driver version

- #define FSL_LPI2C_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))
  LPI2C driver version 2.1.0.

Typedefs

- typedef void(* lpi2c_master_transfer_callback_t) (LPI2C_Type *base, lpi2c_master_handle_t *handle, status_t completionStatus, void *userData)
  Master completion callback function pointer type.
- typedef void(* lpi2c_slave_transfer_callback_t) (LPI2C_Type *base, lpi2c_slave_transfer_t *transfer, void *user↩ Data)
  Slave event callback function pointer type.

Enumerations

- enum _lpi2c_status {
  kStatus_LPI2C_Busy = MAKE_STATUS(kStatusGroup_LPI2C, 0), kStatus_LPI2C_Idle = MAKE_STATUS(k↩
  StatusGroup_LPI2C, 1), kStatus_LPI2C_Nak = MAKE_STATUS(kStatusGroup_LPI2C, 2), kStatus_LPI2C_FifoError
  = MAKE_STATUS(kStatusGroup_LPI2C, 3),
  kStatus_LPI2C_BitError = MAKE_STATUS(kStatusGroup_LPI2C, 4), kStatus_LPI2C_ArbitrationLost = MAKE↩
  _STATUS(kStatusGroup_LPI2C, 5), kStatus_LPI2C_PinLowTimeout, kStatus_LPI2C_NoTransferInProgress,
  kStatus_LPI2C_DmaRequestFail = MAKE_STATUS(kStatusGroup_LPI2C, 7) }

  LPI2C status return codes.
- enum _lpi2c_master_flags {
  kLPI2C_MasterTxReadyFlag = LPI2C_MSR_TDF_MASK, kLPI2C_MasterRxReadyFlag = LPI2C_MSR_RDF↩
  _MASK, kLPI2C_MasterEndOfPacketFlag = LPI2C_MSR_EPF_MASK, kLPI2C_MasterStopDetectFlag = LPI2↩
  C_MSR_SDF_MASK,
  kLPI2C_MasterNackDetectFlag = LPI2C_MSR_NDF_MASK, kLPI2C_MasterArbitrationLostFlag = LPI2C_M↩
  SR_ALF_MASK, kLPI2C_MasterFifoErrFlag = LPI2C_MSR_FEF_MASK, kLPI2C_MasterPinLowTimeoutFlag =
  LPI2C_MSR_PLTF_MASK,
  kLPI2C_MasterDataMatchFlag = LPI2C_MSR_DMF_MASK, kLPI2C_MasterBusyFlag = LPI2C_MSR_MBF_M↩
  ASK, kLPI2C_MasterBusBusyFlag = LPI2C_MSR_BBF_MASK }

  LPI2C master peripheral flags.
- enum lpi2c_direction_t { kLPI2C_Write = 0U, kLPI2C_Read = 1U }

  Direction of master and slave transfers.
- enum lpi2c_master_pin_config_t {
  kLPI2C_2PinOpenDrain = 0x0U, kLPI2C_2PinOutputOnly = 0x1U, kLPI2C_2PinPushPull = 0x2U, kLPI2C_4PinPushPull
  = 0x3U,
  kLPI2C_2PinOpenDrainWithSeparateSlave, kLPI2C_2PinOutputOnlyWithSeparateSlave, kLPI2C_2PinPushPullWithSeparateSlave
  kLPI2C_4PinPushPullWithInvertedOutput = 0x7U }

  LPI2C pin configuration.
- enum lpi2c_host_request_source_t { kLPI2C_HostRequestExternalPin = 0x0U, kLPI2C_HostRequestInputTrigger
  = 0x1U }

  LPI2C master host request selection.
- enum lpi2c_host_request_polarity_t { kLPI2C_HostRequestPinActiveLow = 0x0U, kLPI2C_HostRequestPinActiveHigh
  = 0x1U }

  LPI2C master host request pin polarity configuration.
- enum lpi2c_data_match_config_mode_t {
  kLPI2C_MatchDisabled = 0x0U, kLPI2C_1stWordEqualsM0OrM1 = 0x2U, kLPI2C_AnyWordEqualsM0OrM1 =
  0x3U, kLPI2C_1stWordEqualsM0And2ndWordEqualsM1,
  kLPI2C_AnyWordEqualsM0AndNextWordEqualsM1, kLPI2C_1stWordAndM1EqualsM0AndM1, kLPI2C_AnyWordAndM1EqualsM0
  }

  LPI2C master data match configuration modes.
- enum _lpi2c_master_transfer_flags { kLPI2C_TransferDefaultFlag = 0x00U, kLPI2C_TransferNoStartFlag =
  0x01U, kLPI2C_TransferRepeatedStartFlag = 0x02U, kLPI2C_TransferNoStopFlag = 0x04U }

  Transfer option flags.
- enum _lpi2c_slave_flags {
  kLPI2C_SlaveTxReadyFlag = LPI2C_SSR_TDF_MASK, kLPI2C_SlaveRxReadyFlag = LPI2C_SSR_RDF_MA↩
  SK, kLPI2C_SlaveAddressValidFlag = LPI2C_SSR_AVF_MASK, kLPI2C_SlaveTransmitAckFlag = LPI2C_SS↩
  R_TAF_MASK,
  kLPI2C_SlaveRepeatedStartDetectFlag = LPI2C_SSR_RSF_MASK, kLPI2C_SlaveStopDetectFlag = LPI2C_↩
  SSR_SDF_MASK, kLPI2C_SlaveBitErrFlag = LPI2C_SSR_BEF_MASK, kLPI2C_SlaveFifoErrFlag = LPI2C_S↩
  SR_FEF_MASK,

kLPI2C_SlaveAddressMatch0Flag = LPI2C_SSR_AM0F_MASK, kLPI2C_SlaveAddressMatch1Flag = LPI2C_↩
SSR_AM1F_MASK, kLPI2C_SlaveGeneralCallFlag = LPI2C_SSR_GCF_MASK, kLPI2C_SlaveBusyFlag = LP↩
I2C_SSR_SBF_MASK,
kLPI2C_SlaveBusBusyFlag = LPI2C_SSR_BBF_MASK }

LPI2C slave peripheral flags.

- enum lpi2c_slave_address_match_t { kLPI2C_MatchAddress0 = 0U, kLPI2C_MatchAddress0OrAddress1 = 2U, kLPI2C_MatchAddress0ThroughAddress1 = 6U }

  LPI2C slave address match options.
- enum lpi2c_slave_transfer_event_t {
  kLPI2C_SlaveAddressMatchEvent = 0x01U, kLPI2C_SlaveTransmitEvent = 0x02U, kLPI2C_SlaveReceiveEvent = 0x04U, kLPI2C_SlaveTransmitAckEvent = 0x08U,
  kLPI2C_SlaveRepeatedStartEvent = 0x10U, kLPI2C_SlaveCompletionEvent = 0x20U, kLPI2C_SlaveAllEvents }

  Set of events sent to the callback for non blocking slave transfers.

Functions

Initialization and deinitialization

- void LPI2C_MasterGetDefaultConfig (lpi2c_master_config_t *masterConfig)

  Provides a default configuration for the LPI2C master peripheral.
- void LPI2C_MasterInit (LPI2C_Type *base, const lpi2c_master_config_t *masterConfig, uint32_t source↩
  Clock_Hz)

  Initializes the LPI2C master peripheral.
- void LPI2C_MasterDeinit (LPI2C_Type *base)

  Deinitializes the LPI2C master peripheral.
- void LPI2C_MasterConfigureDataMatch (LPI2C_Type *base, const lpi2c_data_match_config_t *config)

  Configures LPI2C master data match feature.
- static void LPI2C_MasterReset (LPI2C_Type *base)

  Performs a software reset.
- static void LPI2C_MasterEnable (LPI2C_Type *base, bool enable)

  Enables or disables the LPI2C module as master.

Status

- static uint32_t LPI2C_MasterGetStatusFlags (LPI2C_Type *base)

  Gets the LPI2C master status flags.
- static void LPI2C_MasterClearStatusFlags (LPI2C_Type *base, uint32_t statusMask)

  Clears the LPI2C master status flag state.

Interrupts

- static void LPI2C_MasterEnableInterrupts (LPI2C_Type *base, uint32_t interruptMask)

  Enables the LPI2C master interrupt requests.
- static void LPI2C_MasterDisableInterrupts (LPI2C_Type *base, uint32_t interruptMask)

  Disables the LPI2C master interrupt requests.
- static uint32_t LPI2C_MasterGetEnabledInterrupts (LPI2C_Type *base)

  Returns the set of currently enabled LPI2C master interrupt requests.

DMA control

- static void LPI2C_MasterEnableDMA (LPI2C_Type *base, bool enableTx, bool enableRx)

Enables or disables LPI2C master DMA requests.

- static uint32_t LPI2C_MasterGetTxFifoAddress (LPI2C_Type *base)
  Gets LPI2C master transmit data register address for DMA transfer.
- static uint32_t LPI2C_MasterGetRxFifoAddress (LPI2C_Type *base)
  Gets LPI2C master receive data register address for DMA transfer.

FIFO control

- static void LPI2C_MasterSetWatermarks (LPI2C_Type *base, size_t txWords, size_t rxWords)
  Sets the watermarks for LPI2C master FIFOs.
- static void LPI2C_MasterGetFifoCounts (LPI2C_Type *base, size_t *rxCount, size_t *txCount)
  Gets the current number of words in the LPI2C master FIFOs.

Bus operations

- void LPI2C_MasterSetBaudRate (LPI2C_Type *base, uint32_t sourceClock_Hz, uint32_t baudRate_Hz)
  Sets the I2C bus frequency for master transactions.
- static bool LPI2C_MasterGetBusIdleState (LPI2C_Type *base)
  Returns whether the bus is idle.
- status_t LPI2C_MasterStart (LPI2C_Type *base, uint8_t address, lpi2c_direction_t dir)
  Sends a START signal and slave address on the I2C bus.
- static status_t LPI2C_MasterRepeatedStart (LPI2C_Type *base, uint8_t address, lpi2c_direction_t dir)
  Sends a repeated START signal and slave address on the I2C bus.
- status_t LPI2C_MasterSend (LPI2C_Type *base, const void *txBuff, size_t txSize)
  Performs a polling send transfer on the I2C bus.
- status_t LPI2C_MasterReceive (LPI2C_Type *base, void *rxBuff, size_t rxSize)
  Performs a polling receive transfer on the I2C bus.
- status_t LPI2C_MasterStop (LPI2C_Type *base)
  Sends a STOP signal on the I2C bus.

Non-blocking

- void LPI2C_MasterTransferCreateHandle (LPI2C_Type *base, lpi2c_master_handle_t *handle, lpi2c_master_transfer_callback_t callback, void *userData)
  Creates a new handle for the LPI2C master non-blocking APIs.
- status_t LPI2C_MasterTransferNonBlocking (LPI2C_Type *base, lpi2c_master_handle_t *handle, lpi2c_master_transfer_t *transfer)
  Performs a non-blocking transaction on the I2C bus.
- status_t LPI2C_MasterTransferGetCount (LPI2C_Type *base, lpi2c_master_handle_t *handle, size_t *count)
  Returns number of bytes transferred so far.
- void LPI2C_MasterTransferAbort (LPI2C_Type *base, lpi2c_master_handle_t *handle)
  Terminates a non-blocking LPI2C master transmission early.

IRQ handler

- void LPI2C_MasterTransferHandleIRQ (LPI2C_Type *base, lpi2c_master_handle_t *handle)
  Reusable routine to handle master interrupts.

Slave initialization and deinitialization

- void LPI2C_SlaveGetDefaultConfig (lpi2c_slave_config_t *slaveConfig)
  Provides a default configuration for the LPI2C slave peripheral.

- void LPI2C_SlaveInit (LPI2C_Type *base, const lpi2c_slave_config_t *slaveConfig, uint32_t sourceClock_Hz)
  Initializes the LPI2C slave peripheral.
- void LPI2C_SlaveDeinit (LPI2C_Type *base)
  Deinitializes the LPI2C slave peripheral.
- static void LPI2C_SlaveReset (LPI2C_Type *base)
  Performs a software reset of the LPI2C slave peripheral.
- static void LPI2C_SlaveEnable (LPI2C_Type *base, bool enable)
  Enables or disables the LPI2C module as slave.

Slave status

- static uint32_t LPI2C_SlaveGetStatusFlags (LPI2C_Type *base)
  Gets the LPI2C slave status flags.
- static void LPI2C_SlaveClearStatusFlags (LPI2C_Type *base, uint32_t statusMask)
  Clears the LPI2C status flag state.

Slave interrupts

- static void LPI2C_SlaveEnableInterrupts (LPI2C_Type *base, uint32_t interruptMask)
  Enables the LPI2C slave interrupt requests.
- static void LPI2C_SlaveDisableInterrupts (LPI2C_Type *base, uint32_t interruptMask)
  Disables the LPI2C slave interrupt requests.
- static uint32_t LPI2C_SlaveGetEnabledInterrupts (LPI2C_Type *base)
  Returns the set of currently enabled LPI2C slave interrupt requests.

Slave DMA control

- static void LPI2C_SlaveEnableDMA (LPI2C_Type *base, bool enableAddressValid, bool enableRx, bool enableTx)
  Enables or disables the LPI2C slave peripheral DMA requests.

Slave bus operations

- static bool LPI2C_SlaveGetBusIdleState (LPI2C_Type *base)
  Returns whether the bus is idle.
- static void LPI2C_SlaveTransmitAck (LPI2C_Type *base, bool ackOrNack)
  Transmits either an ACK or NAK on the I2C bus in response to a byte from the master.
- static uint32_t LPI2C_SlaveGetReceivedAddress (LPI2C_Type *base)
  Returns the slave address sent by the I2C master.
- status_t LPI2C_SlaveSend (LPI2C_Type *base, const void *txBuff, size_t txSize, size_t *actualTxSize)
  Performs a polling send transfer on the I2C bus.
- status_t LPI2C_SlaveReceive (LPI2C_Type *base, void *rxBuff, size_t rxSize, size_t *actualRxSize)
  Performs a polling receive transfer on the I2C bus.

Slave non-blocking

- void LPI2C_SlaveTransferCreateHandle (LPI2C_Type *base, lpi2c_slave_handle_t *handle, lpi2c_slave_transfer_callback_t callback, void *userData)
  Creates a new handle for the LPI2C slave non-blocking APIs.
- status_t LPI2C_SlaveTransferNonBlocking (LPI2C_Type *base, lpi2c_slave_handle_t *handle, uint32_t eventMask)

Starts accepting slave transfers.

- status_t LPI2C_SlaveTransferGetCount (LPI2C_Type *base, lpi2c_slave_handle_t *handle, size_t *count)

  Gets the slave transfer status during a non-blocking transfer.
- void LPI2C_SlaveTransferAbort (LPI2C_Type *base, lpi2c_slave_handle_t *handle)

  Aborts the slave non-blocking transfers.

Slave IRQ handler

- void LPI2C_SlaveTransferHandleIRQ (LPI2C_Type *base, lpi2c_slave_handle_t *handle)

  Reusable routine to handle slave interrupts.

11.4 platform/drivers/pmic/fsl_pmic.h File Reference

Data Structures

- struct pmic_version_t

  Structure for ID and Revision of PMIC.

Macros

- #define **i2c_error_flags**

Typedefs

- typedef uint8_t pmic_id_t

  This type is used to declare which PMIC to address.

Functions

- status_t i2c_write_sub (uint8_t device_addr, uint8_t reg, void *data, uint32_t dataLength)

  This function is a simple write to an i2c register on the PMIC.
- status_t i2c_write (uint8_t device_addr, uint8_t reg, void *data, uint32_t dataLength)

  This function writes an i2c register on the PMIC device with clock management.
- status_t i2c_read_sub (uint8_t device_addr, uint8_t reg, void *data, uint32_t dataLength)

  This function is a simple read of an i2c register on the PMIC.
- status_t i2c_read (uint8_t device_addr, uint8_t reg, void *data, uint32_t dataLength)

  This function reads an i2c register on the PMIC device with clock management.
- status_t **i2c_j1850_write** (uint8_t device_addr, uint8_t reg, void *data, uint8_t dataLength)
- status_t **i2c_j1850_read** (uint8_t device_addr, uint8_t reg, void *data, uint8_t dataLength)
- uint8_t pmic_get_device_id (uint8_t address)

  This function reads the register at address 0x0 for the Device ID.

11.5 platform/drivers/pmic/pf100/fsl_pf100.h File Reference

Macros

Defines for pf100_vol_regs_t

- #define SW1AB 0x20U
  Base register for SW1AB control.
- #define SW1C 0x2EU
  Base register for SW1C control.
- #define SW2 0x35U
  Base register for SW2 control.
- #define SW3A 0x3cU
  Base register for SW3A control.
- #define SW3B 0x43U
  Base register for SW3B control.
- #define SW4 0x4AU
  Base register for SW4 control.
- #define VGEN1 0x6CU
  Base register for VGEN1 control.
- #define VGEN2 0x6DU
  Base register for VGEN2 control.
- #define VGEN3 0x6EU
  Base register for VGEN3 control.
- #define VGEN4 0x6FU
  Base register for VGEN4 control.
- #define VGEN5 0x70U
  Base register for VGEN5 control.
- #define VGEN6 0x71U
  Base register for VGEN6 control.

Defines for sw_pmic_mode_t

- #define SW_MODE_OFF_STBY_OFF 0x0U
  Normal Mode: OFF, Standby Mode: OFF.
- #define SW_MODE_PWM_STBY_OFF 0x1U
  Normal Mode: PWM, Standby Mode: OFF.
- #define SW_MODE_PFM_STBY_OFF 0x3U
  Normal Mode: PFM, Standby Mode: OFF.
- #define SW_MODE_APS_STBY_OFF 0x4U
  Normal Mode: APS, Standby Mode: OFF.
- #define SW_MODE_PWM_STBY_PWM 0x5U
  Normal Mode: PWM, Standby Mode: PWM.
- #define SW_MODE_PWM_STBY_APS 0x6U
  Normal Mode: PWM, Standby Mode: APS.
- #define SW_MODE_APS_STBY_APS 0x8U
  Normal Mode: APS, Standby Mode: APS.
- #define SW_MODE_APS_STBY_PFM 0xCU
  Normal Mode: APS, Standby Mode: PFM.
- #define SW_MODE_PWM_STBY_PFM 0xDU
  Normal Mode: PWM, Standby Mode: PFM.

Defines for vgen_pmic_mode_t

- #define VGEN_MODE_OFF (0x0U << 4U)
  VGEN always OFF.
- #define VGEN_MODE_ON (0x1U << 4U)
  VGEN always ON.
- #define VGEN_MODE_STBY_OFF (0x3U << 4U)
  VGEN Run: ON STBY: OFF.
- #define VGEN_MODE_LP (0x5U << 4U)
  VGEN Run: LPWR STBY: LPWR.
- #define VGEN_MODE_LP2 (0x7U << 4U)
  VGEN Run: LPWR STBY: LPWR.

Defines for sw_vmode_reg_t

- #define SW_RUN_MODE 0U
  SW run mode voltage.
- #define SW_STBY_MODE 1U
  SW standby mode voltage.
- #define SW_OFF_MODE 2U
  SW off/sleep mode voltage.

Typedefs

- typedef uint8_t pf100_vol_regs_t
  This type is used to indicate which register to address.
- typedef uint8_t sw_pmic_mode_t
  This type is used to indicate a switching regulator mode.
- typedef uint8_t vgen_pmic_mode_t
  This type is used to indicate a VGEN (LDO) regulator mode.
- typedef uint8_t sw_vmode_reg_t
  This type encodes which voltage mode register to set when calling pf100_pmic_set_voltage().

Functions

- pmic_version_t pf100_get_pmic_version (pmic_id_t id)
  This function returns the device ID and revision for the PF100 PMIC.
- sc_err_t pf100_pmic_set_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t vol_mv, uint32_t mode_to_set)
  This function sets the voltage of a corresponding voltage regulator for the PF100 PMIC.
- sc_err_t pf100_pmic_get_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t *vol_mv, uint32_t mode_to_get)
  This function gets the voltage on a corresponding voltage regulator for the PF100 PMIC.
- sc_err_t pf100_pmic_set_mode (pmic_id_t id, uint32_t pmic_reg, uint32_t mode)
  This function sets the mode of the specified regulator.
- uint32_t pf100_get_pmic_temp (pmic_id_t id)
  This function gets the current PMIC temperature as sensed by the PMIC temperature sensor.
- uint32_t pf100_set_pmic_temp_alarm (pmic_id_t id, uint32_t temp)
  This function sets the temp alarm for the PMIC in Celsius.
- sc_err_t **pf100_pmic_register_access** (pmic_id_t id, uint32_t address, sc_bool_t read_write, uint8_t *value)
- sc_bool_t pf100_pmic_irq_service (pmic_id_t id)
  This function services the interrupt for the temp alarm.

11.6 platform/drivers/pmic/pf8100/fsl_pf8100.h File Reference

Macros

- #define **I2C_WRITE** i2c_write
- #define **I2C_READ** i2c_read

Defines for pf8100_vregs_t

- #define PF8100_SW1 0x4DU
  Base register for SW1 regulator control.
- #define PF8100_SW2 0x55U
  Base register for SW2 regulator control.
- #define PF8100_SW3 0x5DU
  Base register for SW3 regulator control.
- #define PF8100_SW4 0x65U
  Base register for SW4 regulator control.
- #define PF8100_SW5 0x6DU
  Base register for SW5 regulator control.
- #define PF8100_SW6 0x75U
  Base register for SW6 regulator control.
- #define PF8100_SW7 0x7DU
  Base register for SW7 regulator control.
- #define PF8100_LDO1 0x85U
  Base register for LDO1 regulator control.
- #define PF8100_LDO2 0x8BU
  Base register for LDO2 regulator control.
- #define PF8100_LDO3 0x91U
  Base register for LDO3 regulator control.
- #define PF8100_LDO4 0x97U
  Base register for LDO4 regulator control.

Defines for sw_mode_t

- #define SW_RUN_OFF 0x0U
  Run mode: OFF.
- #define SW_RUN_PWM 0x1U
  Run mode: PWM.
- #define SW_RUN_PFM 0x2U
  Run mode: PFM.
- #define SW_RUN_ASM 0x3U
  Run mode: ASM.
- #define SW_STBY_OFF (0x0U << 2U)
  Standby mode: OFF.
- #define SW_STBY_PWM (0x1U << 2U)
  Standby mode: PWM.
- #define SW_STBY_PFM (0x2U << 2U)
  Standby mode: PFM.
- #define SW_STBY_ASM (0x3U << 2U)
  Standby mode: ASM.

Defines for ldo_mode_t

- #define RUN_OFF_STBY_OFF 0x0U
  Run mode: OFF, Standby mode: OFF.
- #define RUN_OFF_STBY_EN 0x1U
  Run mode: OFF, Standby mode: ON.
- #define RUN_EN_STBY_OFF 0x2U
  Run mode: ON, Standby mode: OFF.
- #define RUN_EN_STBY_EN 0x3U
  Run mode: ON, Standby mode: ON.

Defines for vmode_reg_t

- #define **REG_STBY_MODE** 0U
- #define **REG_RUN_MODE** 1U

Typedefs

- typedef uint8_t pf8100_vregs_t
  This type is used to indicate which register to address.
- typedef uint8_t sw_mode_t
  This type is used to indicate a switching regulator mode.
- typedef uint8_t ldo_mode_t
  This type is used to indicate an LDO regulator mode.
- typedef uint8_t vmode_reg_t
  This type is used to indicate a Switching regulator voltage setpoint.

Functions

- pmic_version_t pf8100_get_pmic_version (pmic_id_t id)
  This function returns the device ID and revision for the PF8100 PMIC.
- sc_err_t pf8100_pmic_set_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t vol_mv, uint32_t mode_to_set)
  This function sets the voltage of a corresponding voltage regulator for the PF8100 PMIC.
- sc_err_t pf8100_pmic_get_voltage (pmic_id_t id, uint32_t pmic_reg, uint32_t *vol_mv, uint32_t mode_to_get)
  This function gets the voltage on a corresponding voltage regulator for the PF8100 PMIC.
- sc_err_t pf8100_pmic_set_mode (pmic_id_t id, uint32_t pmic_reg, uint32_t mode)
  This function sets the mode of the specified regulator.
- sc_err_t pf8100_pmic_get_mode (pmic_id_t id, uint32_t pmic_reg, uint32_t *mode)
  This function gets the mode of the specified regulator.
- uint32_t pf8100_get_pmic_temp (pmic_id_t id)
  This function gets the current PMIC temperature as sensed by the PMIC temperature sensor.
- uint32_t pf8100_set_pmic_temp_alarm (pmic_id_t id, uint32_t temp)
  This function sets the temp alarm for the PMIC in Celsius.
- sc_err_t **pf8100_pmic_register_access** (pmic_id_t id, uint32_t address, sc_bool_t read_write, uint8_t *value)
- sc_bool_t pf8100_pmic_irq_service (pmic_id_t id)
  This function services the interrupt for the temp alarm.

11.7 platform/main/board.h File Reference

Header file containing the board API.

Macros

- #define **BOARD_PARM_RTN_NOT_USED** 0U
- #define **BOARD_PARM_RTN_USED** 1U
- #define **BOARD_PARM_RTN_EXTERNAL** 2U
- #define **BOARD_PARM_RTN_INTERNAL** 3U
- #define BOARD_PARM_KS1_RETENTION_DISABLE 0U

  Disable retention during KS1.
- #define BOARD_PARM_KS1_RETENTION_ENABLE 1U

  Enable retention during KS1.
- #define BOARD_PARM_KS1_ONOFF_WAKE_DISABLE 0U

  Disable ONOFF wakeup during KS1.
- #define BOARD_PARM_KS1_ONOFF_WAKE_ENABLE 1U

  Enable ONOFF wakeup during KS1.

Macros for DCD processing

- #define **DATA4**(A, V) ∗((volatile uint32_t∗)(A)) = U32(V)
- #define **SET_BIT4**(A, V) ∗((volatile uint32_t∗)(A)) |= U32(V)
- #define **CLR_BIT4**(A, V) ∗((volatile uint32_t∗)(A)) &= ∼(U32(V))
- #define **CHECK_BITS_SET4**(A, M)
- #define **CHECK_BITS_CLR4**(A, M)
- #define **CHECK_ANY_BIT_SET4**(A, M)
- #define **CHECK_ANY_BIT_CLR4**(A, M)

Macro for debug of board calls

- #define **BRD_ERR**(X)

Typedefs

- typedef uint32_t board_parm_rtn_t

  Board config parameter returns.

Enumerations

- enum board_parm_t { **BOARD_PARM_PCIE_PLL** = 0, BOARD_PARM_KS1_RESUME_USEC = 1, BOARD_PARM_KS1_RETENTION = 2, BOARD_PARM_KS1_ONOFF_WAKE = 3 }

  Board config parameter types.
- enum board_cpu_rst_ev_t { **BOARD_CPU_RESET_SELF** = 0, **BOARD_CPU_RESET_WDOG** = 1, **BOARD_CPU_RESET_LOCKUP** = 2, **BOARD_CPU_RESET_MEM_ERR** = 3 }

  Board reset event types for CPUs.
- enum board_ddr_action_t { BOARD_DDR_COLD_INIT = 0, BOARD_DDR_PERIODIC = 1, BOARD_DDR_SR_DRC_ON_ENTER = 2, BOARD_DDR_SR_DRC_ON_EXIT = 3, BOARD_DDR_SR_DRC_OFF_ENTER = 4, BOARD_DDR_SR_DRC_OFF_EXIT = 5 }

  DDR actions (power state transitions, etc.)

Functions

Initialization Functions

- void board_init (uint8_t phase)
  This function initalizes the board.
- LPUART_Type * board_get_debug_uart (uint8_t *inst, uint32_t *baud)
  This function returns the debug UART info.
- void board_config_debug_uart (sc_bool_t early_phase)
  This function initalizes the debug UART.
- void board_config_sc (sc_rm_pt_t pt_sc)
  This function configures SCU resources.
- board_parm_rtn_t board_parameter (board_parm_t parm)
  This function returns board configuration info.
- sc_bool_t board_rsrc_avail (sc_rsrc_t rsrc)
  This function returns resource availability info.
- sc_err_t board_init_ddr (sc_bool_t early, sc_bool_t ddr_initialized)
  This function initalizes DDR.
- sc_err_t board_ddr_config (bool rom_caller, board_ddr_action_t action)
  This function cinfigures the DDR.
- sc_err_t board_system_config (sc_bool_t early, sc_rm_pt_t pt_boot)
  This function allows the board file to do SCFW configuration.
- sc_bool_t board_early_cpu (sc_rsrc_t cpu)
  This function returns SC_TRUE for early CPUs.

Power Functions

- void board_set_power_mode (sc_sub_t ss, uint8_t pd, sc_pm_power_mode_t from_mode, sc_pm_power_mode_t to_mode)
  This function transitions the power state for an external board- level supply which goes to the i.MX8.
- sc_err_t board_set_voltage (sc_sub_t ss, uint32_t new_volt, sc_bool_t wait)
  This function sets the voltage for a PMIC controlled SS.
- sc_err_t board_trans_resource_power (sc_rm_idx_t idx, sc_rm_idx_t rsrc_idx, sc_pm_power_mode_t from↩ _mode, sc_pm_power_mode_t to_mode)
  This function transitions the power state for an external board- level supply which goes to a board component.

Misc Functions

- sc_err_t board_power (sc_pm_power_mode_t mode)
  This function is used to set the board power.
- sc_err_t board_reset (sc_pm_reset_type_t type, sc_pm_reset_reason_t reason)
  This function is used to reset the system.
- void board_cpu_reset (sc_rsrc_t resource, board_cpu_rst_ev_t reset_event)
  This function is called when a CPU encounters a reset event.
- void board_panic (sc_dsc_t dsc)
  This function is called when a DSC reports a panic temp alarm.
- void board_fault (sc_bool_t restarted)
  This function is called when a fault is detected or the SCFW returns from main().
- void board_security_violation (void)
  This function is called when a security violation is reported by the SECO or SNVS.
- sc_bool_t board_get_button_status (void)
  This function is used to return the current status of the ON/OFF button.
- sc_err_t board_set_control (sc_rsrc_t resource, sc_rm_idx_t idx, sc_rm_idx_t rsrc_idx, uint32_t ctrl, uint32_t val)

This function sets a miscellaneous control value.

- sc_err_t board_get_control (sc_rsrc_t resource, sc_rm_idx_t idx, sc_rm_idx_t rsrc_idx, uint32_t ctrl, uint32_t *val)

  This function gets a miscellaneous control value.
- void PMIC_IRQHandler (void)

  Interrupt handler for the PMIC.
- void SNVS_Button_IRQHandler (void)

  Interrupt handler for the SNVS button.

Variables

- const sc_rm_idx_t board_num_rsrc

  External variable for accessing the number of board resources.
- const sc_rsrc_map_t board_rsrc_map [BRD_NUM_RSRC_BRD]

  External variable for accessing the board resource map.
- const uint32_t board_ddr_period_ms

  External variable for specing DDR periodic training.

11.7.1 Detailed Description

Header file containing the board API.

11.8 platform/main/ipc.h File Reference

Header file for the IPC implementation.

Functions

- sc_err_t sc_ipc_open (sc_ipc_t *ipc, sc_ipc_id_t id)

  This function opens an IPC channel.
- void sc_ipc_close (sc_ipc_t ipc)

  This function closes an IPC channel.
- void sc_ipc_read (sc_ipc_t ipc, void *data)

  This function reads a message from an IPC channel.
- void sc_ipc_write (sc_ipc_t ipc, const void *data)

  This function writes a message to an IPC channel.

11.8.1 Detailed Description

Header file for the IPC implementation.

11.8.2 Function Documentation

11.8.2.1 sc_ipc_open()

```
sc_err_t sc_ipc_open (
            sc_ipc_t * ipc,
            sc_ipc_id_t id )
```

This function opens an IPC channel.

Parameters

out	*ipc*	return pointer for ipc handle
in	*id*	id of channel to open

Returns

Returns an error code (SC_ERR_NONE = success, SC_ERR_IPC otherwise).

The *id* parameter is implementation specific. Could be an MU address, pointer to a driver path, channel index, etc.

11.8.2.2 sc_ipc_close()

```
void sc_ipc_close (
            sc_ipc_t ipc )
```

This function closes an IPC channel.

Parameters

in	*ipc*	id of channel to close

11.8.2.3 sc_ipc_read()

```
void sc_ipc_read (
            sc_ipc_t ipc,
            void * data )
```

This function reads a message from an IPC channel.

Parameters

in	*ipc*	id of channel read from
out	*data*	pointer to message buffer to read

This function will block if no message is available to be read.

11.8.2.4 sc_ipc_write()

```
void sc_ipc_write (
            sc_ipc_t ipc,
            const void * data )
```

This function writes a message to an IPC channel.

Parameters

in	*ipc*	id of channel to write to
in	*data*	pointer to message buffer to write

This function will block if the outgoing buffer is full.

11.9 platform/main/types.h File Reference

Header file containing types used across multiple service APIs.

Macros

- #define SC_C_TEMP 0U

  Defnes for sc_ctrl_t.
- #define **SC_C_TEMP_HI** 1U
- #define **SC_C_TEMP_LOW** 2U
- #define **SC_C_PXL_LINK_MST1_ADDR** 3U
- #define **SC_C_PXL_LINK_MST2_ADDR** 4U
- #define **SC_C_PXL_LINK_MST_ENB** 5U
- #define **SC_C_PXL_LINK_MST1_ENB** 6U
- #define **SC_C_PXL_LINK_MST2_ENB** 7U
- #define **SC_C_PXL_LINK_SLV1_ADDR** 8U
- #define **SC_C_PXL_LINK_SLV2_ADDR** 9U
- #define **SC_C_PXL_LINK_MST_VLD** 10U
- #define **SC_C_PXL_LINK_MST1_VLD** 11U
- #define **SC_C_PXL_LINK_MST2_VLD** 12U
- #define **SC_C_SINGLE_MODE** 13U
- #define **SC_C_ID** 14U

- #define **SC_C_PXL_CLK_POLARITY** 15U
- #define **SC_C_LINESTATE** 16U
- #define **SC_C_PCIE_G_RST** 17U
- #define **SC_C_PCIE_BUTTON_RST** 18U
- #define **SC_C_PCIE_PERST** 19U
- #define **SC_C_PHY_RESET** 20U
- #define **SC_C_PXL_LINK_RATE_CORRECTION** 21U
- #define **SC_C_PANIC** 22U
- #define **SC_C_PRIORITY_GROUP** 23U
- #define **SC_C_TXCLK** 24U
- #define **SC_C_CLKDIV** 25U
- #define **SC_C_DISABLE_50** 26U
- #define **SC_C_DISABLE_125** 27U
- #define **SC_C_SEL_125** 28U
- #define **SC_C_MODE** 29U
- #define **SC_C_SYNC_CTRL0** 30U
- #define **SC_C_KACHUNK_CNT** 31U
- #define **SC_C_KACHUNK_SEL** 32U
- #define **SC_C_SYNC_CTRL1** 33U
- #define **SC_C_DPI_RESET** 34U
- #define **SC_C_MIPI_RESET** 35U
- #define **SC_C_DUAL_MODE** 36U
- #define **SC_C_VOLTAGE** 37U
- #define **SC_C_PXL_LINK_SEL** 38U
- #define **SC_C_OFS_SEL** 39U
- #define **SC_C_OFS_AUDIO** 40U
- #define **SC_C_OFS_PERIPH** 41U
- #define **SC_C_OFS_IRQ** 42U
- #define **SC_C_RST0** 43U
- #define **SC_C_RST1** 44U
- #define **SC_C_SEL0** 45U
- #define **SC_C_CALIB0** 46U
- #define **SC_C_CALIB1** 47U
- #define **SC_C_CALIB2** 48U
- #define **SC_C_IPG_DEBUG** 49U
- #define **SC_C_IPG_DOZE** 50U
- #define **SC_C_IPG_WAIT** 51U
- #define **SC_C_IPG_STOP** 52U
- #define **SC_C_IPG_STOP_MODE** 53U
- #define **SC_C_IPG_STOP_ACK** 54U
- #define **SC_C_SYNC_CTRL** 55U
- #define **SC_C_LAST** 56U
- #define SC_P_ALL ((sc_pad_t) UINT16_MAX)

  All pads.

Defines for common frequencies

- #define SC_32KHZ 32768U

  32KHz
- #define SC_10MHZ 10000000U

10MHz

- #define SC_20MHZ 20000000U

  20MHz
- #define SC_25MHZ 25000000U

  25MHz
- #define SC_27MHZ 27000000U

  27MHz
- #define SC_40MHZ 40000000U

  40MHz
- #define SC_45MHZ 45000000U

  45MHz
- #define SC_50MHZ 50000000U

  50MHz
- #define SC_60MHZ 60000000U

  60MHz
- #define SC_66MHZ 66666666U

  66MHz
- #define SC_74MHZ 74250000U

  74.25MHz
- #define SC_80MHZ 80000000U

  80MHz
- #define SC_83MHZ 83333333U

  83MHz
- #define SC_84MHZ 84375000U

  84.37MHz
- #define SC_100MHZ 100000000U

  100MHz
- #define SC_125MHZ 125000000U

  125MHz
- #define SC_133MHZ 133333333U

  133MHz
- #define SC_135MHZ 135000000U

  135MHz
- #define SC_150MHZ 150000000U

  150MHz
- #define SC_160MHZ 160000000U

  160MHz
- #define SC_166MHZ 166666666U

  166MHz
- #define SC_175MHZ 175000000U

  175MHz
- #define SC_180MHZ 180000000U

  180MHz
- #define SC_200MHZ 200000000U

  200MHz
- #define SC_250MHZ 250000000U

  250MHz
- #define SC_266MHZ 266666666U

  266MHz
- #define SC_300MHZ 300000000U

  300MHz
- #define SC_312MHZ 312500000U

  312.5MHZ
- #define SC_320MHZ 320000000U

  320MHz

- #define SC_325MHZ 325000000U
  325MHz
- #define SC_333MHZ 333333333U
  333MHz
- #define SC_350MHZ 350000000U
  350MHz
- #define SC_372MHZ 372000000U
  372MHz
- #define SC_375MHZ 375000000U
  375MHz
- #define SC_400MHZ 400000000U
  400MHz
- #define SC_500MHZ 500000000U
  500MHz
- #define SC_594MHZ 594000000U
  594MHz
- #define SC_625MHZ 625000000U
  625MHz
- #define SC_640MHZ 640000000U
  640MHz
- #define SC_648MHZ 648000000U
  648MHz
- #define SC_650MHZ 650000000U
  650MHz
- #define SC_667MHZ 666666667U
  667MHz
- #define SC_675MHZ 675000000U
  675MHz
- #define SC_700MHZ 700000000U
  700MHz
- #define SC_720MHZ 720000000U
  720MHz
- #define SC_750MHZ 750000000U
  750MHz
- #define SC_753MHZ 753000000U
  753MHz
- #define SC_793MHZ 793000000U
  793MHz
- #define SC_800MHZ 800000000U
  800MHz
- #define SC_850MHZ 850000000U
  850MHz
- #define SC_858MHZ 858000000U
  858MHz
- #define SC_900MHZ 900000000U
  900MHz
- #define SC_953MHZ 953000000U
  953MHz
- #define SC_963MHZ 963000000U
  963MHz
- #define SC_1000MHZ 1000000000U
  1GHz
- #define SC_1060MHZ 1060000000U
  1.06GHz
- #define SC_1068MHZ 1068000000U

1.068GHz

- #define SC_1121MHZ 1121000000U

  1.121GHz
- #define SC_1173MHZ 1173000000U

  1.173GHz
- #define SC_1188MHZ 1188000000U

  1.188GHz
- #define SC_1260MHZ 1260000000U

  1.26GHz
- #define SC_1278MHZ 1278000000U

  1.278GHz
- #define SC_1280MHZ 1280000000U

  1.28GHz
- #define SC_1300MHZ 1300000000U

  1.3GHz
- #define SC_1313MHZ 1313000000U

  1.313GHz
- #define SC_1345MHZ 1345000000U

  1.345GHz
- #define SC_1400MHZ 1400000000U

  1.4GHz
- #define SC_1500MHZ 1500000000U

  1.5GHz
- #define SC_1600MHZ 1600000000U

  1.6GHz
- #define SC_1800MHZ 1800000000U

  1.8GHz
- #define SC_2000MHZ 2000000000U

  2.0GHz
- #define SC_2112MHZ 2112000000U

  2.12GHz

Defines for 24M related frequencies

- #define SC_8MHZ 8000000U

  8MHz
- #define SC_12MHZ 12000000U

  12MHz
- #define SC_19MHZ 19800000U

  19.8MHz
- #define SC_24MHZ 24000000U

  24MHz
- #define SC_48MHZ 48000000U

  48MHz
- #define SC_120MHZ 120000000U

  120MHz
- #define SC_132MHZ 132000000U

  132MHz
- #define SC_144MHZ 144000000U

  144MHz
- #define SC_192MHZ 192000000U

  192MHz
- #define SC_211MHZ 211200000U

  211.2MHz

- #define SC_240MHZ 240000000U
  240MHz
- #define SC_264MHZ 264000000U
  264MHz
- #define SC_352MHZ 352000000U
  352MHz
- #define SC_360MHZ 360000000U
  360MHz
- #define SC_384MHZ 384000000U
  384MHz
- #define SC_396MHZ 396000000U
  396MHz
- #define SC_432MHZ 432000000U
  432MHz
- #define SC_480MHZ 480000000U
  480MHz
- #define SC_600MHZ 600000000U
  600MHz
- #define SC_744MHZ 744000000U
  744MHz
- #define SC_792MHZ 792000000U
  792MHz
- #define SC_864MHZ 864000000U
  864MHz
- #define SC_960MHZ 960000000U
  960MHz
- #define SC_1056MHZ 1056000000U
  1056MHz
- #define SC_1104MHZ 1104000000U
  1104MHz
- #define SC_1200MHZ 1200000000U
  1.2GHz
- #define SC_1464MHZ 1464000000U
  1.464GHz
- #define SC_2400MHZ 2400000000U
  2.4GHz

Defines for A/V related frequencies

- #define SC_62MHZ 62937500U
  62.9375MHz
- #define SC_755MHZ 755250000U
  755.25MHz

Defines for type widths

- #define SC_FADDR_W 36U
  Width of sc_faddr_t.
- #define SC_BOOL_W 1U
  Width of sc_bool_t.
- #define SC_ERR_W 4U
  Width of sc_err_t.
- #define SC_RSRC_W 10U

Width of sc_rsrc_t.

- #define SC_CTRL_W 6U
  Width of sc_ctrl_t.

Defines for sc_bool_t

- #define SC_FALSE ((sc_bool_t) 0U)
  False.
- #define SC_TRUE ((sc_bool_t) 1U)
  True.

Defines for sc_err_t.

- #define SC_ERR_NONE 0U
  Success.
- #define SC_ERR_VERSION 1U
  Incompatible API version.
- #define SC_ERR_CONFIG 2U
  Configuration error.
- #define SC_ERR_PARM 3U
  Bad parameter.
- #define SC_ERR_NOACCESS 4U
  Permission error (no access)
- #define SC_ERR_LOCKED 5U
  Permission error (locked)
- #define SC_ERR_UNAVAILABLE 6U
  Unavailable (out of resources)
- #define SC_ERR_NOTFOUND 7U
  Not found.
- #define SC_ERR_NOPOWER 8U
  No power.
- #define SC_ERR_IPC 9U
  Generic IPC error.
- #define SC_ERR_BUSY 10U
  Resource is currently busy/active.
- #define SC_ERR_FAIL 11U
  General I/O failure.
- #define **SC_ERR_LAST** 12U

Defines for sc_rsrc_t.

- #define **SC_R_A53** 0U
- #define **SC_R_A53_0** 1U
- #define **SC_R_A53_1** 2U
- #define **SC_R_A53_2** 3U
- #define **SC_R_A53_3** 4U
- #define **SC_R_A72** 5U
- #define **SC_R_A72_0** 6U
- #define **SC_R_A72_1** 7U
- #define **SC_R_A72_2** 8U
- #define **SC_R_A72_3** 9U
- #define **SC_R_CCI** 10U
- #define **SC_R_DB** 11U
- #define **SC_R_DRC_0** 12U

- #define **SC_R_DRC_1** 13U
- #define **SC_R_GIC_SMMU** 14U
- #define **SC_R_IRQSTR_M4_0** 15U
- #define **SC_R_IRQSTR_M4_1** 16U
- #define **SC_R_SMMU** 17U
- #define **SC_R_GIC** 18U
- #define **SC_R_DC_0_BLIT0** 19U
- #define **SC_R_DC_0_BLIT1** 20U
- #define **SC_R_DC_0_BLIT2** 21U
- #define **SC_R_DC_0_BLIT_OUT** 22U
- #define **SC_R_DC_0_CAPTURE0** 23U
- #define **SC_R_DC_0_CAPTURE1** 24U
- #define **SC_R_DC_0_WARP** 25U
- #define **SC_R_DC_0_INTEGRAL0** 26U
- #define **SC_R_DC_0_INTEGRAL1** 27U
- #define **SC_R_DC_0_VIDEO0** 28U
- #define **SC_R_DC_0_VIDEO1** 29U
- #define **SC_R_DC_0_FRAC0** 30U
- #define **SC_R_DC_0_FRAC1** 31U
- #define **SC_R_DC_0** 32U
- #define **SC_R_GPU_2_PID0** 33U
- #define **SC_R_DC_0_PLL_0** 34U
- #define **SC_R_DC_0_PLL_1** 35U
- #define **SC_R_DC_1_BLIT0** 36U
- #define **SC_R_DC_1_BLIT1** 37U
- #define **SC_R_DC_1_BLIT2** 38U
- #define **SC_R_DC_1_BLIT_OUT** 39U
- #define **SC_R_DC_1_CAPTURE0** 40U
- #define **SC_R_DC_1_CAPTURE1** 41U
- #define **SC_R_DC_1_WARP** 42U
- #define **SC_R_DC_1_INTEGRAL0** 43U
- #define **SC_R_DC_1_INTEGRAL1** 44U
- #define **SC_R_DC_1_VIDEO0** 45U
- #define **SC_R_DC_1_VIDEO1** 46U
- #define **SC_R_DC_1_FRAC0** 47U
- #define **SC_R_DC_1_FRAC1** 48U
- #define **SC_R_DC_1** 49U
- #define **SC_R_GPU_3_PID0** 50U
- #define **SC_R_DC_1_PLL_0** 51U
- #define **SC_R_DC_1_PLL_1** 52U
- #define **SC_R_SPI_0** 53U
- #define **SC_R_SPI_1** 54U
- #define **SC_R_SPI_2** 55U
- #define **SC_R_SPI_3** 56U
- #define **SC_R_UART_0** 57U
- #define **SC_R_UART_1** 58U
- #define **SC_R_UART_2** 59U
- #define **SC_R_UART_3** 60U
- #define **SC_R_UART_4** 61U
- #define **SC_R_EMVSIM_0** 62U
- #define **SC_R_EMVSIM_1** 63U
- #define **SC_R_DMA_0_CH0** 64U
- #define **SC_R_DMA_0_CH1** 65U
- #define **SC_R_DMA_0_CH2** 66U
- #define **SC_R_DMA_0_CH3** 67U
- #define **SC_R_DMA_0_CH4** 68U
- #define **SC_R_DMA_0_CH5** 69U
- #define **SC_R_DMA_0_CH6** 70U
- #define **SC_R_DMA_0_CH7** 71U
- #define **SC_R_DMA_0_CH8** 72U

- #define **SC_R_DMA_0_CH9** 73U
- #define **SC_R_DMA_0_CH10** 74U
- #define **SC_R_DMA_0_CH11** 75U
- #define **SC_R_DMA_0_CH12** 76U
- #define **SC_R_DMA_0_CH13** 77U
- #define **SC_R_DMA_0_CH14** 78U
- #define **SC_R_DMA_0_CH15** 79U
- #define **SC_R_DMA_0_CH16** 80U
- #define **SC_R_DMA_0_CH17** 81U
- #define **SC_R_DMA_0_CH18** 82U
- #define **SC_R_DMA_0_CH19** 83U
- #define **SC_R_DMA_0_CH20** 84U
- #define **SC_R_DMA_0_CH21** 85U
- #define **SC_R_DMA_0_CH22** 86U
- #define **SC_R_DMA_0_CH23** 87U
- #define **SC_R_DMA_0_CH24** 88U
- #define **SC_R_DMA_0_CH25** 89U
- #define **SC_R_DMA_0_CH26** 90U
- #define **SC_R_DMA_0_CH27** 91U
- #define **SC_R_DMA_0_CH28** 92U
- #define **SC_R_DMA_0_CH29** 93U
- #define **SC_R_DMA_0_CH30** 94U
- #define **SC_R_DMA_0_CH31** 95U
- #define **SC_R_I2C_0** 96U
- #define **SC_R_I2C_1** 97U
- #define **SC_R_I2C_2** 98U
- #define **SC_R_I2C_3** 99U
- #define **SC_R_I2C_4** 100U
- #define **SC_R_ADC_0** 101U
- #define **SC_R_ADC_1** 102U
- #define **SC_R_FTM_0** 103U
- #define **SC_R_FTM_1** 104U
- #define **SC_R_CAN_0** 105U
- #define **SC_R_CAN_1** 106U
- #define **SC_R_CAN_2** 107U
- #define **SC_R_DMA_1_CH0** 108U
- #define **SC_R_DMA_1_CH1** 109U
- #define **SC_R_DMA_1_CH2** 110U
- #define **SC_R_DMA_1_CH3** 111U
- #define **SC_R_DMA_1_CH4** 112U
- #define **SC_R_DMA_1_CH5** 113U
- #define **SC_R_DMA_1_CH6** 114U
- #define **SC_R_DMA_1_CH7** 115U
- #define **SC_R_DMA_1_CH8** 116U
- #define **SC_R_DMA_1_CH9** 117U
- #define **SC_R_DMA_1_CH10** 118U
- #define **SC_R_DMA_1_CH11** 119U
- #define **SC_R_DMA_1_CH12** 120U
- #define **SC_R_DMA_1_CH13** 121U
- #define **SC_R_DMA_1_CH14** 122U
- #define **SC_R_DMA_1_CH15** 123U
- #define **SC_R_DMA_1_CH16** 124U
- #define **SC_R_DMA_1_CH17** 125U
- #define **SC_R_DMA_1_CH18** 126U
- #define **SC_R_DMA_1_CH19** 127U
- #define **SC_R_DMA_1_CH20** 128U
- #define **SC_R_DMA_1_CH21** 129U
- #define **SC_R_DMA_1_CH22** 130U
- #define **SC_R_DMA_1_CH23** 131U
- #define **SC_R_DMA_1_CH24** 132U

- #define **SC_R_DMA_1_CH25** 133U
- #define **SC_R_DMA_1_CH26** 134U
- #define **SC_R_DMA_1_CH27** 135U
- #define **SC_R_DMA_1_CH28** 136U
- #define **SC_R_DMA_1_CH29** 137U
- #define **SC_R_DMA_1_CH30** 138U
- #define **SC_R_DMA_1_CH31** 139U
- #define **SC_R_UNUSED1** 140U
- #define **SC_R_UNUSED2** 141U
- #define **SC_R_UNUSED3** 142U
- #define **SC_R_UNUSED4** 143U
- #define **SC_R_GPU_0_PID0** 144U
- #define **SC_R_GPU_0_PID1** 145U
- #define **SC_R_GPU_0_PID2** 146U
- #define **SC_R_GPU_0_PID3** 147U
- #define **SC_R_GPU_1_PID0** 148U
- #define **SC_R_GPU_1_PID1** 149U
- #define **SC_R_GPU_1_PID2** 150U
- #define **SC_R_GPU_1_PID3** 151U
- #define **SC_R_PCIE_A** 152U
- #define **SC_R_SERDES_0** 153U
- #define **SC_R_MATCH_0** 154U
- #define **SC_R_MATCH_1** 155U
- #define **SC_R_MATCH_2** 156U
- #define **SC_R_MATCH_3** 157U
- #define **SC_R_MATCH_4** 158U
- #define **SC_R_MATCH_5** 159U
- #define **SC_R_MATCH_6** 160U
- #define **SC_R_MATCH_7** 161U
- #define **SC_R_MATCH_8** 162U
- #define **SC_R_MATCH_9** 163U
- #define **SC_R_MATCH_10** 164U
- #define **SC_R_MATCH_11** 165U
- #define **SC_R_MATCH_12** 166U
- #define **SC_R_MATCH_13** 167U
- #define **SC_R_MATCH_14** 168U
- #define **SC_R_PCIE_B** 169U
- #define **SC_R_SATA_0** 170U
- #define **SC_R_SERDES_1** 171U
- #define **SC_R_HSIO_GPIO** 172U
- #define **SC_R_MATCH_15** 173U
- #define **SC_R_MATCH_16** 174U
- #define **SC_R_MATCH_17** 175U
- #define **SC_R_MATCH_18** 176U
- #define **SC_R_MATCH_19** 177U
- #define **SC_R_MATCH_20** 178U
- #define **SC_R_MATCH_21** 179U
- #define **SC_R_MATCH_22** 180U
- #define **SC_R_MATCH_23** 181U
- #define **SC_R_MATCH_24** 182U
- #define **SC_R_MATCH_25** 183U
- #define **SC_R_MATCH_26** 184U
- #define **SC_R_MATCH_27** 185U
- #define **SC_R_MATCH_28** 186U
- #define **SC_R_LCD_0** 187U
- #define **SC_R_LCD_0_PWM_0** 188U
- #define **SC_R_LCD_0_I2C_0** 189U
- #define **SC_R_LCD_0_I2C_1** 190U
- #define **SC_R_PWM_0** 191U
- #define **SC_R_PWM_1** 192U

- #define **SC_R_PWM_2** 193U
- #define **SC_R_PWM_3** 194U
- #define **SC_R_PWM_4** 195U
- #define **SC_R_PWM_5** 196U
- #define **SC_R_PWM_6** 197U
- #define **SC_R_PWM_7** 198U
- #define **SC_R_GPIO_0** 199U
- #define **SC_R_GPIO_1** 200U
- #define **SC_R_GPIO_2** 201U
- #define **SC_R_GPIO_3** 202U
- #define **SC_R_GPIO_4** 203U
- #define **SC_R_GPIO_5** 204U
- #define **SC_R_GPIO_6** 205U
- #define **SC_R_GPIO_7** 206U
- #define **SC_R_GPT_0** 207U
- #define **SC_R_GPT_1** 208U
- #define **SC_R_GPT_2** 209U
- #define **SC_R_GPT_3** 210U
- #define **SC_R_GPT_4** 211U
- #define **SC_R_KPP** 212U
- #define **SC_R_MU_0A** 213U
- #define **SC_R_MU_1A** 214U
- #define **SC_R_MU_2A** 215U
- #define **SC_R_MU_3A** 216U
- #define **SC_R_MU_4A** 217U
- #define **SC_R_MU_5A** 218U
- #define **SC_R_MU_6A** 219U
- #define **SC_R_MU_7A** 220U
- #define **SC_R_MU_8A** 221U
- #define **SC_R_MU_9A** 222U
- #define **SC_R_MU_10A** 223U
- #define **SC_R_MU_11A** 224U
- #define **SC_R_MU_12A** 225U
- #define **SC_R_MU_13A** 226U
- #define **SC_R_MU_5B** 227U
- #define **SC_R_MU_6B** 228U
- #define **SC_R_MU_7B** 229U
- #define **SC_R_MU_8B** 230U
- #define **SC_R_MU_9B** 231U
- #define **SC_R_MU_10B** 232U
- #define **SC_R_MU_11B** 233U
- #define **SC_R_MU_12B** 234U
- #define **SC_R_MU_13B** 235U
- #define **SC_R_ROM_0** 236U
- #define **SC_R_FSPI_0** 237U
- #define **SC_R_FSPI_1** 238U
- #define **SC_R_IEE** 239U
- #define **SC_R_IEE_R0** 240U
- #define **SC_R_IEE_R1** 241U
- #define **SC_R_IEE_R2** 242U
- #define **SC_R_IEE_R3** 243U
- #define **SC_R_IEE_R4** 244U
- #define **SC_R_IEE_R5** 245U
- #define **SC_R_IEE_R6** 246U
- #define **SC_R_IEE_R7** 247U
- #define **SC_R_SDHC_0** 248U
- #define **SC_R_SDHC_1** 249U
- #define **SC_R_SDHC_2** 250U
- #define **SC_R_ENET_0** 251U
- #define **SC_R_ENET_1** 252U

- #define **SC_R_MLB_0** 253U
- #define **SC_R_DMA_2_CH0** 254U
- #define **SC_R_DMA_2_CH1** 255U
- #define **SC_R_DMA_2_CH2** 256U
- #define **SC_R_DMA_2_CH3** 257U
- #define **SC_R_DMA_2_CH4** 258U
- #define **SC_R_USB_0** 259U
- #define **SC_R_USB_1** 260U
- #define **SC_R_USB_0_PHY** 261U
- #define **SC_R_USB_2** 262U
- #define **SC_R_USB_2_PHY** 263U
- #define **SC_R_DTCP** 264U
- #define **SC_R_NAND** 265U
- #define **SC_R_LVDS_0** 266U
- #define **SC_R_LVDS_0_PWM_0** 267U
- #define **SC_R_LVDS_0_I2C_0** 268U
- #define **SC_R_LVDS_0_I2C_1** 269U
- #define **SC_R_LVDS_1** 270U
- #define **SC_R_LVDS_1_PWM_0** 271U
- #define **SC_R_LVDS_1_I2C_0** 272U
- #define **SC_R_LVDS_1_I2C_1** 273U
- #define **SC_R_LVDS_2** 274U
- #define **SC_R_LVDS_2_PWM_0** 275U
- #define **SC_R_LVDS_2_I2C_0** 276U
- #define **SC_R_LVDS_2_I2C_1** 277U
- #define **SC_R_M4_0_PID0** 278U
- #define **SC_R_M4_0_PID1** 279U
- #define **SC_R_M4_0_PID2** 280U
- #define **SC_R_M4_0_PID3** 281U
- #define **SC_R_M4_0_PID4** 282U
- #define **SC_R_M4_0_RGPIO** 283U
- #define **SC_R_M4_0_SEMA42** 284U
- #define **SC_R_M4_0_TPM** 285U
- #define **SC_R_M4_0_PIT** 286U
- #define **SC_R_M4_0_UART** 287U
- #define **SC_R_M4_0_I2C** 288U
- #define **SC_R_M4_0_INTMUX** 289U
- #define **SC_R_M4_0_SIM** 290U
- #define **SC_R_M4_0_WDOG** 291U
- #define **SC_R_M4_0_MU_0B** 292U
- #define **SC_R_M4_0_MU_0A0** 293U
- #define **SC_R_M4_0_MU_0A1** 294U
- #define **SC_R_M4_0_MU_0A2** 295U
- #define **SC_R_M4_0_MU_0A3** 296U
- #define **SC_R_M4_0_MU_1A** 297U
- #define **SC_R_M4_1_PID0** 298U
- #define **SC_R_M4_1_PID1** 299U
- #define **SC_R_M4_1_PID2** 300U
- #define **SC_R_M4_1_PID3** 301U
- #define **SC_R_M4_1_PID4** 302U
- #define **SC_R_M4_1_RGPIO** 303U
- #define **SC_R_M4_1_SEMA42** 304U
- #define **SC_R_M4_1_TPM** 305U
- #define **SC_R_M4_1_PIT** 306U
- #define **SC_R_M4_1_UART** 307U
- #define **SC_R_M4_1_I2C** 308U
- #define **SC_R_M4_1_INTMUX** 309U
- #define **SC_R_M4_1_SIM** 310U
- #define **SC_R_M4_1_WDOG** 311U
- #define **SC_R_M4_1_MU_0B** 312U

- #define **SC_R_M4_1_MU_0A0** 313U
- #define **SC_R_M4_1_MU_0A1** 314U
- #define **SC_R_M4_1_MU_0A2** 315U
- #define **SC_R_M4_1_MU_0A3** 316U
- #define **SC_R_M4_1_MU_1A** 317U
- #define **SC_R_SAI_0** 318U
- #define **SC_R_SAI_1** 319U
- #define **SC_R_SAI_2** 320U
- #define **SC_R_IRQSTR_SCU2** 321U
- #define **SC_R_IRQSTR_DSP** 322U
- #define **SC_R_ELCDIF_PLL** 323U
- #define **SC_R_OCRAM** 324U
- #define **SC_R_AUDIO_PLL_0** 325U
- #define **SC_R_PI_0** 326U
- #define **SC_R_PI_0_PWM_0** 327U
- #define **SC_R_PI_0_PWM_1** 328U
- #define **SC_R_PI_0_I2C_0** 329U
- #define **SC_R_PI_0_PLL** 330U
- #define **SC_R_PI_1** 331U
- #define **SC_R_PI_1_PWM_0** 332U
- #define **SC_R_PI_1_PWM_1** 333U
- #define **SC_R_PI_1_I2C_0** 334U
- #define **SC_R_PI_1_PLL** 335U
- #define **SC_R_SC_PID0** 336U
- #define **SC_R_SC_PID1** 337U
- #define **SC_R_SC_PID2** 338U
- #define **SC_R_SC_PID3** 339U
- #define **SC_R_SC_PID4** 340U
- #define **SC_R_SC_SEMA42** 341U
- #define **SC_R_SC_TPM** 342U
- #define **SC_R_SC_PIT** 343U
- #define **SC_R_SC_UART** 344U
- #define **SC_R_SC_I2C** 345U
- #define **SC_R_SC_MU_0B** 346U
- #define **SC_R_SC_MU_0A0** 347U
- #define **SC_R_SC_MU_0A1** 348U
- #define **SC_R_SC_MU_0A2** 349U
- #define **SC_R_SC_MU_0A3** 350U
- #define **SC_R_SC_MU_1A** 351U
- #define **SC_R_SYSCNT_RD** 352U
- #define **SC_R_SYSCNT_CMP** 353U
- #define **SC_R_DEBUG** 354U
- #define **SC_R_SYSTEM** 355U
- #define **SC_R_SNVS** 356U
- #define **SC_R_OTP** 357U
- #define **SC_R_VPU_PID0** 358U
- #define **SC_R_VPU_PID1** 359U
- #define **SC_R_VPU_PID2** 360U
- #define **SC_R_VPU_PID3** 361U
- #define **SC_R_VPU_PID4** 362U
- #define **SC_R_VPU_PID5** 363U
- #define **SC_R_VPU_PID6** 364U
- #define **SC_R_VPU_PID7** 365U
- #define **SC_R_VPU_UART** 366U
- #define **SC_R_VPUCORE** 367U
- #define **SC_R_VPUCORE_0** 368U
- #define **SC_R_VPUCORE_1** 369U
- #define **SC_R_VPUCORE_2** 370U
- #define **SC_R_VPUCORE_3** 371U
- #define **SC_R_DMA_4_CH0** 372U

- #define **SC_R_DMA_4_CH1** 373U
- #define **SC_R_DMA_4_CH2** 374U
- #define **SC_R_DMA_4_CH3** 375U
- #define **SC_R_DMA_4_CH4** 376U
- #define **SC_R_ISI_CH0** 377U
- #define **SC_R_ISI_CH1** 378U
- #define **SC_R_ISI_CH2** 379U
- #define **SC_R_ISI_CH3** 380U
- #define **SC_R_ISI_CH4** 381U
- #define **SC_R_ISI_CH5** 382U
- #define **SC_R_ISI_CH6** 383U
- #define **SC_R_ISI_CH7** 384U
- #define **SC_R_MJPEG_DEC_S0** 385U
- #define **SC_R_MJPEG_DEC_S1** 386U
- #define **SC_R_MJPEG_DEC_S2** 387U
- #define **SC_R_MJPEG_DEC_S3** 388U
- #define **SC_R_MJPEG_ENC_S0** 389U
- #define **SC_R_MJPEG_ENC_S1** 390U
- #define **SC_R_MJPEG_ENC_S2** 391U
- #define **SC_R_MJPEG_ENC_S3** 392U
- #define **SC_R_MIPI_0** 393U
- #define **SC_R_MIPI_0_PWM_0** 394U
- #define **SC_R_MIPI_0_I2C_0** 395U
- #define **SC_R_MIPI_0_I2C_1** 396U
- #define **SC_R_MIPI_1** 397U
- #define **SC_R_MIPI_1_PWM_0** 398U
- #define **SC_R_MIPI_1_I2C_0** 399U
- #define **SC_R_MIPI_1_I2C_1** 400U
- #define **SC_R_CSI_0** 401U
- #define **SC_R_CSI_0_PWM_0** 402U
- #define **SC_R_CSI_0_I2C_0** 403U
- #define **SC_R_CSI_1** 404U
- #define **SC_R_CSI_1_PWM_0** 405U
- #define **SC_R_CSI_1_I2C_0** 406U
- #define **SC_R_HDMI** 407U
- #define **SC_R_HDMI_I2S** 408U
- #define **SC_R_HDMI_I2C_0** 409U
- #define **SC_R_HDMI_PLL_0** 410U
- #define **SC_R_HDMI_RX** 411U
- #define **SC_R_HDMI_RX_BYPASS** 412U
- #define **SC_R_HDMI_RX_I2C_0** 413U
- #define **SC_R_ASRC_0** 414U
- #define **SC_R_ESAI_0** 415U
- #define **SC_R_SPDIF_0** 416U
- #define **SC_R_SPDIF_1** 417U
- #define **SC_R_SAI_3** 418U
- #define **SC_R_SAI_4** 419U
- #define **SC_R_SAI_5** 420U
- #define **SC_R_GPT_5** 421U
- #define **SC_R_GPT_6** 422U
- #define **SC_R_GPT_7** 423U
- #define **SC_R_GPT_8** 424U
- #define **SC_R_GPT_9** 425U
- #define **SC_R_GPT_10** 426U
- #define **SC_R_DMA_2_CH5** 427U
- #define **SC_R_DMA_2_CH6** 428U
- #define **SC_R_DMA_2_CH7** 429U
- #define **SC_R_DMA_2_CH8** 430U
- #define **SC_R_DMA_2_CH9** 431U
- #define **SC_R_DMA_2_CH10** 432U

- #define **SC_R_DMA_2_CH11** 433U
- #define **SC_R_DMA_2_CH12** 434U
- #define **SC_R_DMA_2_CH13** 435U
- #define **SC_R_DMA_2_CH14** 436U
- #define **SC_R_DMA_2_CH15** 437U
- #define **SC_R_DMA_2_CH16** 438U
- #define **SC_R_DMA_2_CH17** 439U
- #define **SC_R_DMA_2_CH18** 440U
- #define **SC_R_DMA_2_CH19** 441U
- #define **SC_R_DMA_2_CH20** 442U
- #define **SC_R_DMA_2_CH21** 443U
- #define **SC_R_DMA_2_CH22** 444U
- #define **SC_R_DMA_2_CH23** 445U
- #define **SC_R_DMA_2_CH24** 446U
- #define **SC_R_DMA_2_CH25** 447U
- #define **SC_R_DMA_2_CH26** 448U
- #define **SC_R_DMA_2_CH27** 449U
- #define **SC_R_DMA_2_CH28** 450U
- #define **SC_R_DMA_2_CH29** 451U
- #define **SC_R_DMA_2_CH30** 452U
- #define **SC_R_DMA_2_CH31** 453U
- #define **SC_R_ASRC_1** 454U
- #define **SC_R_ESAI_1** 455U
- #define **SC_R_SAI_6** 456U
- #define **SC_R_SAI_7** 457U
- #define **SC_R_AMIX** 458U
- #define **SC_R_MQS_0** 459U
- #define **SC_R_DMA_3_CH0** 460U
- #define **SC_R_DMA_3_CH1** 461U
- #define **SC_R_DMA_3_CH2** 462U
- #define **SC_R_DMA_3_CH3** 463U
- #define **SC_R_DMA_3_CH4** 464U
- #define **SC_R_DMA_3_CH5** 465U
- #define **SC_R_DMA_3_CH6** 466U
- #define **SC_R_DMA_3_CH7** 467U
- #define **SC_R_DMA_3_CH8** 468U
- #define **SC_R_DMA_3_CH9** 469U
- #define **SC_R_DMA_3_CH10** 470U
- #define **SC_R_DMA_3_CH11** 471U
- #define **SC_R_DMA_3_CH12** 472U
- #define **SC_R_DMA_3_CH13** 473U
- #define **SC_R_DMA_3_CH14** 474U
- #define **SC_R_DMA_3_CH15** 475U
- #define **SC_R_DMA_3_CH16** 476U
- #define **SC_R_DMA_3_CH17** 477U
- #define **SC_R_DMA_3_CH18** 478U
- #define **SC_R_DMA_3_CH19** 479U
- #define **SC_R_DMA_3_CH20** 480U
- #define **SC_R_DMA_3_CH21** 481U
- #define **SC_R_DMA_3_CH22** 482U
- #define **SC_R_DMA_3_CH23** 483U
- #define **SC_R_DMA_3_CH24** 484U
- #define **SC_R_DMA_3_CH25** 485U
- #define **SC_R_DMA_3_CH26** 486U
- #define **SC_R_DMA_3_CH27** 487U
- #define **SC_R_DMA_3_CH28** 488U
- #define **SC_R_DMA_3_CH29** 489U
- #define **SC_R_DMA_3_CH30** 490U
- #define **SC_R_DMA_3_CH31** 491U
- #define **SC_R_AUDIO_PLL_1** 492U

- #define **SC_R_AUDIO_CLK_0** 493U
- #define **SC_R_AUDIO_CLK_1** 494U
- #define **SC_R_MCLK_OUT_0** 495U
- #define **SC_R_MCLK_OUT_1** 496U
- #define **SC_R_PMIC_0** 497U
- #define **SC_R_PMIC_1** 498U
- #define **SC_R_SECO** 499U
- #define **SC_R_CAAM_JR1** 500U
- #define **SC_R_CAAM_JR2** 501U
- #define **SC_R_CAAM_JR3** 502U
- #define **SC_R_SECO_MU_2** 503U
- #define **SC_R_SECO_MU_3** 504U
- #define **SC_R_SECO_MU_4** 505U
- #define **SC_R_HDMI_RX_PWM_0** 506U
- #define **SC_R_A35** 507U
- #define **SC_R_A35_0** 508U
- #define **SC_R_A35_1** 509U
- #define **SC_R_A35_2** 510U
- #define **SC_R_A35_3** 511U
- #define **SC_R_DSP** 512U
- #define **SC_R_DSP_RAM** 513U
- #define **SC_R_CAAM_JR1_OUT** 514U
- #define **SC_R_CAAM_JR2_OUT** 515U
- #define **SC_R_CAAM_JR3_OUT** 516U
- #define **SC_R_VPU_DEC_0** 517U
- #define **SC_R_VPU_ENC_0** 518U
- #define **SC_R_CAAM_JR0** 519U
- #define **SC_R_CAAM_JR0_OUT** 520U
- #define **SC_R_PMIC_2** 521U
- #define **SC_R_DBLOGIC** 522U
- #define **SC_R_HDMI_PLL_1** 523U
- #define **SC_R_BOARD_R0** 524U
- #define **SC_R_BOARD_R1** 525U
- #define **SC_R_BOARD_R2** 526U
- #define **SC_R_BOARD_R3** 527U
- #define **SC_R_BOARD_R4** 528U
- #define **SC_R_BOARD_R5** 529U
- #define **SC_R_BOARD_R6** 530U
- #define **SC_R_BOARD_R7** 531U
- #define **SC_R_MJPEG_DEC_MP** 532U
- #define **SC_R_MJPEG_ENC_MP** 533U
- #define **SC_R_VPU_TS_0** 534U
- #define **SC_R_VPU_MU_0** 535U
- #define **SC_R_VPU_MU_1** 536U
- #define **SC_R_VPU_MU_2** 537U
- #define **SC_R_VPU_MU_3** 538U
- #define **SC_R_VPU_ENC_1** 539U
- #define **SC_R_VPU** 540U
- #define **SC_R_DMA_5_CH0** 541U
- #define **SC_R_DMA_5_CH1** 542U
- #define **SC_R_DMA_5_CH2** 543U
- #define **SC_R_DMA_5_CH3** 544U
- #define **SC_R_ATTESTATION** 545U
- #define **SC_R_LAST** 546U
- #define SC_R_ALL ((sc_rsrc_t) UINT16_MAX)

  All resources.

Typedefs

- typedef uint8_t sc_bool_t

  This type is used to store a boolean.
- typedef uint64_t sc_faddr_t

  This type is used to store a system (full-size) address.
- typedef uint8_t sc_err_t

  This type is used to indicate error response for most functions.
- typedef uint16_t sc_rsrc_t

  This type is used to indicate a resource.
- typedef uint8_t sc_ctrl_t

  This type is used to indicate a control.
- typedef uint16_t sc_pad_t

  This type is used to indicate a pad.
- typedef __INT8_TYPE__ int8_t

  Type used to declare an 8-bit integer.
- typedef __INT16_TYPE__ int16_t

  Type used to declare a 16-bit integer.
- typedef __INT32_TYPE__ int32_t

  Type used to declare a 32-bit integer.
- typedef __INT64_TYPE__ int64_t

  Type used to declare a 64-bit integer.
- typedef __UINT8_TYPE__ uint8_t

  Type used to declare an 8-bit unsigned integer.
- typedef __UINT16_TYPE__ uint16_t

  Type used to declare a 16-bit unsigned integer.
- typedef __UINT32_TYPE__ uint32_t

  Type used to declare a 32-bit unsigned integer.
- typedef __UINT64_TYPE__ uint64_t

  Type used to declare a 64-bit unsigned integer.

11.9.1 Detailed Description

Header file containing types used across multiple service APIs.

11.9.2 Typedef Documentation

11.9.2.1 sc_rsrc_t

```
typedef uint16_t sc_rsrc_t
```

This type is used to indicate a resource.

Resources include peripherals and bus masters (but not memory regions). Note items from list should never be changed or removed (only added to at the end of the list).

11.9.2.2 sc_pad_t

```
typedef uint16_t sc_pad_t
```

This type is used to indicate a pad.

Valid values are SoC specific.

Refer to the SoC Pad List for valid pad values.

11.10 platform/svc/pad/api.h File Reference

Header file containing the public API for the System Controller (SC) Pad Control (PAD) function.

Macros

Defines for type widths

- #define SC_PAD_MUX_W 3U
  Width of mux parameter.

Defines for sc_pad_config_t

- #define SC_PAD_CONFIG_NORMAL 0U
  Normal.
- #define SC_PAD_CONFIG_OD 1U
  Open Drain.
- #define SC_PAD_CONFIG_OD_IN 2U
  Open Drain and input.
- #define SC_PAD_CONFIG_OUT_IN 3U
  Output and input.

Defines for sc_pad_iso_t

- #define SC_PAD_ISO_OFF 0U
  ISO latch is transparent.
- #define SC_PAD_ISO_EARLY 1U
  Follow EARLY_ISO.
- #define SC_PAD_ISO_LATE 2U
  Follow LATE_ISO.
- #define SC_PAD_ISO_ON 3U
  ISO latched data is held.

Defines for sc_pad_28fdsoi_dse_t

- #define SC_PAD_28FDSOI_DSE_18V_1MA 0U
  Drive strength of 1mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_2MA 1U

Drive strength of 2mA for 1.8v.

- #define SC_PAD_28FDSOI_DSE_18V_4MA 2U
  Drive strength of 4mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_6MA 3U
  Drive strength of 6mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_8MA 4U
  Drive strength of 8mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_10MA 5U
  Drive strength of 10mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_12MA 6U
  Drive strength of 12mA for 1.8v.
- #define SC_PAD_28FDSOI_DSE_18V_HS 7U
  High-speed drive strength for 1.8v.
- #define SC_PAD_28FDSOI_DSE_33V_2MA 0U
  Drive strength of 2mA for 3.3v.
- #define SC_PAD_28FDSOI_DSE_33V_4MA 1U
  Drive strength of 4mA for 3.3v.
- #define SC_PAD_28FDSOI_DSE_33V_8MA 2U
  Drive strength of 8mA for 3.3v.
- #define SC_PAD_28FDSOI_DSE_33V_12MA 3U
  Drive strength of 12mA for 3.3v.
- #define SC_PAD_28FDSOI_DSE_DV_HIGH 0U
  High drive strength for dual volt.
- #define SC_PAD_28FDSOI_DSE_DV_LOW 1U
  Low drive strength for dual volt.

Defines for sc_pad_28fdsoi_ps_t

- #define SC_PAD_28FDSOI_PS_KEEPER 0U
  Bus-keeper (only valid for 1.8v)
- #define SC_PAD_28FDSOI_PS_PU 1U
  Pull-up.
- #define SC_PAD_28FDSOI_PS_PD 2U
  Pull-down.
- #define SC_PAD_28FDSOI_PS_NONE 3U
  No pull (disabled)

Defines for sc_pad_28fdsoi_pus_t

- #define SC_PAD_28FDSOI_PUS_30K_PD 0U
  30K pull-down
- #define SC_PAD_28FDSOI_PUS_100K_PU 1U
  100K pull-up
- #define SC_PAD_28FDSOI_PUS_3K_PU 2U
  3K pull-up
- #define SC_PAD_28FDSOI_PUS_30K_PU 3U
  30K pull-up

Defines for sc_pad_wakeup_t

- #define SC_PAD_WAKEUP_OFF 0U
  Off.
- #define SC_PAD_WAKEUP_CLEAR 1U

Clears pending flag.

- #define SC_PAD_WAKEUP_LOW_LVL 4U
  Low level.
- #define SC_PAD_WAKEUP_FALL_EDGE 5U
  Falling edge.
- #define SC_PAD_WAKEUP_RISE_EDGE 6U
  Rising edge.
- #define SC_PAD_WAKEUP_HIGH_LVL 7U
  High-level.

Typedefs

- typedef uint8_t sc_pad_config_t
  This type is used to declare a pad config.
- typedef uint8_t sc_pad_iso_t
  This type is used to declare a pad low-power isolation config.
- typedef uint8_t sc_pad_28fdsoi_dse_t
  This type is used to declare a drive strength.
- typedef uint8_t sc_pad_28fdsoi_ps_t
  This type is used to declare a pull select.
- typedef uint8_t sc_pad_28fdsoi_pus_t
  This type is used to declare a pull-up select.
- typedef uint8_t sc_pad_wakeup_t
  This type is used to declare a wakeup mode of a pad.

Functions

Generic Functions

- sc_err_t sc_pad_set_mux (sc_ipc_t ipc, sc_pad_t pad, uint8_t mux, sc_pad_config_t config, sc_pad_iso_t iso)
  This function configures the mux settings for a pad.
- sc_err_t sc_pad_get_mux (sc_ipc_t ipc, sc_pad_t pad, uint8_t *mux, sc_pad_config_t *config, sc_pad_iso_t *iso)
  This function gets the mux settings for a pad.
- sc_err_t sc_pad_set_gp (sc_ipc_t ipc, sc_pad_t pad, uint32_t ctrl)
  This function configures the general purpose pad control.
- sc_err_t sc_pad_get_gp (sc_ipc_t ipc, sc_pad_t pad, uint32_t *ctrl)
  This function gets the general purpose pad control.
- sc_err_t sc_pad_set_wakeup (sc_ipc_t ipc, sc_pad_t pad, sc_pad_wakeup_t wakeup)
  This function configures the wakeup mode of the pad.
- sc_err_t sc_pad_get_wakeup (sc_ipc_t ipc, sc_pad_t pad, sc_pad_wakeup_t *wakeup)
  This function gets the wakeup mode of a pad.
- sc_err_t sc_pad_set_all (sc_ipc_t ipc, sc_pad_t pad, uint8_t mux, sc_pad_config_t config, sc_pad_iso_t iso, uint32_t ctrl, sc_pad_wakeup_t wakeup)
  This function configures a pad.
- sc_err_t sc_pad_get_all (sc_ipc_t ipc, sc_pad_t pad, uint8_t *mux, sc_pad_config_t *config, sc_pad_iso_t *iso, uint32_t *ctrl, sc_pad_wakeup_t *wakeup)
  This function gets a pad's config.

SoC Specific Functions

- sc_err_t sc_pad_set (sc_ipc_t ipc, sc_pad_t pad, uint32_t val)
  This function configures the settings for a pad.
- sc_err_t sc_pad_get (sc_ipc_t ipc, sc_pad_t pad, uint32_t *val)
  This function gets the settings for a pad.

Technology Specific Functions

- sc_err_t sc_pad_set_gp_28fdsoi (sc_ipc_t ipc, sc_pad_t pad, sc_pad_28fdsoi_dse_t dse, sc_pad_28fdsoi_ps_t ps)
  This function configures the pad control specific to 28FDSOI.
- sc_err_t sc_pad_get_gp_28fdsoi (sc_ipc_t ipc, sc_pad_t pad, sc_pad_28fdsoi_dse_t *dse, sc_pad_28fdsoi_ps_t *ps)
  This function gets the pad control specific to 28FDSOI.
- sc_err_t sc_pad_set_gp_28fdsoi_hsic (sc_ipc_t ipc, sc_pad_t pad, sc_pad_28fdsoi_dse_t dse, sc_bool_t hys, sc_pad_28fdsoi_pus_t pus, sc_bool_t pke, sc_bool_t pue)
  This function configures the pad control specific to 28FDSOI.
- sc_err_t sc_pad_get_gp_28fdsoi_hsic (sc_ipc_t ipc, sc_pad_t pad, sc_pad_28fdsoi_dse_t *dse, sc_bool_t *hys, sc_pad_28fdsoi_pus_t *pus, sc_bool_t *pke, sc_bool_t *pue)
  This function gets the pad control specific to 28FDSOI.
- sc_err_t sc_pad_set_gp_28fdsoi_comp (sc_ipc_t ipc, sc_pad_t pad, uint8_t compen, sc_bool_t fastfrz, uint8_t rasrcp, uint8_t rasrcn, sc_bool_t nasrc_sel, sc_bool_t psw_ovr)
  This function configures the compensation control specific to 28FDSOI.
- sc_err_t sc_pad_get_gp_28fdsoi_comp (sc_ipc_t ipc, sc_pad_t pad, uint8_t *compen, sc_bool_t *fastfrz, uint8_t *rasrcp, uint8_t *rasrcn, sc_bool_t *nasrc_sel, sc_bool_t *compok, uint8_t *nasrc, sc_bool_t *psw_ovr)
  This function gets the compensation control specific to 28FDSOI.

11.10.1 Detailed Description

Header file containing the public API for the System Controller (SC) Pad Control (PAD) function.

11.11 platform/svc/timer/api.h File Reference

Header file containing the public API for the System Controller (SC) Timer function.

Macros

Defines for type widths

- #define SC_TIMER_ACTION_W 3U
  Width of sc_timer_wdog_action_t.

Defines for sc_timer_wdog_action_t

- #define SC_TIMER_WDOG_ACTION_PARTITION 0U
  Reset partition.
- #define SC_TIMER_WDOG_ACTION_WARM 1U
  Warm reset system.
- #define SC_TIMER_WDOG_ACTION_COLD 2U
  Cold reset system.
- #define SC_TIMER_WDOG_ACTION_BOARD 3U
  Reset board.
- #define SC_TIMER_WDOG_ACTION_IRQ 4U
  Only generate IRQs.

Typedefs

- typedef uint8_t sc_timer_wdog_action_t

  This type is used to configure the watchdog action.
- typedef uint32_t sc_timer_wdog_time_t

  This type is used to declare a watchdog time value in milliseconds.

Functions

Wathdog Functions

- sc_err_t sc_timer_set_wdog_timeout (sc_ipc_t ipc, sc_timer_wdog_time_t timeout)

  This function sets the watchdog timeout in milliseconds.
- sc_err_t sc_timer_set_wdog_pre_timeout (sc_ipc_t ipc, sc_timer_wdog_time_t pre_timeout)

  This function sets the watchdog pre-timeout in milliseconds.
- sc_err_t sc_timer_start_wdog (sc_ipc_t ipc, sc_bool_t lock)

  This function starts the watchdog.
- sc_err_t sc_timer_stop_wdog (sc_ipc_t ipc)

  This function stops the watchdog if it is not locked.
- sc_err_t sc_timer_ping_wdog (sc_ipc_t ipc)

  This function pings (services, kicks) the watchdog resetting the time before expiration back to the timeout.
- sc_err_t sc_timer_get_wdog_status (sc_ipc_t ipc, sc_timer_wdog_time_t *timeout, sc_timer_wdog_time_t *max_timeout, sc_timer_wdog_time_t *remaining_time)

  This function gets the status of the watchdog.
- sc_err_t sc_timer_pt_get_wdog_status (sc_ipc_t ipc, sc_rm_pt_t pt, sc_bool_t *enb, sc_timer_wdog_time_t *timeout, sc_timer_wdog_time_t *remaining_time)

  This function gets the status of the watchdog of a partition.
- sc_err_t sc_timer_set_wdog_action (sc_ipc_t ipc, sc_rm_pt_t pt, sc_timer_wdog_action_t action)

  This function configures the action to be taken when a watchdog expires.

Real-Time Clock (RTC) Functions

- sc_err_t sc_timer_set_rtc_time (sc_ipc_t ipc, uint16_t year, uint8_t mon, uint8_t day, uint8_t hour, uint8_t min, uint8_t sec)

  This function sets the RTC time.
- sc_err_t sc_timer_get_rtc_time (sc_ipc_t ipc, uint16_t *year, uint8_t *mon, uint8_t *day, uint8_t *hour, uint8_t *min, uint8_t *sec)

  This function gets the RTC time.
- sc_err_t sc_timer_get_rtc_sec1970 (sc_ipc_t ipc, uint32_t *sec)

  This function gets the RTC time in seconds since 1/1/1970.
- sc_err_t sc_timer_set_rtc_alarm (sc_ipc_t ipc, uint16_t year, uint8_t mon, uint8_t day, uint8_t hour, uint8_t min, uint8_t sec)

  This function sets the RTC alarm.
- sc_err_t sc_timer_set_rtc_periodic_alarm (sc_ipc_t ipc, uint32_t sec)

  This function sets the RTC alarm (periodic mode).
- sc_err_t sc_timer_cancel_rtc_alarm (sc_ipc_t ipc)

  This function cancels the RTC alarm.
- sc_err_t sc_timer_set_rtc_calb (sc_ipc_t ipc, int8_t count)

  This function sets the RTC calibration value.

System Counter (SYSCTR) Functions

- sc_err_t sc_timer_set_sysctr_alarm (sc_ipc_t ipc, uint64_t ticks)

  This function sets the SYSCTR alarm.
- sc_err_t sc_timer_set_sysctr_periodic_alarm (sc_ipc_t ipc, uint64_t ticks)

  This function sets the SYSCTR alarm (periodic mode).
- sc_err_t sc_timer_cancel_sysctr_alarm (sc_ipc_t ipc)

  This function cancels the SYSCTR alarm.

11.11.1 Detailed Description

Header file containing the public API for the System Controller (SC) Timer function.

11.12 platform/svc/pm/api.h File Reference

Header file containing the public API for the System Controller (SC) Power Management (PM) function.

Macros

Defines for type widths

- #define SC_PM_POWER_MODE_W 2U
  Width of sc_pm_power_mode_t.
- #define SC_PM_CLOCK_MODE_W 3U
  Width of sc_pm_clock_mode_t.
- #define SC_PM_RESET_TYPE_W 2U
  Width of sc_pm_reset_type_t.
- #define SC_PM_RESET_REASON_W 4U
  Width of sc_pm_reset_reason_t.

Defines for ALL parameters

- #define SC_PM_CLK_ALL ((sc_pm_clk_t) UINT8_MAX)
  All clocks.

Defines for sc_pm_power_mode_t

- #define SC_PM_PW_MODE_OFF 0U
  Power off.
- #define SC_PM_PW_MODE_STBY 1U
  Power in standby.
- #define SC_PM_PW_MODE_LP 2U
  Power in low-power.
- #define SC_PM_PW_MODE_ON 3U
  Power on.

Defines for sc_pm_clk_t

- #define SC_PM_CLK_SLV_BUS 0U
  Slave bus clock.
- #define SC_PM_CLK_MST_BUS 1U
  Master bus clock.
- #define SC_PM_CLK_PER 2U
  Peripheral clock.
- #define SC_PM_CLK_PHY 3U
  Phy clock.
- #define SC_PM_CLK_MISC 4U

Misc clock.

- #define SC_PM_CLK_MISC0 0U
  Misc 0 clock.
- #define SC_PM_CLK_MISC1 1U
  Misc 1 clock.
- #define SC_PM_CLK_MISC2 2U
  Misc 2 clock.
- #define SC_PM_CLK_MISC3 3U
  Misc 3 clock.
- #define SC_PM_CLK_MISC4 4U
  Misc 4 clock.
- #define SC_PM_CLK_CPU 2U
  CPU clock.
- #define SC_PM_CLK_PLL 4U
  PLL.
- #define SC_PM_CLK_BYPASS 4U
  Bypass clock.

Defines for sc_pm_clk_mode_t

- #define SC_PM_CLK_MODE_ROM_INIT 0U
  Clock is initialized by ROM.
- #define SC_PM_CLK_MODE_OFF 1U
  Clock is disabled.
- #define SC_PM_CLK_MODE_ON 2U
  Clock is enabled.
- #define SC_PM_CLK_MODE_AUTOGATE_SW 3U
  Clock is in SW autogate mode.
- #define SC_PM_CLK_MODE_AUTOGATE_HW 4U
  Clock is in HW autogate mode.
- #define SC_PM_CLK_MODE_AUTOGATE_SW_HW 5U
  Clock is in SW-HW autogate mode.

Defines for sc_pm_clk_parent_t

- #define SC_PM_PARENT_XTAL 0U
  Parent is XTAL.
- #define SC_PM_PARENT_PLL0 1U
  Parent is PLL0.
- #define SC_PM_PARENT_PLL1 2U
  Parent is PLL1 or PLL0/2.
- #define SC_PM_PARENT_PLL2 3U
  Parent in PLL2 or PLL0/4.
- #define SC_PM_PARENT_BYPS 4U
  Parent is a bypass clock.

Defines for sc_pm_reset_type_t

- #define SC_PM_RESET_TYPE_COLD 0U
  Cold reset.
- #define SC_PM_RESET_TYPE_WARM 1U
  Warm reset.
- #define SC_PM_RESET_TYPE_BOARD 2U

Board reset.

Defines for sc_pm_reset_reason_t

- #define SC_PM_RESET_REASON_POR 0U
  Power on reset.
- #define SC_PM_RESET_REASON_JTAG 1U
  JTAG reset.
- #define SC_PM_RESET_REASON_SW 2U
  Software reset.
- #define SC_PM_RESET_REASON_WDOG 3U
  Partition watchdog reset.
- #define SC_PM_RESET_REASON_LOCKUP 4U
  SCU lockup reset.
- #define SC_PM_RESET_REASON_SNVS 5U
  SNVS reset.
- #define SC_PM_RESET_REASON_TEMP 6U
  Temp panic reset.
- #define SC_PM_RESET_REASON_MSI 7U
  MSI reset.
- #define SC_PM_RESET_REASON_UECC 8U
  ECC reset.
- #define SC_PM_RESET_REASON_SCFW_WDOG 9U
  SCFW watchdog reset.
- #define SC_PM_RESET_REASON_ROM_WDOG 10U
  SCU ROM watchdog reset.
- #define SC_PM_RESET_REASON_SECO 11U
  SECO reset.
- #define SC_PM_RESET_REASON_SCFW_FAULT 12U
  SCFW fault reset.

Defines for sc_pm_sys_if_t

- #define SC_PM_SYS_IF_INTERCONNECT 0U
  System interconnect.
- #define SC_PM_SYS_IF_MU 1U
  AP -> SCU message units.
- #define SC_PM_SYS_IF_OCMEM 2U
  On-chip memory (ROM/OCRAM)
- #define SC_PM_SYS_IF_DDR 3U
  DDR memory.

Defines for sc_pm_wake_src_t

- #define SC_PM_WAKE_SRC_NONE 0U
  No wake source, used for self-kill.
- #define SC_PM_WAKE_SRC_SCU 1U
  Wakeup from SCU to resume CPU (IRQSTEER & GIC powered down)
- #define SC_PM_WAKE_SRC_IRQSTEER 2U
  Wakeup from IRQSTEER to resume CPU (GIC powered down)
- #define SC_PM_WAKE_SRC_IRQSTEER_GIC 3U
  Wakeup from IRQSTEER+GIC to wake CPU (GIC clock gated)
- #define SC_PM_WAKE_SRC_GIC 4U
  Wakeup from GIC to wake CPU.

Typedefs

- typedef uint8_t sc_pm_power_mode_t

  This type is used to declare a power mode.
- typedef uint8_t sc_pm_clk_t

  This type is used to declare a clock.
- typedef uint8_t sc_pm_clk_mode_t

  This type is used to declare a clock mode.
- typedef uint8_t sc_pm_clk_parent_t

  This type is used to declare the clock parent.
- typedef uint32_t sc_pm_clock_rate_t

  This type is used to declare clock rates.
- typedef uint8_t sc_pm_reset_type_t

  This type is used to declare a desired reset type.
- typedef uint8_t sc_pm_reset_reason_t

  This type is used to declare a reason for a reset.
- typedef uint8_t sc_pm_sys_if_t

  This type is used to specify a system-level interface to be power managed.
- typedef uint8_t sc_pm_wake_src_t

  This type is used to specify a wake source for CPU resources.

Functions

Power Functions

- sc_err_t sc_pm_set_sys_power_mode (sc_ipc_t ipc, sc_pm_power_mode_t mode)

  This function sets the system power mode.
- sc_err_t sc_pm_set_partition_power_mode (sc_ipc_t ipc, sc_rm_pt_t pt, sc_pm_power_mode_t mode)

  This function sets the power mode of a partition.
- sc_err_t sc_pm_get_sys_power_mode (sc_ipc_t ipc, sc_rm_pt_t pt, sc_pm_power_mode_t *mode)

  This function gets the power mode of a partition.
- sc_err_t sc_pm_set_resource_power_mode (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_power_mode_t mode)

  This function sets the power mode of a resource.
- sc_err_t sc_pm_set_resource_power_mode_all (sc_ipc_t ipc, sc_rm_pt_t pt, sc_pm_power_mode_t mode, sc_rsrc_t exclude)

  This function sets the power mode for all the resources owned by a child partition.
- sc_err_t sc_pm_get_resource_power_mode (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_power_mode_t *mode)

  This function gets the power mode of a resource.
- sc_err_t sc_pm_req_low_power_mode (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_power_mode_t mode)

  This function requests the low power mode some of the resources can enter based on their state.
- sc_err_t sc_pm_req_cpu_low_power_mode (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_power_mode_t mode, sc_pm_wake_src_t wake_src)

  This function requests low-power mode entry for CPU/cluster resources.
- sc_err_t sc_pm_set_cpu_resume_addr (sc_ipc_t ipc, sc_rsrc_t resource, sc_faddr_t address)

  This function is used to set the resume address of a CPU.
- sc_err_t sc_pm_set_cpu_resume (sc_ipc_t ipc, sc_rsrc_t resource, sc_bool_t isPrimary, sc_faddr_t address)

  This function is used to set parameters for CPU resume from low-power mode.
- sc_err_t sc_pm_req_sys_if_power_mode (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_sys_if_t sys_↩ if, sc_pm_power_mode_t hpm, sc_pm_power_mode_t lpm)

  This function requests the power mode configuration for system-level interfaces including messaging units, interconnect, and memories.

Clock/PLL Functions

- sc_err_t sc_pm_set_clock_rate (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_clk_t clk, sc_pm_clock_rate_t *rate)
  This function sets the rate of a resource's clock/PLL.
- sc_err_t sc_pm_get_clock_rate (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_clk_t clk, sc_pm_clock_rate_t *rate)
  This function gets the rate of a resource's clock/PLL.
- sc_err_t sc_pm_clock_enable (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_clk_t clk, sc_bool_t enable, sc_bool_t autog)
  This function enables/disables a resource's clock.
- sc_err_t sc_pm_set_clock_parent (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_clk_t clk, sc_pm_clk_parent_t parent)
  This function sets the parent of a resource's clock.
- sc_err_t sc_pm_get_clock_parent (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_clk_t clk, sc_pm_clk_parent_t *parent)
  This function gets the parent of a resource's clock.

Reset Functions

- sc_err_t sc_pm_reset (sc_ipc_t ipc, sc_pm_reset_type_t type)
  This function is used to reset the system.
- sc_err_t sc_pm_reset_reason (sc_ipc_t ipc, sc_pm_reset_reason_t *reason)
  This function gets a caller's reset reason.
- sc_err_t sc_pm_boot (sc_ipc_t ipc, sc_rm_pt_t pt, sc_rsrc_t resource_cpu, sc_faddr_t boot_addr, sc_rsrc_t resource_mu, sc_rsrc_t resource_dev)
  This function is used to boot a partition.
- void sc_pm_reboot (sc_ipc_t ipc, sc_pm_reset_type_t type)
  This function is used to reboot the caller's partition.
- sc_err_t sc_pm_reboot_partition (sc_ipc_t ipc, sc_rm_pt_t pt, sc_pm_reset_type_t type)
  This function is used to reboot a partition.
- sc_err_t sc_pm_cpu_start (sc_ipc_t ipc, sc_rsrc_t resource, sc_bool_t enable, sc_faddr_t address)
  This function is used to start/stop a CPU.

11.12.1 Detailed Description

Header file containing the public API for the System Controller (SC) Power Management (PM) function.

This includes functions for power state control, clock control, reset control, and wake-up event control.

11.13 platform/svc/irq/api.h File Reference

Header file containing the public API for the System Controller (SC) Interrupt (IRQ) function.

Macros

- #define SC_IRQ_NUM_GROUP 5U

  Number of groups.

Defines for sc_irq_group_t

- #define SC_IRQ_GROUP_TEMP 0U

  Temp interrupts.
- #define SC_IRQ_GROUP_WDOG 1U

  Watchdog interrupts.
- #define SC_IRQ_GROUP_RTC 2U

  RTC interrupts.
- #define SC_IRQ_GROUP_WAKE 3U

  Wakeup interrupts.
- #define SC_IRQ_GROUP_SYSCTR 4U

  System counter interrupts.

Defines for sc_irq_temp_t

- #define SC_IRQ_TEMP_HIGH (1UL << 0U)

  Temp alarm interrupt.
- #define SC_IRQ_TEMP_CPU0_HIGH (1UL << 1U)

  CPU0 temp alarm interrupt.
- #define SC_IRQ_TEMP_CPU1_HIGH (1UL << 2U)

  CPU1 temp alarm interrupt.
- #define SC_IRQ_TEMP_GPU0_HIGH (1UL << 3U)

  GPU0 temp alarm interrupt.
- #define SC_IRQ_TEMP_GPU1_HIGH (1UL << 4U)

  GPU1 temp alarm interrupt.
- #define SC_IRQ_TEMP_DRC0_HIGH (1UL << 5U)

  DRC0 temp alarm interrupt.
- #define SC_IRQ_TEMP_DRC1_HIGH (1UL << 6U)

  DRC1 temp alarm interrupt.
- #define SC_IRQ_TEMP_VPU_HIGH (1UL << 7U)

  DRC1 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC0_HIGH (1UL << 8U)

  PMIC0 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC1_HIGH (1UL << 9U)

  PMIC1 temp alarm interrupt.
- #define SC_IRQ_TEMP_LOW (1UL << 10U)

  Temp alarm interrupt.
- #define SC_IRQ_TEMP_CPU0_LOW (1UL << 11U)

  CPU0 temp alarm interrupt.
- #define SC_IRQ_TEMP_CPU1_LOW (1UL << 12U)

  CPU1 temp alarm interrupt.
- #define SC_IRQ_TEMP_GPU0_LOW (1UL << 13U)

  GPU0 temp alarm interrupt.
- #define SC_IRQ_TEMP_GPU1_LOW (1UL << 14U)

  GPU1 temp alarm interrupt.
- #define SC_IRQ_TEMP_DRC0_LOW (1UL << 15U)

  DRC0 temp alarm interrupt.
- #define SC_IRQ_TEMP_DRC1_LOW (1UL << 16U)

DRC1 temp alarm interrupt.

- #define SC_IRQ_TEMP_VPU_LOW (1UL << 17U)
  DRC1 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC0_LOW (1UL << 18U)
  PMIC0 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC1_LOW (1UL << 19U)
  PMIC1 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC2_HIGH (1UL << 20U)
  PMIC2 temp alarm interrupt.
- #define SC_IRQ_TEMP_PMIC2_LOW (1UL << 21U)
  PMIC2 temp alarm interrupt.

Defines for sc_irq_wdog_t

- #define SC_IRQ_WDOG (1U << 0U)
  Watchdog interrupt.

Defines for sc_irq_rtc_t

- #define SC_IRQ_RTC (1U << 0U)
  RTC interrupt.

Defines for sc_irq_wake_t

- #define SC_IRQ_BUTTON (1U << 0U)
  Button interrupt.
- #define SC_IRQ_PAD (1U << 1U)
  Pad wakeup.

Defines for sc_irq_sysctr_t

- #define SC_IRQ_SYSCTR (1U << 0U)
  SYSCTR interrupt.

Typedefs

- typedef uint8_t sc_irq_group_t
  This type is used to declare an interrupt group.
- typedef uint8_t sc_irq_temp_t
  This type is used to declare a bit mask of temp interrupts.
- typedef uint8_t sc_irq_wdog_t
  This type is used to declare a bit mask of watchdog interrupts.
- typedef uint8_t sc_irq_rtc_t
  This type is used to declare a bit mask of RTC interrupts.
- typedef uint8_t sc_irq_wake_t
  This type is used to declare a bit mask of wakeup interrupts.

Functions

- sc_err_t sc_irq_enable (sc_ipc_t ipc, sc_rsrc_t resource, sc_irq_group_t group, uint32_t mask, sc_bool_t enable)
  This function enables/disables interrupts.
- sc_err_t sc_irq_status (sc_ipc_t ipc, sc_rsrc_t resource, sc_irq_group_t group, uint32_t *status)
  This function returns the current interrupt status (regardless if masked).

11.13.1 Detailed Description

Header file containing the public API for the System Controller (SC) Interrupt (IRQ) function.

11.14 platform/svc/misc/api.h File Reference

Header file containing the public API for the System Controller (SC) Miscellaneous (MISC) function.

Macros

- #define SC_MISC_DMA_GRP_MAX 31U
  Max DMA channel priority group.

Defines for type widths

- #define SC_MISC_DMA_GRP_W 5U
  Width of sc_misc_dma_group_t.

Defines for sc_misc_boot_status_t

- #define SC_MISC_BOOT_STATUS_SUCCESS 0U
  Success.
- #define SC_MISC_BOOT_STATUS_SECURITY 1U
  Security violation.

Defines for sc_misc_temp_t

- #define SC_MISC_TEMP 0U
  Temp sensor.
- #define SC_MISC_TEMP_HIGH 1U
  Temp high alarm.
- #define SC_MISC_TEMP_LOW 2U
  Temp low alarm.

Defines for sc_misc_seco_auth_cmd_t

- #define SC_MISC_AUTH_CONTAINER 0U
  Authenticate container.

- #define SC_MISC_VERIFY_IMAGE 1U
  Verify image.
- #define SC_MISC_REL_CONTAINER 2U
  Release container.
- #define SC_MISC_SECO_AUTH_SECO_FW 3U
  SECO Firmware.
- #define SC_MISC_SECO_AUTH_HDMI_TX_FW 4U
  HDMI TX Firmware.
- #define SC_MISC_SECO_AUTH_HDMI_RX_FW 5U
  HDMI RX Firmware.

Defines for sc_misc_bt_t

- #define **SC_MISC_BT_PRIMARY** 0U
- #define **SC_MISC_BT_SECONDARY** 1U
- #define **SC_MISC_BT_RECOVERY** 2U
- #define **SC_MISC_BT_MANUFACTURE** 3U
- #define **SC_MISC_BT_SERIAL** 4U

Typedefs

- typedef uint8_t sc_misc_dma_group_t
  This type is used to store a DMA channel priority group.
- typedef uint8_t sc_misc_boot_status_t
  This type is used report boot status.
- typedef uint8_t sc_misc_seco_auth_cmd_t
  This type is used to issue SECO authenticate commands.
- typedef uint8_t sc_misc_temp_t
  This type is used report boot status.
- typedef uint8_t sc_misc_bt_t
  This type is used report the boot type.

Functions

Control Functions

- sc_err_t sc_misc_set_control (sc_ipc_t ipc, sc_rsrc_t resource, sc_ctrl_t ctrl, uint32_t val)
  This function sets a miscellaneous control value.
- sc_err_t sc_misc_get_control (sc_ipc_t ipc, sc_rsrc_t resource, sc_ctrl_t ctrl, uint32_t *val)
  This function gets a miscellaneous control value.

DMA Functions

- sc_err_t sc_misc_set_max_dma_group (sc_ipc_t ipc, sc_rm_pt_t pt, sc_misc_dma_group_t max)
  This function configures the max DMA channel priority group for a partition.
- sc_err_t sc_misc_set_dma_group (sc_ipc_t ipc, sc_rsrc_t resource, sc_misc_dma_group_t group)
  This function configures the priority group for a DMA channel.

Security Functions

- sc_err_t sc_misc_seco_image_load (sc_ipc_t ipc, sc_faddr_t addr_src, sc_faddr_t addr_dst, uint32_t len, sc_bool_t fw)

  This function loads a SECO image.
- sc_err_t sc_misc_seco_authenticate (sc_ipc_t ipc, sc_misc_seco_auth_cmd_t cmd, sc_faddr_t addr)

  This function is used to authenticate a SECO image or command.
- sc_err_t sc_misc_seco_fuse_write (sc_ipc_t ipc, sc_faddr_t addr)

  This function securely writes a group of fuse words.
- sc_err_t sc_misc_seco_enable_debug (sc_ipc_t ipc, sc_faddr_t addr)

  This function securely enables debug.
- sc_err_t sc_misc_seco_forward_lifecycle (sc_ipc_t ipc, uint32_t change)

  This function updates the lifecycle of the device.
- sc_err_t sc_misc_seco_return_lifecycle (sc_ipc_t ipc, sc_faddr_t addr)

  This function updates the lifecycle to one of the return lifecycles.
- void sc_misc_seco_build_info (sc_ipc_t ipc, uint32_t *version, uint32_t *commit)

  This function is used to return the SECO FW build info.
- sc_err_t sc_misc_seco_chip_info (sc_ipc_t ipc, uint16_t *lc, uint16_t *monotonic, uint32_t *uid_l, uint32_t *uid_h)

  This function is used to return SECO chip info.
- sc_err_t sc_misc_seco_attest_mode (sc_ipc_t ipc, uint32_t mode)

  This function is used to set the attestation mode.
- sc_err_t sc_misc_seco_attest (sc_ipc_t ipc, uint64_t nonce)

  This function is used to request atestation.
- sc_err_t sc_misc_seco_get_attest_pkey (sc_ipc_t ipc, sc_faddr_t addr)

  This function is used to retrieve the attestation public key.
- sc_err_t sc_misc_seco_get_attest_sign (sc_ipc_t ipc, sc_faddr_t addr)

  This function is used to retrieve attestation signature and parameters.
- sc_err_t sc_misc_seco_attest_verify (sc_ipc_t ipc, sc_faddr_t addr)

  This function is used to verify attestation.
- sc_err_t sc_misc_seco_commit (sc_ipc_t ipc, uint32_t *info)

  This function is used to commit into the fuses any new SRK revocation and FW version information that have been found in the primary and secondary containers.

Debug Functions

- void sc_misc_debug_out (sc_ipc_t ipc, uint8_t ch)

  This function is used output a debug character from the SCU UART.
- sc_err_t sc_misc_waveform_capture (sc_ipc_t ipc, sc_bool_t enable)

  This function starts/stops emulation waveform capture.
- void sc_misc_build_info (sc_ipc_t ipc, uint32_t *build, uint32_t *commit)

  This function is used to return the SCFW build info.
- void sc_misc_unique_id (sc_ipc_t ipc, uint32_t *id_l, uint32_t *id_h)

  This function is used to return the device's unique ID.

Other Functions

- sc_err_t sc_misc_set_ari (sc_ipc_t ipc, sc_rsrc_t resource, sc_rsrc_t resource_mst, uint16_t ari, sc_bool_t enable)

  This function configures the ARI match value for PCIe/SATA resources.
- void sc_misc_boot_status (sc_ipc_t ipc, sc_misc_boot_status_t status)

  This function reports boot status.
- sc_err_t sc_misc_boot_done (sc_ipc_t ipc, sc_rsrc_t cpu)

  This function tells the SCFW that a CPU is done booting.
- sc_err_t sc_misc_otp_fuse_read (sc_ipc_t ipc, uint32_t word, uint32_t *val)

This function reads a given fuse word index.

- sc_err_t sc_misc_otp_fuse_write (sc_ipc_t ipc, uint32_t word, uint32_t val)
  This function writes a given fuse word index.
- sc_err_t sc_misc_set_temp (sc_ipc_t ipc, sc_rsrc_t resource, sc_misc_temp_t temp, int16_t celsius, int8_t tenths)
  This function sets a temp sensor alarm.
- sc_err_t sc_misc_get_temp (sc_ipc_t ipc, sc_rsrc_t resource, sc_misc_temp_t temp, int16_t *celsius, int8_t *tenths)
  This function gets a temp sensor value.
- void sc_misc_get_boot_dev (sc_ipc_t ipc, sc_rsrc_t *dev)
  This function returns the boot device.
- sc_err_t sc_misc_get_boot_type (sc_ipc_t ipc, sc_misc_bt_t *type)
  This function returns the boot type.
- void sc_misc_get_button_status (sc_ipc_t ipc, sc_bool_t *status)
  This function returns the current status of the ON/OFF button.
- sc_err_t sc_misc_rompatch_checksum (sc_ipc_t ipc, uint32_t *checksum)
  This function returns the ROM patch checksum.

11.14.1 Detailed Description

Header file containing the public API for the System Controller (SC) Miscellaneous (MISC) function.

11.15 platform/svc/rm/api.h File Reference

Header file containing the public API for the System Controller (SC) Resource Management (RM) function.

Macros

Defines for type widths

- #define SC_RM_PARTITION_W 5U
  Width of sc_rm_pt_t.
- #define SC_RM_MEMREG_W 6U
  Width of sc_rm_mr_t.
- #define SC_RM_DID_W 4U
  Width of sc_rm_did_t.
- #define SC_RM_SID_W 6U
  Width of sc_rm_sid_t.
- #define SC_RM_SPA_W 2U
  Width of sc_rm_spa_t.
- #define SC_RM_PERM_W 3U
  Width of sc_rm_perm_t.

Defines for ALL parameters

- #define SC_RM_PT_ALL ((sc_rm_pt_t) UINT8_MAX)
  All partitions.
- #define SC_RM_MR_ALL ((sc_rm_mr_t) UINT8_MAX)

All memory regions.

Defines for sc_rm_spa_t

- #define SC_RM_SPA_PASSTHRU 0U
  Pass through (attribute driven by master)
- #define SC_RM_SPA_PASSSID 1U
  Pass through and output on SID.
- #define SC_RM_SPA_ASSERT 2U
  Assert (force to be secure/privileged)
- #define SC_RM_SPA_NEGATE 3U
  Negate (force to be non-secure/user)

Defines for sc_rm_perm_t

- #define SC_RM_PERM_NONE 0U
  No access.
- #define SC_RM_PERM_SEC_R 1U
  Secure RO.
- #define SC_RM_PERM_SECPRIV_RW 2U
  Secure privilege R/W.
- #define SC_RM_PERM_SEC_RW 3U
  Secure R/W.
- #define SC_RM_PERM_NSPRIV_R 4U
  Secure R/W, non-secure privilege RO.
- #define SC_RM_PERM_NS_R 5U
  Secure R/W, non-secure RO.
- #define SC_RM_PERM_NSPRIV_RW 6U
  Secure R/W, non-secure privilege R/W.
- #define SC_RM_PERM_FULL 7U
  Full access.

Typedefs

- typedef uint8_t sc_rm_pt_t
  This type is used to declare a resource partition.
- typedef uint8_t sc_rm_mr_t
  This type is used to declare a memory region.
- typedef uint8_t sc_rm_did_t
  This type is used to declare a resource domain ID used by the isolation HW.
- typedef uint16_t sc_rm_sid_t
  This type is used to declare an SMMU StreamID.
- typedef uint8_t sc_rm_spa_t
  This type is a used to declare master transaction attributes.
- typedef uint8_t sc_rm_perm_t
  This type is used to declare a resource/memory region access permission.

Functions

Partition Functions

- sc_err_t sc_rm_partition_alloc (sc_ipc_t ipc, sc_rm_pt_t *pt, sc_bool_t secure, sc_bool_t isolated, sc_bool_t restricted, sc_bool_t grant, sc_bool_t coherent)

  This function requests that the SC create a new resource partition.
- sc_err_t sc_rm_set_confidential (sc_ipc_t ipc, sc_rm_pt_t pt, sc_bool_t retro)

  This function makes a partition confidential.
- sc_err_t sc_rm_partition_free (sc_ipc_t ipc, sc_rm_pt_t pt)

  This function frees a partition and assigns all resources to the caller.
- sc_rm_did_t sc_rm_get_did (sc_ipc_t ipc)

  This function returns the DID of a partition.
- sc_err_t sc_rm_partition_static (sc_ipc_t ipc, sc_rm_pt_t pt, sc_rm_did_t did)

  This function forces a partition to use a specific static DID.
- sc_err_t sc_rm_partition_lock (sc_ipc_t ipc, sc_rm_pt_t pt)

  This function locks a partition.
- sc_err_t sc_rm_get_partition (sc_ipc_t ipc, sc_rm_pt_t *pt)

  This function gets the partition handle of the caller.
- sc_err_t sc_rm_set_parent (sc_ipc_t ipc, sc_rm_pt_t pt, sc_rm_pt_t pt_parent)

  This function sets a new parent for a partition.
- sc_err_t sc_rm_move_all (sc_ipc_t ipc, sc_rm_pt_t pt_src, sc_rm_pt_t pt_dst, sc_bool_t move_rsrc, sc_bool_t move_pads)

  This function moves all movable resources/pads owned by a source partition to a destination partition.

Resource Functions

- sc_err_t sc_rm_assign_resource (sc_ipc_t ipc, sc_rm_pt_t pt, sc_rsrc_t resource)

  This function assigns ownership of a resource to a partition.
- sc_err_t sc_rm_set_resource_movable (sc_ipc_t ipc, sc_rsrc_t resource_fst, sc_rsrc_t resource_lst, sc_bool_t movable)

  This function flags resources as movable or not.
- sc_err_t sc_rm_set_subsys_rsrc_movable (sc_ipc_t ipc, sc_rsrc_t resource, sc_bool_t movable)

  This function flags all of a subsystem's resources as movable or not.
- sc_err_t sc_rm_set_master_attributes (sc_ipc_t ipc, sc_rsrc_t resource, sc_rm_spa_t sa, sc_rm_spa_t pa, sc_bool_t smmu_bypass)

  This function sets attributes for a resource which is a bus master (i.e.
- sc_err_t sc_rm_set_master_sid (sc_ipc_t ipc, sc_rsrc_t resource, sc_rm_sid_t sid)

  This function sets the StreamID for a resource which is a bus master (i.e.
- sc_err_t sc_rm_set_peripheral_permissions (sc_ipc_t ipc, sc_rsrc_t resource, sc_rm_pt_t pt, sc_rm_perm_t perm)

  This function sets access permissions for a peripheral resource.
- sc_bool_t sc_rm_is_resource_owned (sc_ipc_t ipc, sc_rsrc_t resource)

  This function gets ownership status of a resource.
- sc_bool_t sc_rm_is_resource_master (sc_ipc_t ipc, sc_rsrc_t resource)

  This function is used to test if a resource is a bus master.
- sc_bool_t sc_rm_is_resource_peripheral (sc_ipc_t ipc, sc_rsrc_t resource)

  This function is used to test if a resource is a peripheral.
- sc_err_t sc_rm_get_resource_info (sc_ipc_t ipc, sc_rsrc_t resource, sc_rm_sid_t *sid)

  This function is used to obtain info about a resource.

Memory Region Functions

- sc_err_t sc_rm_memreg_alloc (sc_ipc_t ipc, sc_rm_mr_t *mr, sc_faddr_t addr_start, sc_faddr_t addr_end)

This function requests that the SC create a new memory region.

- sc_err_t sc_rm_memreg_split (sc_ipc_t ipc, sc_rm_mr_t mr, sc_rm_mr_t *mr_ret, sc_faddr_t addr_start, sc_faddr_t addr_end)

  This function requests that the SC split a memory region.
- sc_err_t sc_rm_memreg_free (sc_ipc_t ipc, sc_rm_mr_t mr)

  This function frees a memory region.
- sc_err_t sc_rm_find_memreg (sc_ipc_t ipc, sc_rm_mr_t *mr, sc_faddr_t addr_start, sc_faddr_t addr_end)

  Internal SC function to find a memory region.
- sc_err_t sc_rm_assign_memreg (sc_ipc_t ipc, sc_rm_pt_t pt, sc_rm_mr_t mr)

  This function assigns ownership of a memory region.
- sc_err_t sc_rm_set_memreg_permissions (sc_ipc_t ipc, sc_rm_mr_t mr, sc_rm_pt_t pt, sc_rm_perm_t perm)

  This function sets access permissions for a memory region.
- sc_bool_t sc_rm_is_memreg_owned (sc_ipc_t ipc, sc_rm_mr_t mr)

  This function gets ownership status of a memory region.
- sc_err_t sc_rm_get_memreg_info (sc_ipc_t ipc, sc_rm_mr_t mr, sc_faddr_t *addr_start, sc_faddr_t *addr_↩ end)

  This function is used to obtain info about a memory region.

Pad Functions

- sc_err_t sc_rm_assign_pad (sc_ipc_t ipc, sc_rm_pt_t pt, sc_pad_t pad)

  This function assigns ownership of a pad to a partition.
- sc_err_t sc_rm_set_pad_movable (sc_ipc_t ipc, sc_pad_t pad_fst, sc_pad_t pad_lst, sc_bool_t movable)

  This function flags pads as movable or not.
- sc_bool_t sc_rm_is_pad_owned (sc_ipc_t ipc, sc_pad_t pad)

  This function gets ownership status of a pad.

Debug Functions

- void sc_rm_dump (sc_ipc_t ipc)

  This function dumps the RM state for debug.

11.15.1 Detailed Description

Header file containing the public API for the System Controller (SC) Resource Management (RM) function.

This includes functions for partitioning resources, pads, and memory regions.

Index