MGS2401 API Reference Manual

A microcontroller by

MINDGROVE

Table of Contents

List of Figures

List of Tables

Chapter 1. Libraries

1.1 Big Num Library

The Big Number library provides support for handling large integers beyond standard data type limits. It represents numbers using an array of 64-bit digits and includes utilities for initialization, comparison, arithmetic operations, and binary conversion. The design ensures efficient and reliable multi-precision computations suitable for embedded systems.

1.1.1 Required Includes

To use the Big Number functionality, include the *bignum.h* header file in your source code. This header provides the necessary definitions, macros, and functions for performing multi-precision arithmetic operations.

```
#include "bignum.h"
```

1.1.2 Macros, Typedefs and Data Structures

Macros

`BIGNUM_SIZE`

Maximum number of digits (limbs) supported.

`DIGIT_BIT`

Number of bits per digit (limb).

`ZERO_POSITIVE`

Represents zero or positive sign.

`NEGATIVE`

Represents negative sign.

YES

Boolean YES value.

NO

Boolean NO value.

EQUAL_TO

Indicates equality result.

LESS_THAN

Indicates less-than result.

GREATER_THAN

Indicates greater-than result.

BN_INIT(a)

Initializes a big number structure to zero.

BN_ZERO(a)

Sets a big number to zero.

BN_IS_ZERO(a)

Checks if the big number is zero.

BN_FREE(a)

Frees the big number resources and resets it.

BN_IS_EVEN(a)

Checks if the big number is even.

BN_IS_ODD(a)

Checks if the big number is odd.

BN_CLAMP(a)

Removes leading zero digits from the big number.

BN_COPY(a, b)

Copies one big number into another.

BN_ABS(a, b)

Computes absolute value of a big number.

Typedefs

typedef uint64_t **bn_digit**

Defines the digit type for big numbers.

Each big integer is stored as an array of 64-bit unsigned digits (limbs) using this type.

Data Structures

struct **bn_int**

Big integer structure.

Represents a multi-precision integer using an array of fixed-size digits. It stores the digit array, the number of active digits, and the sign of the value.

Public Members

bn_digit **dp**[34U]

Array of digits (least significant limb at index 0)

uint16_t **used**

Number of active digits currently used (0 indicates the value is zero)

uint8_t **sign**

Sign of the integer (POSITIVE (include zero) or NEGATIVE)

1.1.3 API Reference

uint16_t **BigNum_Print_Int_to_Hex**(const *bn_int* *a)

Prints a big integer in hexadecimal format.

This function prints the given big integer in hexadecimal representation, starting from the most significant limb. If the value is zero, it prints 0.

Parameters

a – Pointer to the *bn_int* structure to be printed.

Returns

Returns SUCCESS on successful print; otherwise returns error if the input pointer is NULL.

uint16_t **BigNum_Set_Digit**(*bn_int* *a, *bn_digit* d)

Sets a big integer to a single digit value.

This function initializes the big integer to the specified digit value.

Parameters

- **a** – Pointer to the *bn_int* to be initialized.
- **d** – Digit value to assign.

Returns

Returns SUCCESS on success; otherwise returns error if the pointer is NULL.

uint16_t **BigNum_Left_Shift**(*bn_int* *a, uint16_t n)

Performs left shift operation on a big integer.

This function shifts the big integer left by n bits. It handles both limb-level and bit-level shifts and updates the used length accordingly.

Parameters

- **a** – Pointer to the *bn_int* structure to be shifted.
- **n** – Number of bits to shift.

Returns

Returns SUCCESS on successful shift.

uint16_t **BigNum_Mod**(const *bn_int* *a, const *bn_int* *b, *bn_int* *c)

Computes modulus of two big integers.

This function computes c = a % b using bit-wise long division from MSB to LSB.

Parameters

- **a** – Pointer to dividend.
- **b** – Pointer to divisor.
- **c** – Pointer to result (remainder).

Returns

Returns SUCCESS on success; otherwise returns error if any pointer is NULL.

uint16_t **BigNum_Calculate_R2_Mod_N**(const *bn_int* *n, *bn_int* *result)

Computes R^2 mod n.

This function computes 2^4096 mod n using repeated left shifts and modular reduction at each step.

Parameters

- **n** – Pointer to modulus.
- **result** – Pointer to store computed result.

Returns

Returns SUCCESS on success; otherwise returns error if any pointer is NULL.

int **BigNum_Compare**(const *bn_int* *value1, const *bn_int* *value2)

Compares two big integers.

This function compares two big integers and determines whether one is greater than, less than, or equal to the other.

Parameters

- **a** – Pointer to first operand.
- **b** – Pointer to second operand.

Returns

Returns 1 if , BN_LT, or BN_EQ. Returns error if pointer is NULL.

int **BigNum_Compare_Digit**(const *bn_int* *a, *bn_digit* digit)

Compares a big integer with a single digit.

This function compares the big integer with a given digit value.

Parameters

- **a** – Pointer to big integer.
- **digit** – Digit value for comparison.

Returns

Returns BN_GT, BN_LT, or BN_EQ. Returns error if pointer is NULL.

uint16_t **BigNum_Subract**(const *bn_int* *a, const *bn_int* *b, *bn_int* *c)

Subtracts two big integers.

This function computes c = a - b assuming a >= b. Borrow handling is performed using extended precision.

Parameters

- **a** – Pointer to minuend.
- **b** – Pointer to subtrahend.
- **c** – Pointer to result.

Returns

Returns SUCCESS on success; otherwise returns error code if any pointer is NULL.

uint16_t **BigNum_Unsigned_Bin_Size**(const *bn_int* *a, size_t *size)

Returns the byte size of a big integer.

Calculates the byte size of the big integer in unsigned binary format. Returns 1 if the big integer is zero.

Parameters

- **a** – Pointer to the source big integer.
- **size** – Pointer to a size_t variable that will hold the computed byte size on success.

Returns

Returns SUCCESS on success; otherwise returns error code if any pointer is NULL.

uint16_t **BigNum_Read_Unsigned_Bin**(*bn_int* *a, const uint8_t *b, uint16_t len)

Reads an unsigned binary buffer into a big integer.

This function converts a big-endian byte array into an internal *bn_int* representation.

Parameters

- **a** – Pointer to big integer to be populated.
- **b** – Pointer to input byte buffer.
- **len** – Length of input buffer.

Returns

Returns SUCCESS on success; otherwise returns error code if any pointer is NULL.

uint16_t **BigNum_Write_Unsigned_Bin**(const *bn_int* *a, uint8_t *b, uint16_t len)

Writes a big integer to an unsigned binary buffer.

This function converts the big integer into a big-endian byte array representation.

Parameters

- **a** – Pointer to big integer.
- **b** – Pointer to output buffer.
- **len** – Size of output buffer.

Returns

Returns SUCCESS on success; otherwise returns error code if pointer is NULL or buffer is too small.

1.2 Delays

The file *delays.h* contains delay functions.

1.2.1 API Reference

void **Delay_MS**(uint32_t delayms)

The function `Delay_MS` is used to create a delay in milliseconds.

This function utilizes the RISC-V `mcycle` CSR to perform a busy-wait loop based on the CPU frequency, stalling execution for the requested duration in milliseconds.

Parameters

delayms – The duration of the delay in milliseconds (ms).

Returns

void

void **Delay_US**(uint32_t delayus)

The function `Delay_US` is used to create a delay in microseconds.

This function utilizes the RISC-V `mcycle` CSR to perform a busy-wait loop based on the CPU frequency, stalling execution for the requested duration in microseconds.

Parameters

delayus – The duration of the delay in microseconds (us).

Returns

void

1.3 Error Codes

1.3.1 Macros

Standard Error Codes

SUCCESS

Operation completed successfully.

EPERM

Operation not permitted.

ENOENT

No such file or directory.

ESRCH

No such process.

EINTR

Interrupted system call.

EIO

I/O error.

ENXIO

No such device or address.

E2BIG

Argument list too long.

ENOEXEC

Exec format error.

EBADF

Bad file number.

ECHILD

No child processes.

EAGAIN

Try again.

ENOMEM

Out of memory.

EACCES

Permission denied.

EFAULT

Bad address.

ENOTBLK

Block device required.

EBUSY

Device or resource busy.

EEXIST

File exists.

EXDEV

Cross-device link.

ENODEV

No such device.

ENOTDIR

Not a directory.

EISDIR

Is a directory.

EINVAL

Invalid argument.

ENFILE

File table overflow.

EMFILE

Too many open files.

ENOTTY

Not a typewriter.

ETXTBSY

Text file busy.

EFBIG

File too large.

ENOSPC

No space left on device.

ESPIPE

Illegal seek.

EROFS

Read-only file system.

EMLINK

Too many links.

EPIPE

Broken pipe.

EDOM

Math argument out of domain of function.

ERANGE

Math result not representable.

EDEADLK

Resource deadlock would occur.

ENAMETOOLONG

File name too long.

ENOLCK

No record locks available.

ENOSYS

Invalid system call number.

ENOTEMPTY

Directory not empty.

ELOOP

Too many symbolic links encountered.

EWOULDBLOCK

Operation would block (same as EAGAIN).

ENOMSG

No message of desired type.

EIDRM

Identifier removed.

ECHRNG

Channel number out of range.

EL2NSYNC

Level 2 not synchronized.

EL3HLT

Level 3 halted.

EL3RST

Level 3 reset.

ELNRNG

Link number out of range.

EUNATCH

Protocol driver not attached.

ENOCSI

No CSI structure available.

EL2HLT

Level 2 halted.

EBADE

Invalid exchange.

EBADR

Invalid request descriptor.

EXFULL

Exchange full.

ENOANO

No anode.

EBADRQC

Invalid request code.

EBADSLT

Invalid slot.

EBFONT

Bad font file format.

ENOSTR

Device not a stream.

ENODATA

No data available.

ETIME

Timer expired.

ENOSR

Out of streams resources.

ENONET

Machine is not on the network.

ENOPKG

Package not installed.

EREMOTE

Object is remote.

ENOLINK

Link has been severed.

EADV

Advertise error.

ESRMNT

Srmount error.

ECOMM

Communication error on send.

EPROTO

Protocol error.

EMULTIHOP

Multihop attempted.

EDOTDOT

RFS specific error.

EBADMSG

Not a data message.

EOVERFLOW

Value too large for defined data type.

ENOTUNIQ

Name not unique on network.

EBADFD

File descriptor in bad state.

EREMCHG

Remote address changed.

ELIBACC

Cannot access a needed shared library.

ELIBBAD

Accessing a corrupted shared library.

ELIBSCN

.lib section in a.out corrupted.

ELIBMAX

Attempting to link in too many shared libraries.

ELIBEXEC

Cannot exec a shared library directly.

EILSEQ

Illegal byte sequence.

ERESTART

Interrupted system call should be restarted.

ESTRPIPE

Streams pipe error.

EUSERS

Too many users.

ENOTSOCK

Socket operation on non-socket.

EDESTADDRREQ

Destination address required.

EMSGSIZE

Message too long.

EPROTOTYPE

Protocol wrong type for socket.

ENOPROTOOPT

Protocol not available.

EPROTONOSUPPORT

Protocol not supported.

ESOCKTNOSUPPORT

Socket type not supported.

EOPNOTSUPP

Operation not supported on transport endpoint.

EPFNOSUPPORT

Protocol family not supported.

EAFNOSUPPORT

Address family not supported by protocol.

EADDRINUSE

Address already in use.

EADDRNOTAVAIL

Cannot assign requested address.

ENETDOWN

Network is down.

ENETUNREACH

Network is unreachable.

ENETRESET

Network dropped connection because of reset.

ECONNABORTED

Software caused connection abort.

ECONNRESET

Connection reset by peer.

ENOBUFS

No buffer space available.

EISCONN

Transport endpoint is already connected.

ENOTCONN

Transport endpoint is not connected.

ESHUTDOWN

Cannot send after transport endpoint shutdown.

ETOOMANYREFS

Too many references: cannot splice.

ETIMEDOUT

Connection timed out.

ECONNREFUSED

Connection refused.

EHOSTDOWN

Host is down.

EHOSTUNREACH

No route to host.

EALREADY

Operation already in progress.

EINPROGRESS

Operation now in progress.

ESTALE

Stale file handle.

EUCLEAN

Structure needs cleaning.

ENOTNAM

Not a XENIX named type file.

ENAVAIL

No XENIX semaphores available.

EISNAM

Is a named type file.

EREMOTEIO

Remote I/O error.

EDQUOT

Quota exceeded.

ENOMEDIUM

No medium found.

EMEDIUMTYPE

Wrong medium type.

ECANCELED

Operation Canceled.

ENOKEY

Required key not available.

EKEYEXPIRED

Key has expired.

EKEYREVOKED

Key has been revoked.

EKEYREJECTED

Key was rejected by service.

EOWNERDEAD

Owner died.

ENOTRECOVERABLE

State not recoverable.

ERFKILL

Operation not possible due to RF-kill.

EHWPOISON

Memory page has hardware error.

Driver-Specific Error Codes

ENOACK

No acknowledgement received for data.

ENOACKDEV

No acknowledgement received for slave address.

ETRMODE

Invalid Transmit/Receive mode.

ECBUFFULL

Software buffer is full.

ECBUFEMPTY

Software buffer is empty.

EBUFEMPTY

Built-in hardware buffer is empty.

ELENEXCEED

Length of data field exceeded.

ECLKMODE

Invalid clock mode.

EINFREQ

Invalid frequency (too high or too low).

Platform Security Architecture Error codes

PSA_ERROR_ASN1_OUT_OF_DATA

Out of data when parsing an ASN1 data structure.

PSA_ERROR_ASN1_UNEXPECTED_TAG

ASN1 tag was of an unexpected value.

PSA_ERROR_ASN1_INVALID_LENGTH

Invalid length when parsing ASN1 structure.

PSA_ERROR_ASN1_LENGTH_MISMATCH

Actual length differs from expected length.

PSA_ERROR_ASN1_INVALID_DATA

ASN1 data is invalid.

PSA_ERROR_ASN1_ALLOC_FAILED

Memory allocation failed.

PSA_ERROR_ASN1_BUF_TOO_SMALL

Buffer too small when writing ASN1 data.

PSA_ERROR_ASN1_BAD_INPUT_STATE

Bad input state during ASN1 processing.

PSA_ERROR_RSA_BAD_INPUT_DATA

Bad input parameters to function.

PSA_ERROR_RSA_INVALID_PADDING

Input data contains invalid padding.

PSA_ERROR_RSA_KEY_GEN_FAILED

Key generation failed.

PSA_ERROR_RSA_KEY_CHECK_FAILED

Key failed validity check.

PSA_ERROR_RSA_PUBLIC_FAILED

Public key operation failed.

PSA_ERROR_RSA_PRIVATE_FAILED

Private key operation failed.

PSA_ERROR_RSA_VERIFY_FAILED

PKCS#1 verification failed.

PSA_ERROR_RSA_OUTPUT_TOO_LARGE

Output buffer for decryption too small.

PSA_ERROR_RSA_RNG_FAILED

Random generator failed to generate non-zero bytes.

PSA_ERROR_FPI_ILLEGAL_LENGTH

Illegal length in RSA fp_int.

PSA_ERROR_FPI_BAD_INPUT_DATA

Bad input data from RSA fp_int.

PSA_ERROR_GENERIC_ERROR

An unspecified error occurred.

PSA_ERROR_NOT_PERMITTED

Requested action denied by policy.

PSA_ERROR_NOT_SUPPORTED

Operation or parameter not supported.

PSA_ERROR_INVALID_ARGUMENT

Invalid parameter(s).

PSA_ERROR_INVALID_HANDLE

Key identifier is invalid.

PSA_ERROR_BAD_STATE

Action cannot be performed in current state.

PSA_ERROR_BUFFER_TOO_SMALL

Output buffer too small.

PSA_ERROR_ALREADY_EXISTS

Item already exists.

PSA_ERROR_DOES_NOT_EXIST

Item does not exist.

PSA_ERROR_INSUFFICIENT_MEMORY

Not enough runtime memory.

PSA_ERROR_INSUFFICIENT_STORAGE

Not enough persistent storage.

PSA_ERROR_STORAGE_FAILURE

Permanent storage failure.

PSA_ERROR_HARDWARE_FAILURE

Hardware failure detected.

PSA_ERROR_INSUFFICIENT_ENTROPY

Not enough entropy for requested action.

PSA_ERROR_INVALID_SIGNATURE

Signature, MAC, or hash is incorrect.

PSA_ERROR_INVALID_PADDING

Decrypted padding is incorrect.

PSA_ERROR_CORRUPTION_DETECTED

Tampering detected.

PSA_ERROR_DATA_CORRUPT

Stored data has been corrupted.

PSA_ERROR_DATA_INVALID

Data read from storage is not valid.

PSA_SUCCESS

The action was completed successfully.

Internal Kernel Errors

ERESTARTSYS

Restart the system call.

ERESTARTNOINTR

Restart the system call if not interrupted.

ERESTARTNOHAND

Restart if no handler is present.

ENOIOCTLCMD

No ioctl command.

ERESTART_RESTARTBLOCK

Restart by calling sys_restart_syscall.

EPROBE_DEFER

Driver requests probe retry.

EOPENSTALE

Open found a stale dentry.

NFSv3 Errors

EBADHANDLE

Illegal NFS file handle.

ENOTSYNC

Update synchronization mismatch.

EBADCOOKIE

Cookie is stale.

ENOTSUPP

Operation is not supported.

ETOOSMALL

Buffer or request is too small.

ESERVERFAULT

An untranslatable server error occurred.

EBADTYPE

Type not supported by server.

EJUKEBOX

Request initiated, but will not complete before timeout.

EIOCBQUEUED

IOCB queued; completion event will follow.

Legacy Definitions

ENOINST

Instruction not supported (soon to be deprecated).

1.4 IO Standard Library

The file *io.h* contain functions for standard io utilities.

1.4.1 API Reference

void **_print_**(const char *fmt, va_list ap)

Internal formatted output implementation.

Parses the format string and prints formatted output to the standard output using *putchar()*. Supports character, string, signed/unsigned integers, hexadecimal, octal, float, zero padding and precision handling.

Parameters

- **fmt** – Format control string.
- **ap** – Variable argument list containing values to print.

Returns

None.

int **printf**(const char *fmt, ...)

Prints formatted output to the standard output.

Wrapper function over *_print_()*. Processes the format string and corresponding arguments, then outputs the formatted result to the console.

Parameters

- **fmt** – Format control string.
- **...** – Variable arguments corresponding to format specifiers.

Returns

Returns SUCCESS if successfully printed the output.

int **scanf**(const char *format, ...)

Reads formatted input from the standard input.

Parses input from the console according to the specified format string. Supports string, character, decimal, float, hexadecimal, octal and unsigned conversions.

Parameters

- **format** – Format string specifying expected input pattern.
- **...** – Pointers to variables where parsed values are stored.

Returns

Returns SUCCESS on successful parsing. Returns error code on failure.

int **sscanf**(const char *buf, const char *fmt, ...)

Reads formatted data from a string buffer.

Parses the input buffer according to the given format string and stores extracted values into provided variables. Supports integer, string, character, pointer and scanset conversions with width and length modifiers.

Parameters

- **buf** – Input string buffer to parse.
- **fmt** – Format string specifying parsing rules.
- **...** – Pointers to variables where extracted values are stored.

Returns

Number of successfully assigned input items. Returns -1 if input failure occurs before any conversion.

1.5 Performance monitors API

1.5.1 Perf Monitor Configuration Structure

The Perf monitor configuration is defined by a structure containing various fields to set up Perf monitor parameters. Each field is described below.

```
struct Cache_perf_t
{
    uint64_t Ifbhit;            // Instruction Cache Fill Buffer Hits
    uint64_t Ifbrelease;        // Instruction Cache Fill Buffer␣
↪Releases
    uint64_t Imiss;             // Instruction Cache Misses
```

(continues on next page)

(continued from previous page)

```
    uint64_t Inc_access;       // Non-cached Instruction Access
    uint64_t Iaccess;          // Instruction Cache Accesses

    uint64_t Dread_access;     // Data Cache Read Accesses
    uint64_t Dwrite_access;    // Data Cache Write Accesses
    uint64_t Datomic_access;   // Data Cache Atomic Accesses
    uint64_t Dnc_read_access;  // Non-cached Data Read Accesses
    uint64_t Dnc_write_access; // Non-cached Data Write Accesses

    uint64_t Dread_miss;       // Data Cache Read Misses
    uint64_t Dwrite_miss;      // Data Cache Write Misses
    uint64_t Datomic_miss;     // Data Cache Atomic Misses

    uint64_t Dread_fbhit;      // Data Cache Read Fill Buffer Hits
    uint64_t Dwrite_fbhit;     // Data Cache Write Fill Buffer Hits
    uint64_t Datomic_fbhit;    // Data Cache Atomic Fill Buffer Hits

    uint64_t Dfbrelease;       // Data Cache Fill Buffer Releases
    uint64_t Dline_evictions;  // Data Cache Line Evictions

    uint64_t Itlb_miss;        // Instruction TLB Misses
    uint64_t Dtlb_miss;        // Data TLB Misses
};

struct Stalls_perf_t
{
  uint64_t rawstalls; // RAW stalls
  uint64_t exestalls; // Execution stalls
};

struct Branch_perf_t
{
  uint64_t misprediction; // Branch misprediction
  uint64_t jumps; // Number of jumps
  uint64_t branches; // Number of branches
};
```

(continues on next page)

(continued from previous page)

```
struct Arith_perf_t
{
  uint64_t floats; // Floating-point Operations
  uint64_t muldiv; // Multiply and Divide Operations
};
```

1.5.2 API Reference

Here are the relevant API functions for managing Perf monitor:

void **PERF_Mcycle_Init**()

Brief: Function to Clears the mcycle register.

Arguments: -

Returns: -

uint64_t **PERF_Get_Mcycle**()

Brief: Function to gets the current mcycle value

Arguments: -

Returns: mcycle count

uint64_t **get_mcycle_stop**()

Brief: Function to stop the mcycle value.

Arguments: -

Returns: mcycle count

void **PERF_cache_init**()

Brief: Function to initializes performance monitoring counters and events related to cache accesses.

Arguments: -

Returns: -

void **PERF_Print_Cache**(int iterations)

Brief: Function to prints various performance metrics related to instruction cache (ICache) and data cache (DCache) based on the given number of iterations.

Arguments:

- **iterations** (*int*): It is used to calculate the average value of each metric over a certain number of iterations. By dividing the value read from the perfor-

mance counters by the *iterations*

Returns: -

void **PERF_Print_Cache_MissPercentage**()

Brief: Function that calculates and prints the cache miss percentages for instruction cache (ICache) and data cache (DCache) reads and writes.

Arguments: -

Returns: -

void **PERF_Stalls_Init**()

Brief: Function to initializes performance monitoring counters for raw and execution stalls.

Arguments: -

Returns: -

void **PERF_Print_Stalls**(int iterations)

Brief: Function that prints the number of raw stalls and execution stalls per iteration.

Arguments:

- **iterations** (*int*): It is used to calculate the average value of each metric over a certain number of iterations. By dividing the value read from the performance counters by the *iterations*

Returns: -

void **PERF_Branches_Init**()

Brief: Function to initializes performance monitoring counters for branch instructions.

Arguments: -

Returns: -

void **PERF_Print_Branches**(int iterations)

Brief: Function to calculates and prints the number of branch mispredictions, jumps, branches, and the percentage of branch mispredictions based on the given number of iterations.

Arguments:

- **iterations** (*int*): It is used to calculate the average value of each metric over a certain number of iterations. By dividing the value read from the performance counters by the *iterations*

Returns: -

void **PERF_Arithops_Init**()

Brief: Function to initializes performance monitoring counters for floating-point operations and multiplication/division events.

Arguments: -

Returns: -

void **PERF_Print_Arithops**(int iterations)

Brief: Function to print the average number of floating-point operations and multiplication/division operations per iteration

Arguments:

- **iterations** (*int*): It is used to calculate the average value of each metric over a certain number of iterations. By dividing the value read from the performance counters by the *iterations*

Returns: -

void **PERF_Set_Event**(int counter, int event)

Brief: Function to set a specific event for a performance counter in a system.

Arguments:

- **counter** (*int*): It is used to set the event for the specified counter based on the value of the *counter* parameter.
- **event** (*int*): It is used to set the specific event to set for a performance counter.

Returns: -

struct PERF_Cache_t **PERF_Get_Cache**()

Brief: Function to reads performance counter values and stores them in a struct *PERF_Cache_t*.

Arguments: -

Returns: - It returns a struct of type *PERF_Cache_t* containing various performance data metrics such as instruction cache hits, misses, accesses, data cache reads, writes, misses, atomic accesses, TLB misses, and other cache-related information.

struct PERF_Stalls_t **PERF_Get_Stalls**()

Brief: Function to reads and returns performance stall data from specific counters.

Arguments: -

Returns: - A struct named *PERF_Stalls_t* is being returned. It contains two fields: *rawstalls* and *exestalls*, which are initialized with the values read from perfor-

mance counters *mhpmcounter10* and *mhpmcounter11* respectively.

struct PERF_Branch_t **PERF_Get_Branch**()

Brief: Function to reads performance data related to branch instructions and returns it in a struct.

Arguments: -

Returns: - A struct named *PERF_Branch_t* is being returned, which contains the values of mispredictions, jumps, and branches read from performance counters.

struct PERF_Arith_t **PERF_Get_Arith**()

Brief: Function to reads performance data related to floating-point operations and multiplication/division operations from specific control and status registers.

Arguments: -

Returns: - A struct named *PERF_Arith_t* is being returned by the function *PERF_Get_Arith()*. The struct contains two members: *floats* and *muldiv*, which are initialized with the values read from the performance monitoring counters *mhpmcounter8* and *mhpmcounter9* respectively.

void **PERF_Disable**(int counter)

Brief: Function to disables a specific performance counter based on the input parameter *counter*.

Arguments:

- **counter** (*int*): It is used to determine which performance counter to disable by writing specific values to control and event registers (*mhpmcounterX* and *mhpmeventX*).

Returns: -

void **PERF_Disable_All**()

Brief: Function to disables all performance monitoring event registers.

Arguments: -

Returns: -

void **PERF_Clear_All**()

Brief: Function to clears all performance monitor counter values.

Arguments: -

Returns: -

1.6 Utilities

The file *utils.h* contains basic utility functions.

1.6.1 Macros

CEIL_DIV_US(x, y)

Performs ceiling division: (x + y - 1) / y.

FLOOR_DIV_US(x, y)

Performs floor division: x / y.

POW2_MINUS1(n)

Generates a bitmask with the lowest `n` bits set.

Expands to (2^n - 1). Example: n = 3 → 0b00000111

Parameters

- **n** – Number of least significant bits to set.

Returns

Bitmask with lowest n bits set to 1.

CHECK_NULL(ptr)

Checks if a pointer is NULL and returns EFAULT if true.

IS_ALIGNED(addr, size)

Checks whether an address is aligned to a given size.

Casts the input pointer to `uint8_t*` first to avoid a direct `void*` to integer cast (MISRA compliance). It is then cast to `uintptr_t` so that alignment masking can be performed safely using an integer type capable of holding a pointer.

Parameters

- **addr** – Pointer/address to check.
- **size** – Alignment size (must be power of 2).

Returns

true if aligned, false otherwise.

DATA_SIZE_8

8-bit data width mode.

Selects 8-bit access for registers.

DATA_SIZE_16

16-bit data width mode.

Selects 16-bit access for registers.

DATA_SIZE_32

32-bit data width mode.

Selects 32-bit access for registers.

DATA_SIZE_64

64-bit data width mode.

Selects 64-bit access for registers.

CHECK_BIT_CONDITION(addr, bit_mask, cond, size)

Check whether a register bit matches the expected condition.

This macro performs a volatile read from the given register address, applies the specified bit mask, and compares the resulting bit state with the provided condition.

Parameters

- **addr** – Register address to read from.
- **bit_mask** – Bit mask to apply (example: (1U << bit_position)).
- **cond** – Expected bit condition:
  - 0U → Bit should be clear
  - 1U → Bit should be set
- **size** – Data width selection:
  - DATA_SIZE_8
  - DATA_SIZE_16
  - DATA_SIZE_32
  - DATA_SIZE_64

Returns

- 1U if the actual bit state matches the expected condition.
- 0U otherwise.

CSR_READ(csr, dest)

Reads the value of a Control and Status Register (CSR).

The destination must be a variable compatible with the target register width.

Parameters

- **csr** – The CSR name (e.g., pmpaddr0, pmpcfg0).
- **dest** – Variable where the CSR value will be stored.

CSR_WRITE(csr, val)

Writes a value to a Control and Status Register (CSR).

The value must be compatible with the target register width.

Parameters

- **csr** – The CSR name (e.g., pmpaddr0, pmpcfg0).
- **val** – Value to write to the CSR.

1.6.2 API Reference

uint64_t **Read_Data**(volatile const size_t *addr)

Returns the value stored at a given address.

Parameters

addr – The address at which value is located

Returns

uint64_t value

void **Write_Data**(volatile size_t *addr, size_t val)

The function `Write_Data` writes a value to a specific memory address.

This function performs a direct memory write operation.

Parameters

- **addr** – The target memory address to which data will be written.
- **val** – The value to be written to the address.

void **Millis_Init**(void)

The function `Millis_Init` initializes the millisecond counter.

This function initializes the timer hardware and updates the mtimecmp value to begin generating interrupts for elapsed time calculation.

uint64_t **Millis**(void)

The function `Millis` returns the milliseconds since the program started.

It retrieves the current value of a global millisecond counter which is incremented within the CLINT handler.

Returns

The number of milliseconds elapsed since initialization.

void **Exit**(int status)

The function `Exit` terminates the calling process with a status code.

This function implements process termination for RISC-V using the SBI System Reset Extension. It flushes memory via a FENCE instruction and calls SBI function ID 93.

Note

This function is guaranteed not to return. If SBI is unavailable, it falls back to an infinite Wait-For-Interrupt (WFI) loop.

Parameters

status – Exit status code (0 for success, non-zero for errors).

inline uint16_t **Wait_For_Timeout**(uint32_t address, uint32_t bit_mask, uint8_t cond, uint8_t size, uint64_t timeout)

Wait until a specific register bit matches the expected condition or a timeout occurs.

This function continuously reads the register at the specified memory address using the selected data width. It applies the provided bit mask and checks whether the resulting bit state matches the expected condition (0 or 1).

The timeout mechanism is implemented using the RISC-V `mcycle` CSR (machine cycle counter). The function monitors the elapsed CPU cycles and exits if the specified timeout value is exceeded.

Parameters

- **address** – Memory-mapped register address to monitor.
- **bit_mask** – Bit mask to apply (example: (1U << bit_position)).
- **cond** – Expected bit condition:
  - 0U → Wait until the bit is clear.
  - 1U → Wait until the bit is set.
- **size** – Data width selection for register access:
  - DATA_SIZE_8
  - DATA_SIZE_16
  - DATA_SIZE_32
  - DATA_SIZE_64
- **timeout** – Maximum number of CPU cycles to wait. This value

is compared against the difference between the current `mcycle` count and the start cycle count.

Returns

Returns 0 on SUCCESS, or a timeout/error code if the condition is not met within the specified cycle count.

Chapter 2. Peripherals API

2.1 Analog-to-Digital Converter (ADC) API

2.1.1 Required Includes

To use the ADC functionality, include the `adc.h` header file in your source code. This header provides the necessary definitions, functions, and configuration structures to interface with the adc hardware.

```
#include "adc.h"
```

2.1.2 Macros,Enums, Variables and Data Structures

Macros

ADC_SINGLE_CONVERSION

ADC performs a single conversion when start-of-conversion is triggered.

ADC_FREERUN

ADC continuously performs conversions until disabled.

ADC_INTR_DISABLE

ADC end-of-conversion interrupt is disabled.

ADC_INTR_ENABLE

ADC end-of-conversion interrupt is enabled.

ADC_INT_VREF

Select internal reference voltage.

ADC uses the internally generated bandgap-based reference voltage.

ADC_EXT_VREF

Select external reference voltage.

ADC uses reference voltage provided through external VREF pin.

ADC_INT_VBG

Enable internal bandgap generator.

Enables the on-chip bandgap circuit which generates a stable temperature-independent reference voltage.

ADC_EXT_VBG

Disable internal bandgap generator.

Disables the internal bandgap circuit.

Used when:

- External reference is selected

Enums

enum **adc_channel_t**

ADC Input Channel Selection.

Values:

enumerator **ADC_CHANNEL_0**

enumerator **ADC_CHANNEL_1**

enumerator **ADC_CHANNEL_2**

enumerator **ADC_CHANNEL_3**

enumerator **ADC_CHANNEL_4**

enumerator **ADC_CHANNEL_5**

enumerator **ADC_CHANNEL_6**

enumerator **ADC_CHANNEL_7**

enum **adc_resolution_t**

ADC Bit Resolution Selection.

Values:

enumerator **ADC_RES_6BIT**

enumerator **ADC_RES_8BIT**

enumerator **ADC_RES_10BIT**

enumerator **ADC_RES_12BIT**

Data Structures

struct **ADC_Config_t**

ADC Configuration Structure.

Defines the parameters required to initialize and control the ADC peripheral via the Control Configuration Register (CCR).

Public Members

adc_channel_t **channel**

Selects the ADC input channel.

Determines which analog input pin (channel 0–7) is internally connected to the ADC for conversion. Only the selected channel is sampled during operation.

adc_resolution_t **resolution**

Sets the ADC conversion resolution.

Defines the number of bits used to represent the digital result. Higher resolution increases measurement precision

bool **is_freerun**

Enables or disables free-running mode.

If enabled, the ADC continuously performs conversions. If disabled, the ADC performs a single conversion per trigger.

bool **is_intr_en**

Enables End-of-Conversion (EOC) interrupt.

When enabled, an interrupt is generated after each completed conversion. If disabled, software must poll the status flag to detect completion.

bool **is_vref_int**

Reference voltage source selection. true → Internal reference false → External VREF pin.

bool **is_vbg_int**

Bandgap generator control. true → Internal bandgap enabled false → Bandgap disabled.

2.1.3 API Reference

uint16_t **ADC_CCR**(const *ADC_Config_t* *cfg)

Configures the ADC Control Register.

Validates the provided configuration and writes to the hardware control register to initialize the ADC, select the channel, and start the Start of Conversion (SOC) sequence.

Parameters

cfg – Pointer to a constant *ADC_Config_t* structure.

Returns

SUCCESS (0) on successful configuration, or an error code:

- EFAULT: Null pointer provided.
- ECHRNG: Channel selection out of valid range.
- EINVAL: Invalid resolution or mode parameters.

uint16_t **ADC_Read_Output**(const *ADC_Config_t* *cfg, uint16_t *output)

Reads the converted digital output value.

Blocks until the End of Conversion (EOC) flag is set in hardware. Once ready, it extracts the data from the output register, applies the appropriate bit-mask based on the configured resolution, and shifts the result to provide a normalized value.

Note

This is a blocking call. Ensure the ADC has been triggered via *ADC_CCR()* before calling this function.

Parameters

- **cfg** – Pointer to the configuration structure used during initialization.
- **output** – Pointer to a variable where the converted ADC digital value will be stored.

Returns

SUCCESS (0) on successful completion, or an error code:

- EFAULT: Null pointer provided.
- ETIMEDOUT: Conversion did not complete within timeout.

uint16_t **ADC_Disable**(void)

Deactivates the ADC peripheral by clearing the control register.

Writes the ADC_DISABLE mask to the ADC control register. This immediately stops the peripheral's clock or power gated circuitry depending on the hardware abstraction.

Note

Any conversion currently in progress will be terminated abruptly. To restart, the peripheral must be reconfigured.

Returns

Always returns SUCCESS (0).

2.2 Core Local Interrupt (CLINT) API

2.2.1 Macros, Enums and Variables

Macros

CLINT_DIVISOR

Clock divisor applied to the CLINT input clock.

This value represents the division factor used to derive the effective CLINT timer clock from the system clock.

Variables

uint64_t ***msip**

Machine Software Interrupt Pending register.

uint64_t ***mtime**

Machine Time register.

uint64_t ***mtimecmp**

Machine Time Compare register.

2.2.2 API Reference

uint64_t **Get_MTIME**(void)

The function `Get_MTIME` returns the current 64-bit machine time.

It returns the mtime value.

Returns

64-bit unsigned integer representing the current ticks.

void **Config_Counter**(uint64_t value)

The function `Config_Counter` sets up the CLINT timer threshold.

It is used to set the mtimecmp register to the current mtime value plus the provided delta, scheduling the next timer interrupt.

Parameters

value – The delta value after which the interrupt happens.

Returns

SUCCESS if configuration was successful otherwise error code.

void **CLINT_Timer**(uint64_t timer_value)

The function `CLINT_Timer` configures the CLINT timer and triggers a software interrupt.

Detailed Description:

This function performs the following operations:

- Triggers a software interrupt by writing to the MSIP register.
- Disables specific interrupts by clearing bits in the MIE register.
- Enables global interrupts by setting bits in the MSTATUS register.
- Configures the next timer threshold using *Config_Counter*.

Parameters

`timer_value` – The value to configure the timer via `Config_Counter`.

Returns

SUCCESS if the operation was successful otherwise error code.

2.3 Direct Memory Access (DMA) API

2.3.1 Required Includes

This section lists the header files required to use the DMA driver APIs and associated data structures. These headers provide type definitions, error codes, and hardware abstraction required for DMA configuration and operation.

```
#include <stdint.h>
#include "stdbool.h"
#include "errors.h"
#include "secure_iot.h"
#include "dma.h"
```

2.3.2 Macros, Enums, Variables and Data Structures

Macros

`AES_INP_REG_ADDR`

AES input data register address.

`SHA_INP_REG_ADDR`

SHA input data register address.

RSA_INP_REG_ADDR

RSA input data register address.

AES_OUT_REG_ADDR

AES output data register address.

SHA_OUT_REG_ADDR

SHA output data register address.

RSA_OUT_REG_ADDR

RSA output data register address.

QSPI0_DATA_REG_ADDR

QSPI0 data register address.

QSPI1_DATA_REG_ADDR

QSPI1 data register address.

UART0_TX_REG_ADDR

UART0 TX register address.

UART1_TX_REG_ADDR

UART1 TX register address.

UART2_TX_REG_ADDR

UART2 TX register address.

UART3_TX_REG_ADDR

UART3 TX register address.

UART4_TX_REG_ADDR

UART4 TX register address.

UART0_RX_REG_ADDR

UART0 RX register address.

UART1_RX_REG_ADDR

UART1 RX register address.

UART2_RX_REG_ADDR

UART2 RX register address.

UART3_RX_REG_ADDR

UART3 RX register address.

UART4_RX_REG_ADDR

UART4 RX register address.

SPI0_TX_REG_ADDR

SPI0 TX register address.

SPI1_TX_REG_ADDR

SPI1 TX register address.

SPI2_TX_REG_ADDR

SPI2 TX register address.

SPI3_TX_REG_ADDR

SPI3 TX register address.

SPI0_RX_REG_ADDR

SPI0 RX register address.

SPI1_RX_REG_ADDR

SPI1 RX register address.

SPI2_RX_REG_ADDR

SPI2 RX register address.

SPI3_RX_REG_ADDR

SPI3 RX register address.

ITRACE_DATA_REG_ADDR

Instruction trace data register address.

ADC_DATA_REG_ADDR

ADC data register address.

PRO_IO_DUO_DATA_REG_ADDR

Pro IO Duo data register address.

PRO_IO_TETRA_DATA_REG_ADDR

Pro IO Tetra data register address.

PRO_IO_OCTA_DATA_REG_ADDR

Pro IO Octa data register address.

PRO_IO_FUSION_DATA_REG_ADDR

Pro IO Fusion data register address.

Enums

enum **DMA_Channel_t**

DMA channel identifiers.

Represents the available DMA channels in the controller.

Values:

enumerator **DMA_CHANNEL_0**

DMA Channel 0

enumerator **DMA_CHANNEL_1**

DMA Channel 1

enumerator **DMA_CHANNEL_2**

DMA Channel 2

enumerator **DMA_CHANNEL_3**

DMA Channel 3

enumerator **DMA_CHANNEL_4**

DMA Channel 4

enumerator **DMA_CHANNEL_5**

DMA Channel 5

enumerator **DMA_CHANNEL_6**

DMA Channel 6

enumerator **DMA_CHANNEL_7**

DMA Channel 7

enum **DMA_Priority_Levels**

DMA channel priority levels.

Defines the priority assigned to a DMA channel.

Values:

enumerator **DMA_PRIORITY_LOW**

Low priority

enumerator **DMA_PRIORITY_MEDIUM**

Medium priority

enumerator **DMA_PRIORITY_HIGH**

High priority

enumerator **DMA_PRIORITY_VERY_HIGH**

Very high priority

enum **DMA_Data_Size_t**

DMA data transfer size.

Represents the width of each DMA transfer unit.

Values:

enumerator **DMA_BYTE**

8-bit transfer

enumerator **DMA_TWOBYTE**

16-bit transfer

enumerator **DMA_FOURBYTE**

32-bit transfer

enumerator **DMA_EIGHTBYTE**

64-bit transfer

Data Structures

struct **DMA_Config_t**

DMA configuration structure.

This structure holds all parameters required for DMA transfer setup including source/destination addresses, transfer size, priority and data width.

Public Members

uint32_t ***src_addr**

Source address.

Pointer to the source location for DMA transfer.

- Can be a memory address or a peripheral register address.
- For peripheral transfers, this should point to the peripheral data register (e.g., UART RX, SPI RX).

Note

- Users are recommended to use the predefined macros provided in this driver for peripheral addresses (e.g., UARTx_RX_REG_ADDR, SPIx_RX_REG_ADDR) instead of hard-coding raw addresses.
- Ensure the address is cast to the appropriate type (uint32_t *) before assignment.

uint32_t ***dest_addr**

Destination address.

Pointer to the destination location for DMA transfer.

- Can be a memory address or a peripheral register address.
- For peripheral transfers, this should point to the peripheral data register (e.g., UART TX, SPI TX).

Note

Users are recommended to use the predefined macros provided in this driver for peripheral addresses (e.g., UARTx_TX_ADDR, SPIx_TX_ADDR) instead of hardcoding raw addresses.

- Ensure the address is cast to the appropriate type (uint32_t *) before assignment.

uint32_t **`transfer_length`**

Transfer length.

Specifies the total number of bytes to be transferred.

Note

- This value is specified in bytes.
- Must be aligned with the selected data size:
  - DMA_BYTE : any value
  - DMA_TWOBYTE : multiple of 2
  - DMA_FOURBYTE : multiple of 4
  - DMA_EIGHTBYTE : multiple of 8
- Incorrect alignment may lead to data corruption or transfer errors.
- Maximum transfer length accepted is \(2^{24} - 1\) (UINT24_MAX).
- Values exceeding 24 bits are masked off (upper 8 bits ignored).

DMA_Priority_Levels **`priority`**

DMA channel priority level.

Determines the priority of the DMA channel during arbitration when multiple channels request access.

Higher priority channels are serviced first.

uint8_t **src_data_size**

Source data width.

Specifies the data width for each transfer from source.

Valid values:

- DMA_BYTE : 8-bit transfer
- DMA_TWOBYTE : 16-bit transfer
- DMA_FOURBYTE : 32-bit transfer
- DMA_EIGHTBYTE : 64-bit transfer

uint8_t **dest_data_size**

Destination data width.

Specifies the data width for each transfer to destination.

Valid values:

- DMA_BYTE : 8-bit transfer
- DMA_TWOBYTE : 16-bit transfer
- DMA_FOURBYTE : 32-bit transfer
- DMA_EIGHTBYTE : 64-bit transfer

DMA_Channel_t **chn_no**

DMA channel number.

Selects the DMA channel to be used for the transfer.

Valid values:

- DMA_CHANNEL_0 to DMA_CHANNEL_7

uint32_t **flash_size**

External flash size.

Specifies the total size of the external QSPI flash memory.

- Used to dynamically determine the valid flash address range for DMA memory detection.

Note

- This value must be provided in bytes.
- The valid flash address range is computed as: [DMA_FLASH_START_ADDR, DMA_FLASH_START_ADDR + flash_size - 1]
- Ensure this value correctly matches the actual flash size configured in the system to avoid invalid memory access.

uint32_t **psram_size**

External PSRAM size.

Specifies the total size of the external PSRAM memory.

- Used to dynamically determine the valid PSRAM address range for DMA memory detection.

Note

- This value must be provided in bytes.
- The valid PSRAM address range is computed as: [DMA_PSRAM_START_ADDR, DMA_PSRAM_START_ADDR + psram_size - 1]
- Ensure this value correctly matches the actual PSRAM size configured in the system to avoid invalid memory access.

dma_qspi_type_t **dest_qspi_type**

QSPI destination device type for DMA transfer.

Indicates the type of memory device connected to the QSPI destination address. This field is used by the DMA driver to enforce access restrictions, particularly to prevent invalid write operations to Flash memory.

- When the destination address falls within a QSPI memory region, the DMA driver relies on this field to determine whether the target device is Flash or PSRAM.
- If the destination corresponds to QSPI Flash:
  - DMA write operations are NOT permitted.
  - The driver will reject the transfer and return an error.

- If the destination corresponds to QSPI PSRAM:
  - DMA write operations are allowed.

@usage

- This field MUST be explicitly set by the user when the destination address lies within a QSPI region.
- For non-QSPI destination addresses (e.g., RAM or peripherals), this field MUST be set to DMA_QSPI_UNKNOWN.

@validation

- If the destination address is within QSPI range AND:
  - dest_qspi_type == DMA_QSPI_UNKNOWN → The driver will return an error (invalid configuration).
  - dest_qspi_type == DMA_QSPI_FLASH → The driver will return an error (write to Flash not allowed).
  - dest_qspi_type == DMA_QSPI_PSRAM → The transfer is allowed.

Note

- The DMA driver does NOT perform runtime detection of the connected QSPI device (Flash/PSRAM).
- The correctness of this field is entirely the responsibility of the user/application layer.
- Providing incorrect information may lead to invalid memory access or undefined system behavior.

2.3.3 API Reference

uint16_t **DMA_Interrupt_Status**(const *DMA_Config_t* *dma_config, uint8_t *dma_status)

Get interrupt status for a DMA channel.

Reads the interrupt status register corresponding to the selected DMA channel and returns the status flags.

The interrupt status is returned in the lower 4 bits of `dma_status`, where each bit represents a specific interrupt condition. Multiple interrupt conditions may be set simultaneously.

Possible status values: * 0x0 : No interrupt pending * 0x1 : Global interrupt flag * 0x2 : Transfer complete * 0x4 : Half transfer reached * 0x8 : Transfer error

occurred * Combination of above values indicates multiple events

Parameters

- `dma_config` – **[in]** Pointer to DMA configuration structure.
- `dma_status` – **[out]** Pointer to store interrupt status bits.

Return values

- `SUCCESS` – Operation successful
- `ECHRNG` – Invalid DMA channel number
- `EFAULT` – Null pointer passed

uint16_t **DMA_Clear_Interrupt_Flags**(const *DMA_Config_t* *dma_config, bool transfer_error_int_flag, bool half_transfer_int_flag, bool transfer_complete_int_flag, bool global_int_flag)

Clear DMA interrupt flags.

Clears the selected interrupt flags for the given DMA channel.

Parameters

- `dma_config` – **[in]** Pointer to DMA configuration structure.
- `transfer_error_int_flag` – **[in]** Set to true to clear transfer error flag.
- `half_transfer_int_flag` – **[in]** Set to true to clear half-transfer flag.
- `transfer_complete_int_flag` – **[in]** Set to true to clear transfer complete flag.
- `global_int_flag` – **[in]** Set to true to clear global interrupt flag.

Return values

- `SUCCESS` – Operation successful
- `ECHRNG` – Invalid DMA channel number
- `EFAULT` – Null pointer passed

uint16_t **DMA_Enable_Interrupts**(const *DMA_Config_t* *dma_config, bool transfer_error_int_en, bool half_transfer_int_en, bool transfer_complete_int_en)

Enable selected DMA interrupts.

Enables the specified interrupt sources for the given DMA channel.

Parameters

- **dma_config** – **[in]** Pointer to DMA configuration structure.
- **transfer_error_int_en** – **[in]** Enable transfer error interrupt.
- **half_transfer_int_en** – **[in]** Enable half-transfer interrupt.
- **transfer_complete_int_en** – **[in]** Enable transfer complete interrupt.

Return values

- **SUCCESS** – Operation successful
- **ECHRNG** – Invalid DMA channel number
- **EFAULT** – Null pointer passed

uint16_t **DMA_Disable_Interrupts**(const *DMA_Config_t* *dma_config, bool transfer_error_int_en, bool half_transfer_int_en, bool transfer_complete_int_en)

Disable selected DMA interrupts.

Disables the specified interrupt sources for the given DMA channel.

Parameters

- **dma_config** – **[in]** Pointer to DMA configuration structure.
- **transfer_error_int_en** – **[in]** Disable transfer error interrupt.
- **half_transfer_int_en** – **[in]** Disable half-transfer interrupt.
- **transfer_complete_int_en** – **[in]** Disable transfer complete interrupt.

Return values

- **SUCCESS** – Operation successful
- **ECHRNG** – Invalid DMA channel number
- **EFAULT** – Null pointer passed

uint16_t **DMA_Channel_Status**(const *DMA_Config_t* *dma_config, uint8_t *state)

Get DMA channel status.

Retrieves the current enable/disable status of the DMA channel.

Parameters

- **dma_config** – **[in]** Pointer to DMA configuration structure.
- **state** – **[out]** Pointer to store channel state.
  - 0 : Channel disabled
  - 1 : Channel enabled

Return values

- **SUCCESS** – Operation successful
- **ECHRNG** – Invalid DMA channel number
- **EFAULT** – Null pointer passed
- **ETIMEDOUT** – Timeout while checking channel state

uint16_t **DMA_Channel_Set_State**(const *DMA_Config_t* *dma_config, bool enable)

Set DMA channel state (Enable/Disable).

Enables or disables the specified DMA channel.

Parameters

- **dma_config** – **[in]** Pointer to DMA configuration structure.
- **enable** – **[in]** Channel state control:
  - true : Enable channel
  - false : Disable channel

Return values

- **SUCCESS** – Operation successful
- **ECHRNG** – Invalid DMA channel number
- **EFAULT** – Null pointer passed
- **ETIMEDOUT** – Timeout while enabling/disabling channel

uint16_t **DMA_Transfer_Configure**(const *DMA_Config_t* *dma_config)

Configure DMA transfer parameters.

Configures the DMA channel with source and destination addresses, transfer length, priority level, and data width.

Note

- Channel must be disabled before calling this function.
- Transfer length must be aligned with data size.

Parameters

dma_config – **[in]** Pointer to DMA configuration structure.

Return values

- **SUCCESS** – Operation successful
- **ECHRNG** – Invalid DMA channel number
- **EFAULT** – Null pointer passed
- **ETIMEDOUT** – Timeout while waiting for channel disable

- **EINVAL** – Invalid configuration parameters

2.4 General Purpose Input and Output(GPIO) API

2.4.1 Required Includes

To use the GPIO functionality, include the *gpio.h* header file in your source code. This header provides the necessary definitions, functions, and configuration structures to interface with the GPIO.

```
#include "gpio.h"
```

2.4.2 Macros, Enums and Data Structures

Macros

GPIO_IN

Represents the GPIO input direction.

GPIO_OUT

Represents the GPIO output direction.

GPIO_PIN(x)

Sets bit x in a 64-bit mask. e.g., Assign gpio pin 5 as *GPIO_PIN(5)*

Enums

enum **PRO_IO_Number**

Pro IO width selection.

This enumeration defines the available Pro IO hardware configurations. Each value represents the specific grouping of GPIO pins.

Values:

enumerator **PRO_IO_DUO**

The value of this enum is 0U. It represents a DUO Pro IO

enumerator **PRO_IO_TETRA**

The value of this enum is 1U. It represents a TETRA Pro IO

enumerator **PRO_IO_OCTA**

The value of this enum is 2U. It represents an OCTA Pro IO

enumerator **PRO_IO_FUSION**

The value of this enum is 3U. It represents a combined FUSION Pro IO

enum **PRO_IO_Direction**

Pro IO data direction.

This enumeration defines whether the Pro IO is used for reading data from GPIO pins or writing data to GPIO pins.

Values:

enumerator **PRO_IO_READ**

The value of this enum is 0U. Data is enqueued (read from GPIO pins)

enumerator **PRO_IO_WRITE**

The value of this enum is 1U. Data is dequeued (written to GPIO pins)

enum **PRO_IO_Clock_Edge_Select**

Pro IO clock edge selection.

This enumeration specifies the clock edge on which Pro IO data is sampled or driven.

Values:

enumerator **CLK_POSITIVE_EDGE**

The value of this enum is 0U. Data is captured on the positive clock edge

enumerator **CLK_NEGATIVE_EDGE**

The value of this enum is 1U. Data is captured on the negative clock edge

enum **PRO_IO_Clock_Source**

Pro IO clock source selection.

This enumeration defines whether the Pro IO operates using an internally generated clock or an external clock source.

Values:

enumerator **CLK_INTERNAL**

The value of this enum is 0U. Internal clock source is used

enumerator **CLK_EXTERNAL**

The value of this enum is 1U. External clock source is used

enum **PRO_IO_Mode**

Pro IO operating modes.

This enumeration defines all valid combinations of clock source and data transfer direction for GPIO pro_ioing.

Values:

enumerator **MODE_EXT_CLK_READ**

The value of this enum is 1U. External clock is used and data is read from the Pro IO

enumerator **MODE_EXT_CLK_WRITE**

The value of this enum is 2U. External clock is used and data is written to the Pro IO

enumerator **MODE_INT_CLK_READ**

The value of this enum is 3U. Internal clock is used and data is read from the Pro IO

enumerator **MODE_INT_CLK_WRITE**

The value of this enum is 4U. Internal clock is used and data is written to the Pro IO

Data Structures

struct **PRO_IO_Struct_t**

Pro IO configuration structure.

This structure contains all parameters required to configure a Pro IO for parallel data transfer. It defines the pro_io width, clock source, clock edge, data direction, and data size used during pro_ioed GPIO operations.

Public Members

uint8_t **pro_io_num**

Pro IO width selection.

This parameter selects the number of GPIO pins grouped together to form a pro_io interface.

Note

Uses *PRO_IO_Number*. Valid values are:

- PRO_IO_DUO(0)
- PRO_IO_TETRA(1)
- PRO_IO_OCTA(2)
- PRO_IO_FUSION(3)

uint32_t **prescale**

Pro IO clock prescaler value.

This parameter defines the prescaler applied to the internal clock source used by the Pro IO.

Note

This field is used only when *clk_sel* is set to CLK_INTERNAL.

uint32_t **data_size**

Pro IO data width selection.

This parameter specifies the number of bits transferred per pro_io operation.

Note

Uses PRO_IO_Data_Size. Valid values are:

- *DATA_SIZE_8(1)*
- *DATA_SIZE_16(2)*
- *DATA_SIZE_32(3)*

uint8_t **direction**

Pro IO data transfer direction.

This parameter selects whether data is read from GPIO pins into the pro_io or written from the pro_io to GPIO pins.

Note

Uses *PRO_IO_Direction*. Valid values are:
- PRO_IO_READ(0)
- PRO_IO_WRITE(1)

uint8_t **clk_edge_sel**

Pro IO clock edge selection.

This parameter defines the clock edge on which data is sampled during pro_io operation.

Note

Uses GPIO_Clock_Edge_Select. Valid values are:
- CLK_POSITIVE_EDGE(0)
- CLK_NEGATIVE_EDGE(1)

uint8_t **clk_sel**

Pro IO clock source selection.

This parameter selects whether the Pro IO operates using an internal clock or an external clock source.

Note

Uses GPIO_Clock_Source. Valid values are:
- CLK_INTERNAL(0)
- CLK_EXTERNAL(1)

uint8_t **mode**

Pro IO operating mode.

This parameter defines the complete pro_io operating mode by combining the clock source and data transfer direction.

Note

Uses *PRO_IO_Mode*. Valid values are:

- MODE_EXT_CLK_READ(1)
- MODE_EXT_CLK_WRITE(2)
- MODE_INT_CLK_READ(3)
- MODE_INT_CLK_WRITE(4)

uint64_t **timeout**

Pro IO operation timeout value.

This parameter defines the maximum number of CPU cycles to wait for hardware status flags (e.g., Buffer Ready or Not Full) before aborting an operation.

Note

This value is compared against the RISC-V `mcycle` CSR to ensure non-blocking execution during hardware polling.

2.4.3 API Reference

uint16_t **GPIO_Config**(uint64_t gpio_pins, bool direction)

The function `GPIO_Config` initializes the GPIO instance and set the direction.

It is used to initializes the GPIO instance and set the direction of a GPIO pin which configure the GPIO port as input or output.

Parameters

- **gpio_pins** – The `gpio_pins` parameter specifies the pins that you want to set in the GPIO_DIRECTION register.
- **direction** – The direction parameter is an bool value that specifies the direction of the GPIO pin. It can be either 0 or 1, where 0 represents input and 1 represents output.

Returns

The function `GPIO_Config` returns `SUCCESS`

uint16_t **GPIO_Pin_Set**(uint64_t gpio_pins)

The function `GPIO_Pin_Set` sets multiple GPIO pins based on the input arguments provided.

It is used to set the value of a GPIO pin in a GPIO instance, it also sets for multiple

GPIO pins.

Parameters

gpio_pins – The `gpio_pins` parameter specifies the pins that you want to set in the GPIO_SET register.

Returns

The function `GPIO_Pin_Set` returns `SUCCESS`

uint16_t **GPIO_Pin_Clear**(uint64_t gpio_pins)

The function `GPIO_Pin_Clear` clears multiple GPIO pins based on the input arguments provided.

It is used to clear the value of a GPIO pin in a GPIO instance, it also clears for multiple GPIO pins.

Parameters

gpio_pins – The `gpio_pins` parameter specifies the pins that you want to clear in the GPIO_CLEAR register.

Returns

The function `GPIO_Pin_Clear` returns `SUCCESS`

uint16_t **GPIO_Pin_Toggle**(uint64_t gpio_pins)

The function `GPIO_Pin_Toggle` toggles multiple GPIO pins based on the input arguments provided.

It is used to toggle the value of a GPIO pin in a GPIO instance, it also toggle for multiple GPIO pins.

Parameters

gpio_pins – The `gpio_pins` parameter specifies the pins that you want to toggle in the GPIO_TOGGLE register.

Returns

The function `GPIO_Pin_Toggle` returns `SUCCESS`

uint16_t **GPIO_Interrupt_Config**(uint64_t gpio_pins, uint8_t pin_state)

The function `GPIO_Interrupt_Config` configures the interrupt for a specific GPIO pin.

It is used to configure the interrupt for GPIO pins based on whether it is active low or active high.

Parameters

- **gpio_pin** – The pin number for which the interrupt configuration is being set.
- **pin_state** – This parameter is a boolean value that determines

whether the interrupt is active low or active high. If it is set to true (non-zero), the interrupt is active low, and if it is set to false (zero), the interrupt is active high.

Returns

The function `GPIO_Interrupt_Config` returns `SUCCESS`

uint16_t **GPIO_Read_Data**(uint64_t *read_data)

The function `GPIO_Read_Data` returns the value of the data register in a GPIO instance.

It is used to return the value of the GPIO_DATA register

Parameters

read_data – The `read_data` reads data from GPIO registers and stores the result in the memory location pointed to by `read_data`.

Returns

The function `GPIO_Read_Data` returns `SUCCESS`

uint16_t **GPIO_Read_Pin_Status**(uint64_t gpio_pin, uint8_t *pin_status)

The function `GPIO_Read_Pin_Status` reads the status of GPIO instance.

It is used to read the status of the specific pin in the GPIO instance

Parameters

- **gpio_pin** – gpio_pin The `gpio_pin`represents a GPIO pin.
- **pin_status** – The `pin_status` parameter is a pointer where the status of the GPIO pin will be stored after reading.

Returns

The function `GPIO_Read_Data` returns `SUCCESS`

uint16_t **GPIO_Write_Data**(uint64_t data_word)

The function `GPIO_Write_Data` sets the data register of a GPIO instance to a given data.

It is used to sets the data register of a GPIO instance to a given data.

Parameters

- **gpio_pin** – The pin number to write the data.
- **data_word** – The `data_word` parameter contains data to be written to GPIO pins.

Returns

The function `GPIO_Write_Data` is returns `SUCCESS`.

uint16_t **PRO_IO_Config**(*PRO_IO_Struct_t* *config)

The function `PRO_IO_Config` configures the Pro IO hardware based on the provided configuration parameters.

Parameters

config – Pointer to a constant *PRO_IO_Struct_t* structure. This parameter defines the operational parameters for the Pro IO hardware, including: such as pro_io number, mode, data size, direction, clock edge selection, clock selection, prescale value.

Returns

Returns a 16-bit status code:

- SUCCESS: Configuration applied successfully.
- EFAULT: The provided config pointer is NULL.
- EBUFFERNUM: pro_io_num is out of range (0-3). Check if the correct instance ID is passed.
- EINVALSIZE: data_size is invalid. Ensure it is DATA_SIZE_8, DATA_SIZE_16, or DATA_SIZE_32.
- EBITSEL: Invalid selection for direction, clock edge, or clock source (must be binary 0 or 1).
- EINVALMODE: The operation mode is outside the allowed range (1-4).

uint16_t **PRO_IO_Write**(*PRO_IO_Struct_t* *config, uint32_t data)

The function `PRO_IO_Write` writes data to a specified register based on the configuration provided.

Parameters

- **config** – Pointer to a constant *PRO_IO_Struct_t* structure. This parameter defines the operational parameters for the Pro IO hardware, including: such as pro_io number, data size.
- **data** – The `data` parameter is used to write data to a specific register based on the configuration provided in the `config` parameter.

Returns

Returns a 16-bit status code:

- SUCCESS: Configuration applied successfully.
- EJUKEBOX: Timeout occurred while waiting for the Pro IO FIFO/Status ready flag.
- EBUFFERNUM: pro_io_num is out of range (0-3). Check if the

correct instance ID is passed.

- EINVALSIZE: data_size is invalid. Ensure it is DATA_SIZE_8, DATA_SIZE_16, or DATA_SIZE_32.

uint16_t **PRO_IO_Read**(*PRO_IO_Struct_t* *config, uint32_t *rx_data)

The function `PRO_IO_Read` reads data from a specified PRO_IO port based on the configuration provided.

Parameters

- **config** – Pointer to a constant *PRO_IO_Struct_t* structure. This parameter defines the operational parameters for the Pro IO hardware, including: such as pro_io number,data size.
- **rx_data** – The `rx_data` parameter read from the Pro IO register will be stored. The function reads data from the Pro IO register based on the configuration provided and stores it in the `rx_data

Returns

Returns a 16-bit status code:

- SUCCESS: Configuration applied successfully.
- EJUKEBOX: Timeout occurred while waiting for the Pro IO FIFO/Status ready flag.
- EBUFFERNUM: pro_io_num is out of range (0-3). Check if the correct instance ID is passed.
- EINVALSIZE: data_size is invalid. Ensure it is DATA_SIZE_8, DATA_SIZE_16, or DATA_SIZE_32.

uint16_t **PRO_IO_Wait_Till_tx**(*PRO_IO_Struct_t* *config)

The function `PRO_IO_Wait_Till_tx` waits until the specified PRO_IO channel is ready for transmission.

Parameters

config – Pointer to a constant *PRO_IO_Struct_t* structure. This parameter defines the operational parameters for the Pro IO hardware, such as pro_io number.

Returns

Returns a 16-bit status code:

- SUCCESS: Configuration applied successfully.
- EJUKEBOX: Timeout occurred while waiting for the Pro IO FIFO/Status ready flag.

- EBUFFERNUM: pro_io_num is out of range (0-3). Check if the correct instance ID is passed.

uint16_t **PRO_IO_Disable**(*PRO_IO_Struct_t* *config)

The function PRO_IO_Disable disables a specific PRO_IO pin based on the provided configuration.

Parameters

config – Pointer to a constant *PRO_IO_Struct_t* structure. This parameter defines the operational parameters for the Pro IO hardware, such as pro_io number.

Returns

Returns a 16-bit status code:

- SUCCESS: Configuration applied successfully.
- EBUFFERNUM: pro_io_num is out of range (0-3). Check if the correct instance ID is passed.

2.5 General Purpose Timer (GPTimer) API

2.5.1 Required Includes

To use the GPTimer, include the *gptimer.h* header file in your source code. This header provides the necessary definitions, functions, and configuration structures to interface with the GPTimer hardware.

```
#include "gptimer.h"
```

2.5.2 Macros,Enums and Data Structures

Macros

GPTIMER0

Macro for GPTIMER instance 0 handle.

GPTIMER1

Macro for GPTIMER instance 1 handle.

GPTIMER2

Macro for GPTIMER instance 2 handle.

GPTIMER3

Macro for GPTIMER instance 3 handle.

GPT_PWM_MODE

The value of this macro is 0U. Pulse Width Modulation mode.

GPT_UP_COUNT

The value of this macro is 1U. Standard Up-counting mode.

GPT_DOWN_COUNT

The value of this macro is 2U. Standard Down-counting mode.

GPT_UPDOWN_COUNT

The value of this macro is 3U. Up-Down counting mode.

Data Structures

struct **GPTIMER_Config_t**

GPTIMER configuration structure.

This structure contains all parameters required to initialize and configure a General Purpose Timer (GPTIMER).

Public Members

const GPTIMER_Instance_t ***gpt_num**

GPTIMER hardware instance handle.

This parameter selects the GPTIMER hardware block to be used. Valid instances are GPTIMER1, GPTIMER2, GPTIMER3 and GPTIMER4.

> **Note**
>
> GPTIMER instance 0 is reserved for serial debug communication.

uint8_t **mode**

GPTIMER operating mode.

Selects the counting or PWM mode of the timer.

Note

Valid range: 0 <= mode < 4.

uint32_t **period**

Timer period value.

Specifies the period up to which the GPTIMER counts before overflowing, underflowing, or resetting (depending on the selected mode).

uint32_t **prescaler**

Clock prescaler value.

Divides the input clock frequency before it is fed to the GPTIMER counter.

uint32_t **dutycycle**

PWM duty cycle value.

Defines the duty cycle of the PWM output when the timer operates in PWM mode.

Note

The value provided by the user is normalized internally. The input is taken modulo 100, ensuring the effective duty cycle is always in the range 0 to 100 (inclusive). A modulo result of 0 represents a 100% duty cycle. A duty cycle of 0% is achieved only when the user explicitly provides a value of 0U.

bool **interrupt_en**

Interrupt enable flag.

Enables or disables GPTIMER interrupts based on the selected mode.

Note

1 = Enable interrupt, 0 = Disable interrupt

bool **cnt_en**

Continuous count enable flag.

Enables continuous counting mode for the GPTIMER.

Note

1 = Continuous counting enabled

bool **capture_val**

Capture input enable flag.

Enables capturing of the external input value.

bool **output_en**

Output enable flag.

Enable the output signal in PWM mode by setting the output enable bit in the control/status register.

Note

The value provided by the user is normalized internally using `dutycycle % 100`. This ensures that the effective duty cycle always falls within the range **0 to 100**. A modulo result of `0` is interpreted as a **100% duty cycle**. A true **0% duty cycle** is applied only when the user explicitly provides a value of `0U`.

2.5.3 API Reference

uint16_t **GPT_Init**(*GPTIMER_Config_t* const *gptimer_config)

The function `GPT_Init` initialises the gptimer with different modes and configures necessary registers.

It is used to select the specific GPT instance, set the mode, set the prescaler, set the dutycycle, enable continous count and set the input bit to be captured.

Parameters

gptimer – The `gptimer` parameter in the `GPT_Init` function is a struct that contains several properties related to configuring a General Purpose Timer (GPT).It has the following properties: gpt_num, mode, interrupt_en, period, prescaler, dutycycle, cnt_en, capture_val, output_en.

Returns

SUCCESS if the configuration was successful, else ERROR code.

uint16_t **GPT_Reset**(*GPTIMER_Config_t* const *gptimer_config)

The function ` GPT_Reset` resets the counter, interrupts and register values.

It is used to reset the counter, interrupts and register values. for the specified GPTIMER instance.

Parameters

gptimer – The `gptimer` parameter in the `GPT_Init` function is a struct that contains several properties related to configuring a General Purpose Timer (GPT).It has the following properties: gpt_num, mode, interrupt_en, period, prescaler, dutycycle, cnt_en, capture_val, output_en.

Returns

SUCCESS if the configuration was successful, else ERROR code.

2.6 Inter-Integrated Circuit (I2C) API

2.6.1 Required Includes

To use the I2C functionality, include the *i2c.h* header file in your source code. This header provides the necessary definitions, functions, and configuration structures to interface with the I2C hardware.

```
#include "i2c.h"
```

2.6.2 Macros, Enums and Data Structures

Macros

START_BIT

Sends a START condition on the I2C bus.

STOP_BIT

Sends a STOP condition on the I2C bus.

I2C0

Macro for I2C instance 0 handle.

`I2C1`

Macro for I2C instance 1 handle.

2.6.3 API Reference

uint16_t **I2C_Init**(const I2C_Instance_t *i2c_num, uint32_t clock_frequency)

Used to set clock frequency for I2C communication.

This function initializes the specified I2C instance and sets its clock frequency for master transmit and receive operations.

Parameters

- **i2c_num** – The parameter *i2c_num* is a pointer to the I2C instance handle. Specifies the I2C peripheral instance to configure. Use I2C0 or I2C1 as defined in the I2C instance macros.
- **clock_frequency** – The parameter *clock_frequency* is an unsigned integer specifying the desired I2C communication clock frequency.

Returns

SUCCESS when successfully initialised; on failure, returns an error code.

uint16_t **I2C_Transmit**(const I2C_Instance_t *i2c_num, uint8_t slave_address, uint8_t *data, uint8_t length, uint8_t mode)

Used to send data to an I2C slave device.

This function transmits a sequence of bytes from the specified data buffer to the given I2C slave address.

Parameters

- **i2c_num** – The parameter *i2c_num* is a pointer to the I2C instance handle. Specifies the I2C peripheral instance to configure. Use I2C0 or I2C1 as defined in the I2C instance macros.
- **slave_address** – The parameter *slave_address* is an 7 bit address that represents the I2C slave address.
- **data** – The parameter *data* is an pointer to buffer containing data to send.
- **length** – The parameter *length* is the number of bytes to transmit from the data buffer.

- **mode** – The parameter *mode* is used to set whether to send start bit, stop bit and repeated start bit in transaction.
  - START_BIT : sends start bit
  - STOP_BIT : sends stop bit
  - (START_BIT | STOP_BIT) : sends repeated start bit

Returns

SUCCESS if the transmission is successful; on failure, returns an error code.

uint16_t **I2C_Receive**(const I2C_Instance_t *i2c_num, uint8_t slave_address, uint8_t *data, uint8_t length, uint8_t mode)

Used to receive data from an I2C slave device.

This function reads a sequence of bytes from the specified I2C slave address into the provided data buffer.

Parameters

- **i2c_num** – The parameter *i2c_num* is a pointer to the I2C instance handle. Specifies the I2C peripheral instance to configure. Use I2C0 or I2C1 as defined in the I2C instance macros.
- **slave_address** – The parameter *slave_address* is an 7 bit address that represents the I2C slave address.
- **data** – The parameter *data* is an pointer to buffer containing data to send.
- **length** – The parameter *length* is the number of bytes to transmit from the data buffer.
- **mode** – The parameter *mode* is used to set whether to send start bit, stop bit and repeated start in transaction.
  - START_BIT : sends start bit
  - STOP_BIT : sends stop bit
  - (START_BIT | STOP_BIT) : sends repeated start bit

Returns

SUCCESS if the reception is successful; on failure, returns an error code.

2.7 Itrace API

2.7.1 API Reference

Here are the relevant API functions for managing itrace:

void **ITRACE_Ctrl**()

Brief: This function sets specific bits in a control register to enable instruction tracing.

Arguments: -

Returns: -

void **ITRACE_Ram_Ctrl**(uint32_t buffer)

Brief: This function configures the ITRACE RAM controller and initiates a DMA transfer.

Arguments: -

- **buffer** (*uint32_t*): It is used to send the address through DMA.

Returns: -

int ***ITRACE_Read_Ram_Data**(int *count)

Brief: This function reads data from a RAM buffer and returns the data along with the count of elements read.

Arguments: -

- **count** (*uint32_t*): It is used to store the count of elements read from the ITRACE_RAM data.

Returns: - It returns a pointer to an array, data read from a memory location

void **ITRACE_Filter_val**(uint8_t MatchMode, uint8_t S_Function, uint8_t P_Function)

Brief: This function sets the control register *COMP1_CTRL* with specified match mode, S function, and P function values for ITRACE.

Arguments: -

- **MatchMode** (*uint8_t*): It specifies the comparison mode for the ITRACE filter. It determines how the incoming data is compared with the filter settings.
- **S_Function** (*uint8_t*): It is used to specify the type of source function for the trace comparison.
- **P_Function** (*uint8_t*): It represents a specific function or operation related to the ITRACE component. The exact functionality of *P_Function* would depend

on the specific implementation and design of the ITRACE module in your system.

Returns: -

void **ITRACE_Comp_Ctrl**(uint8_t Comp_num)

Brief: This function sets filter control settings based on the input *Comp_num*.

Arguments: -

- **Comp_num** (*uint8_t*): It is used determine the filter control settings.

Returns: -

void **ITRACE_Disable_Ctrl_Reg**()

Brief: This function disables the control register of the ITRACE module.

Arguments: -

Returns: -

void **ITRACE_Disable_RAM_ctrl_Reg**()

Brief: This function disables the RAM control register of the ITRACE module.

Arguments: -

Returns: -

void **ITRACE_RAM_Wrap**()

Brief: This function sets a flag in the control register to stop tracing when the trace RAM wraps around.

Arguments: -

Returns: -

2.8 Pin Multiplexing (Pinmux) API

2.8.1 Required Includes

Include the below header file to access the required Pinmux functions.

```
#include "pinmux.h"
```

2.8.2 API Reference

uint16_t **PINMUX_Reset**(void)

The function *PINMUX_Reset()* enables GPIO functionality.

The function *PINMUX_Reset()* enables the pinmux for GPIOs 32–44 to select GPIO functionality. Since all other GPIOs are already enabled for GPIO functionality in the default state, they are not reconfigured.

Parameters

none –

Returns

SUCCESS after the Pinmux for GPIO 32-44 is configured.

uint16_t **PINMUX_PWM**(uint8_t num, bool enable)

The function PINMUX_PWM enables PWM functionality for a specified pin.

The function PINMUX_PWM checks whether the provided PWM channel number is less than 14. If it is, the corresponding pinmux configuration register is set to 1 to enable the PWM channel.

Parameters

- **num** – The PWM channel number to be enabled.
- **enable** – If set to true, configures the pin for PWM functionality. If set to false, reverts the pin to GPIO functionality.

Returns

SUCCESS if the pinmux configuration was applied successfully; otherwise, ENODEV if the channel number is invalid.

uint16_t **PINMUX_AllPWM**(bool enable)

The function pinmux__all_pwm enables PWM functionality on all supported pins.

The function pinmux__all_pwm configures the pinmux registers for all PWM-capable pins to enable PWM output.

Parameters

enable – If set to true, configures the pin for PWM functionality. If set to false, reverts the pin to GPIO functionality.

Returns

SUCCESS if all PWM pins are successfully enabled.

uint16_t **PINMUX_SPI**(uint8_t num, bool enable)

The function PINMUX_SPI enables the SPI peripheral(SPI2 or SPI3)

The function PINMUX_SPI checks whether the provided SPI number is 2 or 3 and whether the pin is already in use. If the pin is not in use, the function configures the corresponding pinmux register to enable SPI functionality. If the pin number

is invalid or the pin is already in use, an error is returned.

Parameters

- **num** – The SPI number to be enabled for pin multiplexing.
- **enable** – If set to true, configures the pin for PWM functionality. If set to false, reverts the pin to GPIO functionality.

Returns

SUCCESS if the SPI peripheral is enabled successfully; otherwise, ENODEV if the SPI number is invalid.

uint16_t **PINMUX_UART**(uint8_t num, bool enable)

The function PINMUX_UART enables one of the UART peripherals (UART3 or UART4).

The function PINMUX_UART checks whether the provided UART number is 3 or 4. If valid, it sets the corresponding pinmux configuration register to enable the UART peripheral. If the UART number is invalid, an error is returned.

Parameters

- **num** – The UART number to be enabled for pin multiplexing.
- **enable** – If set to true, configures the pin for PWM functionality. If set to false, reverts the pin to GPIO functionality.

Returns

SUCCESS if the UART peripheral is enabled successfully; otherwise, ENODEV if the UART number is invalid.

uint16_t **PINMUX_GPTimer**(uint8_t num, bool enable)

The function PINMUX_GPTimer enables the GPTimer peripheral (GPTIimer0 to GPTimer3).

The function PINMUX_GPTimer checks whether the provided GPTimer number is in the range 0–3. If valid, it checks whether the corresponding pinmuxed pin (GPIO 38–41) is already in use. If the pin is not in use, the function sets the corresponding pinmux configuration register to enable the GPTimer peripheral. If the number is invalid, an error is returned.

Parameters

- **num** – The GPTIimer number to be enabled for pin multiplexing.
- **enable** – If set to true, configures the pin for PWM functionality. If set to false, reverts the pin to GPIO functionality.

Returns

SUCCESS if the GPTimer peripheral is enabled successfully; other-

wise, ENODEV if the number is invalid.

uint16_t **PINMUX_EnableJTAG**(void)

The function PINMUX_EnableJTAG enables the JTAG pins.

The function PINMUX_EnableJTAG configures the pinmux register to enable the JTAG_TRST functionality.

Parameters

none –

Returns

SUCCESS if the JTAG pin is enabled successfully.

uint16_t **PINMUX_DisableJTAG**(void)

The function PINMUX_DisableJTAG enables the JTAG pin.

The function PINMUX_DisableJTAG configures the pinmux register to disable the JTAG functionality.

Parameters

none –

Returns

SUCCESS if the JTAG pin is disabled successfully.

2.9 Platform Level Interrupt Controller (PLIC) API

2.9.1 Required Includes

To use the PLIC functionality, include the *plic.h* header file in your source code. This header provides the necessary definitions, functions, and configuration structures to interface with the PLIC hardware.

```
#include "plic.h"
```

2.9.2 Writing Handlers

Writing handlers for peripheral interrupts

The handler can be written with or without the argument.

To write handler with argument :

```
void IRQ_Handler(void *args)
   {
      /*work to be done*/
   }
```

To write handler without argument :

```
void IRQ_Handler(void)
   {
      /*work to be done*/
   }
```

Writing handlers for core interrupts

To write handlers include:

```
#include "core_interrupts.h"
```

After this refer to **core_interrupts.h** to find the overiding function for example if we want to write handler for interrupt corresponding to machine with stack push and pop of caller saved registers(only interger registers) then:

```
void on_machine_external_interrupt(void)
   {
      /*work to be done*/
   }
```

If you write without those simply use:

```
void machine_external_interrupt_handler(void)
   {
      /*work to be done*/
   }
```

If you write with pushing and poping integral and floating registers then you can do this:

```
void machine_external_interrupt_handler(void)
     {
```

(continues on next page)

(continued from previous page)

```
        PUSH_CONTEXT_INT_FLOAT;
        on_machine_external_interrupt();
        POP_CONTEXT_INT_FLOAT;
    }
```

2.9.3 Macros, Enums and Data Structures

Macros

PLIC_MAX_INTERRUPT_SRC

2.9.4 API Reference

API Reference for Core Interrupts

void **non_vectored_trap_entry**(void)

Brief: Overrideable function for non vectored trap entry for specific application.

Arguments:

- None

Returns:

- None

void **core_switch_interrupt_mode**(uint8_t mode)

Brief: Switches the core's interrupt mode to the specified mode either NON_VECTORED_MODE or VECTORED.

Arguments:

- **mode:** Interrupt mode to switch to (mode values depend on platform specification).

Returns:

- None

API Reference for PLIC

void **PLIC_Init**(void)

Used to initializes the PLIC interrupt controller.

This function disables all the PLIC interrupts, configures the interrupt threshold, and enables global interrupt handling.

Parameters

none –

Returns

none

void **PLIC_Handler**(void)

Used to handle a PLIC interrupt.

This function performs the following steps when a PLIC interrupt occurs: 1. Reads the claim register to determine the source of the interrupt. 2. Calls the corresponding interrupt handler to service the interrupt. 3. Writes the serviced interrupt ID back to the claim register to notify the PLIC core that the interrupt has been handled.

Parameters

none –

Returns

none

uint16_t **PLIC_Interrupt_Enable**(uint32_t interrupt_id)

Used to enable a particular PLIC interrupt.

This function enables the specified PLIC interrupt source, allowing it to be triggered and serviced by the PLIC.

Parameters

interrupt_id – The parameter *interrupt_id* is an unsigned integer that represents the interrupt id for which interrupt has to be enabled.

Returns

Returns Error code if interrupt id is invalid, else return SUCCESS.

uint16_t **PLIC_Interrupt_Disable**(uint32_t interrupt_id)

Used to disable a particular PLIC interrupt.

This function disables the specified PLIC interrupt source, preventing it from being triggered or serviced by the PLIC.

Parameters

interrupt_id – The parameter *interrupt_id* is an unsigned integer

that represents the interrupt id for which interrupt has to be disabled.

Returns

Returns Error code if interrupt id is invalid, else return SUCCESS.

uint16_t **PLIC_Interrupt_Complete**(uint32_t interrupt_id)

Used to mark a PLIC interrupt as completed.

This function signals that the specified PLIC interrupt has been serviced, allowing the core to handle subsequent interrupts.

Parameters

interrupt_id – The parameter *interrupt_id* is an unsigned integer that identifies the PLIC interrupt to be completed.

Returns

Returns Error code if interrupt id is invalid, else return SUCCESS.

uint16_t **PLIC_Interrupt_Pending**(uint8_t int_id)

Used to check the pending status of a PLIC interrupt.

This function checks whether the specified PLIC interrupt source is pending and awaiting service.

Parameters

interrupt_id – The parameter *interrupt_id* is an unsigned integer that identifies the PLIC interrupt source to be checked.

Returns

Returns 1 if it is pending, 0 if it is not pending or already claimed and Error code if the interrupt ID is invalid.

uint16_t **PLIC_Interrupt_Threshold**(uint32_t priority_value)

Used to set the interrupt priority threshold.

This function configures the minimum priority level required for a PLIC interrupt to be serviced. Interrupts with priority values greater than the configured threshold will be triggered.

Parameters

priority_value – The parameter *priority_value* is an unsigned integer that represents the priority value above which interrupt will be triggered.

Returns

Returns Error code if interrupt id is invalid or if priority value is invalid , else return SUCCESS.

uint16_t **PLIC_Set_Interrupt_Priority**(uint32_t int_id, uint32_t priority_value)

Used to set the priority level for a specific PLIC interrupt.

This function configures the priority of a specified PLIC interrupt. Higher-priority interrupts are serviced before lower-priority ones, subject to the interrupt threshold.

Parameters

- **interrupt_id** – The parameter *interrupt_id* is an unsigned integer that identifies the PLIC interrupt source.
- **priority_value** – The parameter *priority_value* is an unsigned integer that specifies the interrupt priority level.

Returns

Returns Error code if interrupt id is invalid, if priority value is invalid , else return SUCCESS.

uint16_t **PLIC_Set_Handler**(uint8_t interrupt_id, PLIC_IRQHandler_t handler, void *args)

Used to set an interrupt handler for a PLIC interrupt source.

This function associates a user-defined interrupt handler and its argument with the specified PLIC interrupt ID. When the interrupt is triggered, the registered handler is invoked with the provided argument.

Parameters

- **interrupt_id** – The parameter *interrupt_id* is an unsigned integer that represents to which interrupt handler should be mapped.
- **handler** – The parameter *handler* is a function pointer to the interrupt handler function to be executed.
- **args** – The parameter *args* is of void * type used to pass arguement to interrupt handler.

Returns

Returns SUCCESS when successfully initialised or else Error code when device not found.

void **PLIC_Nested_Interrupt**(uint8_t enable)

Used to configure the PLIC interrupt mode as nested or non-nested.

This function configures PLIC interrupt handling mode. In nested mode, higher-priority interrupts preempt lower-priority ones; otherwise, interrupts are serviced sequentially.

Parameters

enable – The parameter *enable* enables nested interrupt mode when non-zero, and disables nested interrupt mode when zero.

Returns

none

2.10 Physical Memory Protection (PMP) API

2.10.1 Including the PMP Header

To use the PMP functions, include the *pmp.h* header file in your source code. This header provides the necessary definitions, functions, and configuration structures to configure PMP entries.

```
#include "pmp.h"
```

2.10.2 Macros,Enums and Data Structures

Macros

PMP_MAX_ENTRIES

Maximum number of PMP entries supported by the hardware.

PMP_GRANULARITY

PMP address granularity in bytes.

PMP_READ_ACCESS

Enables read access for the PMP Entry.

PMP_WRITE_ACCESS

Enables write access for the PMP Entry.

PMP_EXECUTE_ACCESS

Enables execute access for the PMP Entry.

PMP_TOR_MATCHING

Enables TOR (Top of Range) address matching mode.

Uses the previous PMP address register as the lower bound and the current PMP address register as the upper bound.

PMP_NAPOT_MATCHING

Enables NAPOT (Naturally Aligned Power-Of-Two) address matching mode.

PMP_LOCK_BIT

Locks the PMP Entry configuration.

API Reference

uint16_t **PMP_Set_Entry**(uint8_t config, uint8_t entry, uint32_t *address, size_t size)

Configure a PMP (Physical Memory Protection) Entry.

This function sets up a PMP entry with the specified configuration, entry number, base address, and size. It writes the configuration to pmpcfg0 register, encodes and writes the address to pmpaddrX registers as per the given mode

Parameters

- **config** – The PMP configuration byte, defining permissions (read, write, execute), addressing mode (NAPOT, TOR) and lock bit. Use the macros for setting configuration
- **entry** – The pmpaddrX register to be used for PMP Configuration(0-3)
- **address** – The actual address to be protected (8-byte aligned).
- **size** – The size of the memory region in bytes. Must be a power of 2 for NAPOT addressing mode and 0 for TOR Mode.

Returns

SUCCESS when PMP registers are configured, EINVAL for invalid arguments, EPERM if lock bit is set for given entry.

uint16_t **PMP_Clear_Entry**(uint8_t entry)

Clears a specific PMP entry.

Clears the pmpaddrX register and the fields of the pmpcfg0 register for the given entry if the lock bit is not set.

Parameters

entry – PMP entry which is to be cleared (0-3).

Returns

SUCCESS when cleared, EINVAL for invalid PMP entry, EPERM if lock bit for the given entry is set.

uint16_t **PMP_Clear_All**(void)

Clears all PMP entries.

This function resets all pmpaddrX registers and the pmpcfg0 register. If any PMP entry has the lock bit set, the operation is aborted without clearing any of the entries.

Returns

SUCCESS when cleared, EPERM if any entry has its lock bit set.

2.11 Pulse Width Modulation (PWM) API

2.11.1 Required Includes

Include the below mentioned header to manage the PWM peripheral.

```
#include "pwm.h"
```

2.11.2 Configuration Structure

The PWM configuration is defined by a structure containing various fields to set up PWM parameters. Each field is described below.

```
typedef struct
   {
       unsigned int duty;              /* Specifies the duty cycle for␣
↪the PWM signal. This value can range from 0 to 0xFFFF. */
       unsigned int period;            /* Specifies the period of the␣
↪PWM signal. This value can range from 0 to 0xFFFF. */
       uint8_t interrupt_mode;         /* Defines the interrupt mode␣
↪for the PWM signal. Options are no_interrupt, interrupt_on_match,␣
↪etc. */
```

(continues on next page)

(continued from previous page)

```
      uint8_t change_output_polarity; /* Boolean flag to enable or␣
↪disable polarity inversion on the output PWM signal. */
      unsigned int prescaler_value;   /* Specifies the prescaler␣
↪value for the PWM frequency calculation. */
      unsigned short deadband_delay;  /* Defines the delay time in␣
↪terms of PWM cycles to avoid simultaneous switching of high and low␣
↪signals. */
  } PWM_Config_t;
```

2.11.3 Macros and Enums

Macros

PWM_PIN(x)

Macro is used to mention the PWM instances needed.

Note

Enter a number between 0-13 as x

Enums

enum **PWM_Interrupt_Modes**

PWM interrupt configuration modes.

This enumeration defines the available interrupt trigger modes for the PWM peripheral.

Values:

enumerator **PWM_INTR_NONE**

Disable all PWM interrupts

enumerator **PWM_INTR_RISE**

Enable interrupt on rising edge

enumerator **PWM_INTR_FALL**

Enable interrupt on falling edge

enumerator **PWM_INTR_HALFPERIOD**

Enable interrupt on half-period event

2.11.4 API Reference

uint16_t **PWM_Start**(PWM_Config_t *config, uint32_t pwm_pins)

Starts a specific PWM module.

This function initializes and enables the specified PWM module using the provided configuration parameters and activates the selected PWM output channels.

Parameters

- **config** – Pointer to a PWM_Config_t structure containing the parameters required to configure the PWM signal.
- **pwm_pins** – Bitwise OR'ed value of PWM pin identifiers representing the PWM output channels to be enabled.

Returns

Returns SUCCESS if the PWM module is started successfully; ENODEV if the pwm_pins given, is not within 0-13; EINVAL if the configuration values are out of range;

uint16_t **PWM_Stop**(uint32_t pwm_pins)

Stops a specific PWM module.

This function disables the specified PWM output channels and stops PWM signal generation for the selected pins.

Parameters

pwm_pins – Bitwise OR'ed value of PWM pin identifiers representing the PWM output channels to be stopped.

Returns

Returns SUCCESS if the PWM module is stopped successfully; ENODEV if the pwm_pins given, is not within 0-13.

2.12 Quad-Serial Peripheral Interface (QSPI) API

2.12.1 Required Includes

Include the below mentioned header file to access the necessary functions for QSPI peripheral.

```
#include "qspi.h"
```

2.12.2 Structure

qspi_msg

Description:

The “qspi_msg” structure defines a comprehensive message configuration for QSPI (Quad Serial Peripheral Interface) communication. It holds fields that control various aspects of the QSPI command, address, data, and configuration settings, enabling fine-grained control over communication behavior.

```
typedef struct{
    uint8_t functional_mode;        /**< Specifies the QSPI functional␣
↪operating mode. */
    uint8_t instruction;            /**< Specifies the QSPI command␣
↪instruction byte to be transmitted. */
    uint8_t instruction_mode;       /**< Specifies the transmission␣
↪mode of the command instruction. */
    uint8_t address_mode;           /**< Specifies the transmission␣
↪mode used for the address phase. */
    uint8_t address_size;           /**< Specifies the size of the␣
↪address in bytes. */
    uint32_t address;               /**< Specifies the address value␣
↪used for the QSPI transaction. */
    uint8_t alternate_byte_mode;    /**< Specifies the transmission␣
↪mode for alternate bytes. */
    uint8_t alternate_byte;         /**< Specifies the alternate byte␣
↪value used during the transaction. */
```

(continues on next page)

(continued from previous page)

```
    uint8_t dummy_mode;             /**< Indicates whether dummy␣
↪cycles are enabled. */
    uint8_t dummy_bit;              /**< Specifies whether dummy bits␣
↪are used instead of dummy cycles. */
    uint8_t dummy_cycles;           /**< Specifies the number of dummy␣
↪cycles to be inserted. */
    uint8_t sioo;                   /**< If set, the instruction is␣
↪sent only once for consecutive transactions. */
    uint8_t mm_mode;                /**< Enables or disables memory-
↪mapped mode. */
    uint8_t data_mode;              /**< Specifies the data␣
↪transmission mode. */
    uint32_t length;                /**< Specifies the number of data␣
↪bytes to be transmitted or received. */
    uint8_t *data_buffer;           /**< Pointer to the data buffer␣
↪used for data transmission or reception. */
    uint8_t FMEM_SIZE;              /**< Specifies the size of the␣
↪external Flash memory. */
    uint8_t CLK_MODE;               /**< Specifies the QSPI clock mode.
↪ */
    uint32_t TCEN;                  /**< Enables the timeout counter.␣
↪*/
    uint32_t TEIE;                  /**< Enables the transfer error␣
↪interrupt. */
    uint32_t TCIE;                  /**< Enables the transfer complete␣
↪interrupt. */
    uint32_t FTIE;                  /**< Enables the FIFO threshold␣
↪interrupt. */
    uint32_t SMIE;                  /**< Enables the status match␣
↪interrupt. */
    uint32_t TOIE;                  /**< Enables the timeout interrupt.
↪ */
    uint32_t APMS;                  /**< Enables automatic polling␣
↪mode stop. */
    uint32_t PMM;                   /**< Specifies the polling match␣
↪mode. */
    uint32_t PRESCALER;             /**< Specifies the prescaler value␣
```

(continues on next page)

(continued from previous page)

```
↪for the QSPI clock. */
} qspi_msg;
```

2.12.3 QSPI Defined Macros And Enums

Below are the pre-defined macros used in UART configuration and operation:

Instruction mode:

- `CCR_IMODE_NIL`

  Macros for Instruction mode which has no data transaction, assigned to instruction_mode parameter in qspi_msg structure.

- `CCR_IMODE_SINGLE_LINE`

  Macros for Instruction mode which has single line transaction, assigned to instruction_mode parameter in qspi_msg structure.

- `CCR_IMODE_TWO_LINE`

  Macros for Instruction mode which has two line transaction, assigned to instruction_mode parameter in qspi_msg structure.

- `CCR_IMODE_FOUR_LINE`

  Macros for Instruction mode which has four line transaction, assigned to instruction_mode parameter in qspi_msg structure.

Address mode:

- `CCR_ADMODE_NIL`

  Macros for Address mode which has no data transaction, assigned to address_mode parameter in qspi_msg structure.

- `CCR_ADMODE_SINGLE_LINE`

  Macros for Address mode which has single line transaction, assigned to address_mode parameter in qspi_msg structure.

- `CCR_ADMODE_TWO_LINE`

  Macros for Address mode which has two line transaction, assigned to address_mode parameter in qspi_msg structure.

- `CCR_ADMODE_FOUR_LINE`

  Macros for Address mode which has four line transaction, assigned to address_mode parameter in qspi_msg structure.

Address size:

- `CCR_ADSIZE_8_BIT`

  Macros for Address size which has 8-bit address size, assigned to address_size parameter in qspi_msg structure.

- `CCR_ADSIZE_16_BIT`

  Macros for Address size which has 16-bit address size, assigned to address_size parameter in qspi_msg structure.

- `CCR_ADSIZE_24_BIT`

  Macros for Address size which has 24-bit address size, assigned to address_size parameter in qspi_msg structure.

- `CCR_ADSIZE_32_BIT`

  Macros for Address size which has 32-bit address size, assigned to address_size parameter in qspi_msg structure.

Alternate byte mode:

- `CCR_ABMODE_NIL`

  Macros for Alternate byte mode which has no data transaction, assigned to alternate_byte_mode parameter in qspi_msg structure.

- `CCR_ABMODE_SINGLE_LINE`

  Macros for Alternate byte mode which has single line transaction, assigned to alternate_byte_mode parameter in qspi_msg structure.

- `CCR_ABMODE_TW0_LINE`

  Macros for Alternate byte mode which has two line transaction, assigned to alternate_byte_mode parameter in qspi_msg structure.

- `CCR_ABMODE_FOUR_LINE`

  Macros for Alternate byte mode which has four line transaction, assigned to alternate_byte_mode parameter in qspi_msg structure.

Alternate byte size:

- `CCR_ABSIZE_8_BIT`

  Macros for Alternate byte size which has 8-bit alternate byte size, assigned to alternate_byte parameter in qspi_msg structure.

- `CCR_ABSIZE_16_BIT`

  Macros for Alternate byte size which has 16-bit alternate byte size, assigned to alternate_byte parameter in qspi_msg structure.

- `CCR_ABSIZE_24_BIT`

Macros for Alternate byte size which has 24-bit alternate byte size, assigned to alternate_byte parameter in qspi_msg structure.

- `CCR_ABSIZE_32_BIT`

  Macros for Alternate byte size which has 32-bit alternate byte size, assigned to alternate_byte parameter in qspi_msg structure.

Memory map mode:

- `CCR_MM_MODE_XIP`

  Macros for memory map mode which has XIP memory map mode, assigned to mm_mode parameter in qspi_msg structure.

- `CCR_MM_MODE_RAM`

  Macros for memory map mode which has RAM memory map mode, assigned to mm_mode parameter in qspi_msg structure.

Functional mode:

- `CCR_FMODE_INDIRECT_WRITE`

  Macros for functional mode which has indirect write, assigned to functional_mode parameter in qspi_msg structure.

- `CCR_FMODE_INDIRECT_READ`

  Macros for functional mode which has indirect read, assigned to functional_mode parameter in qspi_msg structure.

- `CCR_FMODE_APM`

  Macros for functional mode which has Automatic poll mode, assigned to functional_mode parameter in qspi_msg structure.

- `CCR_FMODE_MMM`

  Macros for functional mode which has Memory map mode, assigned to functional_mode parameter in qspi_msg structure.

Data mode:

- `CCR_IMODE_NIL`

  Macros for data mode which has no data transaction, assigned to data_mode parameter in qspi_msg structure.

- `CCR_IMODE_SINGLE_LINE`

  Macros for data mode which has single line transaction, assigned to data_mode parameter in qspi_msg structure.

- `CCR_IMODE_TWO_LINE`

  Macros for data mode which has two line transaction, assigned to

data_mode parameter in qspi_msg structure.

- `CCR_IMODE_FOUR_LINE`

  Macros for data mode which has four line transaction, assigned to data_mode parameter in qspi_msg structure.

2.12.4 QSPI API Reference

This section provides the API reference for the QSPI function used for data transfer and buffering.

QSPI_Transaction

uint16_t **`QSPI_Transaction`**(uint32_t instance_number, qspi_msg *msg)

Brief: This function performs QSPI data transfer, sending and receiving data based on the QSPI instance and structure.

Arguments:

- **instance_number**: A variable that contains which QSPI is used.
- **flash_msg** : A pointer to a structure that holds the QSPI configuration.

Returns: - **0**: On success. - **Error code**: On failure (indicating an error during the transfer).

2.13 QSPI Flash

Quad Serial Peripheral Interface (QSPI) flash is a non-volatile memory that enables high-speed data transfer using multiple data lines (I/O pins). It supports read and write operations in single, dual, and quad I/O modes, providing features such as fast read, page program, and erase operations for efficient sequential and random memory access. Due to its high data throughput and efficiency in handling high-density storage, QSPI flash is well suited for applications requiring quick access to large volumes of data. In S2401, QSPI flash is accessed through the QSPI interface, an enhanced version of the SPI protocol that enables reliable and high-performance communication with external flash memory devices.

2.13.1 QSPI Specification

For detailed information on QSPI registers, refer to the qspi_section

2.13.2 Required Includes

To use the QSPI Flash functionality, include the qspi_flash.h header file in your source code. This header provides all necessary definitions and API functions required for QSPI flash memory operations.

```
#include "qspi_flash.h"
```

2.13.3 API Reference

uint16_t **Flash_Fast_Read_Quad**(const QSPI_Instance_t *qspi_inst, uint8_t *data, uint32_t address, uint8_t data_length)

Perform Fast Quad Read operation (1-1-4 mode).

Sends instruction over single line, address over single line, and reads data over four data lines.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Buffer to store read data.
- **address** – Flash start address.
- **data_length** – Number of bytes to read.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Fast_Read_Quad_IO**(const QSPI_Instance_t *qspi_inst, uint8_t *data, uint32_t address, uint8_t data_length)

Perform Fast Quad I/O Read operation (1-4-4 mode).

Sends instruction over single line, address over four lines, and reads data over four data lines.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.

- **data** – Buffer to store read data.
- **address** – Flash start address.
- **data_length** – Number of bytes to read.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Fast_Read_Single**(const QSPI_Instance_t *qspi_inst, uint8_t *data, uint32_t address, uint8_t data_length)

Perform Standard Single Line Read (1-1-1 mode).

Instruction, address and data are transferred over a single line.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Buffer to store read data.
- **address** – Flash start address.
- **data_length** – Number of bytes to read.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Input_Page_Quad**(const QSPI_Instance_t *qspi_inst, uint8_t *data, uint32_t address, uint8_t data_length)

Program flash page using Quad mode (1-1-4).

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Data buffer to program.
- **address** – Flash start address.
- **data_length** – Number of bytes to program.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Input_Page_Single**(const QSPI_Instance_t *qspi_inst, uint8_t *data, uint32_t address, uint8_t data_length)

Program flash page using Single line mode (1-1-1).

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Data buffer to program.
- **address** – Flash start address.

- **data_length** – Number of bytes to program.

Returns
SUCCESS on success, error code otherwise.

uint16_t **Flash_Sector_4K_Erase**(const QSPI_Instance_t *qspi_inst, uint32_t address)

Erase 4KB sector.

This function initiates erase and returns immediately. Use status register polling to check completion.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **address** – Sector start address.

Returns
SUCCESS on success, error code otherwise.

uint16_t **Flash_Sector_32K_Erase**(const QSPI_Instance_t *qspi_inst, uint32_t address)

Erase 32KB block.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **address** – Block start address.

Returns
SUCCESS on success, error code otherwise.

uint16_t **Flash_Chip_Erase**(const QSPI_Instance_t *qspi_inst)

Perform complete chip erase.

Blocks internally until erase operation completes.

Parameters
qspi_inst – Pointer to QSPI hardware instance.

Returns
SUCCESS on success, error code otherwise.

uint16_t **Flash_Write_Enable**(const QSPI_Instance_t *qspi_inst)

Enable flash write operations.

Parameters
qspi_inst – Pointer to QSPI hardware instance.

Returns
SUCCESS on success, error code otherwise.

uint16_t **Flash_Write_Disable**(const QSPI_Instance_t *qspi_inst)

Disable flash write operations.

Parameters

qspi_inst – Pointer to QSPI hardware instance.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Suspend**(const QSPI_Instance_t *qspi_inst)

Suspend ongoing erase/program operation.

Parameters

qspi_inst – Pointer to QSPI hardware instance.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Resume**(const QSPI_Instance_t *qspi_inst)

Resume suspended erase/program operation.

Parameters

qspi_inst – Pointer to QSPI hardware instance.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Power_Down**(const QSPI_Instance_t *qspi_inst)

Put flash into deep power-down mode.

Parameters

qspi_inst – Pointer to QSPI hardware instance.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Release_Power_Down**(const QSPI_Instance_t *qspi_inst)

Release flash from deep power-down mode.

Parameters

qspi_inst – Pointer to QSPI hardware instance.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Read_Status_Register1**(const QSPI_Instance_t *qspi_inst, uint8_t *data)

Read Status Register 1.

Retrieves the contents of Status Register 1 from the flash device. Typically contains BUSY bit, Write Enable Latch (WEL) and protection bits.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Pointer to buffer where status byte will be stored.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Read_Status_Register2**(const QSPI_Instance_t *qspi_inst, uint8_t *data)

Read Status Register 2.

Retrieves the contents of Status Register 2 from the flash device.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Pointer to buffer where status byte will be stored.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Read_Status_Register3**(const QSPI_Instance_t *qspi_inst, uint8_t *data)

Read Status Register 3.

Retrieves the contents of Status Register 3 from the flash device.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Pointer to buffer where status byte will be stored.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Read_Flag_Status_Register**(const QSPI_Instance_t *qspi_inst, uint8_t *data)

Read Flag Status Register.

Retrieves the Flag Status Register which indicates program/erase completion and error conditions.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Pointer to buffer where flag status byte will be stored.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Write_Enable_Status_Register**(const QSPI_Instance_t *qspi_inst)

Enable write access to status registers.

Must be called before modifying any status register.

Parameters

`qspi_inst` – Pointer to QSPI hardware instance.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Write_Status_Register1**(const QSPI_Instance_t *qspi_inst, uint8_t *data)

Write Status Register 1.

Programs Status Register 1 with provided value.

Parameters

- `qspi_inst` – Pointer to QSPI hardware instance.
- `data` – Pointer to value to be written.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Write_Status_Register2**(const QSPI_Instance_t *qspi_inst, uint8_t *data)

Write Status Register 2.

Programs Status Register 2 with provided value.

Parameters

- `qspi_inst` – Pointer to QSPI hardware instance.
- `data` – Pointer to value to be written.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Write_Status_Register3**(const QSPI_Instance_t *qspi_inst, uint8_t *data)

Write Status Register 3.

Programs Status Register 3 with provided value.

Parameters

- `qspi_inst` – Pointer to QSPI hardware instance.
- `data` – Pointer to value to be written.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Read_Global_Freeze_Bit**(const QSPI_Instance_t *qspi_inst, uint8_t *data)

Read Global Freeze Bit.

Retrieves the state of the Global Freeze bit.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Pointer where freeze bit value will be stored.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Write_Global_Freeze_Bit**(const QSPI_Instance_t *qspi_inst, uint8_t data)

Set or clear Global Freeze Bit.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Value to program (0 = clear, 1 = set).

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Read_SFDP**(const QSPI_Instance_t *qspi_inst, uint32_t address, uint8_t *data)

Read Serial Flash Discoverable Parameters (SFDP).

Reads SFDP table entry from specified address.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **address** – SFDP table address.
- **data** – Pointer where read byte will be stored.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Read_NVCR**(const QSPI_Instance_t *qspi_inst, uint8_t *data)

Read Non-Volatile Configuration Register (NVCR).

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Pointer to buffer to store NVCR contents.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Write_NVCR**(const QSPI_Instance_t *qspi_inst, uint8_t *data)

Write Non-Volatile Configuration Register (NVCR).

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **data** – Pointer to data to be written to NVCR.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_Read_Jedec_ID**(const QSPI_Instance_t *qspi_inst, uint8_t *id)

Read JEDEC ID.

Retrieves manufacturer ID, memory type and capacity.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **id** – Pointer to 3-byte buffer to store JEDEC ID.

Returns

SUCCESS on success, error code otherwise.

uint16_t **Flash_XIP_Init**(const QSPI_Instance_t *qspi_inst, uint32_t flash_size)

Initialize Flash in Execute-In-Place (XIP) mode.

Configures the QSPI peripheral in Memory-Mapped Mode (MMM) to allow direct execution of code from external flash memory. After initialization, the flash contents can be accessed directly through the mapped memory region.

Parameters

- **qspi_inst** – Pointer to QSPI hardware instance.
- **flash_size** – Flash size configuration value used to program the FSIZE field of QSPI.

Returns

SUCCESS on success, error code otherwise.

2.14 SPI (Serial Peripheral Interface) API

2.14.1 Including SPI headerfile.

To use the SPI functionality, include the *spi.h* header file in your source code. This header provides the necessary definitions, functions, and configuration structures to interface with the spi hardware.

```
#include "spi.h"
```

2.14.2 Macros, Enums and Data Structures

Macros

SPI0

Macro for SPI instance 0 handle.

SPI1

Macro for SPI instance 1 handle.

SPI2

Macro for SPI instance 2 handle.

SPI3

Macro for SPI instance 3 handle.

Enums

enum **SPI_FIFO_Interrupt_t**

SPI FIFO Interrupt Enable values.

This enumeration defines the interrupt enable bit values for TX FIFO and RX FIFO threshold levels. These values can be combined using bitwise OR to enable multiple FIFO interrupts.

Values:

enumerator **TX_FIFO_INTR_EMPTY**

The value of this enum is 1U (ie. 1 << 0). It enables TX FIFO empty interrupt.

enumerator **TX_FIFO_INTR_DUAL**

The value of this enum is 2U (ie. 1 << 1). It enables TX FIFO dual interrupt.

enumerator **TX_FIFO_INTR_QUAD**

The value of this enum is 4U (ie. 1 << 2). It enables TX FIFO quad interrupt.

enumerator **TX_FIFO_INTR_OCTAL**

The value of this enum is 8U (ie. 1 << 3). It enables TX FIFO octal interrupt.

enumerator **TX_FIFO_INTR_HALF**

The value of this enum is 16U (ie. 1 << 4). It enables TX FIFO half interrupt.

enumerator **TX_FIFO_INTR_24**

The value of this enum is 32U (ie. 1 << 5). It enables TX FIFO 24-level interrupt.

enumerator **TX_FIFO_INTR_28**

The value of this enum is 64U (ie. 1 << 6). It enables TX FIFO 28-level interrupt.

enumerator **TX_FIFO_INTR_30**

The value of this enum is 128U (ie. 1 << 7). It enables TX FIFO 30-level interrupt.

enumerator **TX_FIFO_INTR_FULL**

The value of this enum is 256U (ie. 1 << 8). It enables TX FIFO full interrupt.

enumerator **RX_FIFO_INTR_EMPTY**

The value of this enum is 512U (ie. 1 << 9). It enables RX FIFO empty interrupt.

enumerator **RX_FIFO_INTR_DUAL**

The value of this enum is 1024U (ie. 1 << 10). It enables RX FIFO dual interrupt.

enumerator **RX_FIFO_INTR_QUAD**

The value of this enum is 2048U (ie. 1 << 11). It enables RX FIFO quad interrupt.

enumerator **RX_FIFO_INTR_OCTAL**

The value of this enum is 4096U (ie. 1 << 12). It enables RX FIFO octal interrupt.

enumerator **RX_FIFO_INTR_HALF**

The value of this enum is 8192U (ie. 1 << 13). It enables RX FIFO half interrupt.

enumerator **RX_FIFO_INTR_24**

The value of this enum is 16384U (ie. 1 << 14). It enables RX FIFO 24-level interrupt.

enumerator **RX_FIFO_INTR_28**

The value of this enum is 32768U (ie. 1 << 15). It enables RX FIFO 28-level interrupt.

enumerator **RX_FIFO_INTR_30**

The value of this enum is 65536U (ie. 1 << 16). It enables RX FIFO 30-level interrupt.

enumerator **RX_FIFO_INTR_FULL**

The value of this enum is 131072U (ie. 1 << 17). It enables RX FIFO full interrupt.

enum **SPI_DMA_Size**

SPI DMA transfer size identifiers.

This enumeration defines the available bit-widths for Direct Memory Access (DMA) transfers within the SPI module.

Values:

enumerator **SIZE_8**

The value of this enum is 1U. It represents 8-bit data

enumerator **SIZE_16**

The value of this enum is 2U. It represents 16-bit data

enumerator **SIZE_32**

The value of this enum is 3U. It represents 32-bit data

enumerator **SIZE_64**

The value of this enum is 4U. It represents 64-bit data

enum **SPI_Clk_Mode**

SPI Clock Polarity and Phase modes.

This enumeration defines the standard SPI clock modes based on CPOL and CPHA settings.

Values:

enumerator **MODE_0**

The value of this enum is 0U. It represents CPOL = 0, CPHA = 0

enumerator **MODE_1**

The value of this enum is 1U. It represents CPOL = 0, CPHA = 1 (Not supported)

enumerator **MODE_2**

The value of this enum is 2U. It represents CPOL = 1, CPHA = 0 (Not supported)

enumerator **MODE_3**

The value of this enum is 3U. It represents CPOL = 1, CPHA = 1

enum **SPI_Comm_Mode**

SPI Communication Directionality.

This enumeration defines the data flow direction for the SPI transaction.

Values:

enumerator **TX**

The value of this enum is 0U. It represents Transmit only

enumerator **RX**

The value of this enum is 1U. It represents Receive only

enumerator **HALF_DUPLEX**

The value of this enum is 2U. It represents Transmit then Receive

enumerator **FULL_DUPLEX**

The value of this enum is 3U. It represents Simultaneous Transmit and Receive

Data Structures

struct **SPI_Config_t**

SPI configuration structure.

This structure contains all parameters required to initialize and configure an SPI peripheral instance.

Public Members

uint32_t **spi_freq**

SPI clock frequency in mHz.

const SPI_Instance_t ***spi_num**

SPI hardware instance handle.

This parameter selects the SPI hardware block to be used. Valid instances are `SPI1`, `SPI2`, `SPI3` and `SPI4`.

SPI_Clk_Mode **spi_clk_mode**

Clock polarity and phase mode.

Note

Uses SPI_ClkMode_t (`MODE_0` or `MODE_3`). `MODE_1` and `MODE_2` are not supported.

uint8_t **setup_time**

Setup time before data transmission.

uint8_t **hold_time**

Hold time after data transmission.

bool **is_slave_mode**

SPI operating mode: `true` if slave mode or `false` if master mode.

bool **is_lsb**

Bit order: `true` if lsb first or `false` if msb first.

SPI_Comm_Mode **comm_mode**

Communication mode: `TX`, `RX`, `HALF_DUPLEX`, `FULL_DUPLEX`.

uint8_t **spi_size**

Number of bits to be transferred per frame (8, 16, 32).

bool **is_software_ncs**

Chip select control: `true` if software ncs or `false` if `hardware ncs`.

Note

Ignored if SPI is configured as Slave mode.

struct **spi_buffer**

SPI transaction buffer structure.

This structure manages the data buffers used for SPI read and write operations.

Public Members

void ***tx_buf**

Transmit buffer pointer.

Note

Required if comm_mode is TX, HALF_DUPLEX, or FULL_DUPLEX.

void ***rx_buf**

Receive buffer pointer.

Note

Required if comm_mode is RX, HALF_DUPLEX, or FULL_DUPLEX.

uint8_t **data_size**

Data size in bits (8, 16, or 32).

Data size to be read/write from/to RX_reg/TX_reg.

size_t **len**

Number of elements to be transferred.

2.14.3 API Reference

uint16_t **SPI_Config**(const *SPI_Config_t* *spi_config)

Configures and enables the specified SPI instance for communication.

This function sets the SPI parameters such as clock mode, chip select type, frequency, setup/hold times, communication mode, data size, bit order, and master/slave mode based on the provided configuration structure.

Parameters

spi_config – Pointer to an *SPI_Config_t* structure containing all SPI configuration parameters.

Returns

Return SUCCESS if configuration is successful, otherwise returns Error code if any required pointer is null or if spi_freq not in range.

uint16_t **SPI_Disable**(const *SPI_Config_t* *spi_config)

Disables the specified SPI instance.

Parameters

spi_config – Pointer to an *SPI_Config_t* structure specifying the SPI instance.

Returns

Return SUCCESS if spi disabled successful, otherwise returns Error code if any required pointer is null..

uint16_t **Software_Control_NCS**(const *SPI_Config_t* *spi_config, bool ncs_val)

Controls the SPI chip select (NCS) line in software mode.

Parameters

- **spi_config** – Pointer to an *SPI_Config_t* structure specifying the SPI instance.
- **ncs_val** – Boolean value to set the NCS line (0 = deassert, 1 = assert).

Returns

Return SUCCESS if configuration is successful, otherwise returns Error code if any required pointer is null.

uint16_t **Flush_RX_FIFO**(const *SPI_Config_t* *spi_config)

Flushes (clears) the RX FIFO buffer of the specified SPI instance.

Parameters

spi_config – Pointer to an *SPI_Config_t* structure specifying the SPI instance.

Returns

Return SUCCESS if RX FIFO is cleared, otherwise returns Error code if any required pointer is null.

uint8_t **Check_TX_And_Busy_Status**(const *SPI_Config_t* *spi_config)

Checks the SPI TX FIFO empty and busy status.

This function reads the TX FIFO empty flag and SPI busy flag. The returned value indicates whether the SPI peripheral is busy and whether the TX FIFO is empty.

Parameters

spi_config – Pointer to an *SPI_Config_t* structure specifying the SPI instance.

Returns

Returns a byte value where the first 2-bits indicated the status value:

- 00: Not busy, TX FIFO not empty
- 01: Not busy, TX FIFO empty
- 10: Busy, TX FIFO not empty
- 11: Busy, TX FIFO empty

uint16_t **Wait_Till_TX_Complete**(const *SPI_Config_t* *spi_config)

Waits until SPI transmission is complete.

This function continuously monitors the TX FIFO empty status and SPI busy flag.

Parameters

spi_config – Pointer to an *SPI_Config_t* structure specifying the SPI instance.

Returns

Returns SUCCESS if transmission completes successfully; otherwise returns Error code if the operation exceeds the timeout limit.

uint16_t **SPI_Interrupt_Enable**(const *SPI_Config_t* *spi_config, uint32_t interrupt_type)

Enables a interrupt bits based on the provided interrupt.

Parameters

- **spi_config** – Pointer to an *SPI_Config_t* structure specifying the SPI instance.
- **interrupt** – FIFO Interrupts can be enabled using the FIFO INTERRUPT ENABLE macros. Multiple interrupts can be enabled together by OR-ing the macros (e.g., TX_FIFO_INTR_EMPTY | RX_FIFO_INTR_FULL).

Returns

Return SUCCESS if interrupt enabled successful, otherwise returns Error code if any required pointer is null..

uint16_t **SPI_Transceive**(const *SPI_Config_t* *spi_config, const *spi_buffer* *buf)

Transfers data over SPI in TX, RX, HALF_DUPLEX, or FULL_DUPLEX mode.

This function performs SPI data transmission and/or reception depending on the configured communication mode in *spi_config*. It supports master and slave modes with data sizes of 8, 16, or 32 bits.

Parameters

- **spi_config** – Pointer to an *SPI_Config_t* structure specifying the SPI instance and communication mode.
- **buf** – Pointer to a *spi_buffer* structure.

Returns

Return SUCCESS if transfer completed successfully, otherwise returns Error code if any required pointer is null or if an invalid data size is provided.

uint16_t **SPI_DMA**(const *SPI_Config_t* *spi_config, *SPI_DMA_Size* size)

Configures the DMA transfer size for SPI communication.

This function sets the TX and RX DMA transfer sizes based on the provided communication mode and data size. The DMA size can be configured between 8 bits and 64 bits.

Parameters

- **spi_config** – Pointer to an *SPI_Config_t* structure specifying

the SPI instance and communication mode.

- **size** – DMA transfer size *SPI_DMA_Size* (`SIZE_8`, `SIZE_16`, `SIZE_32`, `SIZE_64`).

Returns

Return SUCCESS if configuration is successful, otherwise returns Error code if any required pointer is null or if an invalid data size is provided.

2.15 Universal Asynchronous Receiver-Transmitter (UART) API

2.15.1 Required Includes

To use the UART functionality, include the *uart.h* header file in your source code. This header provides the necessary definitions, functions, and configuration structures to interface with the UART hardware.

```
#include "uart.h"
```

2.15.2 Macros, Enums and Data Structures

Enums

enum **UART_StopBits**

UART stop bit configuration values.

Encoding:

- 00 : 1 stop bit
- 01 : 1.5 stop bits
- 10 : 2 stop bits
- 11 : Reserved (unused)

These values must be used with UART_Config_t::stop_bits.

Values:

enumerator **STOP_BIT_1**

1 stop bit

enumerator **STOP_BIT_1_5**

1.5 stop bits

enumerator **STOP_BIT_2**

2 stop bits

enum **UART_Parity**

UART parity configuration values.

Encoding:

- 00 : No parity
- 01 : Odd parity
- 10 : Even parity
- 11 : Reserved (unused)

These values must be used with UART_Config_t::parity.

Values:

enumerator **NO_PARITY**

No parity

enumerator **ODD_PARITY**

Odd parity

enumerator **EVEN_PARITY**

Even parity

enum **UART_CharSize**

UART character size configuration values.

Encoding:

- 00 : 8-bit character size
- 01 : 7-bit character size
- 10 : 6-bit character size
- 11 : 5-bit character size

These values must be used with UART_Config_t::char_size.

Values:

enumerator **CHAR_SIZE_8**

8-bit character size

enumerator **CHAR_SIZE_7**

7-bit character size

enumerator **CHAR_SIZE_6**

6-bit character size

enumerator **CHAR_SIZE_5**

5-bit character size

Macros

ENABLE_RX_THRESHOLD

Enable RX threshold interrupt.

ENABLE_BREAK_ERROR

Enable break error interrupt.

ENABLE_FRAME_ERROR

Enable frame error interrupt.

ENABLE_OVERRUN

Enable overrun error interrupt.

ENABLE_PARITY_ERROR

Enable parity error interrupt.

ENABLE_RX_FULL

Enable RX full interrupt.

ENABLE_RX_NOT_EMPTY

Enable RX not empty interrupt.

ENABLE_TX_FULL

Enable TX full interrupt.

ENABLE_TX_EMPTY

Enable TX empty interrupt.

UART0

Macro for UART instance 0 handle.

> **Note**
>
> UART instance 0 is reserved for debug and print statements. It should not be used for general-purpose UART communication.

UART1

Macro for UART instance 1 handle.

UART2

Macro for UART instance 2 handle.

UART3

Macro for UART instance 3 handle.

UART4

Macro for UART instance 4 handle.

Data Structures

struct **uart_buf**

#include <uart.h> UART data buffer structure.

This structure is used to pass data buffers to UART transmit and receive APIs.

typedef struct UART_Instance **UART_Instance_t**

Opaque UART hardware instance type.

This type represents a specific UART hardware block on the SoC. The internal structure is hidden from the user and is only known to the UART driver implementation.

Note

This type must only be used as a handle. Do not attempt to allocate or define objects of this type.

struct **UART_Config_t**

#include <uart.h> UART configuration structure.

This structure holds all parameters required for UART initialization and runtime configuration.

2.15.3 API Reference

uint16_t **UART_Available**(*UART_Config_t* const *uart_config)

Check whether data is available in the UART receive buffer.

The function `UART_Available` checks if there is data available to be read from the UART receive buffer.

Parameters

uart_config – Pointer to UART configuration structure used to configure the UART hardware instance.

Returns

Returns 1 if the UART receive buffer is not empty, otherwise returns 0.

uint16_t **UART_RX_Threshold**(*UART_Config_t* const *uart_config)

Set the receive threshold value for UART communication.

The function sets the receive threshold for UART communication.

Parameters

uart_config – Pointer to UART configuration structure used to configure the UART hardware instance.

Returns

Returns 0 on SUCCESS, or a error code on failure.

uint16_t **UART_Interrupt_Enable**(*UART_Config_t* const *uart_config, uint16_t interrupt)

Enable UART interrupts for the specified interrupt mask.

The function enables UART interrupt sources based on the provided interrupt bitmask.

Parameters

- **uart_config** – Pointer to UART configuration structure used to configure the UART hardware instance.
- **interrupt** – Bitmask representing one or more UART interrupt enable bits. The value can be formed by combining macros defined in UART Interrupt Enable Bits using the bitwise OR (|) operator.

Returns

Returns 0 on SUCCESS, or an error code on failure.

uint16_t **UART_Interrupt_Disable**(*UART_Config_t* const *uart_config, uint16_t interrupt)

Disable UART interrupts for the specified interrupt mask.

The function disables UART interrupt sources based on the provided interrupt bitmask.

Parameters

- **uart_config** – Pointer to UART configuration structure used to configure the UART hardware instance.
- **interrupt** – Bitmask representing one or more UART interrupt enable bits. The value can be formed by combining macros defined in UART Interrupt Enable Bits using the bitwise OR (|) operator.

Returns

Returns 0 on SUCCESS, or an error code on failure.

uint16_t **UART_Set_Baudrate**(*UART_Config_t* const *uart_config)

Initialize the baud rate for a UART instance.

The function initializes the baud rate for a UART instance based on the provided configuration.

Parameters

uart_config – Pointer to UART configuration structure used to configure the UART hardware instance.

Returns

It returns 0 which means operation is success.

uint16_t **UART_Config**(*UART_Config_t* const *uart_config)

Configure UART settings for a hardware instance.

The function `UART_Config` configures UART settings based on the provided *UART_Config_t* structure.

Parameters

uart_config – Pointer to a *UART_Config_t* structure used to configure the UART hardware instance.

Returns

Returns 0 on success, or a negative error code on failure.

uint16_t **UART_Init**(*UART_Config_t* const *uart_config)

Initialize a UART hardware instance.

The function `UART_Init` initializes a UART instance based on the provided configuration parameters.

Parameters

uart_config – Pointer to a *UART_Config_t* structure used to configure the UART hardware instance.

Returns

Returns 0 on success, or a negative error code on failure.

uint16_t **UART_Write**(*UART_Config_t* const *uart_config, struct *uart_buf* *rx_bufs)

Transmit data via UART.

The function `UART_Write` transmits data via UART based on the specified transfer mode and UART configuration.

Parameters

- **uart_config** – Pointer to a *UART_Config_t* structure used to configure the UART hardware instance.
- **tx_bufs** – Pointer to a *uart_buf* structure containing the transmit buffer and the length of the data to be transmitted.

Returns

Returns 0 on success, or a negative error code on failure.

uint16_t **UART_Write_Wait**(*UART_Config_t* const *uart_config)

Wait until the UART transmit buffer becomes empty.

The function `UART_Write_Wait` waits for the UART transmit buffer to be empty before returning.

Parameters

uart_config – Pointer to a *UART_Config_t* structure used to configure the UART hardware instance.

Returns

Returns 0 on success, or a negative error code on failure.

uint16_t **UART_Read**(*UART_Config_t* const *uart_config, struct *uart_buf* *rx_bufs, uint64_t timeout)

Receive data from the UART interface.

The function `UART_Read` reads data from a UART interface based on the provided configuration and stores it in the specified receive buffer.

Parameters

- **uart_config** – Pointer to a *UART_Config_t* structure used to configure the UART hardware instance.
- **rx_bufs** – Pointer to a *uart_buf* structure containing the receive buffer and the maximum length of data to be received.
- **timeout** – Maximum number of CPU cycles to wait. This value is compared against the difference between the current `mcycle` count and the start cycle count.

Returns

Returns 0 on success, or a negative error code on failure.

uint16_t **UART_Flush**(*UART_Config_t* const *uart_config)

Flush the UART receive buffer.

The function `UART_Flush` clears the receive buffer of a UART module specified by the given configuration.

Parameters

uart_config – Pointer to a *UART_Config_t* structure used to configure the UART hardware instance.

Returns

Returns 0 on success, or a negative error code on failure.

int **putchar**(int ch)

The function sends a character over UART communication.

This function will be called to printf a single character to the stdout by passing character as an integer.

Note

Instance 0 cannot be initialized as it is reserved for print statements. If instance 0 must be configured, use the alternate APIs provided for performing the required operations.

Parameters

ch – The parameter “ch” is of type int and represents the character to be transmitted over UART (Universal Asynchronous Receiver/Transmitter).

Returns

An integer value of 0.

int **getchar**(void)

Function to read a single character from the standard input device.

The function reads a character from a UART instance and waits until a character is available.

Returns

A character received through UART communication.

2.16 Watchdog Timer (WDT) API

2.16.1 Required Includes

To use the WDTimer functionality, include the *wdtimer.h* header file in your source code. This header provides the necessary definitions, functions, and configuration structures to interface with the WDTimer.

```
#include "wdtimer.h"
```

2.16.2 Macros

Macros

HARD_RESET

Macro is used to mention the WDT HARD reset mode.

Note

Resets the core after downcounting the value written in wcycles.

SOFT_RESET

Macro is used to mention the WDT SOFT reset mode.

Note

Resets the core instantly.

2.16.3 API Reference

uint16_t **WDT_Start**(bool mode, uint64_t wcycles)

Start the watchdog timer.

The function Wdtimer_start enables the watchdog timer and configures the reset behavior based on the selected mode (hard reset or soft reset). The timer will trigger a reset after the specified number of seconds.

Note

wcycles is applicable only for hard reset, as soft reset occurs immediately.

Parameters

- **mode** – Boolean value to select the watchdog reset mode: hard reset or soft reset.
- **wcycles** – Number of seconds before the reset is triggered. Ensure the value, when multiplied by the base frequency, does not exceed the 64-bit range.

Returns

SUCCESS- Watchdog timer is started successfully. EINVAL- wcycles exceeds the 64-bit range when multiplied with base frequency.

uint16_t **WDT_Disable**(void)

The function wdtimer_disable disables the Watchdog Timer.

The function wdtimer_disable stops the Watchdog Timer by clearing the enable bit in the control register, after which the watchdog no longer monitors software execution or generates reset events.

Parameters

None –

Returns

Returns “0” on success.

Chapter 3. Security Accelerators API

3.1 Advanced Encryption Standard (AES) API

3.1.1 Required Includes

To use the AES functionality, include the `aes.h` header file in the source code. This header provides the necessary definitions, functions, and configuration structures to interface with the AES hardware accelerator.

```
#include "aes.h"
```

3.1.2 Enums and Data Structures

Enums

enum **AES_OPERATION**

AES execution operation type.

This enumeration defines whether the AES hardware performs encryption or decryption.

Values:

enumerator **AES_ENCRYPT**

The value of this enum is 0U. It represents AES encryption operation.

enumerator **AES_DECRYPT**

The value of this enum is 1U. It represents AES decryption operation.

enum **AES_MODE**

AES block cipher operating modes.

This enumeration defines the supported AES hardware block cipher modes of operation.

Values:

enumerator **AES_CBC**

The value of this enum is 0U. It represents AES CBC mode.

enumerator **AES_CFB**

The value of this enum is 1U. It represents AES CFB mode.

enumerator **AES_OFB**

The value of this enum is 2U. It represents AES OFB mode.

enumerator **AES_CTR**

The value of this enum is 3U. It represents AES CTR mode.

Data Structures

struct **AES_Config_t**

AES configuration structure.

This structure contains all parameters required to configure and execute an AES operation using the AES hardware module. It provides input/output buffers, key and IV configuration, operation mode selection, and iteration control parameters.

Note

AES_ECB mode is not supported by this implementation.

Public Members

uint8_t ***aes_output**

Pointer to AES output buffer.

Points to the buffer where the encrypted or decrypted output data will be stored.

Note

The buffer must be large enough to hold input_len_bits.

const uint8_t ***input_text**

Pointer to input text buffer.

Points to the input message to be encrypted or decrypted.

> **Note**
>
> The input length must be a multiple of 128 bits.

const uint8_t ***key**

Pointer to AES key.

Points to the encryption/decryption key used by the AES hardware.

> **Note**
>
> Supported key lengths are 128, 192, or 256 bits only.

const uint8_t ***iv**

Pointer to initialization vector (IV).

Points to the 128-bit IV used in supported block cipher modes.

> **Note**
>
> Required for CBC, CFB, OFB, and CTR modes.

size_t **input_len_bits**

Input message length in bits.

Specifies the total length of the input data to be processed.

> **Note**
>
> Must be a multiple of 128 bits.

size_t **key_len_bits**

AES key length in bits.

Defines the size of the AES key.

> **Note**
>
> Valid values are 128U, 192U, or 256U.

AES_MODE **mode**

AES block cipher mode of operation.

Selects the hardware-supported AES mode.

Note

Supported modes: AES_CBC, AES_CFB, AES_OFB, AES_CTR. AES_ECB is not supported.

AES_OPERATION **encrypt_or_decrypt**

AES operation type.

Selects whether the hardware performs encryption or decryption.

Note

AES_ENCRYPT = 0U, AES_DECRYPT = 1U.

size_t **iterated_length_bits**

Previously processed length in bits.

Indicates the number of bits already processed in earlier AES iterations.

Note

Set to 0U for the initial AES operation.

3.1.3 API Reference

uint16_t **AES_Run**(*AES_Config_t* *aes_config)

Performs AES encryption or decryption on input data.

This function encrypts or decrypts the provided input buffer using the AES hardware accelerator. Data is processed in 128-bit blocks and the resulting output is stored in the user-provided output buffer.

This function supports both single-call and multi-call (iterative) operation. In single-call mode, the entire input is processed in one invocation by setting iterated_length_bits to 0. In multi-call mode, the input is split across multiple calls — on the first call iterated_length_bits must be 0, and on subsequent calls it must reflect the total number of bits already processed. The hardware is zeroized and reconfigured only on the first call (iterated_length_bits == 0), and the hardware state is preserved across calls to allow chained block processing.

Parameters

aes_config – Pointer to an *AES_Config_t* structure containing input/output buffers, AES key, IV, operation mode, key length, input

length, and iteration parameters.

Returns

`SUCCESS` if the operation completes successfully, `EFAULT` if any required pointer is NULL, `EINVAL` if any parameter is invalid, `ETIMEDOUT` if the AES hardware does not respond within the timeout window.

uint16_t **AES_Zeroize**(void)

Triggers zeroization of all internal AES hardware registers.

Writes 1 to the AES_ZEROIZE register to initiate a hardware clear of all internal AES state. Polls the AES_ZEROIZE_STATUS register using a constant-time busy-wait until zeroization is confirmed complete or the timeout window expires.

This function may be called independently to ensure the hardware is in a clean state before a new AES operation.

Returns

`SUCCESS` if zeroization completed within the timeout window, `ETIMEDOUT` if the AES hardware zeroization does not complete within the timeout window.

3.2 One-Time Programmable Memory(OTP) API

3.2.1 Required Includes

To use the OTP functionality, include the *otp.h* header file in your source code. This header provides the necessary definitions, functions, and configuration structures to interface with the OTP.

```
#include "otp.h"
```

3.2.2 Configuration Structure

```
typedef struct
{
   uint32_t control;
```

(continues on next page)

(continued from previous page)

```
   uint32_t reserved0;
   volatile uint32_t status;
   uint32_t reserved1;
   uint32_t addr;
   uint32_t reserved2;
   volatile uint32_t read_data;
   uint32_t reserved3;
   uint32_t write_data;
   uint32_t reserved4;
} otp_struct;
```

- **control** : Specifies the control register.
- **reserved0**: Reserved for future use.
- **status**: Specifies the status register
- **reserved1**: Reserved for future use.
- **addr**: Specifies the address where the data need to be read or written
- **reserved2**: Reserved for future use.
- **read_data**: Specifies the data to be read from the otp memory.
- **reserved3**: Reserved for future use.
- **write_data**: Sets the output_enable high in the control and status register
- **reserved4**: Reserved for future use.

3.2.3 OTP Defined Macros

Below are the pre-defined macros used in GPTIMER configuration and operation:

- `OTP_CONTROLREG`

  Macro for the control register to configure and control the operation.

- `OTP_START`

  Macro for the control register used to start the one operation.

- `OTP_R_W`

  Macro for the control register used to control the read and write operation.

- `OTP_STARTUP`

Macro for the control register used to initialize the OTP memory before any operation.

- `OTP_STATUS`

  Macros for the status register provides the status about the current state.

- `OTP_GET_OUTPUT`

  Macros for the status register to get the output from the OTP memory.

- `OTP_PROG_STATUS`

  Macro for the status register to check the programming status on OTP memory.

- `OTP_FATAL`

  Macro for the status register to check if the programming failed on OTP memory.

- `OTP_STARTUP_DONE`

  Macro for the status register tp check if the initialization is done on OTP memory.

- `OTP_ADDR`

  Macro for the address register that holds the memory address for the read and write operation.

- `OTP_READ_DATA`

  Macro for the read register which stores the data read from the specified address.

- `OTP_WRITE_DATA`

  Macro for the write register which stores the data to be written to the specified address.

3.2.4 API Reference

Here are the relevant API functions for managing OTP:

int **`OTP_Init`**`()`

Brief: Function to Initialize the otp based on the Initialization waveforms for dual voltage mode given in the databook.

Arguments: -

Returns: SUCCESS

uint8_t **OTP_Read**(uint32_t addr_in)

Brief: Function to read data from the OTP, The control register is set accordingly and goes into while loop until the read data is ready. The OTP returns an 8 bit data for the address specified.

Arguments:

- addr_in (*uint32_t*) : The address from where the data to be read.

Returns: It returns the value that read from the OTP memory.

int **OTP_Write**(uint32_t addr_in, uint32_t data_in)

Brief: Function to write data into the OTP, The control registers are set accordingly to do a write operation. The function will wait until the PROGRAMMIN_SUCCESS signal has been sent. In case of a failure in programming, the a error message will be printed mentioning it hasfailed to program and returns.

Arguments:

- addr_in (*uint32_t*) : The address from where the data to be written.
- data_in (*uint32_t*) : The data to be written.

Returns: SUCCESS

uint8_t **OTP_Read32bitData**(uint32_t addr_in, uint8_t *data_out, uint8_t data_len)

Brief: The function reads 32-bit data from a specified address in OTP memory and stores it in an output buffer.

Arguments:

- addr_in (*uint32_t*) : The address from where the data to be read.
- data_out (*uint32_t*) : The pointer to an array where the 32-bit data read from the OTP memory will be stored.
- data_len (*uint8_t*) : The length of the data need to be read from OTP memory.

Returns: SUCCESS

uint8_t **reverse_bits**(uint8_t byte)

Brief: Function to reverses the order of bits in an 8-bit unsigned integer.

Arguments:

- byte (*uint8_t*) : It takes the value to reverses the order of its bits

Returns: It returns the reverse order of bits

3.3 Rivest-Shamir-Adleman 2048 bits (RSA) API

3.3.1 Required Includes

To use the RSA-2048 functionality, include the `rsa.h` header file in the source code. This header provides the necessary definitions, functions, and configuration structures to interface with the RSA hardware accelerator.

```
#include "rsa.h"
```

3.3.2 API Reference

uint16_t **RSA_Run**(uint8_t *output, const uint8_t *input, const uint8_t *exponent, const uint8_t *modulus)

Performs an RSA modular exponentiation operation.

This function performs the RSA core operation:

output = input^exponent mod modulus

All operands are expected to be 2048-bit (256-byte) big-endian values. The function computes the Montgomery constant R^2 mod n in software, loads all four operands into the hardware registers, waits for the hardware to complete, reads the 256-byte result, and triggers hardware zeroization before returning.

This API is used for RSA encryption, decryption, signature generation, and signature verification depending on the exponent provided.

Parameters

- **output** – Pointer to a 256-byte buffer where the RSA result will be stored.
- **input** – Pointer to a 256-byte buffer containing the input.
- **exponent** – Pointer to a 256-byte buffer containing the RSA exponent (public or private).
- **modulus** – Pointer to a 256-byte buffer containing the RSA modulus.

Returns

`SUCCESS` if the operation completes successfully, `EFAULT` if any required pointer is NULL, `ETIMEDOUT` if the RSA hardware

does not respond within the timeout window.

uint16_t **RSA_Zeroize**(void)

Triggers zeroization of all internal RSA hardware registers.

Writes 1 to the RSA_ZEROIZE register to initiate a hardware clear of all internal RSA state. Polls the RSA_ZEROIZE_STATUS register using a constant-time busy-wait until zeroization is confirmed complete or the timeout window expires.

This function may be called independently to ensure the hardware is in a clean state before a new RSA operation.

Returns

`SUCCESS` if zeroization completed within the timeout window, `ETIMEDOUT` if the RSA hardware zeroization does not complete within the timeout window.

3.4 Secure Hashing Algorithm 256 bits (SHA256) API

3.4.1 Required Includes

To use the SHA-256 functionality, include the `sha256.h` header file in the source code. This header provides the necessary definitions, functions, and configuration structures to interface with the SHA-256 hardware accelerator.

```
#include "sha256.h"
```

3.4.2 API Reference

uint16_t **SHA256_Single_Run**(uint8_t *sha_output, const uint8_t *input_text, size_t input_len_bits)

Performs a complete SHA-256 hashing operation on a single contiguous input buffer.

This function processes the entire input message in one call using the SHA-256 hardware accelerator. It calculates the required padding, processes each 512-bit block sequentially, and handles the overflow case where the final block requires a double run when padding spills into an additional block.

Once all blocks are processed, the final 256-bit hash is retrieved automatically via SHA256_Read_Output(). The computed 32-byte hash is written into sha_output.

Parameters

- **sha_output** – Pointer to the 32-byte buffer where the final hash output will be stored.
- **input_text** – Pointer to the input message to be hashed.
- **input_len_bits** – Length of the input message in bits.

Returns

`SUCCESS` if hashing completes successfully, `EFAULT` if any required pointer is NULL, `EINVAL` if an invalid block length condition occurs, `ETIMEDOUT` if the SHA-256 hardware does not respond within the timeout window.

uint16_t **SHA256_Multi_Run**(const uint8_t *input_text, size_t input_len_bits, size_t total_length, size_t iterated_length_bits)

Performs incremental SHA256 hashing across multiple runs.

This function allows processing of large messages in multiple calls. Each call processes a message chunk and updates the SHA hardware state.

When the final portion of the message is detected ((iterated_length_bits + input_len_bits) == total_length), padding and finalization are performed automatically.

This function does not return the final hash directly. The user must call SHA256_Read_Output() after completion to retrieve the 256-bit result.

Parameters

- **input_text** – Pointer to the current message chunk.
- **input_len_bits** – Length of the current chunk in bits.
- **total_length** – Total length of the complete message in bits.
- **iterated_length_bits** – Number of bits already processed. For the first call, this value should be 0.

Returns

`SUCCESS` if the chunk is processed successfully, `EFAULT` if input_text is NULL, `EINVAL` if input parameters are invalid, `ETIMEDOUT` if the SHA-256 hardware does not respond within the timeout window.

uint16_t **SHA256_Zeroize**(void)

Triggers zeroization of all internal SHA-256 hardware registers.

Writes 1 to SHA_ZEROIZE register to initiate a hardware clear of all internal SHA-256 state. Polls SHA_ZEROIZE_STATUS register using a constant-time busy-wait until the zeroization is complete or the timeout window expires.

Returns

`SUCCESS` if zeroization completed within the timeout window, `ETIMEDOUT` if the SHA-256 hardware zeroization does not complete within the timeout window.

3.5 TRNG

This section provides the structure and API details for the TRNG (True Random Number Generator) module used for secure random number generation compliant with NIST standards.

3.5.1 TRNG Structure

nist_trng_state

Brief: Internal TRNG state structure used by the driver to maintain internal entropy status and counters.

Note

This structure is used internally by the TRNG driver and its fields are not intended for direct user access. However, the user must create and pass an instance of this structure to the TRNG APIs, as the driver does not allocate it globally.

3.5.2 TRNG APIs

uint32_t **`TRNG_init`**(nist_trng_state *state, uint16_t req_sec_strength)

Brief: Initializes the TRNG with the required security strength and prepares the internal state for entropy collection and random number generation.

Arguments:

- **state** (*nist_trng_state**): Pointer to the TRNG state structure used internally by the driver.

- **req_sec_strength** (*uint16_t*): Required security strength. Valid range: **0–256** (for example, **128** or **256**).

Returns: 0 on success; any non-zero value indicates an error.

uint32_t **TRNG_Generate**(nist_trng_state *state, void *output, uint32_t no_of_bytes)

Brief: Generates cryptographically secure random bytes and stores them in the provided output buffer.

Arguments:

- **state** (*nist_trng_state**): Pointer to the TRNG internal state.
- **output** (*void**): Pointer to the memory buffer to store generated random bytes.
- **no_of_bytes** (*uint32_t*): Number of bytes to generate.

Returns: 0 on success; any non-zero value indicates an error.

uint32_t **TRNG_Uninstantiate**(nist_trng_state *state)

Brief: Securely uninstantiates the TRNG by clearing and zeroizing its internal state.

Arguments:

- **state** (*nist_trng_state**): Pointer to the TRNG state that must be securely cleared.

Returns: 0 on success; any non-zero value indicates an error.

uint8_t **Rand**(void)

Brief: Generates and returns a single random byte using the TRNG.

Returns: uint8_t random 8-bit value.

uint32_t **full_kat_test**(void)

Brief: Performs a full Known-Answer Test (KAT) on the TRNG using all combinations of kat_sel and kat_vec.

Returns: 0 if all KAT tests pass; a non-zero value if any test fails.

Chapter 4. Software Crypto APIs

4.1 Advanced Encryption Standard – Galois/Counter Mode (AES-GCM)

AES-GCM (Galois/Counter Mode) is an authenticated encryption mode that provides both data confidentiality and integrity, combining encryption with cryptographic authentication. It uses AES in Counter (CTR) mode for encryption and Galois field-based hashing (GHASH) for message authentication. The S2401 implements a hybrid hardware-software version of AES-GCM based on the NIST SP 800-38D specification, supporting 128-bit, 192-bit, and 256-bit key sizes with both standard 96-bit (12-byte) and variable-length initialization vectors. In this implementation, AES encryption operations (CTR mode for data encryption/decryption and ECB mode for hash subkey generation) are accelerated through the S2401 hardware AES engine, while the GHASH authentication function is performed in software using finite-field multiplication over GF(2^{128}) to process associated data, ciphertext, and length information for authentication tag generation.

4.1.1 Required Includes

To use the AES-GCM functionality, include the aes_gcm.h header file in your source code. This header provides all necessary definitions, structures, and API functions required for AES-GCM encryption and decryption operations.

```
#include "aes_gcm.h"
```

4.1.2 Enums and Data Structures

Enums

enum **AES_GCM_Mode**

AES-GCM operation modes.

This enumeration defines the supported modes for AES-GCM operations, specifying whether encryption or decryption is performed.

Values:

enumerator **AES_GCM_ENCRYPT**

AES-GCM encryption mode. Used to encrypt plaintext into ciphertext.

enumerator **AES_GCM_DECRYPT**

AES-GCM decryption mode. Used to decrypt ciphertext into plaintext.

Data Structures

struct **AES_GCM_Config**

AES-GCM configuration structure.

This structure contains all parameters required to perform AES encryption or decryption in Galois/Counter Mode (GCM), including input/output buffers, key, initialization vector (IV), Additional Authenticated Data (AAD), and the authentication tag.

Note

1. IV length (`iv_len_bits`) must not be zero, at least 8 bits is expected. 2. The `aad` pointer can be `NULL` if no Additional Authenticated Data is used. 3. All buffers (`cipher_text`, `input_text`, `key`, `iv`, `tag`) must point to valid memory locations with sufficient size. 4. AES key length (`key_len_bits`) must be 128, 192, or 256 bits. 5. Authentication tag length (`tag_len_bits`) must be one of the supported values: 128, 120, 112, 104, 96, 64, or 32 bits.

Public Members

uint8_t ***cipher_text**

Ciphertext buffer pointer.

Pointer to the buffer used for storing ciphertext during encryption or providing ciphertext during decryption.

uint8_t ***input_text**

Plaintext buffer pointer.

Pointer to the buffer used for providing plaintext during encryption or storing decrypted plaintext during decryption.

size_t **input_len_bits**

Length of input data in bits.

uint8_t ***aad**

Additional Authenticated Data (AAD) pointer.

Optional data that is authenticated but not encrypted. Can be `NULL` if no AAD is used.

size_t **aad_len_bits**

Length of AAD in bits.

uint8_t ***key**

AES key pointer.

Pointer to the key used for AES encryption/decryption.

size_t **key_len_bits**

AES key length in bits.

Valid values: 128, 192, 256.

uint8_t ***iv**

Initialization Vector (IV) pointer.

Pointer to the IV used for AES-GCM counter generation.

size_t **iv_len_bits**

IV length in bits.

Note

IV length must be at least 8 bits.

uint8_t ***tag**

Authentication tag buffer pointer.

Used to store the computed authentication tag during encryption or provide the expected tag during decryption for verification.

size_t **tag_len_bits**

Authentication tag length in bits.

Supported lengths: 128, 120, 112, 104, 96, 64, 32.

AES_GCM_Mode **mode**

AES-GCM operation mode.

Determines whether the operation is encryption or decryption. Valid values:

- AES_GCM_ENCRYPT
- AES_GCM_DECRYPT

4.1.3 API Reference

uint16_t **AES_GCM**(*AES_GCM_Config* *cfg)

Performs AES-GCM authenticated encryption or decryption.

This function encrypts or decrypts input data using AES in Galois/Counter Mode (GCM) and computes/verifies the authentication tag.

- In encryption mode, plaintext is converted to ciphertext and an authentication tag is generated.
- In decryption mode, ciphertext is converted to plaintext, and the authentication tag is verified. If verification fails, the plaintext buffer is cleared to prevent use of unauthenticated data.

The function handles key, IV (nonce), Additional Authenticated Data (AAD), and tag lengths. AES block operations (CTR and ECB modes) are performed using hardware acceleration; GCM-specific operations (hash subkey generation, GHASH, pre-counter J0 computation, tag XOR) are performed in software.

Parameters

cfg – Pointer to an *AES_GCM_Config* structure containing input/output buffers, key, IV, AAD, tag, lengths, and operation mode.

Returns

`SUCCESS` if operation succeeds with valid authentication tag, `AUTH_TAG_MISMATCH` if authentication fails during decryption, `EFAULT` if any required pointer in `cfg` is NULL, and `EINVAL` if key length, tag length, IV length, or mode are invalid.

4.2 ASCON

Ascon is a family of lightweight authenticated encryption and hashing algorithms designed for resource-constrained environments and IoT applications. It provides essential cryptographic operations including secure encryption, message authentication, and hashing with minimal resource requirements. This software implementation of Ascon follows the NIST SP 800-232 standard and supports four key variants: AEAD128a for authenticated encryption with associated data (AEAD), Hash256 for generating 256-bit cryptographic digests, XOF128 for extendable-output hashing with 128-bit security, and CXOF128 for customizable extendable-output hashing with domain separation capabilities.

4.2.1 Ascon Variant Specification

Each variant provides distinct cryptographic functionality and the steps below describe the internal processing performed by this software implementation.

1. **AEAD128a (Authenticated Encryption with Associated Data)**

   AEAD128a provides secure encryption while authenticating additional associated data (AAD). It takes a plaintext message and a public nonce as input, encrypts the message, and produces a 128-bit authentication tag that ensures the integrity of both the ciphertext and any associated data.

   Functional Steps:

   - **Initialize State** – Set up the internal state with the key and nonce.
   - **Absorb Associated Data** – Process associated data blocks and incorporate them into the state.
   - **Process Plaintext** – Encrypt plaintext blocks iteratively, updating the state and producing ciphertext.
   - **Finalize Output** – Compute and append the authentication tag to the ciphertext.

2. **Hash256**

   Hash256 is a fixed-length hashing function that generates a 256-bit digest from input data.

   Functional Steps:

   - **Initialize State** – Set up the internal hash state to predefined constants.
   - **Absorb Input** – Process input blocks and update the internal state iteratively.

- **Finalize Output** – Extract the 256-bit digest from the state into the output buffer.

3. **XOF128 (Extendable-Output Function)**

   XOF128 produces variable-length cryptographic output from input data, which provides requested output of any length by generating output blocks iteratively.

   Functional Steps:

   - **Initialize State** – Set up the internal state with predefined constants.
   - **Absorb Input** – Process input blocks and update the internal state iteratively.
   - **Generate Output** – Extract output blocks from the state repeatedly until the requested length is produced.

4. **CXOF128 (Customizable Extendable-Output Function)**

   CXOF128 extends XOF128 by including a customization string that personalizes the output. The customization string is absorbed alongside the input to ensure unique output for different contexts.

   Functional Steps:

   - **Initialize State** – Set up the internal state with predefined constants.
   - **Absorb Input and Customization String** – Process input and customization string blocks, updating the internal state.
   - **Generate Output** – Extract output blocks from the state repeatedly until the requested length is produced.

4.2.2 Required Includes

To use the ASCON cryptographic functionality, include the ascon.h header file in your source code. This header provides all required definitions, macros, and API functions needed for ASCON operations such as authenticated encryption (AEAD), hashing, XOF and CXOF.

```
#include "ascon.h"
```

4.2.3 Enums and Data Structures

Enums

enum **ASCON_Mode**

ASCON AEAD operation modes.

This enumeration defines the supported operation modes for ASCON AEAD, specifying whether encryption or decryption is performed.

Values:

enumerator **ASCON_ENCRYPT**

ASCON encryption mode. Encrypts plaintext and generates authentication tag.

enumerator **ASCON_DECRYPT**

ASCON decryption mode. Decrypts ciphertext and verifies authentication tag.

Data Structures

struct **ASCON_AEAD128a_Config**

ASCON AEAD-128a configuration structure.

This structure contains all parameters required to perform authenticated encryption or decryption using the ASCON AEAD-128a algorithm.

Public Members

uint8_t ***output**

Pointer to the output buffer.

- In encrypt mode: receives the ciphertext followed by the authentication tag.
- In decrypt mode: receives the decrypted plaintext.

size_t ***output_len**

Pointer to a variable that receives the total length of the output buffer.

- In encrypt mode: set to plaintext length + ASCON_TAG_SIZE.
- In decrypt mode: set to ciphertext length - ASCON_TAG_SIZE.

const uint8_t ***input**

Pointer to the input buffer.

- In encrypt mode: the plaintext message.
- In decrypt mode: the ciphertext including the authentication tag.

size_t **input_len**

Length of the input buffer in bytes.

In decrypt mode, the input length must be greater than or equal to ASCON_TAG_SIZE.

const uint8_t ***aad**

Pointer to additional authenticated data (AAD).

If unused, this parameter should be NULL.

size_t **aad_len**

Length of the AAD in bytes.

Ignored if aad is NULL.

const uint8_t ***sec_nonce**

Pointer to the secret nonce.

This parameter is not used and should be NULL.

const uint8_t ***pub_nonce**

Pointer to the public nonce.

The length of the public nonce must be exactly 16 bytes.

const uint8_t ***key**

Pointer to the secret key.

The length of the key must be exactly 16 bytes.

ASCON_Mode **mode**

Operation mode.

Must be one of:

- `ASCON_ENCRYPT` to perform authenticated encryption.

- `ASCON_DECRYPT` to perform authenticated decryption.

struct **ASCON_Hash256_Config**

ASCON Hash256 configuration structure.

This structure contains all parameters required to compute the ASCON-Hash256 digest of an input message.

Public Members

uint8_t ***hash_output**

Pointer to the buffer where the hash output is stored.

The buffer must be at least 32 bytes in size to store the 256-bit (32-byte) hash result.

const uint8_t ***input**

Pointer to the input message.

Points to the message data that will be hashed.

size_t **input_len**

Length of the input message in bytes.

struct **ASCON_XOF128_Config**

ASCON XOF128 configuration structure.

This structure contains all parameters required to compute a variable-length hash output using the ASCON-XOF128 algorithm.

Public Members

uint8_t ***hash_output**

Pointer to the output buffer.

The buffer must be large enough to store the number of bytes specified by hash_output_len.

size_t **hash_output_len**

Requested output length in bytes.

Specifies the number of output bytes to be generated by the ASCON-XOF128 function.

const uint8_t ***input**

Pointer to the input message.

Points to the message data that will be processed by the XOF function.

size_t **input_len**

Length of the input message in bytes.

struct **ASCON_CXOF128_Config**

ASCON CXOF128 configuration structure.

This structure contains all parameters required to compute a customizable variable-length output using the ASCON-CXOF128 algorithm.

Public Members

uint8_t ***hash_output**

Pointer to the output buffer.

The buffer must be large enough to store the number of bytes specified by hash_output_len.

size_t **hash_output_len**

Requested output length in bytes.

Specifies the number of output bytes to be generated by the ASCON-CXOF128 function.

const uint8_t ***input**

Pointer to the input message.

Points to the message data that will be processed by the CXOF function.

size_t **input_len**

Length of the input message in bytes.

const uint8_t ***custom_string**

Pointer to the customization string.

If unused, this parameter should be NULL.

size_t **custom_string_len**

Length of the customization string in bytes.

Ignored if custom_string is NULL.

4.2.4 API Reference

uint16_t **ASCON_AEAD128a**(*ASCON_AEAD128a_Config* *cfg)

Performs authenticated encryption or decryption using the ASCON AEAD algorithm.

This function either encrypts plaintext and generates an authentication tag, or verifies an authentication tag and decrypts ciphertext, depending on the specified mode.

Steps performed: 1. Initializes the ASCON AEAD state using the key and public nonce. 2. Absorbs the associated authenticated data (AAD), if provided. 3. Encrypts or decrypts the input data depending on the mode. 4. Finalizes the state and either appends (encrypt) or verifies (decrypt) the authentication tag.

Parameters

cfg – Pointer to an *ASCON_AEAD128a_Config* structure containing all required input, output, key, nonce, and mode parameters.

Returns

`SUCCESS` on successful encryption or decryption, `AUTH_TAG_MISMATCH` if authentication tag verification fails (decrypt mode only), `EFAULT` if any required buffer pointer inside cfg is NULL, and `EINVAL` if the input length is smaller than the authentication tag length (decrypt mode only).

uint16_t **ASCON_Hash256**(*ASCON_Hash256_Config* *cfg)

Computes the ASCON-Hash256 digest of an input message.

This function computes a fixed-length 256-bit cryptographic hash using the ASCON hash algorithm.

Steps performed: 1. Initializes the ASCON hash state. 2. Absorbs the input message. 3. Applies the ASCON permutation. 4. Produces the final 256-bit hash output.

Parameters

cfg – Pointer to an *ASCON_Hash256_Config* structure containing the input message and output buffer.

Returns

`SUCCESS` on successful hash computation, and `EFAULT` if the

required buffers inside cfg are NULL.

uint16_t **ASCON_XOF128**(*ASCON_XOF128_Config* *cfg)

Computes an extensible-output hash using ASCON-XOF128.

This function generates a variable-length cryptographic hash output from the input message.

Steps performed: 1. Initializes the ASCON XOF state. 2. Absorbs the input message. 3. Applies the ASCON permutation. 4. Squeezes the requested number of output bytes.

Parameters

cfg – Pointer to an *ASCON_XOF128_Config* structure containing the input message, output buffer, and requested output length.

Returns

`SUCCESS` on successful XOF hash computation, and `EFAULT` if the required buffers inside cfg are NULL.

uint16_t **ASCON_CXOF128**(*ASCON_CXOF128_Config* *cfg)

Computes a customized extensible-output hash using ASCON-CXOF128.

This function generates a variable-length hash using a customization string for domain separation.

Steps performed: 1. Initializes the ASCON CXOF state. 2. Absorbs the customization string. 3. Absorbs the input message. 4. Applies the ASCON permutation. 5. Squeezes the requested number of output bytes.

Parameters

cfg – Pointer to an *ASCON_CXOF128_Config* structure containing the input message, customization string, output buffer, and requested output length.

Returns

`SUCCESS` on successful Customizable XOF computation, and `EFAULT` if the required buffers inside cfg are NULL.

4.3 Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) is a public-key cryptographic technique used to provide secure key exchange, digital signatures, and authentication. ECC is based on the mathematical properties of elliptic curves defined over finite fields. Compared to traditional public-key algorithms, ECC provides equivalent security with smaller key sizes,

making it suitable for systems where memory and processing resources are limited. ECC relies on the computational difficulty of the elliptic curve discrete logarithm problem, which makes it infeasible to derive a private key from a corresponding public key using current computational methods. This software implementation aligns with the FIPS 186-4 Digital Signature Standard. It provides core ECC cryptographic operations required for key pair generation, shared secret computation, digital signature generation, and signature verification. This implementation uses hardware TRNG of S2401 for random number generation.

4.3.1 Supported Algorithms

- **Elliptic Curve Digital Signature Algorithm (ECDSA)** Used for generation and verification of digital signatures to ensure data authenticity and integrity.
- **Elliptic Curve Diffie–Hellman (ECDH)** Used for key agreement to derive a shared secret between communicating entities.

4.3.2 Supported Curves

The curve should be chosen in compile time. The following standard elliptic curves are supported by this implementation:

- **secp160r1** Standard elliptic curve defined over a 160-bit prime field.
- **secp192r1** NIST-recommended elliptic curve defined over a 192-bit prime field.
- **secp224r1** NIST-recommended elliptic curve defined over a 224-bit prime field.
- **secp256r1** NIST-recommended elliptic curve defined over a 256-bit prime field.
- **secp256k1** Standard elliptic curve defined over a 256-bit prime field, commonly used in blockchain applications.

4.3.3 Including the header

To use the ECC cryptographic functionality, include the ecc.h header file in your source code. This header provides all required definitions, macros, and API functions needed for ECC operations such as signing, verification and key exchange.

```
#include "ECC.h"
```

4.3.4 API Reference

Here are the relevant API functions for managing ASCON cryptographic functionality:

int **ECC_sign**(const uint8_t *private_key, const uint8_t *message_hash, unsigned hash_size, uint8_t *signature, ECC_Curve curve)

Brief: Generates an ECDSA signature for a given hash value. Compute a hash of the data you wish to sign (SHA-2 is recommended) and pass it along with your private key.

Parameters:

- **private_key** (*const uint8_t **) Pointer to the private key used for signing.
- **message_hash** (*const uint8_t **) Pointer to the hash of the message to sign. SHA-2 or other secure hash functions are recommended.
- **hash_size** (*unsigned*) Size of the *message_hash* in bytes.
- **signature** (*uint8_t **) Pointer to the output buffer that will be filled with the generated signature. Must be at least *2 * curve size* bytes. For example, for *secp256r1* the buffer must be 64 bytes long.
- **curve** (*ECC_Curve*) The elliptic curve to use for signature generation.

Returns:

- *1* if the signature was generated successfully.
- *0* if an error occurred during signature generation.

int **ECC_curve_private_key_size**(ECC_Curve curve)

Brief: Returns the size of a private key for the specified elliptic curve, in bytes.

Parameters:

- **curve** (*ECC_Curve*) The elliptic curve for which the private key size is requested.

Returns:

- The size of the private key in bytes.

int **ECC_curve_public_key_size**(ECC_Curve curve)

Brief: Returns the size of a public key for the specified elliptic curve, in bytes.

Parameters:

- **curve** (*ECC_Curve*) The elliptic curve for which the public key size is requested.

Returns:

- The size of the public key in bytes.

int **ECC_verify**(const uint8_t *public_key, const uint8_t *message_hash, unsigned hash_size, const uint8_t *signature, ECC_Curve curve)

Brief: Verifies an ECDSA signature. Compute the hash of the signed data using the same hash as the signer and pass it to this function along with the signer's public key and the signature values (r and s).

Parameters:

- **public_key** (*const uint8_t **) Pointer to the signer's public key.
- **message_hash** (*const uint8_t **) Pointer to the hash of the signed data. The hash algorithm used must match the one used during signing.
- **hash_size** (*unsigned*) Size of the *message_hash* in bytes.
- **signature** (*const uint8_t **) Pointer to the signature to verify.
- **curve** (*ECC_Curve*) The elliptic curve used for verification.

Returns:

- *1* if the signature is valid.
- *0* if the signature is invalid.

int **ECC_make_key**(uint8_t *public_key, uint8_t *private_key, ECC_Curve curve)

Brief: Generates a public/private key pair for a specified elliptic curve. The generated keys can be used for ECDSA signing or ECDH key agreement.

Parameters:

- **public_key** (*uint8_t **) Pointer to the output buffer that will be filled with the public key. Must be at least *2 * curve size* bytes. For example, for *secp256r1*, the buffer must be 64 bytes.
- **private_key** (*uint8_t **) Pointer to the output buffer that will be filled with the private key. Must be as long as the curve size in bytes. For *secp256r1*, the buffer must be 32 bytes. For *secp160r1*, the private key must be 21 bytes; the first byte will almost always be 0.
- **curve** (*ECC_Curve*) The elliptic curve to use for key pair generation.

Returns:

- *1* if the key pair was generated successfully.
- *0* if an error occurred.

ECC_Curve **ECC_secp160r1**(void)

Brief: Returns the elliptic curve object for secp160r1. Can be used as the *curve* parameter in ECC functions.

Returns: A *ECC_Curve* handle for the secp160r1 curve.

ECC_Curve **ECC_secp192r1**(void)

Brief: Returns the elliptic curve object for secp192r1. Can be used as the *curve* parameter in ECC functions.

Returns: A *ECC_Curve* handle for the secp192r1 curve.

ECC_Curve **ECC_secp224r1**(void)

Brief: Returns the elliptic curve object for secp224r1. Can be used as the *curve* parameter in ECC functions.

Returns: A *ECC_Curve* handle for the secp224r1 curve.

ECC_Curve **ECC_secp256r1**(void)

Brief: Returns the elliptic curve object for secp256r1. Can be used as the *curve* parameter in ECC functions.

Returns: A *ECC_Curve* handle for the secp256r1 curve.

ECC_Curve **ECC_secp256k1**(void)

Brief: Returns the elliptic curve object for secp256k1. Can be used as the *curve* parameter in ECC functions.

Returns: A *ECC_Curve* handle for the secp256k1 curve.

int **ECC_shared_secret**(const uint8_t *public_key, const uint8_t *private_key, uint8_t *secret, ECC_Curve curve)

Brief: Computes a shared secret using your private key and a peer's public key. The resulting secret can be used for symmetric encryption or HMAC after appropriate processing (e.g., hashing).

Parameters:

- **public_key** (*const uint8_t **) Pointer to the peer's public key. The key should be verified before use to ensure it is from a trusted source.
- **private_key** (*const uint8_t **) Pointer to your private key.
- **secret** (*uint8_t **) Pointer to the output buffer that will be filled with the shared secret. Must be the same size as the curve order. For example, for *secp256r1*, the buffer must be 32 bytes long.
- **curve** (*ECC_Curve*) The elliptic curve used for computing the shared secret.

Returns:

- *1* if the shared secret was generated successfully.
- *0* if an error occurred.

4.4 Secure Hashing Algorithm (SHA)

SHA (Secure Hash Algorithm) is a family of cryptographic hash functions designed to convert input data of any size into a fixed-length hash value. It is used to ensure data integrity, authentication, and security in cryptographic systems. SecureIOT provides SHA software implementation for both SHA2 and SHA3

4.4.1 Supported Algorithms

- **SHA2** SHA2 is a NIST-standardized family of cryptographic hash functions (SHA224, SHA256, SHA384, SHA-512) that generate fixed-length message digests and are widely used for ensuring data integrity, authentication, and secure communications. SHA2 is compliant with the NIST FIPS 180-4 standard.
- **SHA3** SHA3is the latest NIST hash standard, based on the Keccak algorithm, using a sponge construction that provides strong security and flexibility, and serves as a robust alternative to SHA2 with a fundamentally different internal design. SHA2 is specified in and compliant with the NIST FIPS 202 standard.

4.4.2 Including the header

To use the SHA hashing functionality, include the sha_software header file in your source code. This header provides all required definitions, macros, and API functions needed for hash generation.

```
#include "sha_software.h"
```

4.4.3 API Reference

API functions for Sha2 :

The following API functions provide support for all SHA2 variants, including SHA-224, SHA-384, and SHA-512.

int **SHA_InitSha224**(SHA_Sha224 *sha224)

Brief: This function initializes SHA224 struct.

Parameters:

- **sha224** (*SHA224 **) Pointer to the Sha224 structure to use for encryption.

Returns:

- **0** Success
- **1** Error returned because sha224 is null.

int **SHA_Sha224Update**(SHA_Sha224 *sha224, const uint8_t *msg, uint32_t len)

Brief: This function processes input data for SHA224 hashing.

Parameters:

- **sha224** (*SHA224 **) Pointer to the SHA224 structure.
- **msg** (*const uint8_t **) Pointer to the input message buffer.
- **len** (*uint32_t*) Length of the input message in bytes.

Returns:

- **0** Success
- **1** Error returned if function fails.
- **BAD_FUNC_ARG** Error returned if sha224 or data is null.

int **SHA_Sha224Final**(SHA_Sha224 *sha224, uint8_t *hash)

Brief: This function finalizes the SHA224 hash computation.

Parameters:

- **sha224** (*SHA224 **) Pointer to the SHA224 structure.
- **hash** (*uint8_t **) Pointer to the output buffer where the hash is stored.

Returns:

- **0** Success
- **1** Error

int **SHA_InitSha384**(SHA_Sha384 *sha384)

Brief: This function initializes the SHA384 structure.

Parameters:

- **sha384** (*SHA384 **) Pointer to the SHA384 structure to initialize.

Returns:

- **0** Returned upon successfully initializing

int **SHA_Sha384Update**(SHA_Sha384 *sha384, const uint8_t *msg, uint32_t len)

Brief: This function processes input data for SHA384 hashing.

Parameters:

- **sha384** (*SHA384 **) Pointer to the SHA384 structure.

- **msg** (*const uint8_t **) Pointer to the input message buffer.
- **len** (*uint32_t*) Length of the input message in bytes.

Returns:

- **0** Returned upon successfully adding the data to the digest.

int **SHA_Sha384Final**(SHA_Sha384 *sha384, uint8_t *hash)

Brief: This function finalizes the SHA384 hash computation.

Parameters:

- **sha384** (*SHA384 **) Pointer to the SHA384 structure.
- **hash** (*uint8_t **) Pointer to the output buffer where the hash is stored.

Returns:

- **0** Success
- **1** Error

int **SHA_InitSha512**(SHA_Sha512 *sha512)

Brief: This function initializes the SHA-512 structure.

Parameters:

- **sha512** (*SHA512 **) Pointer to the SHA-512 structure to initialize.

Returns:

- **0** Returned upon successfully initializing

int **SHA_Sha512Update**(SHA_Sha512 *sha512, const uint8_t *msg, uint32_t len)

Brief: This function processes input data for SHA-512 hashing.

Parameters:

- **sha512** (*SHA512 **) Pointer to the SHA-512 structure.
- **msg** (*const uint8_t **) Pointer to the input message buffer.
- **len** (*uint32_t*) Length of the input message in bytes.

Returns:

- **0** Returned upon successfully adding the data to the digest.

int **SHA_Sha512Final**(SHA_Sha512 *sha512, uint8_t *hash)

Brief: This function finalizes the SHA-512 hash computation.

Parameters:

- **sha512** (*SHA512 **) Pointer to the SHA-512 structure.
- **hash** (*uint8_t **) Pointer to the output buffer where the hash is stored.

Returns:

- **0** Returned upon successfully finalizing the hash.

API functions for Sha2 :

The following API functions provide support for all SHA3 variants, including SHA-224, SHA-384, and SHA-512.

int **SHA_InitSha3_224**(SHA_Sha3 *sha3_224, void *heap, int devId)

Brief: This function initializes the SHA3-224 structure.

Parameters:

- **sha3_224** (*SHA3_224 **) Pointer to the SHA3-224 structure to initialize.
- **heap** (*void **) Pointer to the heap memory (set to NULL if not used).
- **devId** (*int*) Device ID to use (set to INVALID_DEVID if not applicable).

Returns:

- **0** on success
- **Negative** on error (e.g., memory error)

int **SHA_Sha3_224Update**(SHA_Sha3 *sha3_224, const uint8_t *msg, uint32_t len)

Brief: This function processes input data for SHA3-224 hashing.

Parameters:

- **sha3_224** (*SHA3_224 **) Pointer to the SHA3-224 structure.
- **msg** (*const uint8_t **) Pointer to the input message buffer.
- **len** (*uint32_t*) Length of the input message in bytes.

Returns:

- **0** on success
- **Negative** on error

int **SHA_Sha3_224Final**(SHA_Sha3 *sha3_224, uint8_t *hash)

Brief: This function finalizes the SHA3-224 hash computation.

Parameters:

- **sha3_224** (*SHA3_224 **) Pointer to the SHA3-224 structure.
- **hash** (*uint8_t **) Pointer to the output buffer where the hash is stored.

Returns:

- **0** on success
- **Negative** on error

int **SHA_InitSha3_256**(SHA_Sha3 *sha3_256, void *heap, int devId)

Brief: This function initializes the SHA3-256 structure.

Parameters:

- **sha3_256** (*SHA3_256 **) Pointer to the SHA3-256 structure to initialize.
- **heap** (*void **) Pointer to the heap memory (set to NULL if not used).
- **devId** (*int*) Device ID to use (set to INVALID_DEVID if not applicable).

Returns:

- **0** on success
- **Negative** on error

int **SHA_Sha3_256Update**(SHA_Sha3 *sha3_256, const uint8_t *msg, uint32_t len)

Brief: This function processes input data for SHA3-256 hashing.

Parameters:

- **sha3_256** (*SHA3_256 **) Pointer to the SHA3-256 structure.
- **msg** (*const uint8_t **) Pointer to the input message buffer.
- **len** (*uint32_t*) Length of the input message in bytes.

Returns:

- **0** on success
- **Negative** on error

int **SHA_Sha3_256Final**(SHA_Sha3 *sha3_256, uint8_t *hash)

Brief: This function finalizes the SHA3-256 hash computation.

Parameters:

- **sha3_256** (*SHA3_256 **) Pointer to the SHA3-256 structure.
- **hash** (*uint8_t **) Pointer to the output buffer where the hash is stored.

Returns:

- **0** on success
- **Negative** on error

int **SHA_InitSha3_384**(SHA_Sha3 *sha3_384, void *heap, int devId)

Brief: This function initializes the SHA3-384 structure.

Parameters:

- **sha3_384** (*SHA3_384 **) Pointer to the SHA3-384 structure to initialize.
- **heap** (*void **) Pointer to the heap memory (set to NULL if not used).
- **devId** (*int*) Device ID to use (set to INVALID_DEVID if not applicable).

Returns:

- **0** on success
- **Negative** on error

int **SHA_Sha3_384Update**(SHA_Sha3 *sha3_384, const uint8_t *msg, uint32_t len)

Brief: This function processes input data for SHA3-384 hashing.

Parameters:

- **sha3_384** (*SHA3_384 **) Pointer to the SHA3-384 structure.
- **msg** (*const uint8_t **) Pointer to the input message buffer.
- **len** (*uint32_t*) Length of the input message in bytes.

Returns:

- **0** on success
- **Negative** on error

int **SHA_Sha3_384Final**(SHA_Sha3 *sha3_384, uint8_t *hash)

Brief: This function finalizes the SHA3-384 hash computation.

Parameters:

- **sha3_384** (*SHA3_384 **) Pointer to the SHA3-384 structure.
- **hash** (*uint8_t **) Pointer to the output buffer where the hash is stored.

Returns:

- **0** on success
- **Negative** on error

int **SHA_InitSha3_512**(SHA_Sha3 *sha3_512, void *heap, int devId)

Brief: This function initializes the SHA3-512 structure.

Parameters:

- **sha3_512** (*SHA3_512 **) Pointer to the SHA3-512 structure to initialize.
- **heap** (*void **) Pointer to the heap memory (set to NULL if not used).
- **devId** (*int*) Device ID to use (set to INVALID_DEVID if not applicable).

Returns:

- **0** on success
- **Negative** on error

int **SHA_Sha3_512Update**(SHA_Sha3 *sha3_512, const uint8_t *msg, uint32_t len)

Brief: This function processes input data for SHA3-512 hashing.

Parameters:

- **sha3_512** (*SHA3_512 **) Pointer to the SHA3-512 structure.
- **msg** (*const uint8_t **) Pointer to the input message buffer.
- **len** (*uint32_t*) Length of the input message in bytes.

Returns:

- **0** on success
- **Negative** on error

int `SHA_Sha3_512Final`(SHA_Sha3 *sha3_512, uint8_t *hash)

Brief: This function finalizes the SHA3-512 hash computation.

Parameters:

- **sha3_512** (*SHA3_512 **) Pointer to the SHA3-512 structure.
- **hash** (*uint8_t **) Pointer to the output buffer where the hash is stored.

Returns:

- **0** on success
- **Negative** on error

4.5 Hash-based Message Authentication Code (HMAC)

The HMAC driver provides a software implementation of the **Hash-based Message Authentication Code (HMAC)** algorithm. This driver follows the **NIST-specified HMAC construction**. HMAC is used to verify **data integrity** and **authenticity** by combining a cryptographic hash function with a secret key. It is widely used in secure communication protocols, firmware authentication, and message verification. This implementation supports multiple SHA-family hash algorithms.

The HMAC driver supports the following hash algorithms:

- SHA-256
- SHA-384
- SHA-512
- SHA3-224
- SHA3-256
- SHA3-384
- SHA3-512

Each hash algorithm can be selected at initialization time.

4.5.1 Required Includes

To use the HMAC driver, include the `hmac.h` header file in your source code. This header declares the HMAC APIs, data types, and configuration definitions required by the driver.

```
#include "hmac.h"
```

4.5.2 API Usage Sequence

The correct API call order is **mandatory** and must be followed exactly:

1. `HMAC_Set_Key()`
2. `HMAC_Update()`
3. `HMAC_Final()`

Chapter 5. Software USB (Bit-Banged USB Driver)

5.1 Overview of the Bit-Banged USB Driver

The bit-banged USB driver is a **software-only implementation of a USB low-speed interface**, built using general-purpose I/O (GPIO) pins instead of a dedicated USB controller.

The driver manually handles all USB signaling in software, including:

- USB line sampling and driving
- NRZI encoding and decoding
- Bit stuffing and unstuffing
- Packet framing and End-Of-Packet (EOP) detection
- CRC generation and validation
- USB reset detection

5.2 Hardware Interface Used (GPIO)

The driver uses **two GPIO pins** to emulate the USB physical layer:

USB Signal	GPIO Pin
D-	`DMINUS`
D+	`DPLUS`

5.2.1 Hardware Usage Details

- GPIO pins are dynamically reconfigured:
  - **Input mode** during USB reception
  - **Output mode** during USB transmission

- GPIO toggle registers are used to change line states with minimal latency.
- USB signaling states (J, K, SE0) are generated and detected purely through GPIO control.

No external USB PHY or controller is required.

5.3 USB Receive (RX) Flow

The USB receive path is **interrupt-driven** and is initiated by activity on the USB data lines.

5.3.1 RX Initialization

Before enabling USB reception, the application performs the following steps:

- Temporarily disables machine-level external interrupts.
- Registers the USB RX interrupt handler.
- Configures USB GPIO pins as inputs.
- Enables GPIO interrupt on the USB data line.
- Sets the interrupt priority and enables it in the PLIC.
- Re-enables machine-level external interrupts.

Typical RX initialization sequence:

```
asm volatile(
    "li      t0, 0x800\n"        // Bit 11: MEIE
    "csrc    zero, mie, t0\n"    // Disable machine external interrupts
);

PLIC_Set_Handler(GPIO0_IRQn + DPLUS, usb_rx_irq_handler, NULL);

GPIO_Config(GPIO_IN, GPIO_PINS(DMINUS) | GPIO_PINS(DPLUS) | GPIO_
↪PINS(BUTTON));
GPIO_Interrupt_Config(GPIO_PINS(DPLUS), 1);

PLIC_Set_Interrupt_Priority(GPIO0_IRQn + DPLUS, PLIC_PRIORITY_7);
PLIC_Interrupt_Enable(GPIO0_IRQn + DPLUS);
```

(continues on next page)

(continued from previous page)

```
asm volatile(
    "li      t0, 0x800\n"        // Bit 11: MEIE
    "csrrs   zero, mie, t0\n"    // Enable machine external interrupts
);
```

5.3.2 RX Trigger and Capture

- When the **D+ line transitions high**, the GPIO interrupt is triggered.
- This interrupt invokes `usb_rx_irq_handler()`.
- From this point, the driver continuously samples the USB D- line at USB bit intervals.

Inside the RX interrupt handler:

- All incoming USB bits are captured sequentially.
- NRZI decoding and bit unstuffing are applied.
- Decoded bits are packed into bytes.
- Bytes are stored into a shared receive buffer.
- Reception continues until an **End-Of-Packet (SE0)** condition is detected.

5.3.3 RX Completion and Data Availability

- Once the complete USB packet has been received:
  - `buffer[0]` is updated with the packet length (in bytes).
  - `buffer[1..n]` contains the decoded packet data.
  - The global flag `flag` is set to `1`.
- Until `flag` becomes `1`, the application **must wait** and should not access the receive buffer.

5.3.4 Application Usage Model

The application is expected to poll the completion flag and process the data once reception is complete.

Typical usage pattern:

```
while (flag == 0) {
    // wait for USB packet reception
}

uint8_t length = buffer[0];
// buffer[1..length] now contains received USB packet bytes

flag = 0;   // clear flag after processing
```

5.3.5 Alternative Trigger Source (D- Line)

Instead of using the D+ line as the interrupt trigger, the **D- line can also be used** to initiate USB reception.

When using D- as the trigger source:

- The GPIO interrupt must be configured for **low-level triggering**.
- This allows reception to begin when the line is held low (e.g., during SE0 or initial line activity).

Example configuration:

```
GPIO_Interrupt_Config(GPIO_PINS(DMINUS), 0);
```

In this mode, the rest of the RX flow remains unchanged. The interrupt handler still captures all USB bits and stores the decoded packet into the shared buffer.

—

5.4 USB Transmit (TX) Flow

Transmission is **application-driven**.

5.4.1 TX Flow

1. Populate a `Packet` structure:

```
Packet pkt;
pkt.pid = PID_DATA1;
pkt.addr = 1;
pkt.endpoint = 0;
pkt.data = payload;
pkt.data_length = payload_len;
```

2. Transmit the packet:

```
usb_tx_packet_send(&pkt);
```

5.4.2 TX Internals

- Packet fields are assembled in software.
- CRC5 or CRC16 is calculated as required.
- NRZI encoding and bit stuffing are applied.
- GPIO pins are driven at precise USB bit timings.
- End-Of-Packet signaling is generated automatically.

5.5 Features Supported in This Implementation

5.5.1 Protocol Features

- USB **low-speed signaling** (1.5 Mbps)
- NRZI encoding and decoding
- Bit stuffing and unstuffing
- End-Of-Packet (SE0) detection
- USB reset detection
- PID handling with complement bits
- CRC5 (token packets)
- CRC16 (data packets)

5.5.2 Packet Types

- Token packets: IN, OUT, SETUP
- Data packets: DATA0, DATA1, DATA2
- Handshake packets: ACK, NAK, STALL

5.5.3 Implementation Features

- Branchless critical paths for timing stability
- Cycle-accurate delay loops
- Interrupt-driven receive path
- Software-controlled transmit timing
- Optional raw USB bit capture for debugging

5.6 Limitations and Safety Notes

5.6.1 Limitations

- Only **low-speed USB** is supported.
- CPU utilization is high during USB activity.
- USB timing depends on CPU frequency stability.
- No error recovery or retry mechanisms are implemented.

5.6.2 Safety Notes

- No other applications should execute while this USB application is running.
- GPIO interrupts must be carefully managed to avoid timing violations.
- Transmit routines temporarily disable GPIO interrupts.
- Printing or logging inside the RX ISR must be avoided.
- USB timing may break if other high-priority interrupts are active.

5.7 Including the USB Bit-Bang Header

To use the software-based USB (bit-banged) driver, include the `usb_bitbang.h` header file in your source code. This header provides the necessary definitions, data structures, and function prototypes required to implement USB communication using GPIO pins.

```
#include "usb_bitbang.h"
```

5.8 GPIO Configuration

The USB bit-banged driver uses two GPIO pins to emulate the USB D+ and D- data lines.

`DMINUS`
: GPIO pin number used for the USB D- line.

`DPLUS`
: GPIO pin number used for the USB D+ line.

5.9 Shared Global Variables

The following global variables are shared between the USB receive (RX) interrupt handler and the application logic.

`flag`
: A volatile flag set by the RX interrupt handler when a complete USB packet has been successfully received.

`reset_flag`
: A volatile flag set when a USB reset condition (prolonged SE0 state) is detected on the bus.

`buffer`
: A byte array used to store decoded USB packet data.
  - `buffer[0]` contains the packet length in bytes
  - `buffer[1..n]` contain the packet payload

`raw_bits`
: A buffer that stores raw sampled USB D- line bits before NRZI decoding and bit

unstuffing.

5.10 USB Packet Identifier (PID) Types

USB packet types are identified using Packet Identifiers (PIDs). Only the lower 4 bits of the PID are significant; the upper 4 bits are transmitted as the bitwise complement.

The supported USB PID values are defined by the `USB_PID` enumeration.

```
typedef enum {
    PID_OUT   = 1,
    PID_ACK,
    PID_DATA0,
    PID_PING,
    PID_SOF,
    PID_NYET,
    PID_DATA2,
    PID_SPLIT,
    PID_IN,
    PID_NAK,
    PID_DATA1,
    PID_PRE,
    PID_ERR,
    PID_SETUP,
    PID_STALL,
    PID_MDATA
} USB_PID;
```

5.11 USB Packet Structure

USB packets are represented using the `Packet` structure. This structure is used both for decoded received packets and for assembling packets to be transmitted.

```
typedef struct {
    uint8_t  pid;           /* Packet Identifier */
    uint16_t frame_number;  /* Frame number (SOF packets) */
```

(continues on next page)

(continued from previous page)

```
    uint8_t *data;          /* Pointer to payload buffer */
    uint8_t  endpoint;      /* Endpoint number */
    uint8_t  addr;          /* Device address */
    uint8_t  data_length;   /* Payload length in bytes */
} Packet;
```

5.12 API Reference

The following API functions are provided by the USB bit-banged driver.

void **usb_rx_irq_handler**(void *args)

Brief: USB receive interrupt handler.

This function is triggered by GPIO interrupt activity on the USB D+ or D- lines. It samples the raw signal transitions, performs NRZI decoding and bit unstuffing, and stores the decoded USB packet into the shared receive buffer.

Arguments: * **args** (`void*`): Optional interrupt handler argument (unused).

Returns: * None.

int **usb_tx_packet_send**(Packet *packet)

Brief: Assembles and transmits a USB packet on the bus.

This function builds the complete USB packet bitstream, including SYNC pattern, PID, payload data, and CRC. The bitstream is NRZI encoded, bit-stuffed as required, and transmitted via GPIO.

Arguments: * **packet** (`Packet*`): Pointer to a packet structure containing the packet fields to be transmitted.

Returns: * `0` on successful transmission. * A error value if an error occurs during transmission.

Chapter 6. Platform Security Architecture

Platform Security Architecture (PSA) is a framework of security specifications for embedded devices, particularly in the Internet of Things (IoT). It provides a holistic approach to device security by combining hardware and firmware specifications, threat models, security analyses, and open-source reference implementations. Key features include a hardware Root of Trust, a Trusted Execution Environment (TEE), secure boot, secure storage, and a standard API for cryptographic operations.

6.1 Required Includes

To use the PSA functionality, include the *psa.h* header file in your source code. This header provides the necessary definitions, functions, and configuration structures to interface with the PSA.

```
#include "psa.h"
```

6.2 Macros, Data Types, Enums and Data Structures

6.2.1 Macros

PSA_MG_AES_BLOCK_LENGTH

Maximum block size of any supported AES cipher, in bytes. AES always operates on 128-bit (16-byte) blocks regardless of key size.

PSA_MG_MAX_AES_KEY_SIZE

Maximum AES key size supported, in bytes. Supported key sizes are 16 bytes (AES-128), 24 bytes (AES-192), and 32 bytes (AES-256).

PSA_MG_AES_IV_SIZE

IV size used for all supported AES cipher algorithms (CBC, CTR, CFB, OFB), in bytes.

PSA_MG_AES_KEY_SIZE_128_BITS

Valid AES key sizes in bits.

PSA_MG_AES_KEY_SIZE_192_BITS

PSA_MG_AES_KEY_SIZE_256_BITS

PSA_MG_AES_KEY_SIZE_128

Valid AES key sizes in bytes.

PSA_MG_AES_KEY_SIZE_192

PSA_MG_AES_KEY_SIZE_256

PSA_CIPHER_OPERATION_INIT

Initializer for a cipher operation structure. Must be used to initialize a *psa_cipher_operation_t* object before use.

PSA_HASH_OPERATION_INIT

Initializer for a hash operation structure.

PSA_MG_SHA256_OUTPUT_LEN_BITS

SHA256 output length in bits.

PSA_MG_SHA256_OUTPUT_LEN

SHA256 output length in bytes.

PSA_MG_SHA256_MAX_INPUT_LEN_BITS

Maximum input length for SHA256 in bits.

PSA_MG_SHA256_MAX_INPUT_LEN

Maximum input length for SHA256 in bytes.

PSA_MG_HASH_MAX_SIZE

Maximum hash output size in bytes (generic).

PSA_MG_RSA_MAX_SIZE

Maximum RSA key size in bytes (implementation-specific).

PSA_MG_RSA_256_BYTES

Size of a 2048-bit RSA key in bytes.

PSA_MG_RSA_BLOCK_SIZE

RSA block size in bytes.

PSA_MG_RSA_SIGNATURE_SIZE

RSA signature size in bytes.

PSA_MG_RSA_SIGNATURE_SIZE

RSA signature size in bytes.

PSA_MG_RSA_MODULUS_SIZE

RSA modulus size in bytes.

PSA_MG_RSA_PRIVATE_EXPONENT_SIZE

RSA private exponent size in bytes.

PSA_MG_RSA_MIN_PUBLIC_EXPONENT

Minimum valid RSA public exponent.

PSA_MG_RSA_MAX_PUBLIC_EXPONENT

Maximum valid RSA public exponent.

PSA_MG_ASN1_INTEGER_TAG

ASN.1 primitive and constructed type identifiers.

PSA_MG_ASN1_SEQUENCE_TAG

PSA_MG_ASN1_LONG_FORM_FLAG

PSA_MG_ASN1_LENGTH_MASK

PSA_MG_ASN1_MAX_LENGTH_BYTES

PSA_MG_ASN1_BYTE_SHIFT

PSA_ALG_SHA_256

SHA-256 hash algorithm.

PSA_ALG_CTR

CTR stream cipher mode.

PSA_ALG_CFB

CFB stream cipher mode.

PSA_ALG_OFB

OFB stream cipher mode.

PSA_ALG_CBC_NO_PADDING

CBC block cipher mode, no padding.

PSA_ALG_RSA_PKCS1V15_CRYPT

Base algorithms for asymmetric encryption.

PSA_ALG_RSA_OAEP_SHA256

PSA_ALG_RSA_PKCS1V15_SIGN_RAW

Raw PKCS#1 v1.5 signature.

PSA_ALG_RSA_PSS_BASE_ALG

RSA PSS base algorithm.

PSA_KEY_TYPE_AES

AES key for cipher, AEAD, or MAC algorithms.

PSA_KEY_TYPE_RSA_PUBLIC_KEY

RSA public key.

PSA_KEY_TYPE_RSA_KEY_PAIR

RSA key pair (private + public key).

PSA_KEY_ATTRIBUTES_INIT

Returns a suitable initializer for a key attribute object of type *psa_key_attributes_t*.

PSA_KEY_ID_INIT

Initial key identifier value.

PSA_KEY_USAGE_ENCRYPT

PSA_KEY_USAGE_DECRYPT

PSA_KEY_USAGE_SIGN_MESSAGE

PSA_KEY_USAGE_VERIFY_MESSAGE

6.2.2 Data Types

typedef uint32_t **psa_key_id_t**

Encoding of identifiers for persistent keys.

- Applications may choose key identifiers in the range PSA_KEY_ID_USER_MIN to PSA_KEY_ID_USER_MAX.
- Implementations may define additional identifiers in the range PSA_KEY_ID_VENDOR_MIN to PSA_KEY_ID_VENDOR_MAX.
- 0 is reserved as an invalid key identifier.
- Identifiers outside these ranges are reserved for future use.

typedef uint16_t **psa_key_type_t**

Encoding of a key type.

Values are generally constructed by macros `PSA_KEY_TYPE_xxx`.

Note

Values are stored in the persistent key store. Changing them requires bumping the storage format version and providing translation.

typedef uint32_t **psa_key_lifetime_t**

Encoding of key lifetimes.

- Bits 0-7 indicate persistence level (PSA_KEY_LIFETIME_GET_PERSISTENCE).
- Bits 8-31 indicate the key location (PSA_KEY_LIFETIME_GET_LOCATION).

Volatile keys are destroyed automatically on application termination or power reset. Persistent keys survive until explicitly destroyed or by integration-specific events.

Values are constructed using `PSA_KEY_LIFETIME_xxx` macros.

Note

Encoded in persistent storage; changes require storage version bump.

typedef uint32_t **psa_key_usage_t**

Encoding of permitted usage on a key.

Values are bitwise ORs of `PSA_KEY_USAGE_xxx` macros.

Note

Encoded in persistent storage; changes require storage version bump.

typedef uint32_t **psa_algorithm_t**

Encoding of a cryptographic algorithm.

Values are constructed by `PSA_ALG_xxx` macros. For multi-key algorithms, this type encodes the mode/padding, not the key type.

Note

Encoded in persistent storage; changes require storage version bump.

typedef struct rsa_context **mg_rsa_context**

RSA context structure.

This structure holds all parameters required for an RSA key pair, including the public modulus, public exponent, and private exponent. It is used for RSA encryption, decryption, signing, and verification operations.

6.2.3 Enums

enum **mg_operation_t**

Type of cipher operation.

Values:

enumerator **MG_PSA_AES_OPERATION_NONE**

No operation.

enumerator **MG_PSA_AES_ENCRYPT**

Encryption operation.

enumerator **MG_PSA_AES_DECRYPT**

Decryption operation.

6.2.4 Data Structures

struct **psa_key_attributes_t**

Key attributes structure.

Represents metadata for a key object, including type, size, lifetime, usage policy, and internal implementation flags.

Public Members

psa_key_type_t **type**

Key type.

Specifies the type of the key (e.g., symmetric, RSA, ECC).

psa_key_bits_t **bits**

Key size in bits.

Number of bits of the key material.

psa_key_lifetime_t **lifetime**

Lifetime of the key.

Indicates whether the key is volatile or persistent and its storage location.

psa_key_id_t **id**

Key identifier.

Unique identifier for the key. Only valid for persistent keys.

psa_key_policy_t **policy**

Key usage policy.

Embedded `psa_key_policy_t` structure specifying allowed operations and permitted algorithms.

psa_key_attributes_flag_t **flags**

Internal flags.

Implementation-specific flags used internally by the library.

struct **psa_cipher_operation_t**

PSA cipher operation structure.

Represents the complete state of a multi-part PSA cipher operation. Must be initialized by calling psa_cipher_setup() before use. Operations must be finalized with *psa_cipher_finish()* or abandoned with *psa_cipher_abort()*.

Public Members

psa_algorithm_t **alg**

Algorithm used for this cipher operation.

Stores the PSA algorithm identifier (e.g. PSA_ALG_CBC_NO_PADDING, PSA_ALG_CTR) set during psa_cipher_setup(). Used to identify the active algorithm throughout the operation lifecycle.

mg_cipher_context_t **cipher**

Internal cipher context holding full operation state.

Contains key material, IV, mode, unprocessed data buffer, and all other state required to execute and resume the cipher operation across multiple update calls.

uint8_t **iv_set**

Flag indicating whether the IV has been set.

Set to 1U by *psa_cipher_set_iv()* after a valid IV is provided. Set to 0U during psa_cipher_setup(). *psa_cipher_update()* will return PSA_ERROR_BAD_STATE if this flag is not set before calling it.

struct **psa_hash_operation_t**

PSA hash operation structure.

This structure represents an ongoing or completed hash operation in the PSA Crypto API. It encapsulates:

- A unique operation identifier
- The algorithm being used
- The internal SHA-256 context

Public Members

uint32_t **id**

Unique identifier for the hash operation.

Set to 1 when the operation is active, and cleared to 0 when the operation is aborted or completed.

psa_algorithm_t **alg**

Algorithm used for this hash operation.

Set to PSA_ALG_SHA_256 during setup.

mg_sha256_context **sha256_ctx**

Internal SHA-256 context.

Stores intermediate state for the hash computation.

6.3 API references

6.3.1 Core Setup and Randomness

psa_status_t **psa_crypto_init**(void)

Initializes the PSA Crypto subsystem.

This function initializes all internal components required by the PSA Crypto API. It sets up the True Random Number Generator (TRNG), seeds the entropy source,

initializes cryptographic driver wrappers, and prepares the key slot management subsystem. Calling this function multiple times is explicitly permitted — if the subsystem is already fully initialized, the function returns PSA_SUCCESS immediately without reinitializing any resources.

This function must be called before invoking any other PSA Crypto operation such as key import, cipher, hash, or random number generation. Failure to do so will result in PSA_ERROR_BAD_STATE from the called operation.

On any initialization failure, all partially initialized subsystems are cleaned up automatically before the error is returned.

Returns

- PSA_SUCCESS if the crypto subsystem is successfully initialized or was already fully initialized.
- PSA_ERROR_INSUFFICIENT_MEMORY if key slot memory allocation fails (dynamic key store only).
- PSA_ERROR_GENERIC_ERROR if hardware memory protection (PMP) setup fails (dynamic key store only).

group **Random generation**

Functions for cryptographically secure random number generation.

This group provides APIs to generate random data suitable for cryptographic use. The implementation may use a hardware true random number generator (TRNG) when available, or a software-based fallback mechanism otherwise.

Functions

psa_status_t **psa_generate_random**(uint8_t *output, size_t output_size)

Generates cryptographically secure random data.

This function generates random bytes and writes them to the output buffer. If a hardware-based true random number generator (TRNG) is available, it is used as the entropy source. Otherwise, an implementation-defined software random generator is used. This function provides a single-call interface for obtaining random data suitable for cryptographic operations.

Note

When PSA_TRNG_AVAILABLE is not defined, the software fallback is not cryptographically secure and should be used for testing only.

Parameters

- **output** – Pointer to the buffer where generated random bytes will be written.
- **output_size** – Number of random bytes to generate. When PSA_TRNG_AVAILABLE is not defined, must be a multiple of 4 and must not exceed 256 bytes.

Returns

- PSA_SUCCESS if random data is generated successfully.
- PSA_ERROR_INVALID_ARGUMENT if output is NULL, output_size is zero, or when PSA_TRNG_AVAILABLE is not defined, output_size exceeds 256 bytes or is not a multiple of 4 bytes.
- PSA_ERROR_HARDWARE_FAILURE if the TRNG hardware times out while generating random data.

6.3.2 Key Management and Attributes

void **psa_set_key_type**(*psa_key_attributes_t* *attributes, *psa_key_type_t* type)

Sets the key type in a key attributes structure.

This function assigns the key type (for example, AES, RSA) to a key attributes object. The key type defines the cryptographic algorithm family and the structure of the key material. This function does not perform validation; validation occurs when the key is created or imported.

Parameters

- **attributes** – Pointer to the key attributes structure to modify.
- **type** – Key type to assign.

void **psa_set_key_algorithm**(*psa_key_attributes_t* *attributes, *psa_algorithm_t* alg)

Sets the permitted algorithm for a key.

This function specifies the cryptographic algorithm or algorithm policy that the key is allowed to be used with. The algorithm restriction is enforced during cryptographic operations such as encryption, signing, or verification.

Parameters

- **attributes** – Pointer to the key attributes structure to modify.
- **alg** – Algorithm or algorithm policy permitted for the key.

void **psa_set_key_bits**(*psa_key_attributes_t* *attributes, size_t bits)

Sets the key size in bits in a key attributes structure.

This function records the key size to be associated with a key when it is later created or imported. If the provided size exceeds PSA_MAX_KEY_BITS, the size is marked as invalid by storing PSA_KEY_BITS_TOO_LARGE in the attributes structure. The value is not validated until key creation or import is attempted.

Parameters

- **attributes** – Pointer to the key attributes structure to modify.
- **bits** – Key size in bits to set.

void **psa_set_key_lifetime**(*psa_key_attributes_t* *attributes, *psa_key_lifetime_t* lifetime)

Sets the lifetime of a key.

This function assigns a lifetime to the key, defining whether the key is volatile or persistent and where it is stored. If a volatile lifetime is selected, any previously assigned key identifier is cleared to zero to ensure the key is not treated as persistent.

Parameters

- **attributes** – Pointer to the key attributes structure to modify.
- **lifetime** – Lifetime value to assign to the key.

void **psa_set_key_usage_flags**(*psa_key_attributes_t* *attributes, *psa_key_usage_t* usage_flags)

Sets the usage permissions for a key.

This function defines the operations that the key is permitted to perform, such as encryption, decryption, signing, or verification. If hash-based signing or verification permissions are requested, the corresponding message-based permissions are automatically enabled.

Parameters

- **attributes** – Pointer to the key attributes structure to modify.
- **usage_flags** – Bitmask of permitted key usage operations.

void **psa_set_key_id**(*psa_key_attributes_t* *attributes, *psa_key_id_t* key)

Sets the key identifier in a key attributes structure.

This function assigns a key identifier to the attributes structure. If the current lifetime is volatile, it is automatically updated to persistent to reflect that the key now has a fixed identifier.

Parameters

- **attributes** – Pointer to the key attributes structure to modify.
- **key** – Key identifier to assign.

psa_key_type_t **psa_get_key_type**(const *psa_key_attributes_t* *attributes)

Retrieves the key type from a key attributes structure.

This function returns the key type that was previously set in the key attributes structure using *psa_set_key_type()*.

Parameters

attributes – Pointer to the key attributes structure.

Returns

The key type stored in the attributes.

psa_algorithm_t **psa_get_key_algorithm**(const *psa_key_attributes_t* *attributes)

Retrieves the permitted algorithm from a key attributes structure.

This function returns the algorithm policy associated with the key, as previously set using *psa_set_key_algorithm()*.

Parameters

attributes – Pointer to the key attributes structure.

Returns

The algorithm stored in the key policy.

size_t **psa_get_key_bits**(const *psa_key_attributes_t* *attributes)

Retrieves the key size in bits from a key attributes structure.

This function returns the key size that is currently set in the key attributes object. The value reflects what was configured using *psa_set_key_bits()* and is not validated until the key is created or imported.

Parameters

attributes – Pointer to the key attributes structure to query.

Returns

The key size in bits stored in the attributes structure.

psa_key_lifetime_t **psa_get_key_lifetime**(const *psa_key_attributes_t* *attributes)

Retrieves the lifetime of a key from a key attributes structure.

This function returns the key lifetime that was previously set using *psa_set_key_lifetime()*.

Parameters

attributes – Pointer to the key attributes structure.

Returns

The key lifetime stored in the attributes.

psa_key_usage_t **psa_get_key_usage_flags**(const *psa_key_attributes_t* *attributes)

Retrieves the usage flags from a key attributes structure.

This function returns the usage permissions associated with the key, as previously set using *psa_set_key_usage_flags()*.

Parameters

attributes – Pointer to the key attributes structure.

Returns

The key usage flags stored in the attributes.

psa_key_id_t **psa_get_key_id**(const *psa_key_attributes_t* *attributes)

Retrieves the key identifier from a key attributes structure.

This function returns the key identifier associated with the key attributes. For volatile keys, this value is zero until the key is created.

Parameters

attributes – Pointer to the key attributes structure.

Returns

The key identifier stored in the attributes.

psa_status_t **psa_import_key**(const *psa_key_attributes_t* *attributes, const uint8_t *data, size_t data_length, *psa_key_id_t* *key)

Imports a key into the PSA Crypto key store.

This function creates a new key object from raw key material provided in the input buffer and stores it in the PSA Crypto key storage. The key is assigned a unique identifier and its attributes are set according to the provided `attributes` structure. This function handles both volatile and persistent keys, ensures bit-size limits, and invokes the driver to store key material securely.

Parameters

- **attributes** – Pointer to a *psa_key_attributes_t* structure describing the properties of the key (type, usage, algorithm, lifetime, etc.). Must not be NULL.
- **data** – Pointer to the raw key material to import. Must not be NULL.
- **data_length** – Size of the key material in bytes. Must be non-zero.
- **key** – Pointer to a variable where the assigned key identifier will be written upon successful import. Must not be NULL.

Returns

- PSA_SUCCESS if the key is successfully imported and stored.
- PSA_ERROR_INVALID_ARGUMENT if any input pointer is NULL, the key length is zero, or the provided key bits do not match the expected values.
- PSA_ERROR_NOT_SUPPORTED if the key size exceeds PSA_MAX_KEY_BITS or the key type is not supported.
- PSA_ERROR_BAD_STATE if the PSA Crypto subsystem is not initialized.
- PSA_ERROR_BUFFER_TOO_SMALL if the allocated key buffer is too small for the provided key material.
- PSA_ERROR_CORRUPTION_DETECTED if an internal integrity failure occurs during key import.

psa_status_t **mg_psa_rsa_export_key**(*psa_key_type_t* type, *mg_rsa_context* *rsa, uint8_t *data, size_t data_size, size_t *data_length)

Exports an RSA key (public or private) into a buffer in DER format.

This function exports either an RSA private key pair or a public key from an mg_rsa_context into the provided buffer. The key is encoded in DER format as an ASN.1 SEQUENCE containing the modulus and either the private exponent (for key pairs) or the public exponent (for public keys). The exported data is aligned to the beginning of the output buffer.

Note

On any failure, the output buffer is zeroed before returning.

Parameters

- **type** – Type of the key to export. Must be either a key pair type (PSA_KEY_TYPE_IS_KEY_PAIR) or a public key type (PSA_KEY_TYPE_IS_PUBLIC_KEY). Any other type returns PSA_ERROR_INVALID_ARGUMENT.
- **rsa** – Pointer to an initialized mg_rsa_context holding the RSA key material to export. Must not be NULL.
- **data** – Pointer to the output buffer where the DER-encoded key is written. Must not be NULL.
- **data_size** – Size of the output buffer in bytes. Must be non-zero and large enough to hold the DER-encoded key.
- **data_length** – On success, set to the number of bytes written

to the output buffer. Must not be NULL.

Returns

- PSA_SUCCESS on successful export.
- PSA_ERROR_INVALID_ARGUMENT if the key type is neither a key pair nor a public key type, or if any input pointer is NULL, or if data_size is zero.
- PSA_ERROR_BUFFER_TOO_SMALL if the output buffer is too small to hold the DER-encoded key.
- PSA_ERROR_RSA_BAD_INPUT_DATA if a private exponent export is attempted on a context that holds only a public key (no private parameters).
- PSA_ERROR_CORRUPTION_DETECTED if an internal integrity check fails.

psa_status_t **psa_destroy_key**(*psa_key_id_t* key)

Destroys a key identified by its key ID.

This function securely deletes a key from the key store, zeroizing all associated key material and releasing the key slot. If the key ID is zero or the key does not exist in the store, the function treats it as a no-op and returns success. This ensures safe key management without revealing whether a key existed.

Parameters

key – Identifier of the key to destroy.

Returns

- PSA_SUCCESS if the key was successfully destroyed or did not exist.
- PSA_ERROR_CORRUPTION_DETECTED if an internal integrity failure is detected during key slot cleanup.

6.3.3 Symmetric Cipher Operations

group **Cipher operations**

Functions for symmetric cipher encryption and decryption.

This group provides APIs to perform single-shot and multipart symmetric cipher operations using PSA Crypto compliant interfaces. Supported operations include setup, IV configuration, update, finish, and abort of cipher operations. All opera-

tions support AES cipher modes: CBC, CTR, CFB, and OFB with key sizes of 128, 192, and 256 bits.

Functions

psa_status_t **psa_cipher_encrypt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, uint8_t *output, size_t output_size, size_t *output_length)

Encrypts a plaintext using a symmetric cipher key.

This function performs single-shot symmetric encryption using the specified cipher algorithm. A random initialization vector (IV) is generated internally and prepended to the output buffer followed by the ciphertext. The key must permit encryption usage and be compatible with the selected algorithm. The input length must be a non-zero multiple of PSA_MG_AES_BLOCK_LENGTH.

Parameters

- **key** – Identifier of the key to use for encryption. Must be a valid non-zero key identifier with PSA_KEY_USAGE_ENCRYPT permission.
- **alg** – Cipher algorithm to use. Must satisfy PSA_ALG_IS_CIPHER(alg) and be one of the supported algorithms: PSA_ALG_CBC_NO_PADDING, PSA_ALG_CTR, PSA_ALG_CFB, or PSA_ALG_OFB.
- **input** – Pointer to the plaintext input buffer. Must not be NULL.
- **input_length** – Size of the input buffer in bytes. Must be a non-zero multiple of PSA_MG_AES_BLOCK_LENGTH.
- **output** – Buffer where the IV followed by the ciphertext is written. Must not be NULL and must be at least PSA_MG_AES_IV_SIZE + input_length bytes.
- **output_size** – Size of the output buffer in bytes.
- **output_length** – On success, set to the total number of bytes written to the output buffer (PSA_MG_AES_IV_SIZE + input_length). Must not be NULL.

Returns

- PSA_SUCCESS on successful encryption.
- PSA_ERROR_INVALID_ARGUMENT if input, output, or out-

put_length is NULL, the key identifier is zero, the input length is zero or not a multiple of PSA_MG_AES_BLOCK_LENGTH, or the algorithm is not a cipher algorithm.

- PSA_ERROR_NOT_SUPPORTED if the algorithm is not one of the four supported cipher modes.
- PSA_ERROR_BUFFER_TOO_SMALL if output_size is smaller than PSA_MG_AES_IV_SIZE + input_length bytes.
- PSA_ERROR_NOT_PERMITTED if the key does not allow encryption usage.
- PSA_ERROR_INVALID_HANDLE if the key identifier does not refer to a valid key.
- PSA_ERROR_BAD_STATE if the crypto subsystem is not initialized.
- PSA_ERROR_CORRUPTION_DETECTED if an internal integrity check fails.

psa_status_t **psa_cipher_decrypt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, uint8_t *output, size_t output_size, size_t *output_length)

Decrypts a ciphertext using a symmetric cipher key.

This function performs single-shot symmetric decryption using the specified cipher algorithm. The input buffer must contain the IV followed by the ciphertext. The IV is extracted internally and used for decryption. The key must permit decryption usage and be compatible with the selected algorithm. The ciphertext portion (input minus IV) must be a non-zero multiple of PSA_MG_AES_BLOCK_LENGTH.

Parameters

- **key** – Identifier of the key to use for decryption. Must be a valid non-zero key identifier with PSA_KEY_USAGE_DECRYPT permission.
- **alg** – Cipher algorithm to use. Must satisfy PSA_ALG_IS_CIPHER(alg) and be one of the supported algorithms: PSA_ALG_CBC_NO_PADDING, PSA_ALG_CTR, PSA_ALG_CFB, or PSA_ALG_OFB.
- **input** – Pointer to the input buffer containing the IV followed by the ciphertext. Must not be NULL.

- **input_length** – Size of the input buffer in bytes. Must be greater than PSA_MG_AES_IV_SIZE and the ciphertext portion must be a non-zero multiple of PSA_MG_AES_BLOCK_LENGTH.
- **output** – Buffer where the decrypted plaintext is written. Must not be NULL.
- **output_size** – Size of the output buffer in bytes. Must be at least input_length - PSA_MG_AES_IV_SIZE bytes.
- **output_length** – On success, set to the number of bytes of plaintext written to the output buffer. Must not be NULL.

Returns

- PSA_SUCCESS on successful decryption.
- PSA_ERROR_INVALID_ARGUMENT if input, output, or output_length is NULL, the key identifier is zero, the input length is not greater than PSA_MG_AES_IV_SIZE, the ciphertext portion is not a multiple of PSA_MG_AES_BLOCK_LENGTH, or the algorithm is not a cipher algorithm.
- PSA_ERROR_NOT_SUPPORTED if the algorithm is not one of the four supported cipher modes.
- PSA_ERROR_BUFFER_TOO_SMALL if output_size is smaller than input_length - PSA_MG_AES_IV_SIZE bytes.
- PSA_ERROR_NOT_PERMITTED if the key does not allow decryption usage.
- PSA_ERROR_INVALID_HANDLE if the key identifier does not refer to a valid key.
- PSA_ERROR_BAD_STATE if the crypto subsystem is not initialized.
- PSA_ERROR_CORRUPTION_DETECTED if an internal integrity check fails.

psa_status_t **psa_cipher_encrypt_setup**(*psa_cipher_operation_t* *operation, *psa_key_id_t* key, *psa_algorithm_t* alg)

Sets up a multipart cipher operation for encryption.

This function initializes a multipart cipher operation context for encryption using the specified key and algorithm. Once set up, the operation must be used in the following sequence: *psa_cipher_set_iv()* → *psa_cipher_update()*

→ *psa_cipher_finish()*.

psa_cipher_update() may be called once with the entire input or multiple times with block-aligned chunks. Each call to *psa_cipher_update()* must provide a non-zero multiple of PSA_MG_AES_BLOCK_LENGTH bytes.

On any failure after setup, call *psa_cipher_abort()* to clean up.

Parameters

- **`operation`** – Pointer to the cipher operation context to initialize. Must not be NULL and must be in a freshly initialized state (PSA_CIPHER_OPERATION_INIT).
- **`key`** – Identifier of the key to use for encryption. Must be a valid non-zero key identifier with PSA_KEY_USAGE_ENCRYPT permission.
- **`alg`** – Cipher algorithm to use. Must satisfy PSA_ALG_IS_CIPHER(alg) and be one of the supported algorithms: PSA_ALG_CBC_NO_PADDING, PSA_ALG_CTR, PSA_ALG_CFB, or PSA_ALG_OFB.

Returns

- PSA_SUCCESS if the operation is successfully set up.
- PSA_ERROR_BAD_STATE if the operation pointer is NULL or the operation has already been set up (alg or iv_set fields are non-zero).
- PSA_ERROR_INVALID_ARGUMENT if the key identifier is zero or the algorithm is not a cipher algorithm.
- PSA_ERROR_NOT_SUPPORTED if the algorithm is not one of the four supported cipher modes.
- PSA_ERROR_NOT_PERMITTED if the key does not allow encryption usage.
- PSA_ERROR_INVALID_HANDLE if the key identifier does not refer to a valid key.
- PSA_ERROR_CORRUPTION_DETECTED if an internal integrity check fails.

psa_status_t **`psa_cipher_decrypt_setup`**(*psa_cipher_operation_t* *operation, *psa_key_id_t* key, *psa_algorithm_t* alg)

Sets up a multipart cipher operation for decryption.

This function initializes a multipart cipher operation context for decryption using the specified key and algorithm. Once set up, the operation must be used in the following sequence: *psa_cipher_set_iv()* → *psa_cipher_update()* → *psa_cipher_finish()*.

psa_cipher_update() may be called once with the entire input or multiple times with block-aligned chunks. Each call to *psa_cipher_update()* must provide a non-zero multiple of PSA_MG_AES_BLOCK_LENGTH bytes.

On any failure after setup, call *psa_cipher_abort()* to clean up.

Parameters

- **`operation`** – Pointer to the cipher operation context to initialize. Must not be NULL and must be in a freshly initialized state (PSA_CIPHER_OPERATION_INIT).
- **`key`** – Identifier of the key to use for decryption. Must be a valid non-zero key identifier with PSA_KEY_USAGE_DECRYPT permission.
- **`alg`** – Cipher algorithm to use. Must satisfy PSA_ALG_IS_CIPHER(alg) and be one of the supported algorithms: PSA_ALG_CBC_NO_PADDING, PSA_ALG_CTR, PSA_ALG_CFB, or PSA_ALG_OFB.

Returns

- PSA_SUCCESS if the operation is successfully set up.
- PSA_ERROR_BAD_STATE if the operation pointer is NULL or the operation has already been set up (alg or iv_set fields are non-zero).
- PSA_ERROR_INVALID_ARGUMENT if the key identifier is zero or the algorithm is not a cipher algorithm.
- PSA_ERROR_NOT_SUPPORTED if the algorithm is not one of the four supported cipher modes.
- PSA_ERROR_NOT_PERMITTED if the key does not allow decryption usage.
- PSA_ERROR_INVALID_HANDLE if the key identifier does not refer to a valid key.
- PSA_ERROR_CORRUPTION_DETECTED if an internal integrity check fails.

psa_status_t **`psa_cipher_set_iv`**(*psa_cipher_operation_t* *operation, const uint8_t *iv, size_t iv_length)

Sets the initialization vector (IV) for a multipart cipher operation.

This function assigns the IV to an initialized multipart cipher operation context. The IV must be provided exactly once after *psa_cipher_encrypt_setup()* or *psa_cipher_decrypt_setup()* and before any call to *psa_cipher_update()*. On invalid input, the cipher operation is aborted automatically to prevent misuse.

Parameters

- **`operation`** – Pointer to an initialized multipart cipher operation context. Must not be NULL, must have been set up via *psa_cipher_encrypt_setup()* or *psa_cipher_decrypt_setup()*, and IV must not have been set already.
- **`iv`** – Pointer to the initialization vector buffer. Must not be NULL.
- **`iv_length`** – Length of the IV in bytes. Must be exactly PSA_MG_AES_IV_SIZE (16 bytes).

Returns

- PSA_SUCCESS if the IV is successfully set.
- PSA_ERROR_BAD_STATE if the operation pointer is NULL, the operation has not been set up (alg field is zero), or the IV has already been set.
- PSA_ERROR_INVALID_ARGUMENT if the IV pointer is NULL or the IV length does not equal PSA_MG_AES_IV_SIZE.

psa_status_t **`psa_cipher_update`**(*psa_cipher_operation_t* *operation, const uint8_t *input, size_t input_length, uint8_t *output, size_t output_size, size_t *output_length)

Processes input data for a multipart cipher operation.

This function encrypts or decrypts input data as part of an ongoing multipart cipher operation. The operation must have been initialized via *psa_cipher_encrypt_setup()* or *psa_cipher_decrypt_setup()* and the IV must be set via *psa_cipher_set_iv()* before calling this function.

This function may be called once with the entire input or multiple times with sequential block-aligned chunks. In both cases each call must provide a non-zero multiple of PSA_MG_AES_BLOCK_LENGTH bytes. Each call produces output of the same length as the input.

On any error, the cipher operation is aborted automatically to maintain a consistent state.

Parameters

- **operation** – Pointer to an active multipart cipher operation context. Must not be NULL, IV must be set, and alg must be non-zero.
- **input** – Pointer to the input data buffer. Must not be NULL.
- **input_length** – Length of the input data in bytes. Must be a non-zero multiple of PSA_MG_AES_BLOCK_LENGTH.
- **output** – Buffer where the processed output data is written. Must not be NULL and must be at least input_length bytes.
- **output_size** – Size of the output buffer in bytes.
- **output_length** – On success, set to the number of bytes written to the output buffer. Always equal to input_length. Must not be NULL.

Returns

- PSA_SUCCESS if the input data is successfully processed.
- PSA_ERROR_BAD_STATE if the operation pointer is NULL, the IV has not been set, or the algorithm field is zero.
- PSA_ERROR_INVALID_ARGUMENT if any pointer is NULL, the input length is zero, or the input length is not a multiple of PSA_MG_AES_BLOCK_LENGTH.
- PSA_ERROR_BUFFER_TOO_SMALL if output_size is smaller than input_length bytes.
- PSA_ERROR_CORRUPTION_DETECTED if the underlying AES operation fails.

psa_status_t **psa_cipher_finish**(*psa_cipher_operation_t* *operation, uint8_t *output, size_t output_size, size_t *output_length)

Completes a multipart cipher operation.

This function completes an ongoing multipart cipher operation after all input data has been processed using *psa_cipher_update()*. It validates the operation state and ensures that no unprocessed data remains. Since padding is not supported by the underlying AES hardware, no additional output is produced during finalization and output_length is always set to 0 on suc-

cess. Once this function returns, regardless of success or failure, the operation context is automatically zeroized via *psa_cipher_abort()* and must not be used for further processing.

Parameters

- **operation** – Pointer to an active cipher operation context. Must not be NULL, IV must be set, and alg must be non-zero.
- **output** – Buffer provided for final output. No data is written to this buffer by this implementation since padding is not supported.
- **output_size** – Size of the output buffer in bytes.
- **output_length** – On success, always set to 0 since no additional output is produced. Must not be NULL.

Returns

- PSA_SUCCESS if the cipher operation is finalized successfully.
- PSA_ERROR_BAD_STATE if the operation pointer is NULL, the IV has not been set, the algorithm field is zero, or there is remaining unprocessed data (unprocessed_len != 0).
- PSA_ERROR_INVALID_ARGUMENT if the output_length pointer is NULL.

psa_status_t **psa_cipher_abort**(*psa_cipher_operation_t* *operation)

Aborts an ongoing cipher operation and securely clears all state.

This function terminates an active multipart cipher operation. It securely zeroizes the entire operation structure using mg_zeroize(), resetting all internal cipher state including key material, IV, and operation flags. After this call the operation context is reset to an unused state and can be safely reused for a new cipher setup or discarded. This function can be called at any point including after an error to ensure proper cleanup.

Parameters

operation – Pointer to the cipher operation context to abort. Must not be NULL.

Returns

- PSA_SUCCESS if the operation was successfully aborted and cleaned up.
- PSA_ERROR_BAD_STATE if the operation pointer is NULL.

psa_status_t **mg_psa_cipher_multirun**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *iv, size_t iv_length, const uint8_t *input, size_t input_length, uint8_t *output, size_t output_size, size_t *output_length, *mg_operation_t* cipher_operation)

Executes a complete multipart cipher operation in a single call.

This function performs a complete AES cipher operation by internally using the multipart cipher flow — mg_cipher_setup_operation(), *psa_cipher_set_iv()*, *psa_cipher_update()*, and *psa_cipher_finish()*. The entire input is processed in fixed-size chunks of PSA_MG_AES_BLOCK_LENGTH bytes using repeated *psa_cipher_update()* calls. All internal resources are automatically cleaned up on both success and failure.

Warning

This function must be called with the entire input in a single call. It does not support incremental invocation — the complete plaintext or ciphertext must be provided at once. For incremental processing use *psa_cipher_encrypt_setup()* / *psa_cipher_decrypt_setup()* with *psa_cipher_update()* directly.

Parameters

- **key** – Identifier of the key to use. Must be a valid non-zero key identifier with appropriate usage permission (PSA_KEY_USAGE_ENCRYPT for encryption, PSA_KEY_USAGE_DECRYPT for decryption).
- **alg** – Cipher algorithm to use. Must be one of the supported algorithms: PSA_ALG_CBC_NO_PADDING, PSA_ALG_CTR, PSA_ALG_CFB, or PSA_ALG_OFB.
- **iv** – Pointer to the initialization vector buffer. Must not be NULL and must be exactly PSA_MG_AES_IV_SIZE bytes.
- **iv_length** – Length of the IV in bytes. Must equal PSA_MG_AES_IV_SIZE.
- **input** – Pointer to the input data buffer. Must not be NULL.
- **input_length** – Length of the input data in bytes. Must be a non-zero multiple of PSA_MG_AES_BLOCK_LENGTH.
- **output** – Buffer where the encrypted or decrypted data is writ-

ten. Must be at least input_length bytes.

- **output_size** – Size of the output buffer in bytes.
- **output_length** – On success, set to the total number of bytes written to the output buffer. Must not be NULL.
- **cipher_operation** – Selects the direction of the operation. Must be MG_PSA_AES_ENCRYPT or MG_PSA_AES_DECRYPT.

Returns

- PSA_SUCCESS if the operation completes successfully.
- PSA_ERROR_INVALID_ARGUMENT if input_length is zero, not a multiple of PSA_MG_AES_BLOCK_LENGTH, or cipher_operation is neither MG_PSA_AES_ENCRYPT nor MG_PSA_AES_DECRYPT.
- PSA_ERROR_NOT_SUPPORTED if the algorithm is not one of the four supported cipher modes.
- PSA_ERROR_NOT_PERMITTED if the key does not allow the requested usage.
- PSA_ERROR_INVALID_HANDLE if the key identifier does not refer to a valid key.
- PSA_ERROR_BAD_STATE if the internal operation state is invalid.
- PSA_ERROR_CORRUPTION_DETECTED if an internal integrity error is detected.

6.3.4 Hash and Message Digest Operations

group **Hash operations**

Functions for cryptographic hash computation.

This group provides APIs for single-shot and multipart hash operations using PSA Crypto compliant interfaces. Supported algorithms currently include SHA-256.

Functions

psa_status_t **psa_hash_compute**(*psa_algorithm_t* alg, const uint8_t *input, size_t input_length, uint8_t *hash, size_t hash_size, size_t *hash_length)

Computes a hash for the given input in a single call.

This function performs a single-shot hash computation using the specified PSA hash algorithm. The entire input buffer is processed in one call and the resulting hash is written to the output buffer. This implementation currently supports only the SHA-256 algorithm.

Note

For the multipart internal path, use *mg_psa_hash_compute_multirun()* instead.

Parameters

- **`alg`** – Hash algorithm to use. Must satisfy PSA_ALG_IS_HASH(alg).
- **`input`** – Pointer to the input data to be hashed.
- **`input_length`** – Size of the input data in bytes. May be 0 for an empty message.
- **`hash`** – Buffer where the computed hash is written.
- **`hash_size`** – Size of the hash output buffer in bytes. Must be at least PSA_MG_SHA256_OUTPUT_LEN.
- **`hash_length`** – On success, number of bytes written to the hash buffer.

Returns

- PSA_SUCCESS if the hash computation is successful.
- PSA_ERROR_INVALID_ARGUMENT if input, hash, or hash_length is NULL, or the algorithm is not a valid hash algorithm.
- PSA_ERROR_NOT_SUPPORTED if the requested hash algorithm is not supported.
- PSA_ERROR_BUFFER_TOO_SMALL if hash_size is less than PSA_MG_SHA256_OUTPUT_LEN.
- PSA_ERROR_CORRUPTION_DETECTED if the hash computation fails internally.

psa_status_t **`psa_hash_compare`**(*psa_algorithm_t* alg, const uint8_t *input, size_t input_length, const uint8_t *hash, size_t hash_length)

Compares a computed hash of the input data with a reference hash.

This function computes the hash of the provided input buffer using the specified hash algorithm and compares it against the given reference hash. The reference hash length must exactly match PSA_MG_SHA256_OUTPUT_LEN. This implementation currently supports only the SHA-256 algorithm.

Parameters

- **alg** – Hash algorithm to use. Must satisfy PSA_ALG_IS_HASH(alg).
- **input** – Pointer to the input data whose hash is to be computed.
- **input_length** – Size of the input data in bytes. May be 0 for an empty message.
- **hash** – Pointer to the reference hash to compare against.
- **hash_length** – Size of the reference hash in bytes. Must be equal to PSA_MG_SHA256_OUTPUT_LEN.

Returns

- PSA_SUCCESS if the computed hash matches the reference hash.
- PSA_ERROR_INVALID_ARGUMENT if input or hash is NULL, the algorithm is not a valid hash algorithm, or hash_length does not equal PSA_MG_SHA256_OUTPUT_LEN.
- PSA_ERROR_NOT_SUPPORTED if the requested hash algorithm is not supported.
- PSA_ERROR_INVALID_SIGNATURE if the computed hash does not match the provided reference hash.
- PSA_ERROR_CORRUPTION_DETECTED if an internal failure occurs during hash computation or if the computed hash length is unexpected.

psa_status_t **psa_hash_setup**(*psa_hash_operation_t* *operation,
psa_algorithm_t alg)

Initializes a hash operation context for multipart hashing.

This function prepares a hash operation context for incremental hashing using the specified algorithm. The operation context must be initialized to PSA_HASH_OPERATION_INIT before calling this function. Once set up, the operation must be followed by exactly one call to *psa_hash_update()* with

the complete input, and then *psa_hash_finish()* to retrieve the result.

Parameters

- **operation** – Pointer to the hash operation object to be initialized. Must not be NULL and must be in a zeroed or freshly initialized state (PSA_HASH_OPERATION_INIT).
- **alg** – Hash algorithm to use for the operation.

Returns

- PSA_SUCCESS if the hash operation context is successfully initialized.
- PSA_ERROR_BAD_STATE if the operation pointer is NULL or the operation has already been set up.
- PSA_ERROR_INVALID_ARGUMENT if the algorithm is not a valid hash algorithm.
- PSA_ERROR_NOT_SUPPORTED if the algorithm is a valid hash algorithm but is not supported by this implementation.

psa_status_t **psa_hash_update**(*psa_hash_operation_t* *operation, const uint8_t *input, size_t input_length)

Provides input data to an ongoing hash operation.

This function processes the input data as part of a multipart hash operation. The operation must have been successfully initialized using *psa_hash_setup()* before calling this function.

Large inputs are handled transparently by processing them in multiple internal chunks. On any failure, the operation is aborted automatically and must be reinitialized before reuse.

Warning

This implementation supports only a single call to *psa_hash_update()* per operation. The entire input must be provided in one call — multiple sequential calls to *psa_hash_update()* on the same operation are not supported and will produce an incorrect hash result. After this call, *psa_hash_finish()* must be called to retrieve the result.

Parameters

- **operation** – Pointer to an active hash operation context initialized via *psa_hash_setup()*.

- **input** – Pointer to the input data to be hashed. Must not be NULL. A zero-length input is permitted.
- **input_length** – Length of the input data in bytes. A value of 0 is permitted and correctly handles the empty message case.

Returns

- PSA_SUCCESS if the input data is successfully processed.
- PSA_ERROR_BAD_STATE if the operation pointer is NULL or the operation has not been properly initialized via *psa_hash_setup()*.
- PSA_ERROR_INVALID_ARGUMENT if the input pointer is NULL.
- PSA_ERROR_CORRUPTION_DETECTED if an internal processing error occurs during hashing.

psa_status_t **psa_hash_finish**(*psa_hash_operation_t* *operation, uint8_t *hash, size_t hash_size, size_t *hash_length)

Finalizes a multipart hash operation and outputs the computed hash.

This function finalizes an ongoing hash operation that was initialized using *psa_hash_setup()* and updated using *psa_hash_update()*. It retrieves the final hash value and writes it to the provided output buffer. After this call, regardless of success or failure, the operation is aborted and must not be used again unless reinitialized via *psa_hash_setup()*.

Parameters

- **operation** – Pointer to an active hash operation context that has been updated via *psa_hash_update()*.
- **hash** – Buffer where the computed hash value is written.
- **hash_size** – Size of the hash output buffer in bytes. Must be at least PSA_MG_SHA256_OUTPUT_LEN.
- **hash_length** – On success, set to the number of bytes written to the hash buffer. This will equal PSA_MG_SHA256_OUTPUT_LEN.

Returns

- PSA_SUCCESS on successful completion of the hash operation.
- PSA_ERROR_BAD_STATE if the operation pointer is NULL or the operation has not been properly initialized.

- PSA_ERROR_INVALID_ARGUMENT if hash or hash_length is NULL.
- PSA_ERROR_BUFFER_TOO_SMALL if hash_size is less than PSA_MG_SHA256_OUTPUT_LEN.
- PSA_ERROR_CORRUPTION_DETECTED if an internal error occurs during hash finalization or if the output length is unexpected.

psa_status_t **psa_hash_abort**(*psa_hash_operation_t* *operation)

Aborts an ongoing hash operation and clears all associated state.

This function terminates a hash operation that was previously initialized using *psa_hash_setup()*. It securely zeroes all internal state associated with the operation and resets the operation object to its initial state. Calling this function on an inactive or already-aborted operation is permitted and results in no error.

Parameters

operation – Pointer to the hash operation structure to abort.

Returns

- PSA_SUCCESS if the hash operation is successfully aborted or was already inactive.
- PSA_ERROR_BAD_STATE if the operation pointer is NULL.

psa_status_t **mg_psa_hash_compute_multirun**(*psa_algorithm_t* alg, const uint8_t *input, size_t input_length, uint8_t *hash, size_t hash_size, size_t *hash_length)

Computes a hash value using the multipart hash flow in a single call.

This function is a convenience wrapper that performs a complete hash computation by internally executing *psa_hash_setup()*, *psa_hash_update()*, and *psa_hash_finish()* in sequence. It uses the multipart internal path, which allows the hash output to be retrieved after processing. Use this function when the multipart internal path is required. For a direct single-shot computation, use *psa_hash_compute()* instead.

Parameters

- **alg** – Hash algorithm to use. Must be a supported hash algorithm (e.g. PSA_ALG_SHA_256).
- **input** – Pointer to the input data to be hashed. Must not be

NULL. A zero-length input is permitted.

- **input_length** – Size of the input data in bytes.
- **hash** – Buffer where the computed hash is written.
- **hash_size** – Size of the hash output buffer in bytes. Must be at least PSA_MG_SHA256_OUTPUT_LEN.
- **hash_length** – On success, set to the number of bytes written to the hash buffer. This will equal PSA_MG_SHA256_OUTPUT_LEN.

Returns

- PSA_SUCCESS if the hash computation completes successfully.
- PSA_ERROR_BAD_STATE if the internal hash operation cannot be initialized or is in an invalid state.
- PSA_ERROR_INVALID_ARGUMENT if the algorithm is not a valid hash algorithm.
- PSA_ERROR_NOT_SUPPORTED if the requested hash algorithm is not supported.
- PSA_ERROR_BUFFER_TOO_SMALL if hash_size is less than PSA_MG_SHA256_OUTPUT_LEN.
- PSA_ERROR_CORRUPTION_DETECTED if an internal error occurs during hash processing.

6.3.5 Signature and RSA Operations

group **RSA operations**

Functions for RSA encryption, decryption, signing, and verification.

This group provides APIs to perform asymmetric cryptography using PSA Crypto compliant interfaces on Secure-IOT devices. Supported operations include:

- RSA message signing and verification
- RSA public-key encryption and private-key decryption

Supported algorithms:

- PKCS#1 v1.5 signature: *PSA_ALG_RSA_PKCS1V15_SIGN_RAW*
- RSA-PSS signature: #PSA_ALG_RSA_PSS_BASE

- PKCS#1 v1.5 encryption: *PSA_ALG_RSA_PKCS1V15_CRYPT*
- RSA-OAEP encryption: #PSA_ALG_RSA_OAEP_SHA256

The APIs handle key slot management, padding/unpadding, and error handling internally. For RSA-OAEP, the salt parameter acts as the label.

Functions

psa_status_t **psa_sign_message**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, uint8_t *signature, size_t signature_size, size_t *signature_length)

Signs a message using a private key.

This function generates a digital signature over the input message using the private key. It supports RSA-based signature algorithms. The output signature is written to the signature buffer.

Parameters

- **key** – Identifier of the key to use for signing.
- **alg** – Signature algorithm to use. Supported values:
  - *PSA_ALG_RSA_PKCS1V15_SIGN_RAW*
  - #PSA_ALG_RSA_PSS_BASE
- **input** – Pointer to the message to sign.
- **input_length** – Length of the message in bytes.
- **signature** – Pointer to buffer where the signature will be written.
- **signature_size** – Size of the signature buffer in bytes.
- **signature_length** – Pointer to variable that will receive the actual signature length.

Returns

- PSA_SUCCESS on successful signature generation.
- PSA_ERROR_INVALID_ARGUMENT if input arguments are invalid or the algorithm is unsupported.
- PSA_ERROR_NOT_PERMITTED if the key does not allow signing.
- PSA_ERROR_BUFFER_TOO_SMALL if signature_size is insufficient.

- PSA_ERROR_CORRUPTION_DETECTED for internal errors.
- PSA_ERROR_INVALID_HANDLE if the key identifier is invalid.

psa_status_t **psa_sign_message_with_salt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, uint8_t *signature, size_t signature_size, size_t *signature_length, const uint8_t *salt, size_t salt_length)

Signs a message using a private key with a user-provided salt.

Similar to *psa_sign_message()*, but allows the caller to specify a salt for PSS signature algorithms. The salt is only used when alg is #PSA_ALG_RSA_PSS_BASE.

Parameters

- **key** – Identifier of the key to use for signing.
- **alg** – Signature algorithm. Supported values:
  - *PSA_ALG_RSA_PKCS1V15_SIGN_RAW*
  - #PSA_ALG_RSA_PSS_BASE
- **input** – Pointer to the message to sign.
- **input_length** – Length of the message in bytes.
- **signature** – Pointer to buffer where the signature will be written.
- **signature_size** – Size of the signature buffer in bytes.
- **signature_length** – Pointer to variable that will receive the actual signature length.
- **salt** – Pointer to user-provided salt. May be NULL if salt_length is 0.
- **salt_length** – Length of the user-provided salt in bytes.

Returns

- PSA_SUCCESS on successful signature generation.
- PSA_ERROR_INVALID_ARGUMENT if input arguments are invalid or the algorithm is unsupported.
- PSA_ERROR_NOT_PERMITTED if the key does not allow signing.
- PSA_ERROR_BUFFER_TOO_SMALL if signature_size is insuf-

ficient.

- PSA_ERROR_CORRUPTION_DETECTED for internal errors.
- PSA_ERROR_INVALID_HANDLE if the key identifier is invalid.

psa_status_t **psa_verify_message**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, const uint8_t *signature, size_t signature_length)

Verifies a digital signature on a message using a public key.

This function verifies that signature matches the hash of input using the public key identified by key. Supported algorithms are RSA-based signature schemes.

Parameters

- **key** – Identifier of the key to use for verification.
- **alg** – Signature algorithm. Supported values:
  - *PSA_ALG_RSA_PKCS1V15_SIGN_RAW*
  - #PSA_ALG_RSA_PSS_BASE
- **input** – Pointer to the message whose signature is being verified.
- **input_length** – Length of the message in bytes.
- **signature** – Pointer to the signature to verify.
- **signature_length** – Length of the signature in bytes.

Returns

- PSA_SUCCESS if the signature is valid.
- PSA_ERROR_INVALID_SIGNATURE if the signature does not match.
- PSA_ERROR_INVALID_ARGUMENT if input arguments are invalid or the algorithm is unsupported.
- PSA_ERROR_NOT_PERMITTED if the key does not allow verification.
- PSA_ERROR_CORRUPTION_DETECTED for internal errors.
- PSA_ERROR_INVALID_HANDLE if the key identifier is invalid.

psa_status_t **psa_verify_message_with_salt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, const uint8_t *signature, size_t signature_length, size_t expected_salt_length)

Verifies a digital signature on a message with an expected salt length.

Similar to *psa_verify_message()*, but allows specifying the expected salt length for PSS signature verification.

Parameters

- **key** – Identifier of the key to use for verification.
- **alg** – Signature algorithm. Supported values:
  - *PSA_ALG_RSA_PKCS1V15_SIGN_RAW*
  - #PSA_ALG_RSA_PSS_BASE
- **input** – Pointer to the message whose signature is being verified.
- **input_length** – Length of the message in bytes.
- **signature** – Pointer to the signature to verify.
- **signature_length** – Length of the signature in bytes.
- **expected_salt_length** – Expected salt length for PSS verification. Must match the salt length used during signing. Pass 32 for the default randomly generated salt.

Returns

- PSA_SUCCESS if the signature is valid.
- PSA_ERROR_INVALID_SIGNATURE if the signature does not match.
- PSA_ERROR_INVALID_ARGUMENT if input arguments are invalid or the algorithm is unsupported.
- PSA_ERROR_NOT_PERMITTED if the key does not allow verification.
- PSA_ERROR_CORRUPTION_DETECTED for internal errors.
- PSA_ERROR_INVALID_HANDLE if the key identifier is invalid.

psa_status_t **psa_asymmetric_encrypt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, const uint8_t *salt, size_t salt_length, uint8_t *output, size_t output_size, size_t *output_length)

Encrypts a message using a public key.

This function performs asymmetric encryption using the key identified by key. For RSA-OAEP, the salt parameter serves as the label.

Parameters

- **key** – Identifier of the key to use for encryption.
- **alg** – Encryption algorithm. Supported values:
  - *PSA_ALG_RSA_PKCS1V15_CRYPT*
  - #PSA_ALG_RSA_OAEP_SHA256
- **input** – Pointer to the plaintext input.
- **input_length** – Length of the plaintext in bytes.
- **salt** – Optional label for OAEP encryption. Ignored for PKCS#1 v1.5. May be NULL if salt_length is 0.
- **salt_length** – Length of the salt/label in bytes.
- **output** – Pointer to buffer where ciphertext will be written.
- **output_size** – Size of the output buffer in bytes.
- **output_length** – Pointer to variable that will receive the ciphertext length.

Returns

- PSA_SUCCESS on successful encryption.
- PSA_ERROR_INVALID_ARGUMENT if input arguments are invalid or the algorithm is unsupported.
- PSA_ERROR_NOT_PERMITTED if the key does not allow encryption.
- PSA_ERROR_BUFFER_TOO_SMALL if output_size is less than MG_RSA_256_BYTES.
- PSA_ERROR_CORRUPTION_DETECTED for internal errors.
- PSA_ERROR_INVALID_HANDLE if the key identifier is invalid.

psa_status_t **psa_asymmetric_decrypt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, const uint8_t *salt, size_t salt_length, uint8_t *output, size_t output_size, size_t *output_length)

Decrypts a message using a private key.

This function performs asymmetric decryption using the key identified by key. For RSA-OAEP, the salt parameter serves as the label.

Parameters

- **key** – Identifier of the key to use for decryption.
- **alg** – Decryption algorithm. Supported values:
  - *PSA_ALG_RSA_PKCS1V15_CRYPT*
  - #PSA_ALG_RSA_OAEP_SHA256
- **input** – Pointer to the ciphertext input.
- **input_length** – Length of the ciphertext in bytes. Must be exactly MG_RSA_256_BYTES.
- **salt** – Optional label for OAEP decryption. Ignored for PKCS#1 v1.5. May be NULL if salt_length is 0.
- **salt_length** – Length of the salt/label in bytes.
- **output** – Pointer to buffer where plaintext will be written.
- **output_size** – Size of the output buffer in bytes.
- **output_length** – Pointer to variable that will receive the plaintext length.

Returns

- PSA_SUCCESS on successful decryption.
- PSA_ERROR_INVALID_ARGUMENT if input arguments are invalid, the algorithm is unsupported, or input_length is not MG_RSA_256_BYTES.
- PSA_ERROR_NOT_PERMITTED if the key does not allow decryption.
- PSA_ERROR_BUFFER_TOO_SMALL if output_size is insufficient.
- PSA_ERROR_CORRUPTION_DETECTED for internal errors.
- PSA_ERROR_INVALID_HANDLE if the key identifier is invalid.

Chapter 7. Examples

This section describes how to use the terminal to build and run applications. The below are the two ways using which we can run the applications on the Secure IoT SoC using OCSRAM.

7.1 Bare Metal Examples

This section describes how to use the terminal to build and run applications. The below are the two ways using which we can run the applications on the Secure IoT SoC using OCSRAM.

7.1.1 Delays

Functions Usage

1. **Delay Function (Milliseconds)**

This function uses the *mcycle* to perform a wait loop that delays execution for the requested duration in milliseconds.

Example Usage

To create a delay of 500 milliseconds, call the delay function as shown below.

```
#include<io.h>
#include"log.h"
#include "delays.h"


int main()
{
    delays_ms(500);
    return 0;
}
```

2. **Delay Function (Microseconds)**

This function uses the *mcycle* to perform a wait loop that delays execution for the requested duration in microseconds.

Example Usage

To create a delay of 500 microseconds, call the delay function as shown below.

```
#include<io.h>
#include "log.h"
#include "delays.h"


int main()
{
    delays_us(500);
    return 0;
}
```

7.1.2 Performance monitors

Performance test for matrix multiplication

The following example demonstrates the performance test for matrix multiplication.

Initialization and Configuration

1. **Initialization**

   To initialization the cache, stalls, branches and Arithmetic operation.

   ```
   PERF_cache_init();
   PERF_Stalls_Init();
   PERF_Branches_Init();
   PERF_Arithops_Init();
   ```

2. **Disable the perf events**

   Before printing the performance events, Need to disable the performance event registers.

```
PERF_Disable_All();
```

3. **Print the events**

   To print the cache, cache miss percentage, stalls, branches, arithmetic operation.

```
PERF_Print_Cache(1);
PERF_Print_Cache_MissPercentage();
PERF_Print_Stalls(1);
PERF_Print_Branches(1);
PERF_Print_Arithops(1);
```

Example Usage

To check the performance test for matrix multiplication.

```
#include "io.h"
#include "perf_monitors.h"

#define R1 5 // number of rows in Matrix-1
#define C1 3 // number of columns in Matrix-1
#define R2 3 // number of rows in Matrix-2
#define C2 3 // number of columns in Matrix-2

void Matrix_Multiply(int m1[][C1], int m2[][C2])
{
   int result[R1][C2];

   printf("Resultant Matrix is:\n");

   for (int i = 0; i < R1; i++) {
      for (int j = 0; j < C2; j++) {
         result[i][j] = 0;

         for (int k = 0; k < R2; k++) {
            result[i][j] += m1[i][k] * m2[k][j];
         }

         printf("%d\t", result[i][j]);
```

(continues on next page)

(continued from previous page)

```
      }

      printf("\n");
   }
   printf("\n");
}

int main()
{
   PERF_cache_init();  // Initialize the counters and events for cache
   PERF_Stalls_Init();  // Initialize the counters and events for raw␣
↪and execution stalls
   PERF_Branches_Init();  // Initialize the counters for branch
   PERF_Arithops_Init();  // Initialize the counters for arithmetic␣
↪operation

   int m1[R1][C1] = { { 1, 1, 1}, { 2, 2, 2 }, { 3, 3, 3 }, { 4, 4, 4 }
↪, { 5, 5, 5 } };

   int m2[R2][C2] = { { 1, 1, 1 }, { 2, 2, 2 } , { 3, 3, 3} };

   // if coloumn of m1 not equal to rows of m2
   if (C1 != R2) {
      printf("The number of columns in Matrix-1 must be "
         "equal to the number of rows in "
         "Matrix-2\n");
      printf("Please update MACROs value according to "
         "your array dimension in "
         "#define section\n");
   }

   Matrix_Multiply(m1, m2);

   PERF_Disable_All(); // Disables all event registers

   PERF_Print_Cache(1); // prints the cache based on iterations
   PERF_Print_Cache_MissPercentage(); // prints the cache miss␣
```

(continues on next page)

(continued from previous page)

```
↪percentages
   PERF_Print_Stalls(1); // prints the number of raw and execution␣
↪stalls based on iterations
   PERF_Print_Branches(1); // prints the branches based on iterations
   PERF_Print_Arithops(1); // prints the arithmetic operations based␣
↪on iterations

   return 0;
}
```

Output:

```
Fill buffer hit(ICache):  471
Fill buffer release(ICache):      25
Number of Cache Misses(ICache):   497
Non-cached Memory fetches(ICache):        0
Number of instruction requests to ICache:         1576118
Number of Read access(DCache):    175981
Number of Write access(DCache):   1831
Number of Atomic access(DCache):  0
Number of Non-Cachable Read access(DCache):       0
Number of Non-Cachable Write access(DCache):      0
Number of Cache misses for Read operations(DCache):       174692
Number of Cache misses for Write operations(DCache):      1488
Number of Cache misses for Atomic operations(DCache):     0
Fill Buffer Hits for Reads(DCache):       174649
Fill Buffer Hits for Writes(DCache):      1098
Fill Buffer Hits for Atomic operations(DCache):   0
Fill Buffer Releases(DCache):     738
Number of cache lines evicted from DCache:        0
Instruction Translation Lookaside buffer misses:  0
Data Translation Lookaside buffer misses:         0
Miss percentage(ICache):  0.0315
Read Miss percentage(DCache):     99.2675
Write Miss percentage(DCache):    81.2671
Number of Rawstalls:      5221123
Number of Exestalls:      2343
Number of Branch Misprediction:   330
```

(continues on next page)

(continued from previous page)

```
Number of Jumps:  530
Number of Branches:       174523
precentage of Misprediction:      0.1891
Number of Floats:         0
Number of Muldiv:         116
```

Calculating mcycles

The following example demonstrates how to calculate number of cycles taken.

Initialization

1. **Initialization**

   To initialize the mcycle to clear the mcycle registers.

   ```
   PERF_Mcycle_Init();
   ```

2. **Getting Total mcycles**

   To get the total cycle,see the following example.

   ```
   // Code implementation
   uint64_t x = PERF_Get_Mcycle();
   uint64_t y = PERF_Get_Mcycle();
   uint64_t z = y - x;
   ```

7.1.3 Utilities

Functions Usage

1. **Reading data at a given address**

   ```
   uint64_t *addr=(uint64_t *)0x11300;
   ReadData(addr);
   ```

2. **Writing data at a given address**

```
uint64_t *addr=(uint64_t *)0x11300;
unsigned long val=8;
WriteData(addr,val);
```

3. **Example Usage for exit()**

```
int main() {
 // Program logic here

 if (error_condition) {
     exit(1);  // Exit with error status
 }

 exit(0);  // Normal termination
}
```

Example Usage for millis

This example code shows how to use *millis()* ` to calculate the milliseconds taken.

Sample Application for testing millis and delays.

SPDX-License-Identifier: GPL-3.0-or-later

Copyright

Copyright (c) 2021-2026 Mindgrove Technologies. All rights reserved.

License

Licensed under the GNU General Public License v3.0 or later (see LICENSE)

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see https://www.gnu.org/licenses/.

This application demonstrates basic millis functionality and delays functionality.

Authors

Deeptha G (deeptha@mindgrovetech.in)

Date

12-02-2026

Functions

void **main**()

Entry point for millis test application.

Initializes the millis timer, measures elapsed time after a delay, and prints the measured difference.

```
#include<io.h>
#include"log.h"
#include "delays.h"
#include "utils.h"


void main()
{
    millis_init();
    uint64_t a=millis();
    delays_ms(100);
    uint64_t b=millis();
    printf("\n The number of milliseconds is:%d",b-a);

}
```

Output

```
The number of milliseconds is:100
```

Troubleshooting

1. Ensure that you have called *millis_init()* before using *millis()*.
2. Ensure that you have included utils.h before calling *millis()*.

7.1.4 ADC (Analog-to-Digital Converter)

Functions Usage

1. **Initializing ADC**

   To initialize the ADC module and configure its resolution and channels, follow the steps below.

```
ADC_Config_t ADC1;

ADC1.channel     = ADC_CHANNEL_0;     /* Select ADC input channel␣
↪(ADC_CHANNEL_0-ADC_CHANNEL_x) */
ADC1.resolution  = ADC_RES_12BIT;     /* ADC conversion resolution␣
↪(6/8/10/12-bit) */
ADC1.is_intr_en  = ADC_INTR_DISABLE;  /* Enable or disable ADC␣
↪interrupt */
ADC1.is_freerun  = ADC_FREERUN;       /* Enable or disable free-
↪running conversion mode */
ADC1.is_vref_int = ADC_INT_VREF;      /* Enable or disable␣
↪internal reference voltage */
ADC1.is_vbg_int  = ADC_INT_VBG;       /* Enable or disable␣
↪internal bandgap reference */

ADC_CCR(&ADC1);                      /* Apply configuration to ADC␣
↪*/
```

2. **ADC_ReadOutput**

   Waits for the ADC conversion to complete and reads the conversion result.The function extracts and returns the ADC output aligned according to the configured resolution.

```
ADC_Read_Output(&ADC1, &output);
```

ADC Sample Application

This example demonstrates how to initialize and use an ADC to read an analog signal, convert it to digital, and display the result.

```
#include "adc.h"
#include "io.h"

void main()
{
   /*Example application for ADC with interrupt disabled and in
   freerun mode for 12 bit resolution*/
   uint16_t freerun_samp = 10;
   uint16_t output;
   ADC_Config_t ADC1;

   ADC1.channel = ADC_CHANNEL_0;
   ADC1.resolution = ADC_RES_12BIT;
   ADC1.is_intr_en = ADC_INTR_DISABLE;
   ADC1.is_freerun = ADC_FREERUN;
   ADC1.is_vref_int = ADC_INT_VREF;
   ADC1.is_vbg_int = ADC_INT_VBG;

   ADC_CCR(&ADC1);
   printf("----ADC in freerun mode for 12 bit resolution----\n");

   while (freerun_samp--)
   {
      ADC_Read_Output(&ADC1, &output);
      printf("output:%d\n\r", output);
   }
   ADC_Disable();
}
```

Output

```
----ADC in freerun mode for 12 bit resolution----
output:3390
output:3390
output:3390
output:3390
output:3390
```

(continues on next page)

(continued from previous page)

```
output:3390
output:3390
output:3390
output:3390
output:3390
```

7.1.5 Core Local Interrupt (CLINT)

Example Code

```
#include "traps.h" /*Included to access the trap*/
#include "clint_driver.h" /*Included to access CLINT peripheral API's*/

void main(void)
{
  volatile uint64_t* msip = (volatile uint64_t*)0x02000000; // Assign␣
↪CLINT base address
  volatile uint64_t* mtimecmp = (volatile uint64_t*)0x02004000; //␣
↪Assign MTIMECMP address to a variable
  uint64_t value = 4000; // Delta value to add with the `mtime`

  CLINT_Timer(value); // Configure the value to set up the clint timer
}
```

Output

```
Timer Interrupt Handled!
```

Troubleshooting

1. Ensure that, the *mtimecmp* value given is greater than *mtime*.
2. Check if the MSIP and MTIMECMP addresses are correct.

7.1.6 DMA (Direct Memory Access)

Functions Usage

DMA allows high-speed data transfer without CPU intervention. It is commonly used for:

- Memory-to-memory transfers
- Peripheral-to-memory or memory-to-peripheral transfers
- Peripheral-to-peripheral transfers
- Offloading repetitive copy/move operations from the CPU
- Improving system performance in data-intensive applications

1. **Configuring DMA Transfer**

   To configure a DMA transfer, specify:

   - Channel number
   - Source address
   - Destination address
   - Number of bytes to move
   - Priority level
   - Source and destination data width

   Example Code:

```
uint32_t src_buffer[10] = {
    0xAAAAAAAA, 0xBBBBBBBB, 0xCCCCCCCC, 0xDDDDDDDD,
    0xEEEEEEEE, 0xFFFFFFFF, 0x11111111, 0x22222222,
    0x33333333, 0x44444444
};

uint32_t dest_buffer[10] = {0};

DMA_Config_t dma_config;

dma_config.chn_no          = DMA_CHANNEL_0;
dma_config.src_addr        = (uint32_t *)src_buffer;
dma_config.dest_addr       = (uint32_t *)dest_buffer;
dma_config.src_data_size   = DMA_FOURBYTE;
```

(continues on next page)

(continued from previous page)

```
dma_config.dest_data_size  = DMA_FOURBYTE;
dma_config.transfer_length = 40U;  /* 10 x 4 bytes */
dma_config.priority        = DMA_PRIORITY_HIGH;

if (DMA_Transfer_Configure(&dma_config) != SUCCESS)
{
    /* Handle error */
}
```

2. **Checking DMA Transfer Status**

   DMA status can be checked to determine:

   - Whether a transfer has finished
   - Whether an error occurred
   - Whether a half-transfer event happened

   Example Code:

```
uint8_t status = 0U;

if (DMA_Interrupt_Status(&dma_config, &status) == SUCCESS)
{
    if ((status & 0x07U) == 0x07U)
    {
        /* Transfer complete */
    }
}
```

3. **Clearing DMA Interrupt Flags**

   After checking status, clear the interrupt flags:

   Example Code:

```
DMA_Clear_Interrupt_Flags(&dma_config,
                          true,   /* Transfer error */
                          true,   /* Half transfer */
                          true,   /* Transfer complete */
                          true);  /* Global flag */
```

4. **Enabling / Disabling a DMA Channel**

   A DMA channel should be enabled to start transfer and disabled after completion:

Example Code:

```
/* Enable DMA */
DMA_Channel_Set_State(&dma_config, true);

/* Disable DMA */
DMA_Channel_Set_State(&dma_config, false);
```

DMA Sample Application (M2M)

The following example demonstrates configuring a DMA channel for memory-to-memory (M2M) transfer using the DMA driver APIs. The example initializes a source buffer with data, transfers it to a destination buffer using DMA, and verifies the transferred data.

```
#include "io.h"
#include "dma.h"

#define BUFFER_SIZE 10

void main(void)
{
    uint8_t  status = 0U;
    uint16_t ret    = 0U;

    /* Source and destination buffers */
    static uint8_t src_buffer[BUFFER_SIZE];
    static uint8_t dest_buffer[BUFFER_SIZE];

    /* Initialize source buffer */
    for (uint32_t i = 0U; i < BUFFER_SIZE; i++)
    {
        src_buffer[i] = (uint8_t)(i + 1U);
    }

    /* Clear destination buffer */
    for (uint32_t i = 0U; i < BUFFER_SIZE; i++)
    {
```

(continues on next page)

(continued from previous page)

```
    dest_buffer[i] = 0U;
}

/* ================ DMA CONFIG ================ */

DMA_Config_t dma_config;

dma_config.chn_no           = DMA_CHANNEL_3;
dma_config.src_addr         = (uint32_t *)src_buffer;
dma_config.dest_addr        = (uint32_t *)dest_buffer;
dma_config.src_data_size    = DMA_TWOBYTE;
dma_config.dest_data_size   = DMA_TWOBYTE;
dma_config.transfer_length  = BUFFER_SIZE;
dma_config.priority         = DMA_PRIORITY_HIGH;

ret = DMA_Transfer_Configure(&dma_config);
if (ret != SUCCESS)
{
    printf("DMA config failed: %u\r\n", ret);
    return;
}

/* Ensure instruction/data synchronization */
asm volatile ("fence.i");

/* Enable DMA channel */
ret = DMA_Channel_Set_State(&dma_config, true);
if (ret != SUCCESS)
{
    printf("DMA enable failed: %u\r\n", ret);
    return;
}

/* ================ WAIT FOR COMPLETION ================ */

while (1)
{
```

(continues on next page)

(continued from previous page)

```
        ret = DMA_Interrupt_Status(&dma_config, &status);
        if (ret != SUCCESS)
        {
            printf("DMA status failed: %u\r\n", ret);
            return;
        }

        if ((status & 0x07U) == 0x07U)
        {
            break;
        }
    }

    /* ================= STOP DMA ================= */

    ret = DMA_Channel_Set_State(&dma_config, false);
    if (ret != SUCCESS)
    {
        printf("DMA disable failed: %u\r\n", ret);
        return;
    }

    ret = DMA_Clear_Interrupt_Flags(&dma_config, 1U, 1U, 1U, 1U);
    if (ret != SUCCESS)
    {
        printf("DMA clear flags failed: %u\r\n", ret);
        return;
    }

    /* ================= VERIFY DATA ================= */

    for (uint32_t i = 0U; i < BUFFER_SIZE; i++)
    {
        if (src_buffer[i] != dest_buffer[i])
        {
            printf("Transfer Error at index %lu\r\n", i);
            while (1);
```

(continues on next page)

(continued from previous page)

```
        }
        else
        {
            printf("DATA[%lu] = %u\r\n", i, dest_buffer[i]);
        }
    }

    printf("DMA M2M Transfer Successful\r\n");
}
```

Output

```
DATA[0] = 1
DATA[1] = 2
DATA[2] = 3
DATA[3] = 4
DATA[4] = 5
DATA[5] = 6
DATA[6] = 7
DATA[7] = 8
DATA[8] = 9
DATA[9] = 10
DMA M2M Transfer Successful
```

DMA Sample Application (M2P - UART Transmission)

The following example demonstrates a memory-to-peripheral (M2P) DMA transfer where data from a memory buffer is transmitted over UART using DMA. A loopback connection is used to verify the transmitted data.

```
#include "io.h"
#include "uart.h"
#include "string_functions.h"
#include "dma.h"

void main(void)
```

(continues on next page)

(continued from previous page)

```
{
    /* ================= REQUIRED HARDWARE CONNECTIONS␣
↪=================

    UART2 (Loopback for Verification)
    -------------------------------
    TX    ------------------> RX


  ␣
↪=====================================================================␣
↪*/

    uint8_t  status = 0U;
    uint32_t length;

    /* ================= UART CONFIGURATION ================= */

    UART_Config_t uart_config;

    uart_config.uart_num      = UART2;
    uart_config.baudrate      = 115200U;
    uart_config.char_size     = CHAR_SIZE_8;
    uart_config.delay         = 0U;
    uart_config.parity        = NO_PARITY;
    uart_config.stop_bits     = STOP_BIT_1;
    uart_config.transfer_mode = DATA_SIZE_8;
    uart_config.receive_mode  = DATA_SIZE_8;
    uart_config.pullup        = 0U;

    if (UART_Init(&uart_config) != SUCCESS)
    {
        printf("UART init failed\r\n");
        while (1);
    }

    /* ================= SOURCE DATA ================= */
```

(continues on next page)

(continued from previous page)

```
    static char tx_buffer[] = "MINDGROVE FACTS";
    length = (uint32_t)StrLen(tx_buffer);

    /* ================ DMA CONFIGURATION ================ */

    DMA_Config_t dma_config;

    dma_config.chn_no          = DMA_CHANNEL_0;
    dma_config.src_addr        = (uint32_t *)tx_buffer;
    dma_config.dest_addr       = (uint32_t *)UART2_TX_REG_ADDR;
    dma_config.src_data_size   = DMA_BYTE;
    dma_config.dest_data_size  = DMA_BYTE;
    dma_config.transfer_length = length;
    dma_config.priority        = DMA_PRIORITY_HIGH;

    if (DMA_Transfer_Configure(&dma_config) != SUCCESS)
    {
        printf("DMA configuration failed\r\n");
        while (1);
    }

    /* Ensure memory consistency before DMA starts */
    __asm__ volatile ("fence");

    /* Enable DMA */
    if (DMA_Channel_Set_State(&dma_config, true) != SUCCESS)
    {
        printf("DMA enable failed\r\n");
        while (1);
    }

    /* ================ WAIT FOR COMPLETION ================ */

    while (1)
    {
        if (DMA_Interrupt_Status(&dma_config, &status) != SUCCESS)
        {
```

(continues on next page)

(continued from previous page)

```
            printf("DMA status read failed\r\n");
            while (1);
        }

        if ((status & 0x07U) == 0x07U)
        {
            if (DMA_Channel_Set_State(&dma_config, false) != SUCCESS)
            {
                printf("DMA disable failed\r\n");
                while (1);
            }

            if (DMA_Clear_Interrupt_Flags(&dma_config,
                                          true, true, true, true) !=␣
↪SUCCESS)
            {
                printf("DMA flag clear failed\r\n");
                while (1);
            }

            break;
        }
    }

    /* ================= RECEIVE FOR VERIFICATION ================= */

    char rx_buffer[20] = {0};

    struct uart_buf rx =
    {
        .uart_data = rx_buffer,
        .len       = length
    };

    if (UART_Read(&uart_config, &rx, 1000000U) != SUCCESS)
    {
        printf("UART read failed\r\n");
```

(continues on next page)

(continued from previous page)

```
        while (1);
    }

    /* ================= OUTPUT RESULT ================= */

    printf("---- DMA M2P Transfer Successful ----\r\n");
    printf("\r\n**%s**\r\n", rx_buffer);
}
```

Output

```
---- DMA M2P Transfer Successful ----

**MINDGROVE FACTS**
```

DMA Sample Application (P2M - UART Receive to Memory)

The following example demonstrates a peripheral-to-memory (P2M) DMA transfer where data received from a UART peripheral is stored into a memory location using DMA. A loopback connection is used to generate input data.

```
#include "io.h"
#include "uart.h"
#include "string_functions.h"
#include "dma.h"

void main(void)
{
    /* ================= REQUIRED HARDWARE CONNECTIONS␣
↪================

    UART2 (Loopback for Verification)
    -------------------------------
    TX    ------------------> RX


  ␣
```

(continues on next page)

(continued from previous page)

```
↪=================================================================
↪*/

    uint8_t  status = 0U;
    uint32_t length;

    /* ================= SOURCE DATA (FOR LOOPBACK) ================
↪*/

    char tx_buffer[16] = "MINDGROVE FACTS";
    length = (uint32_t)StrLen(tx_buffer);

    struct uart_buf tx =
    {
        .uart_data = tx_buffer,
        .len       = length
    };

    /* ================ UART CONFIGURATION ================ */

    UART_Config_t uart_config;

    uart_config.uart_num      = UART2;
    uart_config.baudrate      = 115200U;
    uart_config.char_size     = CHAR_SIZE_8;
    uart_config.delay         = 0U;
    uart_config.parity        = NO_PARITY;
    uart_config.stop_bits     = STOP_BIT_1;
    uart_config.transfer_mode = DATA_SIZE_8;
    uart_config.receive_mode  = DATA_SIZE_8;
    uart_config.pullup        = 0U;

    if (UART_Init(&uart_config) != SUCCESS)
    {
        printf("UART init failed\r\n");
        while (1);
    }
```

(continues on next page)

(continued from previous page)

```
    /* Send data to generate RX activity (loopback) */
    if (UART_Write(&uart_config, &tx) != SUCCESS)
    {
        printf("UART write failed\r\n");
        while (1);
    }

    if (UART_Write_Wait(&uart_config) != SUCCESS)
    {
        printf("UART write wait failed\r\n");
        while (1);
    }

    /* ================= DMA CONFIGURATION ================= */

    DMA_Config_t dma_config;

    dma_config.chn_no          = DMA_CHANNEL_0;
    dma_config.src_addr        = (uint32_t *)UART2_RX_REG_ADDR;    /*␣
↪Peripheral (UART RX) */
    dma_config.dest_addr       = (uint32_t *)0x80010000U;          /*␣
↪Memory */
    dma_config.src_data_size   = DMA_BYTE;
    dma_config.dest_data_size  = DMA_BYTE;
    dma_config.transfer_length = 15U;
    dma_config.priority        = DMA_PRIORITY_HIGH;

    if (DMA_Transfer_Configure(&dma_config) != SUCCESS)
    {
        printf("DMA configuration failed\r\n");
        while (1);
    }

    /* Ensure instruction/data synchronization */
    __asm volatile ("fence.i");
```

(continues on next page)

(continued from previous page)

```
    /* Enable DMA */
    if (DMA_Channel_Set_State(&dma_config, true) != SUCCESS)
    {
        printf("DMA enable failed\r\n");
        while (1);
    }

    /* ================= WAIT FOR COMPLETION ================= */

    while (1)
    {
        if (DMA_Interrupt_Status(&dma_config, &status) != SUCCESS)
        {
            printf("DMA status read failed\r\n");
            while (1);
        }

        if ((status & 0x07U) == 0x07U)
        {
            if (DMA_Channel_Set_State(&dma_config, false) != SUCCESS)
            {
                printf("DMA disable failed\r\n");
                while (1);
            }

            if (DMA_Clear_Interrupt_Flags(&dma_config,
                                          true, true, true, true) !=␣
↪SUCCESS)
            {
                printf("DMA flag clear failed\r\n");
                while (1);
            }

            break;
        }
    }
```

(continues on next page)

(continued from previous page)

```
    /* ================ OUTPUT RESULT ================ */

    printf("---- DMA P2M Transfer Successful ----\r\n");
    printf("\r\nData at 0x80010000: ");

    uint8_t *data = (uint8_t *)0x80010000U;

    for (uint32_t i = 0U; i < 15U; i++)
    {
        printf("%c", data[i]);
    }
}
```

Output

```
---- DMA P2M Transfer Successful ----

Data at 0x80010000: MINDGROVE FACTS
```

DMA Sample Application (P2P - UART to UART Transfer)

The following example demonstrates a peripheral-to-peripheral (P2P) DMA transfer where data received from UART2 is directly transferred to UART1 using DMA. The received data is then read from UART1 to verify successful transfer.

```
#include "io.h"
#include "uart.h"
#include "string_functions.h"
#include "dma.h"

void main(void)
{
    /* ================ REQUIRED HARDWARE CONNECTIONS␣
↪================

    UART2 (Loopback)
```

(continues on next page)

(continued from previous page)

```
    -------------------------------
    TX    ------------------> RX

    UART1 (Loopback)
    -------------------------------
    TX    ------------------> RX


  ␣
↪=============================================================================␣
↪*/

    uint8_t  status = 0U;
    uint32_t length;

    /* ================ SOURCE DATA ================ */

    char tx_buffer[16] = "MINDGROVE FACTS";
    length = (uint32_t)StrLen(tx_buffer);

    struct uart_buf tx =
    {
        .uart_data = tx_buffer,
        .len       = length
    };

    /* ================ UART2 CONFIGURATION (TX SOURCE)␣
↪================ */

    UART_Config_t uart_tx;

    uart_tx.uart_num      = UART2;
    uart_tx.baudrate      = 115200U;
    uart_tx.char_size     = CHAR_SIZE_8;
    uart_tx.delay         = 0U;
    uart_tx.parity        = NO_PARITY;
    uart_tx.stop_bits     = STOP_BIT_1;
    uart_tx.transfer_mode = DATA_SIZE_8;
```

(continues on next page)

(continued from previous page)

```
    uart_tx.receive_mode  = DATA_SIZE_8;
    uart_tx.pullup        = 0U;

    if (UART_Init(&uart_tx) != SUCCESS)
    {
        printf("UART2 init failed\r\n");
        while (1);
    }

    /* ================= UART1 CONFIGURATION (DESTINATION)␣
↪================= */

    UART_Config_t uart_rx;

    uart_rx.uart_num      = UART1;
    uart_rx.baudrate      = 115200U;
    uart_rx.char_size     = CHAR_SIZE_8;
    uart_rx.delay         = 0U;
    uart_rx.parity        = NO_PARITY;
    uart_rx.stop_bits     = STOP_BIT_1;
    uart_rx.transfer_mode = DATA_SIZE_8;
    uart_rx.receive_mode  = DATA_SIZE_8;
    uart_rx.pullup        = 0U;

    if (UART_Init(&uart_rx) != SUCCESS)
    {
        printf("UART1 init failed\r\n");
        while (1);
    }

    /* ================= INITIAL DATA GENERATION ================= */

    if (UART_Write(&uart_tx, &tx) != SUCCESS)
    {
        printf("UART write failed\r\n");
        while (1);
    }
```

(continues on next page)

(continued from previous page)

```
    if (UART_Write_Wait(&uart_tx) != SUCCESS)
    {
        printf("UART write wait failed\r\n");
        while (1);
    }

    /* ================= DMA CONFIGURATION (P2P) ================= */

    DMA_Config_t dma_config;

    dma_config.chn_no          = DMA_CHANNEL_0;
    dma_config.src_addr        = (uint32_t *)UART2_RX_REG_ADDR; /*␣
↪Source: UART2 RX */
    dma_config.dest_addr       = (uint32_t *)UART1_TX_REG_ADDR; /*␣
↪Destination: UART1 TX */
    dma_config.src_data_size   = DMA_BYTE;
    dma_config.dest_data_size  = DMA_BYTE;
    dma_config.transfer_length = 15U;
    dma_config.priority        = DMA_PRIORITY_HIGH;

    if (DMA_Transfer_Configure(&dma_config) != SUCCESS)
    {
        printf("DMA config failed\r\n");
        while (1);
    }

    __asm__ volatile ("fence.i");

    if (DMA_Channel_Set_State(&dma_config, true) != SUCCESS)
    {
        printf("DMA enable failed\r\n");
        while (1);
    }

    /* ================= WAIT FOR COMPLETION ================= */
```

(continues on next page)

(continued from previous page)

```
    while (1)
    {
        if (DMA_Interrupt_Status(&dma_config, &status) != SUCCESS)
        {
            printf("DMA status read failed\r\n");
            while (1);
        }

        if ((status & 0x07U) == 0x07U)
        {
            if (DMA_Channel_Set_State(&dma_config, false) != SUCCESS)
            {
                printf("DMA disable failed\r\n");
                while (1);
            }

            if (DMA_Clear_Interrupt_Flags(&dma_config,
                                          true, true, true, true) !=␣
↪SUCCESS)
            {
                printf("DMA clear flags failed\r\n");
                while (1);
            }

            break;
        }
    }

    printf("P2P Transfer Complete\r\n");

    /* ================= VERIFY USING UART1 ================= */

    char rx_buffer[16] = {0};

    struct uart_buf rx =
    {
        .uart_data = rx_buffer,
```

(continues on next page)

(continued from previous page)

```
        .len       = 15U
    };

    if (UART_Read(&uart_rx, &rx, 1000000U) != SUCCESS)
    {
        printf("UART read failed\r\n");
        while (1);
    }

    printf("\r\n**%s**\r\n", rx_buffer);
}
```

Output

```
P2P Transfer Complete

**MINDGROVE FACTS**
```

Troubleshooting

1. **Transfer Does Not Start**
   - Verify channel number is valid.
   - Ensure DMA request source is correctly selected (if applicable).
   - Confirm `DMA_Transfer_Configure` was called before polling status.
2. **Incorrect Data in Destination Buffer**
   - Validate source and destination addresses.
   - Ensure data width (BYTE, TWOBYTE, FOURBYTE, EIGHTBYTE) matches buffer type.
   - Check for buffer overlap.
3. **Transfer Error Flag is Set**
   - Confirm addresses point to accessible memory or peripheral regions.
   - Ensure peripheral trigger source is correctly configured (if used).
4. **Transfer Never Completes**

- Verify interrupts and status bits.
- Ensure source peripheral is generating DMA requests (if P2M or M2P mode).
- Double-check size and alignment constraints.

5. **No Output Observed but Transfer is complete**
   - Make sure to use `fence` before configuring the DMA transfer, as it will flush the cache and then begin the transfer.

7.1.7 GPIO (General Purpose Input and Output)

GPIO APIs provide a interface for configuring and controlling GPIO pins in the system. The typical GPIO usage flow consists of configuring the pin direction, followed by read or write operations depending on the configured mode.

1. **GPIO Configuration**

Before using any GPIO pin, it must be properly configured as either an input or an output. GPIO configuration is performed using the *GPIO_Config()* API, which sets the pin direction and performs the required pin multiplexing validation.

```
GPIO_Config(GPIO_OUT, GPIO_PINS(2) | GPIO_PINS(4) | GPIO_PINS(6));
```

In the example above, GPIO pins 2, 4, and 6 are configured as output pins.

2. **Writing GPIO Data**

Once a GPIO pin is configured as an output, its logic level can be controlled using by the following functions are provided:

- *GPIO_Pin_Set()* — Sets the specified GPIO pins to logic HIGH
- *GPIO_Pin_Clear()* — Clears the specified GPIO pins to logic LOW

```
/* Set GPIO pins 2, 4, and 6 to HIGH */
GPIO_Pin_Set(GPIO_PINS(2) | GPIO_PINS(4) | GPIO_PINS(6));

/* Set GPIO pins 2, 4, and 6 to LOW */
GPIO_Pin_Clear(GPIO_PINS(2) | GPIO_PINS(4) | GPIO_PINS(6));
```

These operations affect only the specified pins and do not modify the state of other GPIO pins.

3. **Reading GPIO Data**

GPIO pins configured as inputs can be sampled using the GPIO read API. The current logic level of the pin is returned as a software-readable value.

```
/* Configure GPIO pin 2 as input */
GPIO_Config(GPIO_IN, GPIO_PINS(2));

/* Read the current status of GPIO pin 2 */
int value = GPIO_Read_Pin_Status(GPIO_PINS(2));
```

The returned value indicates the current logic level present on the GPIO pin (typically 0 for LOW and 1 for HIGH).

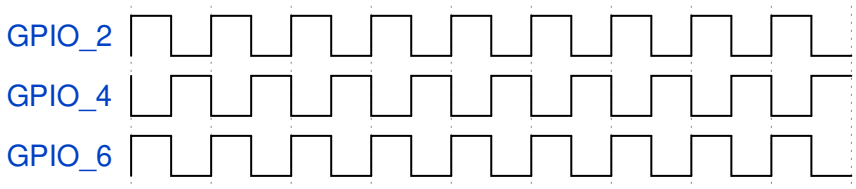
GPIO Example:Blink LED

This example explores how to blink an LED using GPIO.

```
#include <stdio.h>
#include <stdlib.h>
#include "gpio.h"

int main()
{
   GPIO_Config(GPIO_OUT, GPIO_PINS(2) | GPIO_PINS(4) | GPIO_PINS(6));

   while (1)
   {
      GPIO_Pin_Toggle(GPIO_PINS(2) | GPIO_PINS(4) | GPIO_PINS(6));
   }
   return 0;
}
```

Corresponding GPIO Waveform:

The following GPIO waveform corresponds to the toggling behavior of the GPIO pins:



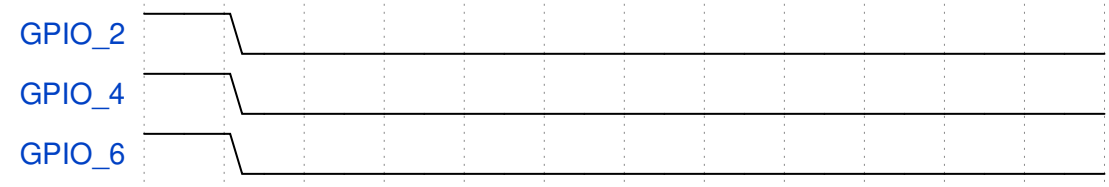
GPIO Example: Set and Clear

This section demonstrates the GPIO waveform when GPIO_PIN_0 is set to HIGH and GPIO_PIN_1 is set to LOW. The following C code shows how these states are configured.

```
#include <stdio.h>
#include <stdlib.h>
#include "gpio.h"

int main()
{
   GPIO_Config(GPIO_OUT, GPIO_PINS(2) | GPIO_PINS(4) | GPIO_PINS(6));

   while (1)
   {
      GPIO_Pin_Set(GPIO_PINS(2) | GPIO_PINS(4) | GPIO_PINS(6)); //␣
↪Sets the GPIO PINS state to HIGH.
      GPIO_Pin_Clear(GPIO_PINS(2) | GPIO_PINS(4) | GPIO_PINS(6)); //␣
↪Sets the GPIO PINS state to LOW.
   }
}
```

Output GPIO Waveform:

The following GPIO waveform illustrates the constant states of two GPIO pins:



ProIO

Example 1: External Clock, Read ProIO

1. **ProIO Configuration**

   The given example is for DUO group with 8 bits transaction. To configure ProIO for the external clock, read.

```
PRO_IO_Struct_t proio_config;

proio_config.pro_io_num   = PRO_IO_DUO;
proio_config.clk_sel      = CLK_EXTERNAL;
proio_config.clk_edge_sel = CLK_POSITIVE_EDGE;
proio_config.direction    = PRO_IO_READ;
proio_config.data_size    = DATA_SIZE_8;
proio_config.mode         = MODE_EXT_CLK_READ;
proio_config.timeout      = 70000000U;

PRO_IO_Config(&proio_config);
```

2. **Read**

In this example, we use spi for the clock and get the data from spi and receive it in the ProIO. To read the ProIO, call the read function:

```
uint32_t rx_data[4];

for (int i = 0; i < 4; i++) {
    PRO_IO_Read(&proio_config, &rx_data[i]);
}
```

3. **Disable ProIO**

   After receiving the data from the ProIO, disable the ProIO using the given function.

```
PRO_IO_Disable(&proio_config);
```

Usage:

This example demonstrates the external clock, transmit data from spi and receive it in ProIO for DUO group:

```
#include "io.h"
#include "secure_iot.h"
#include "gpio.h"
#include "pro_io.h"
#include "spi.h"

/*  External clk Generated, Read
        - Configured SPI as master(Clk generated from spi)
```

(continues on next page)

(continued from previous page)

```
        - Send data from SPI -> Receive the data in ProIO
        - Connect MOSI -> LSB(ProIO)
*/
void main()
{
  unsigned short *baud_reg = (unsigned short*)0x11300;
    *baud_reg = 16;

    PRO_IO_Struct_t gpio_config;
    gpio_config.pro_io_num = 0;
    gpio_config.clk_sel = 1;
    gpio_config.clk_edge_sel = 0;
    gpio_config.direction = 0;
    gpio_config.data_size = 1;
    gpio_config.mode = 1;

    PRO_IO_Config(&gpio_config);

    SPI_Config_t config;
    config.spi_num = SPI2;
    config.spi_clk_mode = MODE_0;
    config.spi_freq = 50000;
    config.setup_time = 0;
    config.hold_time = 0;
    config.is_slave_mode = 0;
    config.is_lsb = 1;
    config.comm_mode = TX;
    config.spi_size = 8;
    config.is_software_ncs = 0;

    uint8_t tx_buff[4] = {0xff, 0x11, 0x22, 0x33};
    spi_buffer buf = {.tx_buf = tx_buff, .rx_buf = NULL, .
↪data_size = 8, .len = 4};
    printf("\ntx:");
    int len = sizeof(tx_buff) / sizeof(tx_buff[0]);
    for(int i = 0;i<4;i++){
        printf("\t%x", tx_buff[i]);
```

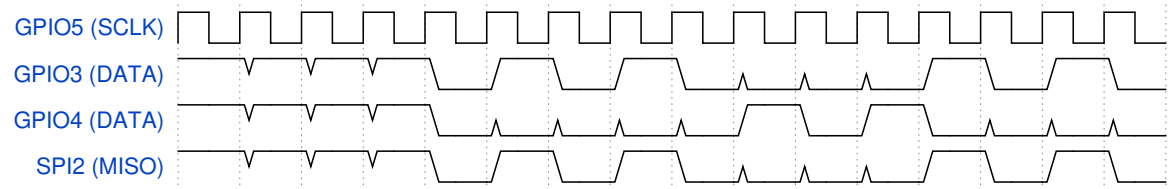
(continues on next page)

(continued from previous page)

```
    }

    SPI_Config(&config);
    SPI_Transceive(&config, &buf);
    if (Wait_Till_TX_Complete(&config) == SUCCESS) {
            printf("Transfer Completed\n\r");
    }

    uint32_t rx_data[4];
    for (int i=0; i<4; i++) {
        PRO_IO_Read(&gpio_config, &rx_data[i]);
    }
    PRO_IO_Disable(&gpio_config);

    printf("\nrx:");
    for (int k=0; k<16; k++) {
        printf("\t%x", rx_data[k]);
    }
    printf("\n");
    SPI_Disable(&config);
}
```

Output ProIO Waveform:

The following GPIO waveform illustrates the DUO group external clock, read:



Note: In this example, GPIO PIN 3 is connected to the SPI2(MISO).

Example 2: External Clock, Write ProIO

1. ProIO Configuration

The given example is for a PRO_IO DUO group with 8-bit transactions. In this mode, the PRO_IO block is configured to use an external clock generated by SPI. Data is written into the ProIO and shifted out synchronized with the SPI clock.

```
PRO_IO_Struct_t gpio_config;
gpio_config.pro_io_num = PRO_IO_DUO;
gpio_config.clk_sel = CLK_EXTERNAL;
gpio_config.clk_edge_sel = CLK_POSITIVE_EDGE;
gpio_config.direction = PRO_IO_WRITE;
gpio_config.data_size = DATA_SIZE_8;
gpio_config.mode = MODE_EXT_CLK_WRITE;
gpio_config.timeout = 70000000U;

PRO_IO_Config(&gpio_config);
```

2. SPI Configuration

SPI is configured as a master to generate the clock required for the PRO_IO transfer. The SPI interface receives data from the PRO_IO during transmission.

```
SPI_Config_t config;

config.spi_num = SPI2;
config.spi_clk_mode = MODE_0;
config.spi_freq = 50000;
config.setup_time = 0;
config.hold_time = 0;
config.is_slave_mode = 0;
config.is_lsb = 1;
config.comm_mode = RX;
config.spi_size = 8;
config.is_software_ncs = 0;
```

3. Write ProIO

Data is written into the ProIO before initiating the SPI transaction. It will then provide data synchronized with the external clock.

```
for (int i = 0; i < 4; i++) {
    PRO_IO_Write(&gpio_config, tx_data[i]);
}
```

4. SPI Transfer and Read

SPI is used to generate the clock and receive the transmitted data from the ProIO.

```
SPI_Config(&config);
SPI_Transceive(&config, &buf);

if (Wait_Till_TX_Complete(&config) == SUCCESS) {
    printf("Transfer Completed\n\r");
}
```

5. Wait and Disable ProIO

After completing the transfer, ensure the ProIO has finished transmitting all data before disabling it.

```
PRO_IO_Wait_Till_tx(&gpio_config);
PRO_IO_Disable(&gpio_config);
```

Usage

This example demonstrates external clock-driven data transmission where SPI acts as the clock source and ProIO acts as the data source. Data written into the ProIO is transmitted out on each SPI clock edge.

```
#include "io.h"
#include "secure_iot.h"
#include "gpio.h"
#include "pro_io.h"
#include "spi.h"
#include "utils.h"

/* External clk Generated, Write
   - Configured SPI as master (Clk generated from SPI)
   - Send data from ProIO -> Receive the data in SPI
   - Connect MISO -> LSB(ProIO)
*/
```

(continues on next page)

(continued from previous page)

```
void main()
{
    unsigned short *baud_reg = (unsigned short*)0x11300;
    *baud_reg = 16;

    PRO_IO_Struct_t gpio_config;
    gpio_config.pro_io_num = PRO_IO_DUO;
    gpio_config.clk_sel = CLK_EXTERNAL;
    gpio_config.clk_edge_sel = CLK_POSITIVE_EDGE;
    gpio_config.direction = PRO_IO_WRITE;
    gpio_config.data_size = DATA_SIZE_8;
    gpio_config.mode = MODE_EXT_CLK_WRITE;
    gpio_config.timeout = 70000000U;

    PRO_IO_Config(&gpio_config);

    uint8_t tx_data[8] = {
        0x66, 0x11, 0x22, 0x33,
        0x44, 0x55, 0x66, 0x99
    };

    SPI_Config_t config;

    config.spi_num = SPI2;
    config.spi_clk_mode = MODE_0;
    config.spi_freq = 50000;
    config.setup_time = 0;
    config.hold_time = 0;
    config.is_slave_mode = 0;
    config.is_lsb = 1;
    config.comm_mode = RX;
    config.spi_size = 8;
    config.is_software_ncs = 0;

    uint8_t rx_buff[8];
    spi_buffer buf = {
        .tx_buf = NULL,
```

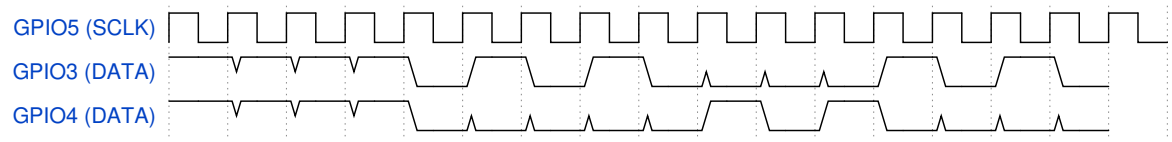
(continues on next page)

(continued from previous page)

```
        .rx_buf = rx_buff,
        .data_size = 8,
        .len = 8
    };

    printf("\ntx:");
    for (int k = 0; k < 8; k++) {
        printf("\t%x", tx_data[k]);
    }

    for (int i = 0; i < 4; i++) {
        PRO_IO_Write(&gpio_config, tx_data[i]);
    }

    SPI_Config(&config);
    SPI_Transceive(&config, &buf);

    if (Wait_Till_TX_Complete(&config) == SUCCESS) {
        printf("Transfer Completed\n\r");
    }

    printf("\nrx:");
    for (int j = 0; j < 8; j++) {
        printf("\t%x", rx_buff[j]);
    }

    PRO_IO_Wait_Till_tx(&gpio_config);
    PRO_IO_Disable(&gpio_config);
}
```

Output ProIO Waveform:

The following ProIO waveform illustrates the DUO group External clock, write:



Example 3: Internal Clock, Read ProIO

1. **ProIO Configuration**

   The given example is for a PRO_IO DUO group with 8-bit transactions operating in **internal clock read mode**. In this mode, the PRO_IO block generates its own clock, and data is sampled from the SPI interface into the ProIO.

```
PRO_IO_Struct_t gpio_config;
gpio_config.pro_io_num = PRO_IO_DUO;
gpio_config.clk_sel = CLK_INTERNAL;
gpio_config.clk_edge_sel = CLK_POSITIVE_EDGE;
gpio_config.direction = PRO_IO_READ;
gpio_config.data_size = DATA_SIZE_8;
gpio_config.prescale = 29;
gpio_config.mode = MODE_INT_CLK_READ;
```

2. **SPI Configuration**

   SPI is configured in slave mode since the clock is generated internally by PRO_IO. Data is transmitted from the SPI interface and captured by the ProIO .

```
SPI_Config_t config;
config.spi_num = SPI2;
config.spi_clk_mode = MODE_0;
config.spi_freq = 50000;
config.setup_time = 5;
config.hold_time = 5;
config.is_slave_mode = 1;
config.is_lsb = 0;
config.comm_mode = TX;
config.spi_size = 8;
config.is_software_ncs = 1;
```

3. **Enable and Configure PRO_IO**

   SPI transfer is initiated first, followed by PRO_IO configuration to start capturing incoming data using the internal clock.

```
PRO_IO_Config(&gpio_config);
```

4. **SPI Data Transmission**

   A transmit buffer is prepared and sent via SPI. This data will be received by the

ProIO.

```
uint8_t tx_data[24] = {
   0xff, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
   0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0x12,
   0x23, 0x34, 0xab, 0xbc, 0xcd, 0xde, 0xef, 0xff
};

spi_buffer buf = {
   .tx_buf = tx_data,
   .rx_buf = NULL,
   .data_size = 8,
   .len = 24
};

SPI_Config(&config);
SPI_Transceive(&config, &buf);
```

5. **Read ProIO**

   Data captured by the ProIO is read sequentially.

```
uint32_t rx_data[192];

for (int i = 0; i < 192; i++) {
   PRO_IO_Read(&gpio_config, &rx_data[i]);
}
```

6. **Disable ProIO**

   After all data has been received, ProIO is disabled.

```
PRO_IO_Disable(&gpio_config);
```

Usage:

This example demonstrates ProIO operating in internal clock mode where ProIO generates the clock and captures incoming SPI data into its buffer.

```
#include "io.h"
#include "secure_iot.h"
#include "gpio.h"
#include "pro_io.h"
```

(continues on next page)

(continued from previous page)

```
#include "spi.h"

void main()
{
   unsigned short * baud_reg = (unsigned short*) 0x11300;
   *baud_reg = 16;

   PRO_IO_Struct_t gpio_config;
   gpio_config.pro_io_num = PRO_IO_DUO;
   gpio_config.clk_sel = CLK_INTERNAL;
   gpio_config.clk_edge_sel = CLK_POSITIVE_EDGE;
   gpio_config.direction = PRO_IO_READ;
   gpio_config.data_size = DATA_SIZE_8;
   gpio_config.prescale = 29;
   gpio_config.mode = MODE_INT_CLK_READ;

   SPI_Config_t config;
   config.spi_num = SPI2;
   config.spi_clk_mode = MODE_0;
   config.spi_freq = 50000;
   config.setup_time = 5;
   config.hold_time = 5;
   config.is_slave_mode = 1;
   config.is_lsb = 0;
   config.comm_mode = TX;
   config.spi_size = 8;
   config.is_software_ncs = 1;

   uint8_t tx_data[24] = {
      0xff, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
      0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0x12,
      0x23, 0x34, 0xab, 0xbc, 0xcd, 0xde, 0xef, 0xff
   };

   spi_buffer buf = {
      .tx_buf = tx_data,
      .rx_buf = NULL,
```

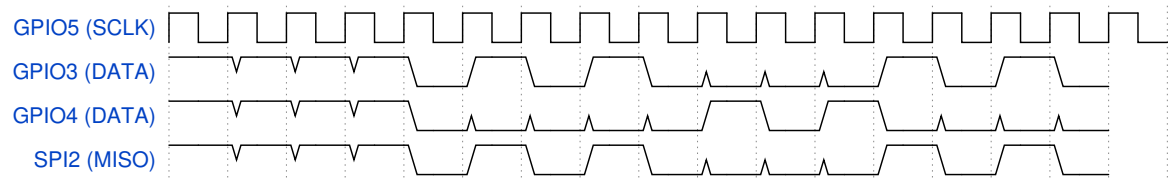
(continues on next page)

(continued from previous page)

```
        .data_size = 8,
        .len = 24
    };

    SPI_Config(&config);
    SPI_Transceive(&config, &buf);

    PRO_IO_Config(&gpio_config);

    printf("\ntx:");
    for (int k = 0; k < 24; k++) {
        printf("\t%x", tx_data[k]);
    }

    uint32_t rx_data[192];
    for (int i = 0; i < 192; i++) {
        PRO_IO_Read(&gpio_config, &rx_data[i]);
    }

    PRO_IO_Disable(&gpio_config);

    printf("\nrx:\n");
    for (int k = 0; k < 192; k++) {
        printf("\t%x", rx_data[k]);
        if ((k + 1) % 8 == 0) {
            printf("\n");
        }
    }
}
```

Output ProIO Waveform:

The following ProIO waveform illustrates the ProIO DUO Internal clock, read:



Note: In this example, GPIO PIN 3 is connected to the SPI2(MISO).

Example 4: Internal Clock, Write ProIO

1. **ProIO Configuration**

   The given example is for a PRO_IO DUO group with 8-bit transactions operating in **internal clock write mode**. In this mode, PRO_IO generates the clock internally and transmits data from the ProIO to the SPI interface.

```
PRO_IO_Struct_t gpio_config;
gpio_config.pro_io_num = PRO_IO_DUO;
gpio_config.clk_sel = CLK_INTERNAL;
gpio_config.clk_edge_sel = CLK_POSITIVE_EDGE;
gpio_config.direction = PRO_IO_WRITE;
gpio_config.data_size = DATA_SIZE_8;
gpio_config.prescale = 29;
gpio_config.mode = MODE_INT_CLK_WRITE;
gpio_config.timeout = 70000000U;
```

2. **SPI Configuration**

   SPI is configured in slave mode since the clock is generated internally by ProIO. Data transmitted from the ProIO is received by the SPI interface.

```
SPI_Config_t config;
config.spi_num = SPI2;
config.spi_clk_mode = MODE_0;
config.spi_freq = 1000000;
config.setup_time = 5;
config.hold_time = 5;
config.is_slave_mode = 1;
config.is_lsb = 0;
config.comm_mode = RX;
config.spi_size = 8;
config.is_software_ncs = 1;
```

3. **Enable and Configure ProIO**

   ProIO is configured prior to writing data into the buffer.

```
PRO_IO_Config(&gpio_config);
```

4. **Write Data to ProIO**

   Data is written into the PRO_IO sequentially. This data will be transmitted over

the internally generated clock and received by SPI.

```
uint8_t tx_data[24] = {
   0xff, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
   0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0x12,
   0x23, 0x34, 0xab, 0xbc, 0xcd, 0xde, 0xef, 0xff
};

for (int i = 0; i < 16; i++) {
   PRO_IO_Write(&gpio_config, tx_data[i]);
}
```

Note

Only the first 16 bytes of the transmit buffer are written into the ProIO in this example.

5. **SPI Transaction**

   SPI is triggered to receive the data transmitted from PRO_IO.

```
uint8_t rx_buff[32];
spi_buffer buf = {
   .tx_buf = NULL,
   .rx_buf = rx_buff,
   .data_size = 8,
   .len = 8
};

SPI_Config(&config);
SPI_Transceive(&config, &buf);
```

6. **Wait for Transmission Completion**

   Wait until ProIO completes transmitting all buffered data.

```
PRO_IO_Wait_Till_tx(&gpio_config);
```

7. **Disable ProIO**

   After the transaction is complete, disable ProIO.

```
PRO_IO_Disable(&gpio_config);
```

Usage:

This example demonstrates ProIO operating in internal clock mode where ProIO generates the clock and transmits buffered data to SPI, which receives it in slave mode.

```
#include "io.h"
#include "secure_iot.h"
#include "gpio.h"
#include "pro_io.h"
#include "spi.h"
#include "utils.h"

void main()
{
   unsigned short * baud_reg = (unsigned short*) 0x11300;
   *baud_reg = 16;

   PRO_IO_Struct_t gpio_config;
   gpio_config.pro_io_num = PRO_IO_DUO;
   gpio_config.clk_sel = CLK_INTERNAL;
   gpio_config.clk_edge_sel = CLK_POSITIVE_EDGE;
   gpio_config.direction = PRO_IO_WRITE;
   gpio_config.data_size = DATA_SIZE_8;
   gpio_config.prescale = 29;
   gpio_config.mode = MODE_INT_CLK_WRITE;
   gpio_config.timeout = 70000000U;

   SPI_Config_t config;
   config.spi_num = SPI2;
   config.spi_clk_mode = MODE_0;
   config.spi_freq = 1000000;
   config.setup_time = 5;
   config.hold_time = 5;
   config.is_slave_mode = 1;
   config.is_lsb = 0;
   config.comm_mode = RX;
   config.spi_size = 8;
   config.is_software_ncs = 1;

   uint8_t tx_data[24] = {
```

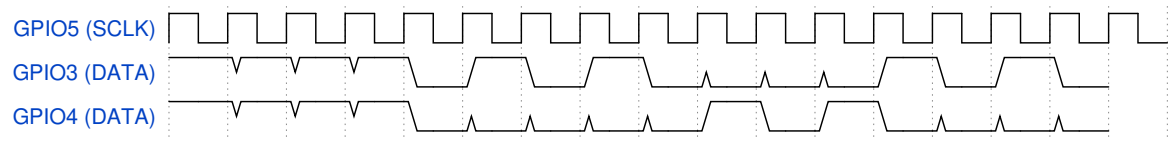
(continues on next page)

(continued from previous page)

```
        0xff, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
        0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0x12,
        0x23, 0x34, 0xab, 0xbc, 0xcd, 0xde, 0xef, 0xff
    };

    uint8_t rx_buff[32];

    spi_buffer buf = {
        .tx_buf = NULL,
        .rx_buf = rx_buff,
        .data_size = 8,
        .len = 8
    };

    SPI_Config(&config);

    printf("\ntx:");
    for (int k = 0; k < 16; k++) {
        printf("\t%x", tx_data[k]);
    }

    PRO_IO_Config(&gpio_config);

    for (int i = 0; i < 16; i++) {
        PRO_IO_Write(&gpio_config, tx_data[i]);
    }

    SPI_Transceive(&config, &buf);

    printf("\nrx:");
    for (int j = 0; j < 32; j++) {
        printf("\t%x", rx_buff[j]);
    }

    PRO_IO_Wait_Till_tx(&gpio_config);
    PRO_IO_Disable(&gpio_config);
}
```

Output ProIO Waveform:

The following ProIO waveform illustrates the PRO_IO DUO Internal clock, write:



7.1.8 GPTimer (General Purpose Timer)

Example Code

This example shows how to trigger the GPTimer interrupt.

```
#include "io.h"  /*Included to access functions for basic IO␣
↪operations such as printf,etc*/
#include "gptimer.h" /*Included to access GPTimer peripheral API's*/
#include "plic.h"

uint8_t interrupt_triggered=0;

void gptimer_isr()
{
    printf("Interrupt triggered\n\r");
    interrupt_triggered = 1;
    PLIC_Interrupt_Disable(GPTIMER1_IRQn); //Disable interrupt to␣
↪ensure ISR is called only once
}

int main()
{
  GPTIMER_Config_t gptimer = {
      .gpt_num = 1,
      .mode = GPT_UP_COUNT,
      .interrupt_en = 1, //enabling interrupts to verify the reset␣
↪occurs
      .period = 20,
      .prescalar = 1,
      .cnt_en = 0,
```

(continues on next page)

(continued from previous page)

```
      .output_en = 0
  };

  PLIC_Set_Handler(GPTIMER1_IRQn, gptimer_isr,NULL);
  PLIC_Set_Interrupt_Priority(GPTIMER1_IRQn, PLIC_PRIORITY_3);
  PLIC_Interrupt_Enable(GPTIMER1_IRQn);
  GPT_Init(&gptimer);
  while(interrupt_triggered == 0); // Wait till interrupt has been␣
↪triggered.
  GPT_Reset(&gptimer); // will reset the counter.
  return 0;
}
```

Output

```
Interrupt Triggered
```

Troubleshooting

If you encounter issues while working with the GPTimer driver, follow these steps to identify and resolve the problem:

1. **Driver Initialization Issues**

   The GPTimer driver fails to initialize:

   - clock source configuration: Verify that the clock source is correctly configured in the control register.
   - Invalid prescaler value: Ensure the prescaler value is within the valid range.

2. **No Timer Output**

   The timer does not produce any output signal:

   - Check *gpt_enable*: Ensure that *gpt_enable* is set to 1 to enable the timer.
   - Check *output_enable*: Verify that *output_enable* is configured as needed.
   - Verify mode selection: Check the *gpt_mode* field to ensure the correct mode is selected (PWM, up, down, up-down).

7.1.9 I2C (Inter-Integrated Circuit)

I2C RTC Example

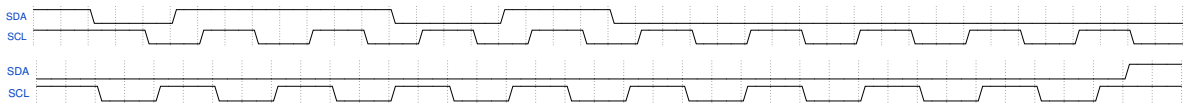
To use I2C, follow these steps:

- Step 1: Initialize I2C with SCL frequency.
- Step 2: Use I2C_Transmit to send data to slave and I2C_Recieve to read data from slave.

Example Code:

```
#include "io.h"         /*Included to access functions for basic IO␣
↪operations such as printf,etc*/
#include "i2c.h"        /*Included to access I2C peripheral API's*/

void main(){
  I2C_Init(0,100000);
  uint8_t data = 0x00,seconds;
  uint8_t i2c_instance = 0x00;
  uint8_t slave_address = 0x68;
  while(1)/*Read time from RTC continuously*/
  {
    I2C_Transmit(i2c_instance,slave_address,&data,1,START_BIT|STOP_
↪BIT);//Setting reg address to read seconds from RTC
    I2C_Receive(i2c_instance,slave_address,&seconds,1,START_BIT|STOP_
↪BIT);//Reading from given register address from RTC
    printf("\n Seconds: %d\n",seconds);
  }
}
```

Output



Troubleshooting

1. If you encounter mcause exception,make sure that the data lines and clock signal are correctly wired.
2. If code is stuck in infinite loop in I2C transmit/recieve function, reset the chip, sometimes I2C may keep on generating clock even after ending transaction due to some glitch.
3. Ensure that pull resistors are in range of 2kOmega to 5kOmega.

7.1.10 ITRACE

Example Code

Filter

```
// Filter
#include "itrace.h"
#include "uart.h"

void filter_val()
{
printf("\nFiltered");
printf("\nFilter_control: %x",*(filter_cntrl));
printf("\nComp1_control: %x",*(comp1_cntrl));
printf("\nP1: %x",*(comp1_p_match_low));
printf("\nS1: %x",*(comp1_s_match_low));
}

void main()
{
    int (*putchar_ptr)(int) = &putchar;
    uint32_t buffer = 0x11404;
    UART_Config_t uart_config; /*Structure variable which␣
↪hold the UART's specifications */
    uart_config.uart_num = 1; /*Specifies the UART instance*/
    uart_config.baudrate = 115200; /*Specifies the baudrate*/
```

(continues on next page)

(continued from previous page)

```
    uart_config.char_size=8; /*Specifies the Char size*/
    uart_config.delay=0;/*Specifies the delay*/
    uart_config.parity=0;/*Specifies the parity*/
    uart_config.stop_bits=1;/*Specifies the stopbits*/
    uart_config.transfer_mode=3;
    uart_config.receive_mode=3;
    uart_config.pullup=0;

    printf("\ncode started !\n");
    UART_Init(&uart_config);

    ITRACE_Ram_Ctrl(buffer);
    ITRACE_Comp_Ctrl(1);
    ITRACE_Filter_val(2, 3, 5);

    *comp1_p_match_low  = (int)putchar_ptr;
    *comp1_p_match_high = 0;
    *comp1_s_match_low  = (int)putchar_ptr + 62;
    *comp1_s_match_high = 0;

    ITRACE_Ctrl();
    printf("hello world\n");
    ITRACE_Disable_Ctrl_Reg();
    filter_val();
}
```

Ram Sink

```
// Ram sink
#include "itrace.h"
#include "uart.h"
void main()
{
    uint32_t buffer = 1;

    UART_Config_t uart_config; /*Structure variable which␣
```

(continues on next page)

(continued from previous page)

```
↪hold the UART's specifications */
    uart_config.uart_num = 1; /*Specifies the UART instance*/
    uart_config.baudrate = 115200; /*Specifies the baudrate*/
    uart_config.char_size=8; /*Specifies the Char size*/
    uart_config.delay=0;/*Specifies the delay*/
    uart_config.parity=0;/*Specifies the parity*/
    uart_config.stop_bits=1;/*Specifies the stopbits*/
    uart_config.transfer_mode=3;
    uart_config.receive_mode=3;
    uart_config.pullup=0;

    printf("\ncode started !\n");
    UART_Init(&uart_config);

    ITRACE_Ram_Ctrl(buffer);
    ITRACE_Ctrl();
    // printf("hello world\n");
    int x = 7*4;
    int b = 7*4;
    int c = 7*4;
    ITRACE_Disable_Ctrl_Reg();

    uint32_t a;
    struct uart_buf rx = { .uart_data = &a, .len = 1};
    for (int i = 0; i < 23; i++)
    {
        UART_Read_Character(&uart_config,&rx); /*Read all the␣
↪data in Built-in buffer*/
        printf("%08x\n", a);
    }

    for (int i=0; i<200; i++)
    asm volatile("nop");
}
```

7.1.11 Pinmux driver example

PINMUX APIs provide a interface for configuring PINMUX pins in the system.

PINMUX Example :

This example explores how to blink an LED using PWM followed by switching ON the LED using GPIO.

```
#include "pinmux.h"
#include "gpio.h"
#include "pwm.h"

void main() {
    PINMUX_PWM(0, true);
    PWM_Config_t config;
    config.duty = 0x0032;
    config.period = 0x0064;
    config.interrupt_mode = no_interrupt;
    config.change_output_polarity = false ;
    config.prescalar_value = 0x009;
    config.deadband_delay = 0x0005;
    PWM_Start(&config, PWM_PIN(0));

    PINMUX_PWM(0, false);
    GPIO_Config(GPIO_OUT, GPIO_PINS(0));
    GPIO_Pin_Set(GPIO_PINS(0));
}
```

Corresponding PINMUX Waveform:

The following PINMUX waveform corresponds to the behavior of the PIN:

7.1.12 Platform Level Interrupt Controller (PLIC)

PLIC Sample Application

This example shows how to initialize PLIC and to trigger an ISR function when GPIO pin gets low as input.

Example Code:

```
#include"io.h"
#include "gpio.h"
#include "plic.h"
#include"interrupt_control.h"
#define GPIO_PIN_A 28
#define GPIO_PIN_B 10


void GPIO_IRQHandler(void *args)
{   //Testing PLIC functionality disable function
    PLIC_Interrupt_Disable(GPIO0_IRQn+(uint32_t)args);
    printf("\r\nButton pressed is %d",(uint32_t)args);
    //Testing PLIC functionality enable function
    PLIC_Interrupt_Enable(GPIO0_IRQn+(uint32_t)args);
}
void main(void){
   // core_switch_interrupt_mode(NON_VECTORED_MODE);//By default␣
↪vectored mode
   PLIC_Interrupt_Threshold(PLIC_PRIORITY_1);/*masking interrupts with␣
↪priority <= 2*/

   PLIC_Set_Handler(GPIO0_IRQn+GPIO_PIN_A, GPIO_IRQHandler,(void␣
↪*)GPIO_PIN_A);
   PLIC_Set_Handler(GPIO0_IRQn+GPIO_PIN_B, GPIO_IRQHandler,(void␣
↪*)GPIO_PIN_B);

   /*Initializing GPIO 28 as input pin*/
   GPIO_Config(GPIO_IN, GPIO_PINS(GPIO_PIN_A));
   /*Configuring interrupt to GPIO 28 such when it read low (change 0␣
↪to 1 for high)interrupt will be trigerred */
```

(continues on next page)

(continued from previous page)

```
   GPIO_Interrupt_Config( GPIO_PINS(GPIO_PIN_A), 0);
   /*Setting interrupt priority*/
   PLIC_Set_Interrupt_Priority(GPIO0_IRQn+GPIO_PIN_A, PLIC_PRIORITY_3);
   /*Enabling Interrupt*/
   PLIC_Interrupt_Enable(GPIO0_IRQn+GPIO_PIN_A);

   /*Initializing GPIO 10 as input pin*/
   GPIO_Config(GPIO_IN, GPIO_PINS(GPIO_PIN_B));
   /*Configuring interrupt to GPIO 10 such when it read low (change 0␣
↪to 1 for high)interrupt will be trigerred */
   PLIC_Set_Interrupt_Priority(GPIO0_IRQn+GPIO_PIN_B, PLIC_PRIORITY_3);
↪/*Setting interrupt priority*/
   GPIO_Interrupt_Config( GPIO_PINS(GPIO_PIN_B), 0);
   /*Enabling Interrupt*/
   PLIC_Interrupt_Enable(GPIO0_IRQn+GPIO_PIN_B);

   while(1)
   {
       printf("\r\nWaiting in loop");
   }
}
```

Output

```
Waiting in loop
Waiting in loop
Waiting in loop
Button pressed is 10
Button pressed is 10
Button pressed is 10
Waiting in loop
```

Troubleshooting

1. **Interrupt Not Triggering**

- **Possible Causes**: - The interrupt source is not enabled in the Interrupt Enabled Register. - The interrupt priority is set to *0*, which disables the interrupt. - The priority threshold is set too high, masking interrupts with lower priority.
- **Solutions**: - Ensure the corresponding bit in the Interrupt Enabled Register. - Check and set the interrupt priority. - Verify and adjust the Priority Threshold Register.

2. **Interrupt Not Cleared**
   - **Possible Causes**: - The interrupt completion process is not performed. - The interrupt source continues to assert the interrupt signal.
   - **Solutions**: - Write the interrupt ID to the Interrupt Completion Register. - Ensure the interrupt source de-asserts the interrupt signal after handling.
3. **Spurious Interrupts**
   - **Possible Causes**: - Incorrect interrupt ID written during completion.
   - **Solutions**: - Ensure the correct interrupt ID is written during the completion process.

7.1.13 PMP (Physical Memory Protection)

Functions Usage

1. **Configuring a PMP entry**

   To configure a PMP entry with permissions and address space:

   Example Code:

```
uint8_t config = PMP_NAPOT_MATCHING | PMP_READ_ACCESS | PMP_
↪LOCK_BIT;
uint8_t region = 5;
uint32_t* pmp_addr = (uint32_t*)0x80010000;
size_t size = 16384;
PMP_Set_Region(config, region, pmp_addr, size);
```

2. **Clear a PMP entry**

   Example Code:

   To clear a specific PMP entry:

```
uint8_t region = 5;
PMP_Clear_Region(region);
```

To clear all PMP entries:

```
PMP_Clear_All();
```

PMP Sample Application

This example shows how to set and clear a PMP entry.

```
#include "io.h"
#include "pmp.h"
#include "errors.h"

void main() {
 uint8_t config1, config2, region1, region2 ;
 uint32_t size, address;
 uint32_t* pmp_addr = (uint32_t*)0x80010000;
 config1 = PMP_NAPOT_MATCHING | PMP_READ_ACCESS | PMP_LOCK_BIT;
 config2 = PMP_TOR_MATCHING | PMP_READ_ACCESS | PMP_LOCK_BIT;
 region1 = 3;
 region2 = 4;
 size = 16384; //Not needed for TOR mode

 uint16_t ret;
 ret = PMP_Set_Entry(config1, region1, pmp_addr, size);
 if(ret == SUCCESS) {
     printf("PMP Entry %d Configured successfully\r\n", region1);
 }

 ret = PMP_Set_Entry(config2, region2, (uint32_t*)0x80016000, 0);
 if(ret == SUCCESS) {
     printf("PMP Entry %d Configured successfully\r\n", region2);
 }

 uint32_t *ptr = (uint32_t *)0x80010000;
 *ptr = 19;  //This will trigger store access fault(If lock bit set)␣
```

(continues on next page)

(continued from previous page)

```
↪as only read permission is given


 PMP_Clear_All();

 while(1);
}
```

Output

```
PMP Entry 3 Configured successfully
PMP Entry 4 Configured successfully
Store/AMO access fault at 80008924
```

Troubleshooting

1. Ensure that the hart is in Machine mode before accessing the PMP functions.
2. If the lock bit is enabled for a region, PMP_Set_Region and PMP_Clear_Region will not apply to that region.
3. If TOR matching is used, ensure that the starting address of the region to be protected is correctly configured in the previous PMP entry.

7.1.14 PWM (Pulse Width Modulation)

PWM Example

PWM is used for generating analog-like signals, controlling motors, blinking LEDs, or in communication protocols. It is often used in applications where power control or signal modulation is required.

1. **Initializing PWM**

To initialize a PWM instance with specific duty cycle, period, prescalar, and other configurations:

```
PWM_Config_t config;  /* Structure variable to hold PWM specifications
↪*/
config.duty = 0x0032;  /* Sets the duty cycle */
config.period = 0x0064;  /* Sets the period */
config.interrupt_mode = no_interrupt;  /* Disables interrupts */
config.change_output_polarity = false;  /* Disables output polarity
↪change */
config.prescalar_value = 0x4E20;  /* Sets the prescalar value */
config.deadband_delay = 0x00;  /* Sets the deadband delay */
```

2. **PWM Start function**

This example shows how to start PWM using the specified configuration values. It demonstrates a simple PWM initialization and starting process:

Example Code:

```
PWM_Start(&config, PWM_PIN(0) | PWM_PIN(1));
```

Example Code:

```
PWM_Start(&config, PWM_PIN(0) | PWM_PIN(1));
```

Application

Example Code:

This code is for blinking an LED using PWM.

```
#include "pwm.h"

int main()
{
     PWM_Config_t config;
     config.duty = 0x0032;
     config.period = 0x0064;
     config.interrupt_mode = no_interrupt;
     config.change_output_polarity = true;
     config.prescalar_value = 0x4E20;
     config.deadband_delay = 0x00;
```

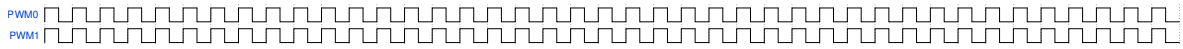
(continues on next page)

(continued from previous page)

```
    PWM_Start(&config, PWM_PIN(0) | PWM_PIN(1));

    return 0;
}
```

PWM Blink Application Waveform

The following waveform represents the PWM signal based on the configured parameters in the example code:



Troubleshooting

1. **Incorrect Pin-Mux Configuration**
   - Verify that the selected pins are configured for PWM through pin-mux.
   - Ensure the pins are not assigned to another peripheral.
2. **Invalid Duty or Period Values**
   - Valid ranges:
     - *period: 1 – 0xFFFF (65535)*
     - *duty: 0 – period*
   - duty must be <= period; otherwise the PWM output may remain constantly high or low.
3. **Prescaler Out of Range**
   - Valid range: 1 – 32768
   - PWM frequency formula:

     `Freq = Clock_freq / (period * (prescalar_value + 1))`
   - Incorrect prescaler settings can produce an unusable PWM frequency.
4. **Interrupts Not Working**
   - If interrupts are enabled:
     - Ensure the ISR is implemented and registered.
     - Confirm that global interrupt enable is set.
   - If interrupts are not required, set `PWM_INT_NONE`.

5. **Inverted Output Signal**
   - If the waveform appears inverted, check the `change_output_polarity` setting.
6. **Invalid Deadband Configuration**
   - Valid range: 0 – 0xFF (255)
   - Out-of-range deadband values may distort the PWM signal.

7.1.15 QSPI (Quad-Serial Peripheral Interface)

Include the header file and configure the QSPI for transaction.

Header File Descriptions

- **qspi.h** This file contains the function declaration, data structure, and macros necessary for configuring and using the QSPI (Quad Serial Peripheral Interface) protocol. Key components of *qspi.h* include:
- **qspi_msg**: A structure for setting up QSPI parameters such as clock polarity, phase, prescaler, and communication mode.
- **QSPI_Transaction**: Performs the data transmission and reception over the QSPI interface.

Configure QSPI Parameters

The QSPI configuration is set up using the *qspi_msg* structure. Each field in this structure customizes a specific aspect of the QSPI communication.

```
qspi_msg flash_msg;

flash_msg.functional_mode = CCR_FMODE_INDIRECT_READ;/*Setting␣
↪functional mode*/
flash_msg.instruction = 0x5A;                       /*Setting␣
↪instruction to be sent*/
flash_msg.instruction_mode = CCR_IMODE_SINGLE_LINE; /*Setting␣
↪instruction mode*/
flash_msg.address_mode = CCR_ADMODE_SINGLE_LINE;    /*Setting Address␣
```

(continues on next page)

(continued from previous page)

```
↪mode */
flash_msg.address_size = CCR_ADSIZE_24_BIT;          /*Setting Address␣
↪size*/
flash_msg.alternate_byte_mode = CCR_ABMODE_NIL;      /*Setting␣
↪alternate byte mode*/
flash_msg.dummy_mode = 0;                            /*Setting dummy␣
↪mode*/
flash_msg.dummy_bit = 0;                             /*Setting dummy␣
↪bit*/
flash_msg.dummy_cycles = 7;                          /*Setting no of␣
↪dummy cycles*/
flash_msg.mm_mode = CCR_MM_MODE_XIP;                 /*Setting which␣
↪memory map mode had to be set*/
flash_msg.data_mode = CCR_DMODE_SINGLE_LINE;         /*Setting data␣
↪mode*/
flash_msg.length = 1;                                /*Setting length␣
↪of data be read*/
flash_msg.data_buffer = &data;
flash_msg.FMEM_SIZE =27;
flash_msg.CLK_MODE = 1;                             /*Setting QSPI clock␣
↪mode*/
flash_msg.TCEN = 0;
flash_msg.TEIE = 0;
flash_msg.SMIE = 0;
flash_msg.FTIE = 0;
flash_msg.TOIE = 0;
flash_msg.APMS = 0;
flash_msg.PMM = 0;
flash_msg.PRESCALER = GET_QSPI_PRESCALAR(6000000);/*Setting QSPI␣
↪prescalar to fix QSPI clock frequency*/
```

Configuration Fields

- The *instruction* is set to *CCR_IMODE_SINGLE_LINE*.
- The *address_mode* is set to *CCR_ADMODE_SINGLE_LINE*.
- The *address_size* is set to *CCR_ADSIZE_24_BIT*.
- The *alternate_byte_mode* is set to *CCR_ABMODE_NIL*.

- The *dummy_mode* is set to *0*.
- The *dummy_bit* is set to *0*.
- The *dummy_cycles* is set to *7*.
- The *mm_mode* is set to *CCR_MM_MODE_XIP*.
- The *data_mode* is set to *CCR_DMODE_SINGLE_LINE*.
- The *length* is set to *1*.
- The *data* is set to *&data*.
- The *FMEM_SIZE* is set to *27*;
- The *CLK_MODE* is set to *1*.
- The *TCEN* is set to *0*.
- The *TEIE* is set to *0*.
- The *SMIE* is set to *0*.
- The *FTIE* is set to *0*.
- The *TOIE* is set to *0*.
- The *APMS* is set to *0*.
- The *PMM* is set to *0*.
- The *PRESCALER* is set to *8*.

This configuration prepares the QSPI protocol for communication according to the specified parameters.

QSPI Example Code

This example demonstrates how to configure and use QSPI communication with transmit and receive buffers.

```
#include "qspi.h"      /*Included to access QSPI Flash driver API's*/

#define GET_QSPI_PRESCALAR(QSPI_FREQUENCY) ((CLOCK_FREQUENCY/QSPI_
↪FREQUENCY)-1)

void main()
{
    uint8_t qspi_instance = 0;
    qspi_msg flash_msg;
```

(continues on next page)

(continued from previous page)

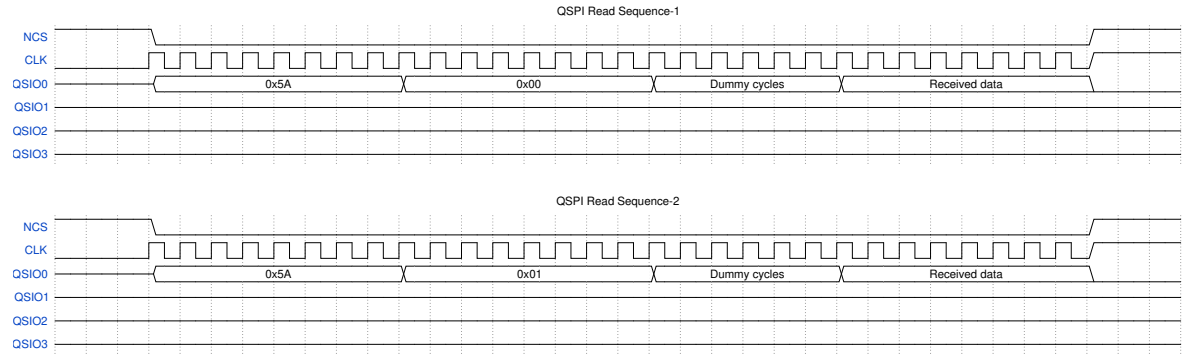
```
    uint8_t data;
    flash_msg.PRESCALER = GET_QSPI_PRESCALAR(6000000);/*Setting QSPI␣
↪prescalar to fix QSPI clock frequency*/
    flash_msg.CLK_MODE = 1;                          /*Setting QSPI␣
↪clock mode to fix QSPI clock frequency*/
    flash_msg.FMEM_SIZE =27;

    /*Setting all interrupts to 0 as it is unncessary*/
    flash_msg.FTIE = 0;
    flash_msg.TCEN = 0;
    flash_msg.TEIE = 0;
    flash_msg.TOIE = 0;
    flash_msg.SMIE = 0;

    flash_msg.APMS = 0;
    flash_msg.PMM = 0;
    flash_msg.address_mode = CCR_ADMODE_SINGLE_LINE;    /*Setting␣
↪Address mode */
    flash_msg.address_size = CCR_ADSIZE_8_BIT;         /*Setting␣
↪Address size*/
    flash_msg.instruction = 0x5A;                       /*Setting␣
↪instruction to be sent*/
    flash_msg.instruction_mode = CCR_IMODE_SINGLE_LINE; /*Setting␣
↪instruction mode*/
    flash_msg.data_mode = CCR_DMODE_SINGLE_LINE;        /*Setting data␣
↪mode*/
    flash_msg.functional_mode = CCR_FMODE_INDIRECT_READ;/*Setting␣
↪functional mode*/
    flash_msg.dummy_mode = 0;                           /*Setting␣
↪dummy mode*/
    flash_msg.dummy_cycles = 7;                         /*Setting no␣
↪of dummy cycles*/
    flash_msg.dummy_bit = 0;                            /*Setting␣
↪dummy bit*/
    flash_msg.mm_mode = CCR_MM_MODE_XIP;                                /
↪*Setting which memory map mode had to be set*/
    flash_msg.alternate_byte_mode = CCR_ABMODE_NIL;     /*Setting␣
```

(continues on next page)

(continued from previous page)

```
↪alternate byte mode*/
    flash_msg.length = 1;                                   /*Setting␣
↪length of data be read*/
    flash_msg.data_buffer = &data;                          /*Setting␣
↪pointer to address of data variable to be updated*/
    printf("\n\r BA \t Value\n\r");
    for(int i = 0; i  < 304; i++)
    {
        flash_msg.address = i;                              /*Setting␣
↪address to be read*/
        QSPI_Transaction(qspi_instance,&flash_msg);         /*Performing␣
↪the QSPI transaction*/
        printf("%03x \t %04x \n\r", i, data);
    }
    while(1);
}
```

QSPI Waveform



QSPI Troubleshooting

1. **Data Corruption**: If data appears garbled or incorrect.

   Check Clock and Data Lines: Inspect all QSPI lines (CLK, CS, IO0-IO3) for loose connections or interference. Ensure signals are stable and free from noise.

2. **No Data Transmission or Reception**: If data is not being sent or received:

Confirm Pin Configuration: Check that all necessary QSPI pins (including any control signals like CS) are correctly mapped and initialized. Verify Initialization Sequence: Ensure that the QSPI peripheral is initialized with the correct settings, and the mode.

3. **Incorrect Data Lengths** :If data read or written is shorter or longer than expected:

   Check Command and Address Phases: Ensure that the QSPI command and address lengths match the expected configuration for the connected device (e.g., 8-bit, 16-bit, 24-bit or 32-bit commands). Set the Right Data Width: Ensure the data width (number of lines used, NIL, 1, 2, or 4) is correctly configured in the QSPI peripheral settings.

7.1.16 QSPI Flash

1. **Include the necessay headerfiles:**

The QSPI Flash communication program requires the main header file to function correctly. This header file is included at the beginning to provide the necessary definitions, macros, and function declarations for QSPI Flash operations:

```
#include "qspi_flash.h"
```

This ensures that the QSPI_Flash functions and I/O operations are available throughout the program.

2. **Configure QSPI_Flash Parameters:**

The QSPI_Flash configuration is set up. Each variable customizes a specific aspect of the QSPI_Flash communication.

Variables

- **qspinum** : This parameter is an unsigned integer that represents the QSPI instance number.
- **data** : This parameter is a pointer to array which contains data to be read from flash.
- **starting_address** : This parameter specifies from which start address data to be read from flash through QSPI.
- **data_length** : This parameter specifies how many bytes of data should be read from flash.

```
uint8_t qspi_instance = 0;
uint8_t data[16] = {[0 ... 7] = 0x11,[8 ... 15] = 0x22};
uint8_t starting_address = 0x600;
uint8_t data_length = 16;
```

QSPI_Flash Example Code

The following examples demonstrates how to use QSPI_Flash communication.

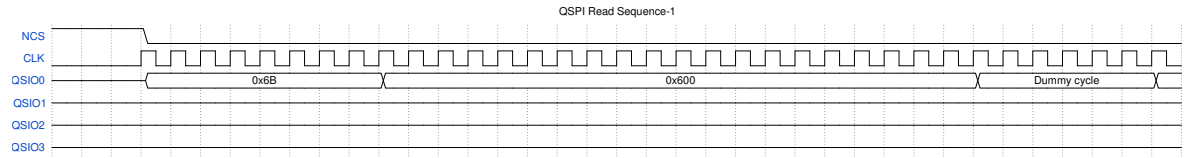
Flash_read

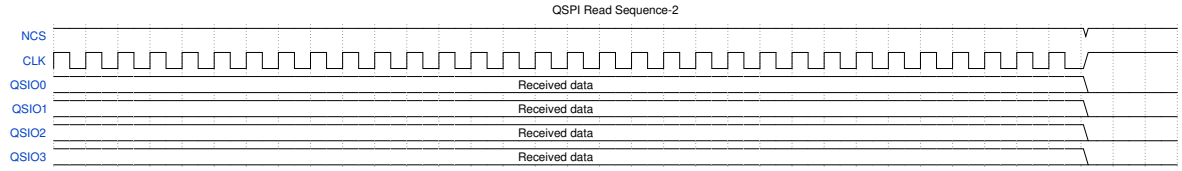
Example code to read data from flash memory through QSPI.

```
#include "qspi_flash.h"/*Included to access QSPI Flash driver API's*/

void main(){
    uint8_t qspi_instance = 0;
    uint8_t data[16];
    uint8_t starting_address = 0x600;
    uint8_t data_length = 16;
    fastReadQuad(qspi_instance,data,starting_address,data_length);/
↪*Read data from flash*/
    for(uint8_t i = 0; i < 16; i++){
        printf("Data[%d] = %x\r \n\r",i,data[i]);
    }
    while(1);
}
```

Waveform

The following waveform corresponds to the Flash_read.





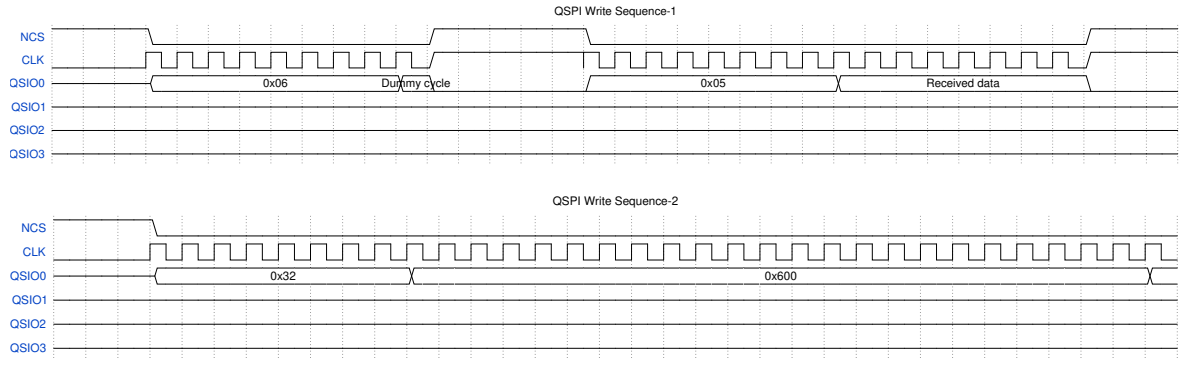
Flash_write

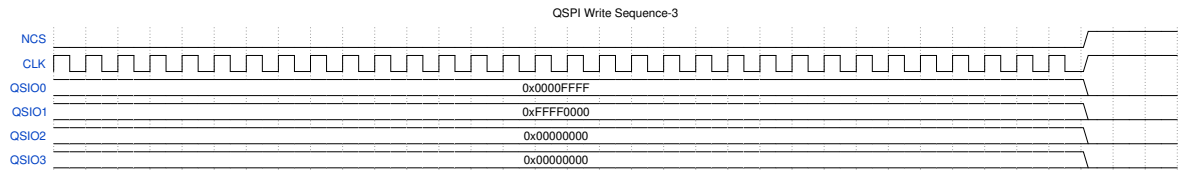
Example code to write data into flash memory through QSPI.

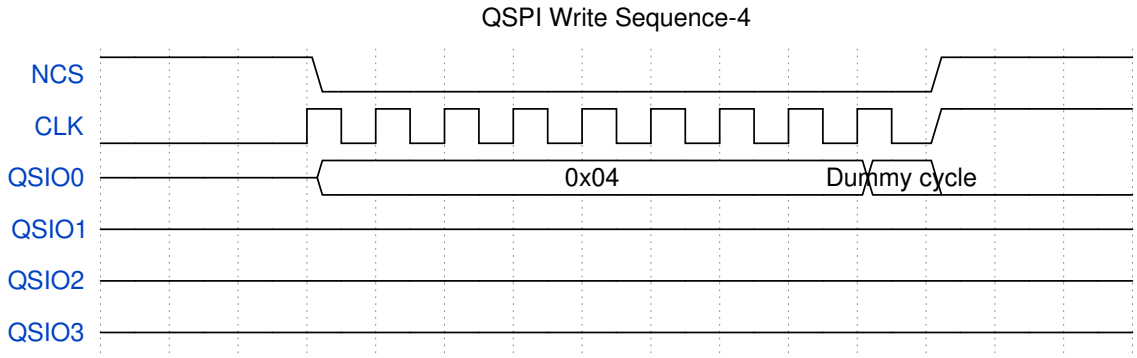
```
#include "qspi_flash.h"/*Included to access QSPI Flash driver API's*/

void main()
{
  uint8_t qspi_instance = 0;
  uint8_t data[16] = {[0 ... 7] = 0x11,[8 ... 15] = 0x22};
  uint8_t starting_address = 0x600;
  uint8_t data_length = 16;
  writeEnable(qspi_instance);/*Enable write operation*/
  inputpageQuad(qspi_instance,data,starting_address,data_length);/*To␣
↪write data to flash*/
  writeDisable(qspi_instance);/*Disable write operation*/
  while(1);
}
```

Waveform

The following waveform corresponds to the Flash_write.







Flash_erase

Code to erase whole chip through QSPI.

```
#include "qspi_flash.h"/*Included to access QSPI Flash driver API's*/

void main(){
uint8_t sr;
        uint8_t qspi_instance = 0;
        printf("Wait Chip Erase in Progress!!");
        writeEnable(qspi_instance);/*Enable write operation*/
        chipErase(qspi_instance);/*Send chip erase command*/
        writeDisable(qspi_instance);/*Disable write operation*/
        uint8_t temp = 0;
        while(1)
        {
        readFlagStatusRegister(qspi_instance, &temp);
        readStatusRegister1(qspi_instance, &sr);
        temp = temp & (1<<7);
        if(temp == (1<<7)&&(sr == 0)){/*Wait till ready bit is set,
↪used to check if erase operation is in progress*/
            break;
        }
    }
```

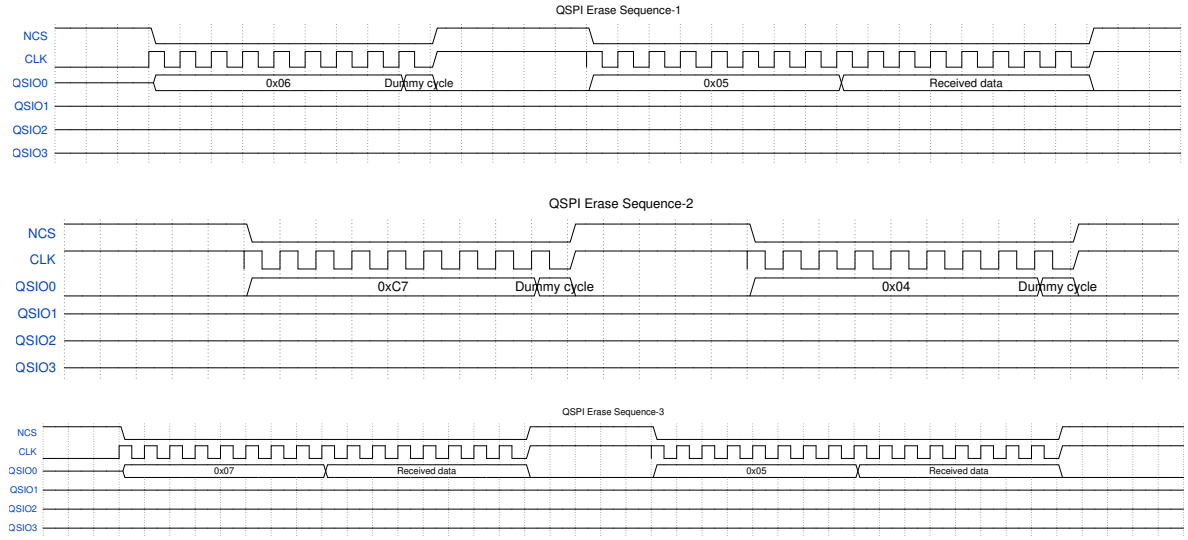
(continues on next page)

(continued from previous page)

```
    printf("Chip Erase Complete!");
}
```

Waveform

The following waveform corresponds to the Flash_erase.



QSPI Flash Troubleshooting

1. **Data Corruption**: If data appears garbled or incorrect.

   Check signal integrity (CLK, CS, IO0-IO3) for noise. Verify timing parameters (clock frequency, setup/hold times). Ensure stable power supply with proper decoupling.

2. **No Data Transmission or Reception**: If data is not being sent or received:

   Confirm Pin Configuration: Check that all necessary QSPI pins (including any control signals like CS) are correctly mapped and initialized. Verify Initialization Sequence: Ensure that the QSPI peripheral is initialized with the correct settings, and the mode.

3. **Slow Data Transfer**:If data transfer takes longer than expected.

   Maximize QSPI clock frequency. Use Quad I/O mode for faster data rates. Enable continuous read or burst mode.

7.1.17 SPI (Serial Peripheral Interface)

Below are example codes for SPI master and SPI slave operating in different transfer modes.

SPI Master(TX) and Slave(RX) Example code

This example demonstrates how to configure and use SPI0 as `MASTER` in `TX` mode.

```
#include "spi.h"
#include "io.h"
#define DATA_LEN 10 //  Number of data
  int main()
  {
   for (uint32_t i = 0; i < DATA_LEN ; i++){
       tx_data[i] = i+1;
   }

  spi_buffer buf = {.tx_buf = tx_data, .rx_buf = NULL, .data_size = 8,␣
↪.len = DATA_LEN};

  SPI_Config_t spi_config;
  spi_config.spi_freq      = 6000000;
  spi_config.spi_number    = 0;
  spi_config.spi_clk_mode  = MODE_0;
  spi_config.setup_time    = 0;
  spi_config.hold_time     = 0;
  spi_config.spi_mode      = MASTER;
  spi_config.is_lsb        = MSB_FIRST;
  spi_config.comm_mode     = TX;
  spi_config.spi_size      = 8;
  spi_config.ncs_type      = HARDWARE_NCS;

  SPI_Config(&spi_config);
  SPI_Transceive(&spi_config, &buf);

  Wait_Till_TX_Complete(&spi_config, 1);
  SPI_Disable(&spi_config);
```

(continues on next page)

(continued from previous page)

```
  return 0;
  }
```

This example demonstrates how to configure and use SPI1 as SLAVE in RX mode.

```
#include "spi.h"
#include "io.h"
#define DATA_LEN 10 //  Number of data

int main()
  {
  uint8_t rx_data[DATA_LEN];
  spi_buffer buf = {.tx_buf = NULL, .rx_buf = rx_data, .data_size = 8,␣
↪.len = DATA_LEN};
  SPI_Config_t spi_config;
  spi_config.spi_number    = 1;
  spi_config.spi_clk_mode  = MODE_0;
  spi_config.setup_time    = 0;
  spi_config.hold_time     = 0;
  spi_config.spi_mode      = SLAVE;
  spi_config.is_lsb        = MSB_FIRST;
  spi_config.comm_mode     = RX;
  spi_config.spi_size      = 8;
  spi_config.ncs_type      = HARDWARE_NCS;
  SPI_Config(&spi_config);
  SPI_Transceive(&spi_config, &buf);
  printf("Received Data:\n");
   for (uint32_t i=0 ; i < DATA_LEN  ; i++){
       printf("%d \n", rx_data[i]);
   }
  return 0;
  }
```

Output

```
Received Data:
1
2
3
4
5
6
7
8
9
10
```

SPI Master(RX) and Slave(TX) Example code

This example demonstrates how to configure and use SPI0 as `MASTER` in `RX` mode.

```
#include "spi.h"
#include "io.h"
#define DATA_LEN 10 //  Number of data
  int main()
  {
   uint8_t rx_data[DATA_LEN];
   spi_buffer buf = {.tx_buf = NULL, .rx_buf = rx_data, .data_size = 8,
↪ .len = DATA_LEN};

   SPI_Config_t spi_config;
   spi_config.spi_freq      = 6000000;
   spi_config.spi_number    = 0;
   spi_config.spi_clk_mode  = MODE_0;
   spi_config.setup_time    = 0;
   spi_config.hold_time     = 0;
   spi_config.spi_mode      = MASTER;
   spi_config.is_lsb        = MSB_FIRST;
   spi_config.comm_mode     = RX;
   spi_config.spi_size      = 8;
   spi_config.ncs_type      = HARDWARE_NCS;
```

(continues on next page)

(continued from previous page)

```
  SPI_Config(&spi_config);
  SPI_Transceive(&spi_config, &buf);

  SPI_Disable(&spi_config);
  Flush_RX_FIFO(&spi_config);
  printf("Received Data:\n");
  for (uint32_t i=0 ; i < DATA_LEN  ; i++){
      printf("%d \n", rx_data[i]);
  }
 return 0;
 }
```

This example demonstrates how to configure and use SPI1 as `SLAVE` in `TX` mode.

```
#include "spi.h"
#include "io.h"
#define DATA_LEN 10 //  Number of data

int main()
  {
  uint8_t rx_data[DATA_LEN];
  spi_buffer buf = {.tx_buf = NULL, .rx_buf = rx_data, .data_size = 8,␣
↪.len = DATA_LEN};
  SPI_Config_t spi_config;
  spi_config.spi_number    = 1;
  spi_config.spi_clk_mode  = MODE_0;
  spi_config.setup_time    = 0;
  spi_config.hold_time     = 0;
  spi_config.spi_mode      = SLAVE;
  spi_config.is_lsb        = MSB_FIRST;
  spi_config.comm_mode     = RX;
  spi_config.spi_size      = 8;
  spi_config.ncs_type      = HARDWARE_NCS;
  SPI_Config(&spi_config);
  SPI_Transceive(&spi_config, &buf);
  printf("Received Data:\n");
   for (uint32_t i=0 ; i < DATA_LEN  ; i++){
```

(continues on next page)

(continued from previous page)

```
      printf("%d \n", rx_data[i]);
   }
  return 0;
  }
```

Output

```
Received Data:
0 // Slave always transmit `0` as the first byte.
1
2
3
4
5
6
7
8
9
```

SPI Master(FULL-DUPLEX) and Slave(FULL-DUPLEX) Example code

This example demonstrates how to configure and use SPI0 as `MASTER` in `FULL_DUPLEX` mode.

```
#include "spi.h"
#include "io.h"
#define DATA_LEN 10 //  Number of data

int main()
{
  uint8_t tx_data[DATA_LEN];
  uint8_t rx_data[DATA_LEN];
  for (uint32_t i = 0; i < DATA_LEN ; i++){
      tx_data[i] = i+1;
  }
```

(continues on next page)

(continued from previous page)

```
  spi_buffer buf = {.tx_buf = tx_data, .rx_buf = rx_data, .data_size =␣
↪8, .len = DATA_LEN};

  SPI_Config_t spi_config;
  spi_config.spi_freq      = 6000000;
  spi_config.spi_number    = 0;
  spi_config.spi_clk_mode  = MODE_0;
  spi_config.setup_time    = 0;
  spi_config.hold_time     = 0;
  spi_config.spi_mode      = MASTER;
  spi_config.is_lsb        = MSB_FIRST;
  spi_config.comm_mode     = FULL_DUPLEX;
  spi_config.spi_size      = 8;
  spi_config.ncs_type      = SOFTWARE_NCS;

  SPI_Config(&spi_config);
  Software_Control_NCS(&spi_config, 0);
  SPI_Transceive(&spi_config, &buf);
  Wait_Till_TX_Complete(&spi_config, 1);
  Software_Control_NCS(&spi_config, 1);
  SPI_Disable(&spi_config);

  return 0;
}
```

This example demonstrates how to configure and use SPI1 as `SLAVE` in `FULL-DUPLEX` mode.

```
#include "spi.h"
#include "io.h"
#define DATA_LEN 10 //  Number of data

int main()
{
   uint8_t tx_data[DATA_LEN];
   uint8_t rx_data[DATA_LEN];
   for (uint32_t i = 0; i < DATA_LEN ; i++){
       tx_data[i] = i+1;
```

(continues on next page)

(continued from previous page)

```
    }

    spi_buffer buf = {.tx_buf = tx_data, .rx_buf = rx_data, .data_size␣
↪= 8, .len = DATA_LEN};

    SPI_Config_t spi_config;
    spi_config.spi_number    = 1;
    spi_config.spi_clk_mode  = MODE_0;
    spi_config.setup_time    = 0;
    spi_config.hold_time     = 0;
    spi_config.spi_mode      = SLAVE;
    spi_config.is_lsb        = MSB_FIRST;
    spi_config.comm_mode     = FULL_DUPLEX;
    spi_config.spi_size      = 8;
    spi_config.ncs_type      = SOFTWARE_NCS;

    SPI_Config(&spi_config);
    SPI_Transceive(&spi_config, &buf);

  printf ("Received Data:\n");
  for (uint32_t i=0 ; i < DATA_LEN  ; i++){
      printf ("%d \n", rx_data[i]);
  }
  return 0;
}
```

Output

```
Received Data by Master:
0
1
2
3
4
5
6
```

(continues on next page)

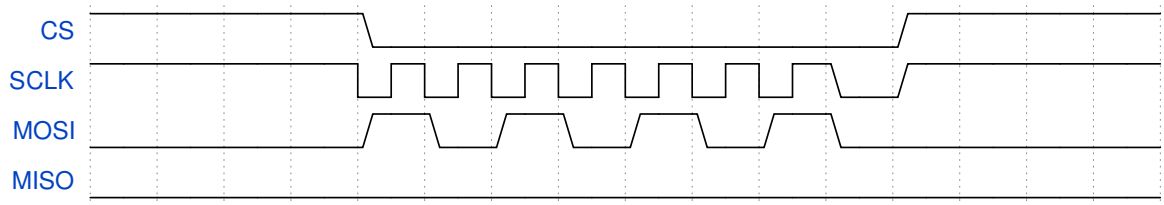
(continued from previous page)

```
7
8
9

 Received Data by Slave:
1
2
3
4
5
6
7
8
9
10
```

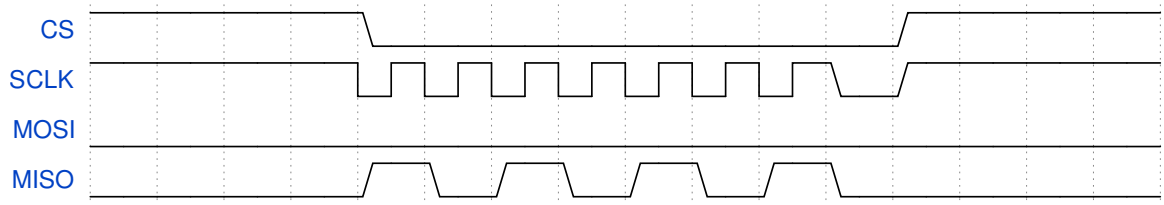
SPI Protocol Timing Diagram

The following diagram illustrates an example SPI protocol timing diagram when clock mode is 3:

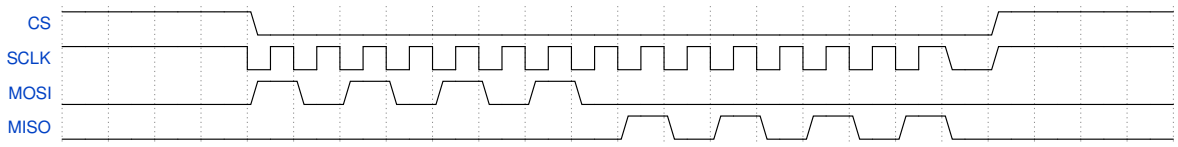
Simplex TX



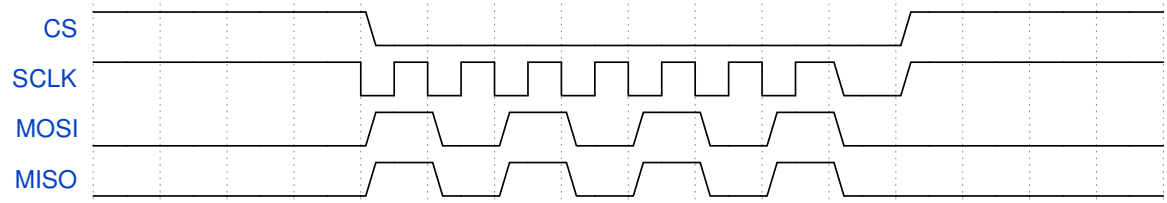
Simplex RX



Half-Duplex



Full-Duplex



How to Interpret the Diagram

- **CS (Chip Select):** Goes low to initiate communication.
- **SCLK (Serial Clock):** Provides clock pulses for data transmission.
- **MOSI (Master Out Slave In):** Data sent by the master (*0xA*, *0xB*).
- **MISO (Master In Slave Out):** Data received by the master (*0xA*, *0xB*).

Troubleshooting

1. **Clock Polarity and Phase Mismatch (CPOL/CPHA)**

   Our SPI supports only two valid modes: {CPOL = 0, CPHA = 0} and {CPOL = 1, CPHA = 1}.

   If the master and slave are not configured with the same mode, data may be shifted, corrupted, or not sampled correctly.

2. **Incorrect Clock Frequency**

   If the SPI clock frequency is too high for the slave device, issues such as the following may occur:

   - Data corruption
   - Intermittent communication failures
   - Timing violations

3. **SPI Slave Limitations**

- In Slave mode, `Half-Duplex` operation is not supported.
- For TX and FULL_DUPLEX, the first transmitted byte will always be *0*, followed by the user-provided data.

4. **Incorrect Configuration Between Master and Slave**

   Ensure that both the master and slave are configured identically, including settings such as:

   - Bit order (*LSB_FIRST* or *MSB_FIRST*)
   - Transfer mode
   - CPOL/CPHA mode
   - Frame size and other configuration parameters

5. **Chip Select Handling**

   When using software-controlled chip select, ensure that the function `Software_Control_NCS(SPI_Config_t *spi_config, bool ncs_val)` is used to assert and deassert CS when the SPI is configured with *SOFTWARE_NCS*.

7.1.18 UART (Universal Asynchronous Receiver-Transmitter)

Functions Usage

UART can be used in serial communication with peripheral devices, debugging, logging data, or interfacing with sensors and controllers. Its full-duplex capability makes it suitable for applications requiring simultaneous data transmission and reception.

1. **Initializing UART**

   To Initialize the particular UART Port with specific baud rate, data bits, stop bits, parity, delay, data size and timeout:

```
UART_Config_t uart_config;  // Structure variable which hold␣
↪the UART's specifications
uart_config.uart_num = 4;  // Specifies the UART instance
uart_config.baudrate = 115200;  // Specifies the baudrate
uart_config.char_size = 8;  // Specifies the Char size
uart_config.delay = 0; // Specifies the delay
uart_config.parity = 0; // Specifies the parity
uart_config.stop_bits = 1; // Specifies the stopbits
```

(continues on next page)

(continued from previous page)

```
uart_config.transfer_mode = 2; // Specifies the transfer mode
uart_config.receive_mode = 2; // Specifies the receive mode
uart_config.pullup = 0; // Specifies the pullup
uart_config.timeout = 100;  // Specifies the timeout function
UART_Init(&uart_config); // Initializing the particular uart␣
↪instance
```

2. **Transmitting Data via UART**

   To transmit data from UART:

   Example Code:

```
char str[] = "mind";  // Automatically null-terminated
uint32_t x = strlen(str);  // Use standard function
struct uart_buf tx = {.uart_data = str, .len = x};
UART_Write(&uart_config,&tx); /*Transmitting the data via the␣
↪specified UART Port*/
```

3. **Read Data via UART**

   To read the received data from UART:

   Example Code:

```
char new[100] = {0};
struct uart_buf rx = {.uart_data = new, .len = x};
UART_Read(&uart_config,&rx);
```

UART Sample Application

This example shows how to initialize UART, transmit data, and receive data. This code is a example of *uartloopback* application (so short rx and tx of the same uart Port).

```
#include "string_functions.h"
#include "uart.h"  // Included to access UART peripheral API's
#include "io.h"   // Included to access functions for basic IO␣
↪operations such as printf,etc

 void main()
 {
```

(continues on next page)

(continued from previous page)

```
  UART_Config_t uart_config;  // Structure variable which hold the␣
↪UART's specifications
  uart_config.uart_num = 4;  // Specifies the UART instance
  uart_config.baudrate = 115200;  // Specifies the baudrate
  uart_config.char_size = 8;  // Specifies the Char size
  uart_config.delay = 0; // Specifies the delay
  uart_config.parity = 0; // Specifies the parity
  uart_config.stop_bits = 1; // Specifies the stopbits
  uart_config.transfer_mode = 2; // Specifies the transfer mode
  uart_config.receive_mode = 2; // Specifies the receive mode
  uart_config.pullup = 0; // Specifies the pullup
  uart_config.timeout = 100;  // Specifies the timeout function
  UART_Init(&uart_config); // Initializing the particular uart␣
↪instance

  char str[] = "mind";  // Automatically null-terminated
  uint32_t x = StrLen(str);  // Use standard function
  struct uart_buf tx = {.uart_data = str, .len = x};
  while (1)
  {
      char new[100] = {0}; // Clear buffer before each read
      UART_Write(&uart_config, &tx);  // Transmit
      struct uart_buf rx = {.uart_data = new, .len = x};
      UART_Write_Wait(&uart_config);
      UART_Read(&uart_config, &rx);   // Receive
      printf("\nReceived string: ");
      for (uint32_t i = 0; i < x; i++) {
          printf("%c", new[i]);
      }
      printf("\n");
  }
}
```

Output

```
Received string:
mind
mind
mind
mind
mind
```

Troubleshooting

1. **Data Corruption** If data appears garbled or corrupted:
   - **Verify Baud Rate**: Ensure that the baud rate matches between UART devices.
   - **Check Parity and Stop Bits**: Confirm that both devices are set to the same parity and stop bit configuration.
2. **No Data Transmission or Reception** If data is not being transmitted or received:
   - **Check Connections**: Verify that TX and RX lines are correctly connected and not swapped.
   - **Ensure Proper Initialization**: Confirm that the UART Port is correctly initialized with the required settings.
3. **Framing Errors** If framing errors occur frequently:
   - **Verify Data Bits**: Check that both devices are using the same data bit configuration (e.g., 8 data bits).
   - **Check Signal Quality**: Inspect for signal interference or poor connections that could disrupt communication.
4. **Slow Transmission Speeds** If data transfer is unexpectedly slow:
   - **Optimize Baud Rate**: Set the baud rate as high as possible within the limits of the application.
   - **Reduce Delays**: Ensure that any delay settings are configured appropriately to avoid unnecessary pauses in communication.

7.1.19 Watchdog Timer

Example Code for Hard reset

```
#include "wdtimer.h"

  int main()
  {
      wdtimer_start(HARD_RESET, 0xABA9500);
      return 0;
  }
```

Output

```
Boot message
```

Example Code for Soft reset

```
#include "wdtimer.h"

  int main()
  {
      wdtimer_start(SOFT_RESET, 0);
      return 0;
  }
```

Output

```
Boot message
```

7.1.20 AES (Advanced Encryption Standard)

Example Code

Similar execution can also be perfomed for other modes. Please ensure that the input is a block of the order 16.

```
#include "aes.h"            /* Included to access AES API's */
#include "crypto_defines.h" /* Included to access AES related␣
↪constants */
#include "io.h"             /* Included to access IO operations API's */
#include "errors.h"         /* Included to access errors codes */

/* AES mode of operation. Change this to switch modes. */
#define AES_SELECTED_MODE AES_CFB

/* Number of test blocks in the plaintext test vector */
#define AES_TEST_NUM_BLOCKS 3

/* Total plaintext length in bytes */
#define AES_TEST_PLAINTEXT_LEN (AES_TEST_NUM_BLOCKS * AES_BLOCK_SIZE) /
↪* 48 bytes */

int main(void)
{
    /* 128-bit AES secret key */
    uint8_t aes_key_128bit[AES128_KEY_SIZE] = {0x2B, 0x7E, 0x15, 0x16,␣
↪0x28, 0xAE, 0xD2, 0xA6,
                                               0xAB, 0xF7, 0x15, 0x88,␣
↪0x09, 0xCF, 0x4F, 0x3C};

    /* 128-bit Initialization Vector (IV) */
    uint8_t aes_iv_128bit[AES_IV_SIZE] = {0x00, 0x01, 0x02, 0x03, 0x04,
↪ 0x05, 0x06, 0x07,
                                          0x08, 0x09, 0x0A, 0x0B, 0x0C,␣
↪0x0D, 0x0E, 0x0F};

    /* 48-byte (3-block) plaintext test input */
    uint8_t plaintext[AES_TEST_PLAINTEXT_LEN] = {
```

(continues on next page)

(continued from previous page)

```
        0x6B, 0xC1, 0xBE, 0xE2, 0x2E, 0x40, 0x9F, 0x96,
        0xE9, 0x3D, 0x7E, 0x11, 0x73, 0x93, 0x17, 0x2A,
        0xAE, 0x2D, 0x8A, 0x57, 0x1E, 0x03, 0xAC, 0x9C,
        0x9E, 0xB7, 0x6F, 0xAC, 0x45, 0xAF, 0x8E, 0x51,
        0x30, 0xC8, 0x1C, 0x46, 0xA3, 0x5C, 0xE4, 0x11,
        0xE5, 0xFB, 0xC1, 0x19, 0x1A, 0x0A, 0x52, 0xEF};

    uint16_t key_length_bits   = (uint16_t)(sizeof(aes_key_128bit) *␣
↪8U);
    size_t   plaintext_len_bits = AES_TEST_PLAINTEXT_LEN * 8U;

    uint8_t single_shot_ciphertext[AES_TEST_PLAINTEXT_LEN];
    uint8_t single_shot_decrypted[AES_TEST_PLAINTEXT_LEN];
    uint8_t multi_shot_ciphertext[AES_TEST_PLAINTEXT_LEN];
    uint8_t multi_shot_decrypted[AES_TEST_PLAINTEXT_LEN];

    AES_Config_t aes_config;

    printf("\r\nPlaintext         (%d bytes) : ", AES_TEST_PLAINTEXT_
↪LEN);
    for (size_t i = 0; i < AES_TEST_PLAINTEXT_LEN; i++)
        printf("%02x", plaintext[i]);
    printf("\r\nPlaintext Length             : %d bits (%d bytes)\r\n",
        plaintext_len_bits, AES_TEST_PLAINTEXT_LEN);
    printf("Key               (16 bytes) : ");
    for (size_t i = 0; i < sizeof(aes_key_128bit); i++)
        printf("%02x", aes_key_128bit[i]);
    printf("\r\nKey Length                   : %d bits\r\n", key_
↪length_bits);
    printf("IV                (16 bytes) : ");
    for (size_t i = 0; i < sizeof(aes_iv_128bit); i++)
        printf("%02x", aes_iv_128bit[i]);
    printf("\r\nMode                         : AES_CFB\r\n");

    /* ======================= SINGLE SHOT - ENCRYPT␣
↪==================== */
```

(continues on next page)

(continued from previous page)

```
    printf("\r\nRunning Single-Shot Encryption...\r\n");

    aes_config.aes_output          = single_shot_ciphertext;
    aes_config.input_text          = plaintext;
    aes_config.key                 = aes_key_128bit;
    aes_config.iv                  = aes_iv_128bit;
    aes_config.input_len_bits      = plaintext_len_bits;
    aes_config.key_len_bits        = key_length_bits;
    aes_config.mode                = AES_SELECTED_MODE;
    aes_config.encrypt_or_decrypt  = AES_ENCRYPT;
    aes_config.iterated_length_bits = 0U;

    AES_Run(&aes_config);

    printf("Ciphertext        (%d bytes) : ", AES_TEST_PLAINTEXT_LEN);
    for (size_t i = 0; i < AES_TEST_PLAINTEXT_LEN; i++)
        printf("%02x", single_shot_ciphertext[i]);
    printf("\r\nAES_Run Encrypt Successful\r\n");

    /* ======================= SINGLE SHOT - DECRYPT␣
↪====================== */

    printf("\r\nRunning Single-Shot Decryption...\r\n");

    aes_config.aes_output          = single_shot_decrypted;
    aes_config.input_text          = single_shot_ciphertext;
    aes_config.key                 = aes_key_128bit;
    aes_config.iv                  = aes_iv_128bit;
    aes_config.input_len_bits      = plaintext_len_bits;
    aes_config.key_len_bits        = key_length_bits;
    aes_config.mode                = AES_SELECTED_MODE;
    aes_config.encrypt_or_decrypt  = AES_DECRYPT;
    aes_config.iterated_length_bits = 0U;

    AES_Run(&aes_config);

    printf("Decrypted Text    (%d bytes) : ", AES_TEST_PLAINTEXT_LEN);
```

(continues on next page)

(continued from previous page)

```
    for (size_t i = 0; i < AES_TEST_PLAINTEXT_LEN; i++)
        printf("%02x", single_shot_decrypted[i]);
    printf("\r\nAES_Run Decrypt Successful\r\n");

    /* ====================== MULTI SHOT - ENCRYPT␣
↪====================== */

    printf("\r\nRunning Multi-Shot Encryption...\r\n");

    for (uint8_t block_index = 0; block_index < AES_TEST_NUM_BLOCKS;␣
↪block_index++)
    {
        size_t block_offset_bytes = (size_t)block_index * AES_BLOCK_
↪SIZE;

        aes_config.aes_output          = multi_shot_ciphertext + block_
↪offset_bytes;
        aes_config.input_text          = plaintext + block_offset_
↪bytes;
        aes_config.key                 = aes_key_128bit;
        aes_config.iv                  = aes_iv_128bit;
        aes_config.input_len_bits      = AES_BLOCK_SIZE * 8U;
        aes_config.key_len_bits        = key_length_bits;
        aes_config.mode                = AES_SELECTED_MODE;
        aes_config.encrypt_or_decrypt  = AES_ENCRYPT;
        aes_config.iterated_length_bits = block_offset_bytes * 8U;

        printf("\r\nBlock %d:\r\n", block_index);
        printf("  Input Data      (%d bytes) : ", AES_BLOCK_SIZE);
        for (size_t i = 0; i < AES_BLOCK_SIZE; i++)
            printf("%02x", plaintext[block_offset_bytes + i]);
        printf("\r\n  Input Length              : %d bits (%d bytes)\
↪r\n",
            AES_BLOCK_SIZE * 8U, AES_BLOCK_SIZE);
        printf("  Processed So Far          : %d bits\r\n", block_
↪offset_bytes * 8U);
```

(continues on next page)

(continued from previous page)

```
        AES_Run(&aes_config);

        printf("  Ciphertext      (%d bytes) : ", AES_BLOCK_SIZE);
        for (size_t i = 0; i < AES_BLOCK_SIZE; i++)
            printf("%02x", multi_shot_ciphertext[block_offset_bytes +␣
↪i]);
        printf("\r\n  AES_Run Encrypt Block %d Successful\r\n", block_
↪index);
    }

    /* ====================== MULTI SHOT - DECRYPT␣
↪====================== */

    printf("\r\nRunning Multi-Shot Decryption...\r\n");

    for (uint8_t block_index = 0; block_index < AES_TEST_NUM_BLOCKS;␣
↪block_index++)
    {
        size_t block_offset_bytes = (size_t)block_index * AES_BLOCK_
↪SIZE;

        aes_config.aes_output          = multi_shot_decrypted + block_
↪offset_bytes;
        aes_config.input_text          = multi_shot_ciphertext + block_
↪offset_bytes;
        aes_config.key                 = aes_key_128bit;
        aes_config.iv                  = aes_iv_128bit;
        aes_config.input_len_bits      = AES_BLOCK_SIZE * 8U;
        aes_config.key_len_bits        = key_length_bits;
        aes_config.mode                = AES_SELECTED_MODE;
        aes_config.encrypt_or_decrypt  = AES_DECRYPT;
        aes_config.iterated_length_bits = block_offset_bytes * 8U;

        printf("\r\nBlock %d:\r\n", block_index);
        printf("  Ciphertext      (%d bytes) : ", AES_BLOCK_SIZE);
        for (size_t i = 0; i < AES_BLOCK_SIZE; i++)
            printf("%02x", multi_shot_ciphertext[block_offset_bytes +␣
```

(continues on next page)

(continued from previous page)

```
↪i]);
        printf("\r\n  Input Length                : %d bits (%d bytes)\
↪r\n",
            AES_BLOCK_SIZE * 8U, AES_BLOCK_SIZE);
        printf("  Processed So Far            : %d bits\r\n", block_
↪offset_bytes * 8U);

        AES_Run(&aes_config);

        printf("  Decrypted Text  (%d bytes) : ", AES_BLOCK_SIZE);
        for (size_t i = 0; i < AES_BLOCK_SIZE; i++)
            printf("%02x", multi_shot_decrypted[block_offset_bytes +␣
↪i]);
        printf("\r\n  AES_Run Decrypt Block %d Successful\r\n", block_
↪index);
    }

    return SUCCESS;
}
```

Output

```
Plaintext         (48 bytes) :␣
↪6bc1bee22e409f96e93d7e117393172aae2d8a571e03a
                              ␣
↪c9c9eb76fac45af8e5130c81c46a35ce411e5fbc1191a0a52ef
Plaintext Length             : 384 bits (48 bytes)
Key               (16 bytes) : 2b7e151628aed2a6abf7158809cf4f3c
Key Length                   : 128 bits
IV                (16 bytes) : 000102030405060708090a0b0c0d0e0f
Mode                         : AES_CFB

Running Single-Shot Encryption...
Ciphertext        (48 bytes) :␣
↪3b3fd92eb72dad20333449f8e83cfb4ac8a64537a0b3a
                              ␣
```

(continues on next page)

(continued from previous page)

```
↪93fcde3cdad9f1ce58b26751f67a3cbb140b1808cf187a4f4df
AES_Run Encrypt Successful

Running Single-Shot Decryption...
Decrypted Text    (48 bytes) :␣
↪6bc1bee22e409f96e93d7e117393172aae2d8a571e03a
                              ␣
↪c9c9eb76fac45af8e5130c81c46a35ce411e5fbc1191a0a52ef
AES_Run Decrypt Successful

Running Multi-Shot Encryption...

Block 0:
Input Data      (16 bytes) : 6bc1bee22e409f96e93d7e117393172a
Input Length               : 128 bits (16 bytes)
Processed So Far           : 0 bits
Ciphertext      (16 bytes) : 3b3fd92eb72dad20333449f8e83cfb4a
AES_Run Encrypt Block 0 Successful

Block 1:
Input Data      (16 bytes) : ae2d8a571e03ac9c9eb76fac45af8e51
Input Length               : 128 bits (16 bytes)
Processed So Far           : 128 bits
Ciphertext      (16 bytes) : c8a64537a0b3a93fcde3cdad9f1ce58b
AES_Run Encrypt Block 1 Successful

Block 2:
Input Data      (16 bytes) : 30c81c46a35ce411e5fbc1191a0a52ef
Input Length               : 128 bits (16 bytes)
Processed So Far           : 256 bits
Ciphertext      (16 bytes) : 26751f67a3cbb140b1808cf187a4f4df
AES_Run Encrypt Block 2 Successful

Running Multi-Shot Decryption...

Block 0:
Ciphertext      (16 bytes) : 3b3fd92eb72dad20333449f8e83cfb4a
```

(continues on next page)

(continued from previous page)

```
Input Length               : 128 bits (16 bytes)
Processed So Far           : 0 bits
Decrypted Text  (16 bytes) : 6bc1bee22e409f96e93d7e117393172a
AES_Run Decrypt Block 0 Successful

Block 1:
Ciphertext      (16 bytes) : c8a64537a0b3a93fcde3cdad9f1ce58b
Input Length               : 128 bits (16 bytes)
Processed So Far           : 128 bits
Decrypted Text  (16 bytes) : ae2d8a571e03ac9c9eb76fac45af8e51
AES_Run Decrypt Block 1 Successful

Block 2:
Ciphertext      (16 bytes) : 26751f67a3cbb140b1808cf187a4f4df
Input Length               : 128 bits (16 bytes)
Processed So Far           : 256 bits
Decrypted Text  (16 bytes) : 30c81c46a35ce411e5fbc1191a0a52ef
AES_Run Decrypt Block 2 Successful
```

Example Code with PSA APIs

Similar execution can also be perfomed for other modes. Please ensure that the input is a block of the order 16.

```c
#include "psa.h" /* Included to access PSA Crypto API's */
#include "io.h"  /* Included to access printf etc */

int main(void)
{
    /* 128-bit AES key */
    uint8_t key[PSA_MG_AES_KEY_SIZE_128] = {0x00, 0x01, 0x02, 0x03,
                                            0x04, 0x05, 0x06, 0x07,
                                            0x08, 0x09, 0x0A, 0x0B,
                                            0x0C, 0x0D, 0x0E, 0x0F};

    /* 128-bit plaintext input */
    uint8_t input[PSA_MG_AES_BLOCK_LENGTH] = {0x00, 0x11, 0x22, 0x33,
```

(continues on next page)

(continued from previous page)

```
                                        0x44, 0x55, 0x66, 0x77,
                                        0x88, 0x99, 0xAA, 0xBB,
                                        0xCC, 0xDD, 0xEE, 0xFF};

    /* 128-bit IV */
    uint8_t iv[PSA_MG_AES_IV_SIZE] = {0x00, 0x01, 0x02, 0x03,
                                      0x04, 0x05, 0x06, 0x07,
                                      0x08, 0x09, 0x0A, 0x0B,
                                      0x0C, 0x0D, 0x0E, 0x0F};

    /* Single-shot buffers */
    uint8_t single_shot_ciphertext[sizeof(input) + PSA_MG_AES_IV_SIZE]␣
↪= {0};
    uint8_t single_shot_decrypted[sizeof(input)] = {0};
    size_t  single_shot_cipher_len    = 0;
    size_t  single_shot_decrypted_len = 0;

    /* Multi-shot encrypt buffers */
    uint8_t multi_shot_ciphertext[sizeof(input)] = {0};
    size_t  multi_shot_update_len  = 0;
    size_t  multi_shot_finish_len  = 0;

    /* Multi-shot decrypt buffers */
    uint8_t multi_shot_decrypted[sizeof(input)] = {0};
    size_t  multi_shot_dec_update_len = 0;
    size_t  multi_shot_dec_finish_len = 0;

    psa_key_attributes_t    key_attributes = PSA_KEY_ATTRIBUTES_INIT;
    psa_cipher_operation_t  operation      = PSA_CIPHER_OPERATION_INIT;
    psa_key_id_t            key_handle     = 0;
    psa_status_t            status;

    psa_crypto_init();

    printf("\r\nPlaintext         (%d bytes) : ", sizeof(input));
    for (size_t i = 0; i < sizeof(input); i++)
        printf("%02x", input[i]);
```

(continues on next page)

(continued from previous page)

```
    printf("\r\nInput Length                : %d bytes\r\n",␣
↪sizeof(input));
    printf("Key               (16 bytes) : ");
    for (size_t i = 0; i < sizeof(key); i++)
        printf("%02x", key[i]);
    printf("\r\nKey Length                  : %d bits\r\n", PSA_MG_
↪AES_KEY_SIZE_128_BITS);
    printf("IV                (16 bytes) : ");
    for (size_t i = 0; i < sizeof(iv); i++)
        printf("%02x", iv[i]);
    printf("\r\nAlgorithm                   : PSA_ALG_CTR\r\n");

    /* Key setup */
    psa_set_key_usage_flags(&key_attributes, PSA_KEY_USAGE_ENCRYPT |␣
↪PSA_KEY_USAGE_DECRYPT);
    psa_set_key_algorithm(&key_attributes, PSA_ALG_CTR);
    psa_set_key_type(&key_attributes, PSA_KEY_TYPE_AES);
    psa_set_key_bits(&key_attributes, PSA_MG_AES_KEY_SIZE_128_BITS);

    status = psa_import_key(&key_attributes, key, PSA_MG_AES_KEY_SIZE_
↪128, &key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_import_key Failed. Status: %d\r\n", status);
        return status;
    }
    printf("\r\npsa_import_key Successful\r\n");

    /* ======================= SINGLE SHOT - ENCRYPT␣
↪====================== */

    printf("\r\nRunning Single-Shot Encryption...\r\n");

    status = psa_cipher_encrypt(key_handle, PSA_ALG_CTR, input,␣
↪sizeof(input),
                                single_shot_ciphertext, sizeof(single_
↪shot_ciphertext),
```

(continues on next page)

(continued from previous page)

```
                                &single_shot_cipher_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_encrypt Failed. Status: %d\r\n", status);
        return status;
    }

    printf("Ciphertext        (%d bytes) : ", single_shot_cipher_len);
    for (size_t i = 0; i < single_shot_cipher_len; i++)
        printf("%02x", single_shot_ciphertext[i]);
    printf("\r\nOutput Length                : %d bytes\r\n", single_
↪shot_cipher_len);
    printf("psa_cipher_encrypt Successful\r\n");

    /* ======================= SINGLE SHOT - DECRYPT␣
↪====================== */

    printf("\r\nRunning Single-Shot Decryption...\r\n");

    status = psa_cipher_decrypt(key_handle, PSA_ALG_CTR, single_shot_
↪ciphertext,
                                sizeof(single_shot_ciphertext), single_
↪shot_decrypted,
                                sizeof(single_shot_decrypted), &single_
↪shot_decrypted_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_decrypt Failed. Status: %d\r\n", status);
        return status;
    }

    printf("Decrypted Text    (%d bytes) : ", single_shot_decrypted_
↪len);
    for (size_t i = 0; i < single_shot_decrypted_len; i++)
        printf("%02x", single_shot_decrypted[i]);
    printf("\r\nOutput Length                : %d bytes\r\n", single_
↪shot_decrypted_len);
```

(continues on next page)

(continued from previous page)

```
    printf("psa_cipher_decrypt Successful\r\n");

    /* ======================= MULTI SHOT - ENCRYPT␣
↪====================== */

    printf("\r\nRunning Multi-Shot Encryption...\r\n");

    status = psa_cipher_encrypt_setup(&operation, key_handle, PSA_ALG_
↪CTR);
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_setup Failed. Status: %d\r\n", status);
        return status;
    }
    printf("psa_cipher_setup Successful\r\n");

    status = psa_cipher_set_iv(&operation, iv, sizeof(iv));
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_set_iv Failed. Status: %d\r\n", status);
        return status;
    }
    printf("psa_cipher_set_iv Successful\r\n");

    status = psa_cipher_update(&operation, input, sizeof(input),
                               multi_shot_ciphertext, sizeof(multi_shot_
↪ciphertext),
                               &multi_shot_update_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_update Failed. Status: %d\r\n", status);
        return status;
    }

    printf("Ciphertext        (%d bytes) : ", multi_shot_update_len);
    for (size_t i = 0; i < multi_shot_update_len; i++)
        printf("%02x", multi_shot_ciphertext[i]);
```

(continues on next page)

(continued from previous page)

```
    printf("\r\nOutput Length                : %d bytes\r\n", multi_
↪shot_update_len);
    printf("psa_cipher_update Successful\r\n");

    status = psa_cipher_finish(&operation, multi_shot_ciphertext,␣
↪sizeof(multi_shot_ciphertext),
                               &multi_shot_finish_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_finish Failed. Status: %d\r\n", status);
        return status;
    }
    printf("Output Length (finish)       : %d bytes\r\n", multi_shot_
↪finish_len);
    printf("psa_cipher_finish Successful\r\n");

    status = psa_cipher_abort(&operation);
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_abort Failed. Status: %d\r\n", status);
        return status;
    }
    printf("psa_cipher_abort Successful\r\n");

    /* ======================== MULTI SHOT - DECRYPT␣
↪======================== */

    printf("\r\nRunning Multi-Shot Decryption...\r\n");

    status = psa_cipher_decrypt_setup(&operation, key_handle, PSA_ALG_
↪CTR);
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_setup Failed. Status: %d\r\n", status);
        return status;
    }
    printf("psa_cipher_setup Successful\r\n");
```

(continues on next page)

(continued from previous page)

```
    status = psa_cipher_set_iv(&operation, iv, sizeof(iv));
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_set_iv Failed. Status: %d\r\n", status);
        return status;
    }
    printf("psa_cipher_set_iv Successful\r\n");

    status = psa_cipher_update(&operation, multi_shot_ciphertext,␣
↪sizeof(multi_shot_ciphertext),
                               multi_shot_decrypted, sizeof(multi_shot_
↪decrypted),
                               &multi_shot_dec_update_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_update Failed. Status: %d\r\n", status);
        return status;
    }

    printf("Decrypted Text    (%d bytes) : ", multi_shot_dec_update_
↪len);
    for (size_t i = 0; i < multi_shot_dec_update_len; i++)
        printf("%02x", multi_shot_decrypted[i]);
    printf("\r\nOutput Length                 : %d bytes\r\n", multi_
↪shot_dec_update_len);
    printf("psa_cipher_update Successful\r\n");

    status = psa_cipher_finish(&operation, multi_shot_decrypted,␣
↪sizeof(multi_shot_decrypted),
                               &multi_shot_dec_finish_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_finish Failed. Status: %d\r\n", status);
        return status;
    }
    printf("Output Length (finish)       : %d bytes\r\n", multi_shot_
```

(continues on next page)

(continued from previous page)

```
↪dec_finish_len);
    printf("psa_cipher_finish Successful\r\n");

    status = psa_cipher_abort(&operation);
    if (status != PSA_SUCCESS)
    {
        printf("psa_cipher_abort Failed. Status: %d\r\n", status);
        return status;
    }
    printf("psa_cipher_abort Successful\r\n");

    return PSA_SUCCESS;
}
```

Output

```
Plaintext          (16 bytes) : 00112233445566778899aabbccddeeff
Input Length                  : 16 bytes
Key                (16 bytes) : 000102030405060708090a0b0c0d0e0f
Key Length                    : 128 bits
IV                 (16 bytes) : 000102030405060708090a0b0c0d0e0f
Algorithm                     : PSA_ALG_CTR

psa_import_key Successful

Running Single-Shot Encryption...
Ciphertext         (32 bytes) :␣
↪cf8cbdf4293147f7b45dfd7bd859797f4d4dbd8736ca56894f6613999b4e2116
Output Length                 : 32 bytes
psa_cipher_encrypt Successful

Running Single-Shot Decryption...
Decrypted Text     (16 bytes) : 00112233445566778899aabbccddeeff
Output Length                 : 16 bytes
psa_cipher_decrypt Successful
```

(continues on next page)

(continued from previous page)

```
Running Multi-Shot Encryption...
psa_cipher_setup Successful
psa_cipher_set_iv Successful
Ciphertext        (16 bytes) : 0a852986053b9632795a3ee30a8e04a5
Output Length                : 16 bytes
psa_cipher_update Successful
Output Length (finish)       : 0 bytes
psa_cipher_finish Successful
psa_cipher_abort Successful

Running Multi-Shot Decryption...
psa_cipher_setup Successful
psa_cipher_set_iv Successful
Decrypted Text    (16 bytes) : 00112233445566778899aabbccddeeff
Output Length                : 16 bytes
psa_cipher_update Successful
Output Length (finish)       : 0 bytes
psa_cipher_finish Successful
psa_cipher_abort Successful
```

Troubleshooting

1. Ensure that AES engine is in correct state before loading input.
2. Ensure the correct endianess before loading input abnd while checking output.

7.1.21 OTP (One-Time Programmable Memory)

Example Code

```c
#include "otp.h"

int main()
{
    OTP_Init();
```

(continues on next page)

(continued from previous page)

```
    OTP_Write(0, 1);
    OTP_Write(2, 1);
    OTP_Write(4, 1);
    OTP_Write(6, 1);

    OTP_Write(8, 1);
    OTP_Write(9, 1);
    OTP_Write(10, 1);
    OTP_Write(11, 1);

    printf("Data1 : %02x\n\r", OTP_Read(0));
    printf("Data2 : %02x", OTP_Read(8));

    return 0;
}
```

Output

When reading from the OTP, the output will be in little endian format.

```
Data1 :  0x55
Data2 :  0x0F
```

7.1.22 Rivest Shamir Adleman (RSA)

Example Code

```
#include "rsa.h"            /* Included to access RSA API's */
#include "crypto_defines.h" /* Included to access RSA-2048 related␣
↪constants */
#include "rsa_padding.h"    /* Included to access RSA Padding API's */
#include "io.h"             /* Included to access IO operations API's */
#include "errors.h"         /* Included to access errors codes */

int main(void)
```

(continues on next page)

(continued from previous page)

```
{
    /* Input message */
    uint8_t input[] = "Hello RSA!";

    /* 2048-bit public exponent (e = 65537) */
    uint8_t public_exponent[RSA_PUBLIC_EXPONENT_MAX_SIZE] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
↪0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,␣
```

(continues on next page)

(continued from previous page)

```
↪0x00, 0x00, 0x00, 0x01, 0x00, 0x01};

    /* 2048-bit private exponent */
    uint8_t private_exponent[RSA_PRIVATE_EXPONENT_SIZE] = {
        0x1d, 0xbc, 0xa9, 0x2e, 0x42, 0x45, 0xc2, 0xd5, 0x7b, 0xfb,␣
↪0xa7, 0x62, 0x10, 0xcc, 0x06, 0x02,
        0x9b, 0x50, 0x27, 0x53, 0xb7, 0xc8, 0x21, 0xa3, 0x2b, 0x79,␣
↪0x9f, 0xbd, 0x33, 0xc9, 0x8b, 0x49,
        0xdb, 0x10, 0x22, 0x6b, 0x1e, 0xac, 0x01, 0x43, 0xc8, 0x57,␣
↪0x4e, 0xf6, 0x52, 0x83, 0x3b, 0x96,
        0x37, 0x4d, 0x03, 0x4e, 0xf8, 0x4d, 0xaa, 0x55, 0x59, 0xc6,␣
↪0x93, 0xf3, 0xf0, 0x28, 0xd4, 0x97,
        0x16, 0xb8, 0x2e, 0x87, 0xa3, 0xf6, 0x82, 0xf2, 0x54, 0x24,␣
↪0x56, 0x3b, 0xd9, 0x40, 0x9d, 0xcf,
        0x9d, 0x08, 0x11, 0x05, 0x00, 0xf7, 0x3f, 0x74, 0x07, 0x6f,␣
↪0x28, 0xe7, 0x5e, 0x01, 0x99, 0xb1,
        0xf2, 0x9f, 0xa2, 0xf7, 0x0b, 0x9a, 0x31, 0x19, 0x0d, 0xec,␣
↪0x54, 0xe8, 0x72, 0xa7, 0x40, 0xe7,
        0xa1, 0xb1, 0xe3, 0x8c, 0x3d, 0x11, 0xbc, 0xa8, 0x26, 0x7d,␣
↪0xeb, 0x84, 0x2c, 0xef, 0x42, 0x62,
        0x23, 0x7a, 0xc8, 0x75, 0x72, 0x50, 0x68, 0xf3, 0x25, 0x63,␣
↪0xb4, 0x78, 0xac, 0xa8, 0xd6, 0xa9,
        0x9f, 0x34, 0xcb, 0x88, 0x76, 0xb9, 0x71, 0x45, 0xb2, 0xe8,␣
↪0x52, 0x9e, 0xc8, 0xad, 0xea, 0x83,
        0xea, 0xd4, 0xec, 0x63, 0xe3, 0xff, 0x2d, 0x17, 0xa2, 0xff,␣
↪0xef, 0xb0, 0x5c, 0x90, 0x2c, 0xa7,
        0xa9, 0x21, 0x68, 0x37, 0x8c, 0x89, 0xf7, 0x5c, 0x92, 0x8f,␣
↪0xc4, 0xf0, 0x70, 0x7e, 0x43, 0x48,
        0x7a, 0x4f, 0x47, 0xdf, 0x70, 0xca, 0xe8, 0x7e, 0x24, 0x27,␣
↪0x2c, 0x13, 0x6d, 0x3e, 0x98, 0xcf,
        0x59, 0x06, 0x6d, 0x41, 0xa3, 0xd0, 0x38, 0x85, 0x7d, 0x07,␣
↪0x3d, 0x8b, 0x4d, 0x2c, 0x27, 0xb8,
        0xf0, 0xea, 0x6b, 0xfa, 0x50, 0xd2, 0x63, 0x09, 0x1a, 0x4a,␣
↪0x18, 0xc6, 0x3f, 0x44, 0x6b, 0xc9,
        0xa6, 0x1e, 0x8c, 0x4a, 0x68, 0x83, 0x47, 0xb2, 0x43, 0x5e,␣
↪0xc8, 0xe7, 0x2e, 0xdd, 0xae, 0xa7};
```

(continues on next page)

(continued from previous page)

```
    /* 2048-bit modulus */
    uint8_t modulus[RSA_MODULUS_SIZE] = {
        0xe0, 0xb1, 0x4b, 0x99, 0xcd, 0x61, 0xcd, 0x3d, 0xb9, 0xc2,␣
↪0x07, 0x66, 0x68, 0x84, 0x13, 0x24,
        0xfa, 0x31, 0x74, 0xf3, 0x3c, 0xe6, 0x6f, 0xfd, 0x51, 0x43,␣
↪0x94, 0xd3, 0x41, 0x78, 0xd2, 0x9a,
        0x49, 0x49, 0x32, 0x76, 0xb6, 0x77, 0x72, 0x33, 0xe7, 0xd4,␣
↪0x6a, 0x3e, 0x68, 0xbc, 0x7c, 0xa7,
        0xe8, 0x99, 0xe9, 0x01, 0xd5, 0x4f, 0x6d, 0xee, 0x07, 0x49,␣
↪0xc3, 0xe4, 0x8d, 0xdf, 0x68, 0x68,
        0x58, 0x67, 0xee, 0x2a, 0xe6, 0x6d, 0xf8, 0x8e, 0xb5, 0x63,␣
↪0xf6, 0xdb, 0x13, 0x7a, 0x9f, 0x6b,
        0x17, 0x5a, 0x11, 0x2e, 0x0e, 0xda, 0x83, 0x68, 0xe8, 0x8e,␣
↪0x45, 0xef, 0xe1, 0xce, 0x14, 0xbc,
        0x60, 0x16, 0xd5, 0x26, 0x39, 0x62, 0x70, 0x66, 0xaf, 0x18,␣
↪0x72, 0xc7, 0x2f, 0x60, 0xb9, 0x16,
        0x1c, 0x1d, 0x23, 0x7e, 0xeb, 0x34, 0xb0, 0xf8, 0x41, 0xb3,␣
↪0xf0, 0x89, 0x6f, 0x9f, 0xe0, 0xe1,
        0x6b, 0x0f, 0x74, 0x35, 0x2d, 0x10, 0x12, 0x92, 0xcc, 0x46,␣
↪0x4a, 0x7e, 0x78, 0x61, 0xbb, 0xeb,
        0x86, 0xf6, 0xdf, 0x61, 0x51, 0xcb, 0x26, 0x54, 0x17, 0xc6,␣
↪0x6c, 0x56, 0x5e, 0xd8, 0x97, 0x4b,
        0xd8, 0xfc, 0x98, 0x4d, 0x5d, 0xdf, 0xd4, 0xeb, 0x91, 0xa3,␣
↪0xd5, 0x23, 0x4c, 0xe1, 0xb5, 0x46,
        0x7f, 0x3a, 0xde, 0x37, 0x5f, 0x80, 0x2e, 0xc0, 0x72, 0x93,␣
↪0xf1, 0x23, 0x6e, 0xfa, 0x30, 0x68,
        0xbc, 0x91, 0xb1, 0x58, 0x55, 0x1c, 0x87, 0x5c, 0x5d, 0xc0,␣
↪0xa9, 0xd6, 0xfa, 0x32, 0x1b, 0xf9,
        0x42, 0x1f, 0x08, 0xde, 0xac, 0x91, 0x0e, 0x35, 0xc1, 0xc2,␣
↪0x85, 0x49, 0xee, 0x8e, 0xed, 0x83,
        0x30, 0xcf, 0x70, 0x59, 0x5f, 0xf7, 0x0b, 0x94, 0xb4, 0x99,␣
↪0x07, 0xe2, 0x76, 0x98, 0xa9, 0xd9,
        0x11, 0xf7, 0xac, 0x07, 0x06, 0xaf, 0xcb, 0x1a, 0x4a, 0x39,␣
↪0xfe, 0xb3, 0x8b, 0x0a, 0x80, 0x49};

    /* Deterministic PSS salt */
    uint8_t pss_salt[20] = {
```

(continues on next page)

(continued from previous page)

```
        0xe1, 0x25, 0x6f, 0xc1, 0xee, 0xef, 0x81, 0x77, 0x3f, 0xdd,
        0x54, 0x65, 0x7e, 0x40, 0x07, 0xfd, 0xe6, 0xbc, 0xb9, 0xb1};

    /* PKCS#1 v1.5 encryption buffers */
    uint8_t PKCS_padded_input[RSA_BLOCK_SIZE]   = {0};
    uint8_t PKCS_ciphertext[RSA_BLOCK_SIZE]     = {0};
    uint8_t PKCS_decoded_output[RSA_BLOCK_SIZE] = {0};
    uint8_t PKCS_decrypted[RSA_BLOCK_SIZE]      = {0};
    size_t  PKCS_decrypted_len       = 0;

    /* OAEP encryption buffers */
    uint8_t OAEP_padded_input[RSA_BLOCK_SIZE]   = {0};
    uint8_t OAEP_ciphertext[RSA_BLOCK_SIZE]     = {0};
    uint8_t OAEP_decoded_output[RSA_BLOCK_SIZE] = {0};
    uint8_t OAEP_decrypted[RSA_BLOCK_SIZE]      = {0};
    size_t  OAEP_decrypted_len       = 0;

    /* PKCS#1 v1.5 signature buffers */
    uint8_t PKCS_sign_encoded[RSA_BLOCK_SIZE]   = {0};
    uint8_t PKCS_signature[RSA_SIGNATURE_SIZE]  = {0};
    uint8_t PKCS_verify_decoded[RSA_BLOCK_SIZE] = {0};

    /* PSS deterministic salt signature buffers */
    uint8_t PSS_det_sign_encoded[RSA_BLOCK_SIZE]   = {0};
    uint8_t PSS_det_signature[RSA_SIGNATURE_SIZE]  = {0};
    uint8_t PSS_det_verify_decoded[RSA_BLOCK_SIZE] = {0};

    /* PSS random salt signature buffers */
    uint8_t PSS_rand_sign_encoded[RSA_BLOCK_SIZE]   = {0};
    uint8_t PSS_rand_signature[RSA_SIGNATURE_SIZE]  = {0};
    uint8_t PSS_rand_verify_decoded[RSA_BLOCK_SIZE] = {0};
    uint8_t pss_rand_salt_len                       = 32; /* salt_
↪length passed to Sign controls how many random bytes are generated;
                                                            Verify␣
↪needs the same value as expected_salt_length */
    uint8_t result;
```

(continues on next page)

(continued from previous page)

```
    printf("\r\nInput Message                 : %s\r\n", input);
    printf("Input Length                  : %d bytes\r\n",␣
↪sizeof(input));
    printf("Key Size                      : 2048 bits (256 bytes)\r\n");
    printf("Public Exponent  (256 bytes) : ");
    for (uint16_t i = 0; i < sizeof(public_exponent); i++)
        printf("%02x", public_exponent[i]);
    printf("\r\nPrivate Exponent (256 bytes) : ");
    for (uint16_t i = 0; i < sizeof(private_exponent); i++)
        printf("%02x", private_exponent[i]);
    printf("\r\nModulus          (256 bytes) : ");
    for (uint16_t i = 0; i < sizeof(modulus); i++)
        printf("%02x", modulus[i]);
    printf("\r\n");

    /* ================ PKCS#1 v1.5 - ENCRYPT / DECRYPT␣
↪================= */

    printf("\r\nRunning PKCS#1 v1.5 Encryption...\r\n");

    RSAES_PKCS1_v1_5_Encrypt(PKCS_padded_input, sizeof(PKCS_padded_
↪input),
                             input, sizeof(input));
    printf("RSAES_PKCS1_v1_5_Encrypt Successful\r\n");

    RSA_Run(PKCS_ciphertext, PKCS_padded_input, public_exponent,␣
↪modulus);

    printf("Ciphertext       (256 bytes) : ");
    for (uint16_t i = 0; i < sizeof(PKCS_ciphertext); i++)
        printf("%02x", PKCS_ciphertext[i]);
    printf("\r\nOutput Length                 : %d bytes\r\n",␣
↪sizeof(PKCS_ciphertext));
    printf("RSA_Run Encrypt Successful\r\n");

    printf("\r\nRunning PKCS#1 v1.5 Decryption...\r\n");
```

(continues on next page)

(continued from previous page)

```
    RSA_Run(PKCS_decoded_output, PKCS_ciphertext, private_exponent,␣
↪modulus);
    printf("RSA_Run Decrypt Successful\r\n");

    result = RSAES_PKCS1_v1_5_Decrypt(PKCS_decrypted, &PKCS_decrypted_
↪len,
                                      PKCS_decoded_output, sizeof(PKCS_
↪decoded_output));
    if (result != SUCCESS)
    {
        printf("RSAES_PKCS1_v1_5_Decrypt Failed. Status: %d\r\n",␣
↪result);
        return result;
    }

    printf("Decrypted Output (%d bytes) : %s\r\n", PKCS_decrypted_len,␣
↪PKCS_decrypted);
    printf("Output Length                : %d bytes\r\n", PKCS_
↪decrypted_len);
    printf("RSAES_PKCS1_v1_5_Decrypt Successful\r\n");

    /* ==================== OAEP - ENCRYPT / DECRYPT␣
↪==================== */

    printf("\r\nRunning OAEP Encryption...\r\n");

    RSAES_OAEP_Encrypt(OAEP_padded_input, sizeof(OAEP_padded_input),
                       input, sizeof(input), NULL, 0);
    printf("RSAES_OAEP_Encrypt Successful\r\n");

    RSA_Run(OAEP_ciphertext, OAEP_padded_input, public_exponent,␣
↪modulus);

    printf("Ciphertext       (256 bytes) : ");
    for (uint16_t i = 0; i < sizeof(OAEP_ciphertext); i++)
        printf("%02x", OAEP_ciphertext[i]);
    printf("\r\nOutput Length                : %d bytes\r\n",␣
```

(continues on next page)

(continued from previous page)

```
↪sizeof(OAEP_ciphertext));
    printf("RSA_Run Encrypt Successful\r\n");

    printf("\r\nRunning OAEP Decryption...\r\n");

    RSA_Run(OAEP_decoded_output, OAEP_ciphertext, private_exponent,␣
↪modulus);
    printf("RSA_Run Decrypt Successful\r\n");

    result = RSAES_OAEP_Decrypt(OAEP_decoded_output, sizeof(OAEP_
↪decoded_output),
                                OAEP_decrypted, &OAEP_decrypted_len,␣
↪NULL, 0);
    if (result != SUCCESS)
    {
        printf("RSAES_OAEP_Decrypt Failed. Status: %d\r\n", result);
        return result;
    }

    printf("Decrypted Output (%d bytes) : %s\r\n", OAEP_decrypted_len,␣
↪OAEP_decrypted);
    printf("Output Length                : %d bytes\r\n", OAEP_
↪decrypted_len);
    printf("RSAES_OAEP_Decrypt Successful\r\n");

    /* ================ PKCS#1 v1.5 - SIGN / VERIFY␣
↪==================== */

    printf("\r\nRunning PKCS#1 v1.5 Sign...\r\n");

    RSASSA_PKCS1_v1_5_Sign(PKCS_sign_encoded, sizeof(PKCS_sign_
↪encoded),
                           input, sizeof(input));
    printf("RSASSA_PKCS1_v1_5_Sign Successful\r\n");

    RSA_Run(PKCS_signature, PKCS_sign_encoded, private_exponent,␣
↪modulus);
```

(continues on next page)

(continued from previous page)

```
    printf("Signature        (256 bytes) : ");
    for (uint16_t i = 0; i < sizeof(PKCS_signature); i++)
        printf("%02x", PKCS_signature[i]);
    printf("\r\nOutput Length                : %d bytes\r\n",␣
↪sizeof(PKCS_signature));
    printf("RSA_Run Sign Successful\r\n");

    printf("\r\nRunning PKCS#1 v1.5 Verify...\r\n");

    RSA_Run(PKCS_verify_decoded, PKCS_signature, public_exponent,␣
↪modulus);
    printf("RSA_Run Verify Successful\r\n");

    result = RSASSA_PKCS1_v1_5_Verify(PKCS_verify_decoded, sizeof(PKCS_
↪verify_decoded),
                                      input, sizeof(input));
    if (result != SUCCESS)
    {
        printf("RSASSA_PKCS1_v1_5_Verify Failed. Status: %d\r\n",␣
↪result);
        return result;
    }

    printf("Output Status                : SUCCESS - Signature Valid\r\
↪n");
    printf("RSASSA_PKCS1_v1_5_Verify Successful\r\n");

    /* ================ PSS - SIGN / VERIFY (DETERMINISTIC SALT)␣
↪======== */

    printf("\r\nRunning PSS Sign (Deterministic Salt)...\r\n");
    printf("Salt             (20 bytes)  : ");
    for (uint8_t i = 0; i < sizeof(pss_salt); i++)
        printf("%02x", pss_salt[i]);
    printf("\r\nSalt Length                  : %d bytes\r\n",␣
↪sizeof(pss_salt));
```

(continues on next page)

(continued from previous page)

```
    RSASSA_PSS_Sign(PSS_det_sign_encoded, sizeof(PSS_det_sign_encoded),
                    input, sizeof(input), pss_salt, sizeof(pss_salt),␣
↪1);
    printf("RSASSA_PSS_Sign Successful\r\n");

    RSA_Run(PSS_det_signature, PSS_det_sign_encoded, private_exponent,␣
↪modulus);

    printf("Signature        (256 bytes) : ");
    for (uint16_t i = 0; i < sizeof(PSS_det_signature); i++)
        printf("%02x", PSS_det_signature[i]);
    printf("\r\nOutput Length                : %d bytes\r\n",␣
↪sizeof(PSS_det_signature));
    printf("RSA_Run Sign Successful\r\n");

    printf("\r\nRunning PSS Verify (Deterministic Salt)...\r\n");

    RSA_Run(PSS_det_verify_decoded, PSS_det_signature, public_exponent,
↪ modulus);
    printf("RSA_Run Verify Successful\r\n");

    result = RSASSA_PSS_Verify(PSS_det_verify_decoded, sizeof(PSS_det_
↪verify_decoded),
                               input, sizeof(input), sizeof(pss_
↪salt));
    if (result != SUCCESS)
    {
        printf("RSASSA_PSS_Verify (deterministic salt) Failed. Status:
↪%d\r\n", result);
        return result;
    }

    printf("Output Status                : SUCCESS - Signature Valid\r\
↪n");
    printf("RSASSA_PSS_Verify (deterministic salt) Successful\r\n");
```

(continues on next page)

(continued from previous page)

```
    /* ================ PSS - SIGN / VERIFY (RANDOM SALT) ================ */

    printf("\r\nRunning PSS Sign (Random Salt)...\r\n");
    printf("Salt Length                  : %d bytes (randomly generated)\r\n", pss_rand_salt_len);

    RSASSA_PSS_Sign(PSS_rand_sign_encoded, sizeof(PSS_rand_sign_encoded),
                    input, sizeof(input), NULL, pss_rand_salt_len, 0);
    printf("RSASSA_PSS_Sign Successful\r\n");

    RSA_Run(PSS_rand_signature, PSS_rand_sign_encoded, private_exponent, modulus);

    printf("Signature        (256 bytes) : ");
    for (uint16_t i = 0; i < sizeof(PSS_rand_signature); i++)
        printf("%02x", PSS_rand_signature[i]);
    printf("\r\nOutput Length                : %d bytes\r\n", sizeof(PSS_rand_signature));
    printf("RSA_Run Sign Successful\r\n");

    printf("\r\nRunning PSS Verify (Random Salt)...\r\n");

    RSA_Run(PSS_rand_verify_decoded, PSS_rand_signature, public_exponent, modulus);
    printf("RSA_Run Verify Successful\r\n");

    result = RSASSA_PSS_Verify(PSS_rand_verify_decoded, sizeof(PSS_rand_verify_decoded),
                               input, sizeof(input), pss_rand_salt_len);
    if (result != SUCCESS)
    {
        printf("RSASSA_PSS_Verify (random salt) Failed. Status: %d\r\n", result);
        return result;
```

(continues on next page)

(continued from previous page)

```
    }

    printf("Output Status                : SUCCESS - Signature Valid\r\
↪n");
    printf("RSASSA_PSS_Verify (random salt) Successful\r\n");

    return SUCCESS;
}
```

Output

```
Input Message                : Hello RSA!
Input Length                 : 11 bytes
Key Size                     : 2048 bits (256 bytes)
Public Exponent  (256 bytes)  :␣
↪0000000000000000000000000000000000000000000000000
                              ␣
↪0000000000000000000000000000000000000000000000000
                              ␣
↪0000000000000000000000000000000000000000000000000
                              ␣
↪0000000000000000000000000000000000000000000000000
                              ␣
↪0000000000000000000000000000000000000000000000000
                               00000000000000000000000000000010001
Private Exponent (256 bytes)  :␣
↪1dbca92e4245c2d57bfba76210cc06029b502753b7c8
                              ␣
↪21a32b799fbd33c98b49db10226b1eac0143c8574ef6
                              ␣
↪52833b96374d034ef84daa5559c693f3f028d49716b8
                              ␣
↪2e87a3f682f25424563bd9409dcf9d08110500f73f74
                              ␣
↪076f28e75e0199b1f29fa2f70b9a31190dec54e872a7
                              ␣
```

(continues on next page)

(continued from previous page)

```
↪40e7a1b1e38c3d11bca8267deb842cef4262237ac875
                                    ␣
↪725068f32563b478aca8d6a99f34cb8876b97145b2e8
                                    ␣
↪529ec8adea83ead4ec63e3ff2d17a2ffefb05c902ca7
                                    ␣
↪a92168378c89f75c928fc4f0707e43487a4f47df70ca
                                    ␣
↪e87e24272c136d3e98cf59066d41a3d038857d073d8b
                                    ␣
↪4d2c27b8f0ea6bfa50d263091a4a18c63f446bc9a61e
                                     8c4a688347b2435ec8e72eddaea7
Modulus           (256 bytes)  :␣
↪e0b14b99cd61cd3db9c20766688841324fa3174f33ce
                                    ␣
↪66ffd514394d34178d29a49493276b6777233e7d46a3
                                    ␣
↪e68bc7ca7e899e901d54f6dee0749c3e48ddf6868586
                                    ␣
↪7ee2ae66df88eb563f6db137a9f6b175a112e0eda836
                                    ␣
↪8e88e45efe1ce14bc6016d52639627066af1872c72f6
                                    ␣
↪0b9161c1d237eeb34b0f841b3f0896f9fe0e16b0f743
                                    ␣
↪52d101292cc464a7e7861bbeb86f6df6151cb265417c
                                    ␣
↪66c565ed8974bd8fc984d5ddfd4eb91a3d5234ce1b54
                                    ␣
↪67f3ade375f802ec07293f1236efa3068bc91b158551
                                    ␣
↪c875c5dc0a9d6fa321bf9421f08deac910e35c1c2854
                                    ␣
↪9ee8eed8330cf70595ff70b94b49907e27698a9d911f
                                     7ac0706afcb1a4a39feb38b0a8049

Running PKCS#1 v1.5 Encryption...
```

(continues on next page)

(continued from previous page)

```
RSAES_PKCS1_v1_5_Encrypt Successful
Ciphertext       (256 bytes)  :␣
↪4145098110bd7cc06815fe933c161f62863ca257458d
                              ␣
↪6ee4b1a4b502c5564c58b8de6b122b0fb13b7e3437fa
                              ␣
↪e3b1da7600c401ab05b0e6bc695cfd46eb8ed9e405d1
                              ␣
↪2f3d3ebb677857e42423e5dfb11432c9aa39c80949e2
                              ␣
↪32190b7a4051a33c8d2bcf512a5a4d68822607ececfd
                              ␣
↪0568a61c50834a30367dd4b4e46eb1a5398846c8aa60
                              ␣
↪dfca458faad4240a28084e42c5da0949bf4485d0b440
                              ␣
↪4cd5e95a9e2d568b33b5e45f37db52a847eefd019b5e
                              ␣
↪4265e5e631fd6d06b40d6b55234924e3caa4adb589c6
                              ␣
↪09824a5233e78cac18ecef73dbf50a3521ce74ce162a
                              ␣
↪08824f13a7b62a506f76ab29c4e7b1e9dce304b20050
                                a50f3823c44ef789a5cc48f6c8347
Output Length                : 256 bytes
RSA_Run Encrypt Successful

Running PKCS#1 v1.5 Decryption...
RSA_Run Decrypt Successful
Decrypted Output (11 bytes)   : Hello RSA!
Output Length                : 11 bytes
RSAES_PKCS1_v1_5_Decrypt Successful

Running OAEP Encryption...
RSAES_OAEP_Encrypt Successful
Ciphertext       (256 bytes)  :␣
↪708d4119869f9e5b44ea3486eb855f3b1a6ef8c116ef
```

(continues on next page)

(continued from previous page)

```
                                    ␣
↪33417579757bdc02c3f78f31fe69bb9e1d5b0858dae3
                                    ␣
↪17eb8fb35070e6908c036b07b036b6087e584a1eef28
                                    ␣
↪9d544152582bfe0c99475b10e9c2092dba114dc9ba48
                                    ␣
↪75c9d2fe65b3c6a300db0e16b7b8b9ab2327c26c9c81
                                    ␣
↪0e8c02e0ee6babbb3af15d9751b43d4bfa3c326a314d
                                    ␣
↪9de189dd3c2c90195fa8454c2d87cd97a8f7f963cdc7
                                    ␣
↪66afe33401bb5242a6b1bd88f18ad9ef2ad6fde5639a
                                    ␣
↪2dfdfa9411067a663e4c8046dcc3e4d34ffa74ab7b45
                                    ␣
↪4d0d2013dd818a03e46a1b91a32aaf510b53e8924c64
                                    ␣
↪527f00bc13156eb6bc45dc3d7f2f3c93d2707d5ae07e
                                     ec401347888c41e1faf9ac9366e5
Output Length                  : 256 bytes
RSA_Run Encrypt Successful

Running OAEP Decryption...
RSA_Run Decrypt Successful
Decrypted Output (11 bytes)    : Hello RSA!
Output Length                  : 11 bytes
RSAES_OAEP_Decrypt Successful

Running PKCS#1 v1.5 Sign...
RSASSA_PKCS1_v1_5_Sign Successful
Signature        (256 bytes)  :␣
↪ccb63d6974574b361ff67e90f55567d7010b212c2dbe
                                    ␣
↪3b6d7d15c229916ce2edcac992c98136df0fdd175c34
                                    ␣
```

(continues on next page)

(continued from previous page)

```
↪c1a658406726e8aec7871f02a4fb3c809531bc27b849
                                    ␣
↪4819ec917350c73a66eade177eb1fce892e231fa36bf
                                    ␣
↪711b0aba3bbbc7ca89b56bea198ce5e54a210269a8af
                                    ␣
↪1a9cb47a213832c5630eda76aecf483208bf4b678efc
                                    ␣
↪611cb13a61f4a98856a8ab3407efa6d0eec934423b88
                                    ␣
↪8f9a689c5424e27c3d4fdfcfb5a9b6f4a1a6f562ee5e
                                    ␣
↪97fd712663f03cf77deb632b962ba43051465d863ef8
                                    ␣
↪6060b11148e1d28c796cecff4dd1c47fc2f35d779a31
                                    ␣
↪4dda39c5a2ce3305f8aef422804dec89a522a8b54238
                                     728dd6397d931386e7beb13e95f1
Output Length                   : 256 bytes
RSA_Run Sign Successful

Running PKCS#1 v1.5 Verify...
RSA_Run Verify Successful
Output Status                   : SUCCESS - Signature Valid
RSASSA_PKCS1_v1_5_Verify Successful

Running PSS Sign (Deterministic Salt)...
Salt              (20 bytes)    :␣
↪e1256fcleeef81773fdd54657e4007fde6bcb9b1
Salt Length                     : 20 bytes
RSASSA_PSS_Sign Successful
Signature         (256 bytes)   :␣
↪cbf1e7fa37673728155e62277bb4b9da8e5b661fecf3
                                    ␣
↪c67bc83a61c75830be70ca4ba38387b56e240ef0ea1c
                                    ␣
↪8a9fc1513ee72a403b6f1f438dc98d2d7bbdf3fbe8bc
```

(continues on next page)

(continued from previous page)

```
                                    ␣
↪d033af4cc6180c59135045f159a10bb08c375f773ef3
                                    ␣
↪d42cbbc88533feb67e1b863c8995c7869e8b72d52ed2
                                    ␣
↪1eab4a3cbee2b515e35d5d92bbf0f6deb7fc30b1a0cc
                                    ␣
↪1c69144e7469837a0521757558bffb4502654a637169
                                    ␣
↪6220a9c06a19d5efa682fdb6d10fffd4bc08b29522a8
                                    ␣
↪6c9deb88e05623c6c7906a18d7f75e84306f6b661e4e
                                    ␣
↪097b7e96b6d4c52f6e640c3174991a7a49d7081a7f93
                                    ␣
↪598a83aec02301cae5c1843f6185983ab9fc72a915ae
                                     3725c504d75ccc769e10c354bc02
Output Length                   : 256 bytes
RSA_Run Sign Successful

Running PSS Verify (Deterministic Salt)...
RSA_Run Verify Successful
Output Status                   : SUCCESS - Signature Valid
RSASSA_PSS_Verify (deterministic salt) Successful

Running PSS Sign (Random Salt)...
Salt Length                     : 32 bytes (randomly generated)
RSASSA_PSS_Sign Successful
Signature        (256 bytes)    :␣
↪ce662b965f2f97443632b88bd9191a4587096181fb73
                                    ␣
↪b003235cde8840d3873deefbeb26ee551f80da780e0d
                                    ␣
↪9ed68298fb528cf093c4d2d2489618f76e393cd4c546
                                    ␣
↪6a9396c0d2b9dade952a5272b8734615a5a18a76952b
                                    ␣
```

(continues on next page)

(continued from previous page)

```
↪658448b562ff68060d34a435e77f52fee18c3df60a1a
                                ␣
↪433f574e1eb509ec5ce2d4e0f87fe7e98c4f15ae6aa7
                                ␣
↪230984c7facf2257eb3ce6595995ecd8db1fc4baa839
                                ␣
↪202e27210ad8a28e9d391167d1946ffdf6ddd41a938b
                                ␣
↪f303a34350c72fd99f663012f8fb60d0c21aa27e4d00
                                ␣
↪f21d20f827239bdb4a40c7533addd40c9c91d3ddc923
                                ␣
↪94cba3fc0369934cb9ef89cd1411f4de2bb12f08d9d3
                                 40ea575c9129a738745dd5910c7
Output Length                  : 256 bytes
RSA_Run Sign Successful

Running PSS Verify (Random Salt)...
RSA_Run Verify Successful
Output Status                  : SUCCESS - Signature Valid
RSASSA_PSS_Verify (random salt) Successful
```

Example Code with PSA APIs

```
#include "psa.h"                /* Included to access PSA Crypto API's␣
↪*/
#include "io.h"                 /* Included to access printf, etc */
#include "bignum.h"             /* Included to access Bignum library␣
↪API's*/
#include "memory_functions.h"   /* Included to access memset API*/

int main(void)
{
    /* 128-byte input message */
    uint8_t input[128] = {
        0x5a, 0xf2, 0x83, 0xb1, 0xb7, 0x6a, 0xb2, 0xa6, 0x95, 0xd7,␣
```

(continues on next page)

(continued from previous page)

```
↪0x94, 0xc2, 0x3b, 0x35, 0xca, 0x73,
       0x71, 0xfc, 0x77, 0x9e, 0x92, 0xeb, 0xf5, 0x89, 0xe3, 0x04,␣
↪0xc7, 0xf9, 0x23, 0xd8, 0xcf, 0x97,
       0x63, 0x04, 0xc1, 0x98, 0x18, 0xfc, 0xd8, 0x9d, 0x6f, 0x07,␣
↪0xc8, 0xd8, 0xe0, 0x8b, 0xf3, 0x71,
       0x06, 0x8b, 0xdf, 0x28, 0xae, 0x6e, 0xe8, 0x3b, 0x2e, 0x02,␣
↪0x32, 0x8a, 0xf8, 0xc0, 0xe2, 0xf9,
       0x6e, 0x52, 0x8e, 0x16, 0xf8, 0x52, 0xf1, 0xfc, 0x54, 0x55,␣
↪0xe4, 0x77, 0x2e, 0x28, 0x8a, 0x68,
       0xf1, 0x59, 0xca, 0x6b, 0xdc, 0xf9, 0x02, 0xb8, 0x58, 0xa1,␣
↪0xf9, 0x47, 0x89, 0xb3, 0x16, 0x38,
       0x23, 0xe2, 0xd0, 0x71, 0x7f, 0xf5, 0x66, 0x89, 0xee, 0xc7,␣
↪0xd0, 0xe5, 0x4d, 0x93, 0xf5, 0x20,
       0xd9, 0x6e, 0x1e, 0xb0, 0x45, 0x15, 0xab, 0xc7, 0x0a, 0xe9,␣
↪0x05, 0x78, 0xff, 0x38, 0xd3, 0x1b};

    /* 2048-bit modulus */
    uint8_t modulus[PSA_MG_RSA_MODULUS_SIZE] = {
        0xCE, 0xA8, 0x04, 0x75, 0x32, 0x4C, 0x1D, 0xC8, 0x34, 0x78,␣
↪0x27, 0x81, 0x8D, 0xA5, 0x8B, 0xAC,
       0x06, 0x9D, 0x34, 0x19, 0xC6, 0x14, 0xA6, 0xEA, 0x1A, 0xC6,␣
↪0xA3, 0xB5, 0x10, 0xDC, 0xD7, 0x2C,
       0xC5, 0x16, 0x95, 0x49, 0x05, 0xE9, 0xFE, 0xF9, 0x08, 0xD4,␣
↪0x5E, 0x13, 0x00, 0x6A, 0xDF, 0x27,
       0xD4, 0x67, 0xA7, 0xD8, 0x3C, 0x11, 0x1D, 0x1A, 0x5D, 0xF1,␣
↪0x5E, 0xF2, 0x93, 0x77, 0x1A, 0xEF,
       0xB9, 0x20, 0x03, 0x2A, 0x5B, 0xB9, 0x89, 0xF8, 0xE4, 0xF5,␣
↪0xE1, 0xB0, 0x50, 0x93, 0xD3, 0xF1,
       0x30, 0xF9, 0x84, 0xC0, 0x7A, 0x77, 0x2A, 0x36, 0x83, 0xF4,␣
↪0xDC, 0x6F, 0xB2, 0x8A, 0x96, 0x81,
       0x5B, 0x32, 0x12, 0x3C, 0xCD, 0xD1, 0x39, 0x54, 0xF1, 0x9D,␣
↪0x5B, 0x8B, 0x24, 0xA1, 0x03, 0xE7,
       0x71, 0xA3, 0x4C, 0x32, 0x87, 0x55, 0xC6, 0x5E, 0xD6, 0x4E,␣
↪0x19, 0x24, 0xFF, 0xD0, 0x4D, 0x30,
       0xB2, 0x14, 0x2C, 0xC2, 0x62, 0xF6, 0xE0, 0x04, 0x8F, 0xEF,␣
↪0x6D, 0xBC, 0x65, 0x2F, 0x21, 0x47,
       0x9E, 0xA1, 0xC4, 0xB1, 0xD6, 0x6D, 0x28, 0xF4, 0xD4, 0x6E,␣
```

(continues on next page)

(continued from previous page)

```
↪0xF7, 0x18, 0x5E, 0x39, 0x0C, 0xBF,
       0xA2, 0xE0, 0x23, 0x80, 0x58, 0x2F, 0x31, 0x88, 0xBB, 0x94,␣
↪0xEB, 0xBF, 0x05, 0xD3, 0x14, 0x87,
       0xA0, 0x9A, 0xFF, 0x01, 0xFC, 0xBB, 0x4C, 0xD4, 0xBF, 0xD1,␣
↪0xF0, 0xA8, 0x33, 0xB3, 0x8C, 0x11,
       0x81, 0x3C, 0x84, 0x36, 0x0B, 0xB5, 0x3C, 0x7D, 0x44, 0x81,␣
↪0x03, 0x1C, 0x40, 0xBA, 0xD8, 0x71,
       0x3B, 0xB6, 0xB8, 0x35, 0xCB, 0x08, 0x09, 0x8E, 0xD1, 0x5B,␣
↪0xA3, 0x1E, 0xE4, 0xBA, 0x72, 0x8A,
       0x8C, 0x8E, 0x10, 0xF7, 0x29, 0x4E, 0x1B, 0x41, 0x63, 0xB7,␣
↪0xAE, 0xE5, 0x72, 0x77, 0xBF, 0xD8,
       0x81, 0xA6, 0xF9, 0xD4, 0x3E, 0x02, 0xC6, 0x92, 0x5A, 0xA3,␣
↪0xA0, 0x43, 0xFB, 0x7F, 0xB7, 0x8D};

   /* 2048-bit private exponent */
   uint8_t private_exponent[PSA_MG_RSA_PRIVATE_EXPONENT_SIZE] = {
       0x09, 0x97, 0x63, 0x4C, 0x47, 0x7C, 0x1A, 0x03, 0x9D, 0x44,␣
↪0xC8, 0x10, 0xB2, 0xAA, 0xA3, 0xC7,
       0x86, 0x2B, 0x0B, 0x88, 0xD3, 0x70, 0x82, 0x72, 0xE1, 0xE1,␣
↪0x5F, 0x66, 0xFC, 0x93, 0x89, 0x70,
       0x9F, 0x8A, 0x11, 0xF3, 0xEA, 0x6A, 0x5A, 0xF7, 0xEF, 0xFA,␣
↪0x2D, 0x01, 0xC1, 0x89, 0xC5, 0x0F,
       0x0D, 0x5B, 0xCB, 0xE3, 0xFA, 0x27, 0x2E, 0x56, 0xCF, 0xC4,␣
↪0xA4, 0xE1, 0xD3, 0x88, 0xA9, 0xDC,
       0xD6, 0x5D, 0xF8, 0x62, 0x89, 0x02, 0x55, 0x6C, 0x8B, 0x6B,␣
↪0xB6, 0xA6, 0x41, 0x70, 0x9B, 0x5A,
       0x35, 0xDD, 0x26, 0x22, 0xC7, 0x3D, 0x46, 0x40, 0xBF, 0xA1,␣
↪0x35, 0x9D, 0x0E, 0x76, 0xE1, 0xF2,
       0x19, 0xF8, 0xE3, 0x3E, 0xB9, 0xBD, 0x0B, 0x59, 0xEC, 0x19,␣
↪0x8E, 0xB2, 0xFC, 0xCA, 0xAE, 0x03,
       0x46, 0xBD, 0x8B, 0x40, 0x1E, 0x12, 0xE3, 0xC6, 0x7C, 0xB6,␣
↪0x29, 0x56, 0x9C, 0x18, 0x5A, 0x2E,
       0x0F, 0x35, 0xA2, 0xF7, 0x41, 0x64, 0x4C, 0x1C, 0xCA, 0x5E,␣
↪0xBB, 0x13, 0x9D, 0x77, 0xA8, 0x9A,
       0x29, 0x53, 0xFC, 0x5E, 0x30, 0x04, 0x8C, 0x0E, 0x61, 0x9F,␣
↪0x07, 0xC8, 0xD2, 0x1D, 0x1E, 0x56,
       0xB8, 0xAF, 0x07, 0x19, 0x3D, 0x0F, 0xDF, 0x3F, 0x49, 0xCD,␣
```

(continues on next page)

(continued from previous page)

```
↪0x49, 0xF2, 0xEF, 0x31, 0x38, 0xB5,
       0x13, 0x88, 0x62, 0xF1, 0x47, 0x0B, 0xD2, 0xD1, 0x6E, 0x34,␣
↪0xA2, 0xB9, 0xE7, 0x77, 0x7A, 0x6C,
       0x8C, 0x8D, 0x4C, 0xB9, 0x4B, 0x4E, 0x8B, 0x5D, 0x61, 0x6C,␣
↪0xD5, 0x39, 0x37, 0x53, 0xE7, 0xB0,
       0xF3, 0x1C, 0xC7, 0xDA, 0x55, 0x9B, 0xA8, 0xE9, 0x8D, 0x88,␣
↪0x89, 0x14, 0xE3, 0x34, 0x77, 0x3B,
       0xAF, 0x49, 0x8A, 0xD8, 0x8D, 0x96, 0x31, 0xEB, 0x5F, 0xE3,␣
↪0x2E, 0x53, 0xA4, 0x14, 0x5B, 0xF0,
       0xBA, 0x54, 0x8B, 0xF2, 0xB0, 0xA5, 0x0C, 0x63, 0xF6, 0x7B,␣
↪0x14, 0xE3, 0x98, 0xA3, 0x4B, 0x0D};

    /* 2048-bit public exponent */
    uint8_t public_exponent[PSA_MG_RSA_MAX_PUBLIC_EXPONENT] = {0};

    /* PSS deterministic salt */
    uint8_t pss_salt[20] = {
        0xe1, 0x25, 0x6f, 0xc1, 0xee, 0xef, 0x81, 0x77, 0x3f, 0xdd,
        0x54, 0x65, 0x7e, 0x40, 0x07, 0xfd, 0xe6, 0xbc, 0xb9, 0xb1};

    /* e = 0x260445 */
    public_exponent[253] = 0x26;
    public_exponent[254] = 0x04;
    public_exponent[255] = 0x45;

    /* ASN.1 encoded key buffers */
    uint8_t private_asn1_key[1024] = {0};
    size_t  private_asn1_key_len   = 0;
    uint8_t public_asn1_key[1024]  = {0};
    size_t  public_asn1_key_len    = 0;

    /* PKCS#1 v1.5 encryption buffers */
    uint8_t PKCS_ciphertext[PSA_MG_RSA_BLOCK_SIZE] = {0};
    size_t  PKCS_cipher_len      = 0;
    uint8_t PKCS_decrypted[PSA_MG_RSA_BLOCK_SIZE]  = {0};
    size_t  PKCS_decrypted_len   = 0;
```

(continues on next page)

(continued from previous page)

```
    /* OAEP encryption buffers */
    uint8_t OAEP_ciphertext[PSA_MG_RSA_BLOCK_SIZE] = {0};
    size_t  OAEP_cipher_len      = 0;
    uint8_t OAEP_decrypted[PSA_MG_RSA_BLOCK_SIZE]  = {0};
    size_t  OAEP_decrypted_len   = 0;

    /* PKCS#1 v1.5 signature buffers */
    uint8_t PKCS_signature[PSA_MG_RSA_SIGNATURE_SIZE] = {0};
    size_t  PKCS_signature_len  = 0;

    /* PSS random salt signature buffers */
    uint8_t PSS_rand_signature[PSA_MG_RSA_SIGNATURE_SIZE] = {0};
    size_t  PSS_rand_sig_len        = 0;

    /* PSS deterministic salt signature buffers */
    uint8_t PSS_det_signature[PSA_MG_RSA_SIGNATURE_SIZE] = {0};
    size_t  PSS_det_sig_len        = 0;

    mg_rsa_context          rsa_priv;
    mg_rsa_context          rsa_pub;
    psa_key_attributes_t    private_key_attributes = PSA_KEY_
↪ATTRIBUTES_INIT;
    psa_key_attributes_t    public_key_attributes  = PSA_KEY_
↪ATTRIBUTES_INIT;
    psa_key_id_t            priv_key_handle        = 0;
    psa_key_id_t            pub_key_handle         = 0;
    psa_status_t            status;

    psa_crypto_init();

    printf("\r\nInput Data        (128 bytes) : ");
    for (uint16_t i = 0; i < sizeof(input); i++)
        printf("%02x", input[i]);
    printf("\r\nInput Length                  : %d bytes\r\n",␣
↪sizeof(input));
    printf("Key Size                     : 2048 bits (256 bytes)\r\n
↪");
```

(continues on next page)

(continued from previous page)

```
    printf("Public Exponent  (256 bytes)  : ");
    for (uint16_t i = 0; i < sizeof(public_exponent); i++)
        printf("%02x", public_exponent[i]);
    printf("\r\nPrivate Exponent (256 bytes)  : ");
    for (uint16_t i = 0; i < sizeof(private_exponent); i++)
        printf("%02x", private_exponent[i]);
    printf("\r\nModulus          (256 bytes)  : ");
    for (uint16_t i = 0; i < sizeof(modulus); i++)
        printf("%02x", modulus[i]);
    printf("\r\n");

    /* Build ASN.1 encoded key pair */
    memset(&rsa_priv, 0, sizeof(mg_rsa_context));
    BN_INIT(&rsa_priv.N);
    BN_INIT(&rsa_priv.D);
    BigNum_Read_Unsigned_Bin(&rsa_priv.N, modulus, PSA_MG_RSA_MODULUS_
↪SIZE);
    BigNum_Read_Unsigned_Bin(&rsa_priv.D, private_exponent, PSA_MG_RSA_
↪PRIVATE_EXPONENT_SIZE);
    rsa_priv.len = 256;

    status = mg_psa_rsa_export_key(PSA_KEY_TYPE_RSA_KEY_PAIR, &rsa_
↪priv,
                                   private_asn1_key, sizeof(private_asn1_
↪key),
                                   &private_asn1_key_len);
    if (status != PSA_SUCCESS)
    {
        printf("mg_psa_rsa_export_key (private) Failed. Status: %d\r\n
↪", status);
        return status;
    }
    printf("\r\nmg_psa_rsa_export_key (private) Successful\r\n");

    memset(&rsa_pub, 0, sizeof(mg_rsa_context));
    BN_INIT(&rsa_pub.N);
    BN_INIT(&rsa_pub.E);
```

(continues on next page)

(continued from previous page)

```
    BigNum_Read_Unsigned_Bin(&rsa_pub.N, modulus, PSA_MG_RSA_MODULUS_
↪SIZE);
    BigNum_Read_Unsigned_Bin(&rsa_pub.E, public_exponent, PSA_MG_RSA_
↪MAX_PUBLIC_EXPONENT);

    status = mg_psa_rsa_export_key(PSA_KEY_TYPE_RSA_PUBLIC_KEY, &rsa_
↪pub,
                                   public_asn1_key, sizeof(public_asn1_
↪key),
                                   &public_asn1_key_len);
    if (status != PSA_SUCCESS)
    {
        printf("mg_psa_rsa_export_key (public) Failed. Status: %d\r\n",
↪ status);
        return status;
    }
    printf("mg_psa_rsa_export_key (public) Successful\r\n");

    /* ============= PKCS#1 v1.5 - ENCRYPT / DECRYPT␣
↪==================== */

    printf("\r\nRunning PKCS#1 v1.5 Encryption...\r\n");

    psa_key_attributes_t pkcs_pub_attr = PSA_KEY_ATTRIBUTES_INIT;
    psa_set_key_usage_flags(&pkcs_pub_attr, PSA_KEY_USAGE_ENCRYPT);
    psa_set_key_algorithm(&pkcs_pub_attr, PSA_ALG_RSA_PKCS1V15_CRYPT);
    psa_set_key_type(&pkcs_pub_attr, PSA_KEY_TYPE_RSA_PUBLIC_KEY);

    status = psa_import_key(&pkcs_pub_attr, public_asn1_key,
                            public_asn1_key_len, &pub_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_import_key (PKCS encrypt) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }
    printf("psa_import_key Successful\r\n");
```

(continues on next page)

(continued from previous page)

```
    status = psa_asymmetric_encrypt(pub_key_handle, PSA_ALG_RSA_
↪PKCS1V15_CRYPT,
                                    input, sizeof(input), NULL, 0,
                                    PKCS_ciphertext, sizeof(PKCS_
↪ciphertext),
                                    &PKCS_cipher_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_asymmetric_encrypt (PKCS) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }

    printf("Ciphertext        (%d bytes) : ", PKCS_cipher_len);
    for (uint16_t i = 0; i < PKCS_cipher_len; i++)
        printf("%02x", PKCS_ciphertext[i]);
    printf("\r\nOutput Length                 : %d bytes\r\n", PKCS_
↪cipher_len);
    printf("psa_asymmetric_encrypt Successful\r\n");

    status = psa_destroy_key(pub_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_destroy_key Failed. Status: %d\r\n", status);
        return status;
    }

    printf("\r\nRunning PKCS#1 v1.5 Decryption...\r\n");

    psa_key_attributes_t pkcs_priv_attr = PSA_KEY_ATTRIBUTES_INIT;
    psa_set_key_usage_flags(&pkcs_priv_attr, PSA_KEY_USAGE_DECRYPT);
    psa_set_key_algorithm(&pkcs_priv_attr, PSA_ALG_RSA_PKCS1V15_CRYPT);
    psa_set_key_type(&pkcs_priv_attr, PSA_KEY_TYPE_RSA_KEY_PAIR);

    status = psa_import_key(&pkcs_priv_attr, private_asn1_key,
                            private_asn1_key_len, &priv_key_handle);
```

(continues on next page)

(continued from previous page)

```
    if (status != PSA_SUCCESS)
    {
        printf("psa_import_key (PKCS decrypt) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }
    printf("psa_import_key Successful\r\n");

    status = psa_asymmetric_decrypt(priv_key_handle, PSA_ALG_RSA_
↪PKCS1V15_CRYPT,
                                    PKCS_ciphertext, PKCS_cipher_len,␣
↪NULL, 0,
                                    PKCS_decrypted, sizeof(PKCS_
↪decrypted),
                                    &PKCS_decrypted_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_asymmetric_decrypt (PKCS) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }

    printf("Decrypted Output  (%d bytes) : ", PKCS_decrypted_len);
    for (uint16_t i = 0; i < PKCS_decrypted_len; i++)
        printf("%02x", PKCS_decrypted[i]);
    printf("\r\nOutput Length                 : %d bytes\r\n", PKCS_
↪decrypted_len);
    printf("psa_asymmetric_decrypt Successful\r\n");

    status = psa_destroy_key(priv_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_destroy_key Failed. Status: %d\r\n", status);
        return status;
    }

    /* ================== OAEP - ENCRYPT / DECRYPT␣
```

(continues on next page)

(continued from previous page)

```
↪====================== */

    printf("\r\nRunning OAEP Encryption...\r\n");

    psa_key_attributes_t oaep_pub_attr = PSA_KEY_ATTRIBUTES_INIT;
    psa_set_key_usage_flags(&oaep_pub_attr, PSA_KEY_USAGE_ENCRYPT);
    psa_set_key_algorithm(&oaep_pub_attr, PSA_ALG_RSA_OAEP_SHA256);
    psa_set_key_type(&oaep_pub_attr, PSA_KEY_TYPE_RSA_PUBLIC_KEY);

    status = psa_import_key(&oaep_pub_attr, public_asn1_key,
                            public_asn1_key_len, &pub_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_import_key (OAEP encrypt) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }
    printf("psa_import_key Successful\r\n");

    status = psa_asymmetric_encrypt(pub_key_handle, PSA_ALG_RSA_OAEP_
↪SHA256,
                                    input, sizeof(input), NULL, 0,
                                    OAEP_ciphertext, sizeof(OAEP_
↪ciphertext),
                                    &OAEP_cipher_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_asymmetric_encrypt (OAEP) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }

    printf("Ciphertext        (%d bytes) : ", OAEP_cipher_len);
    for (uint16_t i = 0; i < OAEP_cipher_len; i++)
        printf("%02x", OAEP_ciphertext[i]);
    printf("\r\nOutput Length                 : %d bytes\r\n", OAEP_
↪cipher_len);
```

(continues on next page)

(continued from previous page)

```
    printf("psa_asymmetric_encrypt Successful\r\n");

    status = psa_destroy_key(pub_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_destroy_key Failed. Status: %d\r\n", status);
        return status;
    }

    printf("\r\nRunning OAEP Decryption...\r\n");

    psa_key_attributes_t oaep_priv_attr = PSA_KEY_ATTRIBUTES_INIT;
    psa_set_key_usage_flags(&oaep_priv_attr, PSA_KEY_USAGE_DECRYPT);
    psa_set_key_algorithm(&oaep_priv_attr, PSA_ALG_RSA_OAEP_SHA256);
    psa_set_key_type(&oaep_priv_attr, PSA_KEY_TYPE_RSA_KEY_PAIR);

    status = psa_import_key(&oaep_priv_attr, private_asn1_key,
                            private_asn1_key_len, &priv_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_import_key (OAEP decrypt) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }
    printf("psa_import_key Successful\r\n");

    status = psa_asymmetric_decrypt(priv_key_handle, PSA_ALG_RSA_OAEP_
↪SHA256,
                                    OAEP_ciphertext, OAEP_cipher_len,␣
↪NULL, 0,
                                    OAEP_decrypted, sizeof(OAEP_
↪decrypted),
                                    &OAEP_decrypted_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_asymmetric_decrypt (OAEP) Failed. Status: %d\r\n",␣
↪status);
```

(continues on next page)

(continued from previous page)

```
        return status;
    }

    printf("Decrypted Output  (%d bytes) : ", OAEP_decrypted_len);
    for (uint16_t i = 0; i < OAEP_decrypted_len; i++)
        printf("%02x", OAEP_decrypted[i]);
    printf("\r\nOutput Length                 : %d bytes\r\n", OAEP_
↪decrypted_len);
    printf("psa_asymmetric_decrypt Successful\r\n");

    status = psa_destroy_key(priv_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_destroy_key Failed. Status: %d\r\n", status);
        return status;
    }

    /* ============= PKCS#1 v1.5 - SIGN / VERIFY␣
↪======================== */

    printf("\r\nRunning PKCS#1 v1.5 Sign...\r\n");

    psa_key_attributes_t pkcs_sign_attr = PSA_KEY_ATTRIBUTES_INIT;
    psa_set_key_usage_flags(&pkcs_sign_attr, PSA_KEY_USAGE_SIGN_
↪MESSAGE);
    psa_set_key_algorithm(&pkcs_sign_attr, PSA_ALG_RSA_PKCS1V15_SIGN_
↪BASE);
    psa_set_key_type(&pkcs_sign_attr, PSA_KEY_TYPE_RSA_KEY_PAIR);

    status = psa_import_key(&pkcs_sign_attr, private_asn1_key,
                            private_asn1_key_len, &priv_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_import_key (PKCS sign) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }
```

(continues on next page)

(continued from previous page)

```
    printf("psa_import_key Successful\r\n");

    status = psa_sign_message(priv_key_handle, PSA_ALG_RSA_PKCS1V15_
↪SIGN_BASE,
                              input, sizeof(input),
                              PKCS_signature, sizeof(PKCS_signature),
                              &PKCS_signature_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_sign_message (PKCS) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }

    printf("Signature         (%d bytes) : ", PKCS_signature_len);
    for (uint16_t i = 0; i < PKCS_signature_len; i++)
        printf("%02x", PKCS_signature[i]);
    printf("\r\nOutput Length                 : %d bytes\r\n", PKCS_
↪signature_len);
    printf("psa_sign_message Successful\r\n");

    status = psa_destroy_key(priv_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_destroy_key Failed. Status: %d\r\n", status);
        return status;
    }

    printf("\r\nRunning PKCS#1 v1.5 Verify...\r\n");

    psa_key_attributes_t pkcs_verify_attr = PSA_KEY_ATTRIBUTES_INIT;
    psa_set_key_usage_flags(&pkcs_verify_attr, PSA_KEY_USAGE_VERIFY_
↪MESSAGE);
    psa_set_key_algorithm(&pkcs_verify_attr, PSA_ALG_RSA_PKCS1V15_SIGN_
↪BASE);
    psa_set_key_type(&pkcs_verify_attr, PSA_KEY_TYPE_RSA_PUBLIC_KEY);
```

(continues on next page)

(continued from previous page)

```
    status = psa_import_key(&pkcs_verify_attr, public_asn1_key,
                            public_asn1_key_len, &pub_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_import_key (PKCS verify) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }
    printf("psa_import_key Successful\r\n");

    status = psa_verify_message(pub_key_handle, PSA_ALG_RSA_PKCS1V15_
↪SIGN_BASE,
                                input, sizeof(input),
                                PKCS_signature, PKCS_signature_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_verify_message (PKCS) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }

    printf("Output Status                 : PSA_SUCCESS\r\n");
    printf("psa_verify_message Successful\r\n");

    status = psa_destroy_key(pub_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_destroy_key Failed. Status: %d\r\n", status);
        return status;
    }

    /* ============= PSS - SIGN / VERIFY (RANDOM SALT)␣
↪================== */

    printf("\r\nRunning PSS Sign (Random Salt)...\r\n");

    psa_key_attributes_t pss_rand_sign_attr = PSA_KEY_ATTRIBUTES_INIT;
```

(continues on next page)

(continued from previous page)

```
    psa_set_key_usage_flags(&pss_rand_sign_attr, PSA_KEY_USAGE_SIGN_
↪MESSAGE);
    psa_set_key_algorithm(&pss_rand_sign_attr, PSA_ALG_RSA_PSS_BASE);
    psa_set_key_type(&pss_rand_sign_attr, PSA_KEY_TYPE_RSA_KEY_PAIR);

    status = psa_import_key(&pss_rand_sign_attr, private_asn1_key,
                            private_asn1_key_len, &priv_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_import_key (PSS rand sign) Failed. Status: %d\r\n",
↪ status);
        return status;
    }
    printf("psa_import_key Successful\r\n");

    status = psa_sign_message(priv_key_handle, PSA_ALG_RSA_PSS_BASE,
                            input, sizeof(input),
                            PSS_rand_signature, sizeof(PSS_rand_
↪signature),
                            &PSS_rand_sig_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_sign_message (PSS random) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }

    printf("Signature          (%d bytes) : ", PSS_rand_sig_len);
    for (uint16_t i = 0; i < PSS_rand_sig_len; i++)
        printf("%02x", PSS_rand_signature[i]);
    printf("\r\nOutput Length                 : %d bytes\r\n", PSS_
↪rand_sig_len);
    printf("psa_sign_message Successful\r\n");

    status = psa_destroy_key(priv_key_handle);
    if (status != PSA_SUCCESS)
    {
```

(continues on next page)

(continued from previous page)

```
        printf("psa_destroy_key Failed. Status: %d\r\n", status);
        return status;
    }

    printf("\r\nRunning PSS Verify (Random Salt)...\r\n");

    psa_key_attributes_t pss_rand_verify_attr = PSA_KEY_ATTRIBUTES_
↪INIT;
    psa_set_key_usage_flags(&pss_rand_verify_attr, PSA_KEY_USAGE_
↪VERIFY_MESSAGE);
    psa_set_key_algorithm(&pss_rand_verify_attr, PSA_ALG_RSA_PSS_BASE);
    psa_set_key_type(&pss_rand_verify_attr, PSA_KEY_TYPE_RSA_PUBLIC_
↪KEY);

    status = psa_import_key(&pss_rand_verify_attr, public_asn1_key,
                            public_asn1_key_len, &pub_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_import_key (PSS rand verify) Failed. Status: %d\r\n
↪", status);
        return status;
    }
    printf("psa_import_key Successful\r\n");

    status = psa_verify_message(pub_key_handle, PSA_ALG_RSA_PSS_BASE,
                                input, sizeof(input),
                                PSS_rand_signature, PSS_rand_sig_len);
    if (status != PSA_SUCCESS)
    {
        printf("psa_verify_message (PSS random) Failed. Status: %d\r\n
↪", status);
        return status;
    }

    printf("Output Status                  : PSA_SUCCESS\r\n");
    printf("psa_verify_message Successful\r\n");
```

(continues on next page)

(continued from previous page)

```
    status = psa_destroy_key(pub_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_destroy_key Failed. Status: %d\r\n", status);
        return status;
    }

    /* ============= PSS - SIGN / VERIFY (DETERMINISTIC SALT) ␣
↪=========== */

    printf("\r\nRunning PSS Sign (Deterministic Salt)...\r\n");
    printf("Salt              (20 bytes)  : ");
    for (uint8_t i = 0; i < sizeof(pss_salt); i++)
        printf("%02x", pss_salt[i]);
    printf("\r\nSalt Length                  : %d bytes\r\n",␣
↪sizeof(pss_salt));

    psa_key_attributes_t pss_det_sign_attr = PSA_KEY_ATTRIBUTES_INIT;
    psa_set_key_usage_flags(&pss_det_sign_attr, PSA_KEY_USAGE_SIGN_
↪MESSAGE);
    psa_set_key_algorithm(&pss_det_sign_attr, PSA_ALG_RSA_PSS_BASE);
    psa_set_key_type(&pss_det_sign_attr, PSA_KEY_TYPE_RSA_KEY_PAIR);

    status = psa_import_key(&pss_det_sign_attr, private_asn1_key,
                            private_asn1_key_len, &priv_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_import_key (PSS det sign) Failed. Status: %d\r\n",␣
↪status);
        return status;
    }
    printf("psa_import_key Successful\r\n");

    status = psa_sign_message_with_salt(priv_key_handle, PSA_ALG_RSA_
↪PSS_BASE,
                                        input, sizeof(input),
                                        PSS_det_signature, sizeof(PSS_
```

(continues on next page)

(continued from previous page)

```
↪det_signature),
                                        &PSS_det_sig_len,
                                        pss_salt, sizeof(pss_salt));
    if (status != PSA_SUCCESS)
    {
        printf("psa_sign_message_with_salt Failed. Status: %d\r\n",␣
↪status);
        return status;
    }

    printf("Signature         (%d bytes) : ", PSS_det_sig_len);
    for (uint16_t i = 0; i < PSS_det_sig_len; i++)
        printf("%02x", PSS_det_signature[i]);
    printf("\r\nOutput Length                 : %d bytes\r\n", PSS_det_
↪sig_len);
    printf("psa_sign_message_with_salt Successful\r\n");

    status = psa_destroy_key(priv_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_destroy_key Failed. Status: %d\r\n", status);
        return status;
    }

    printf("\r\nRunning PSS Verify (Deterministic Salt)...\r\n");

    psa_key_attributes_t pss_det_verify_attr = PSA_KEY_ATTRIBUTES_INIT;
    psa_set_key_usage_flags(&pss_det_verify_attr, PSA_KEY_USAGE_VERIFY_
↪MESSAGE);
    psa_set_key_algorithm(&pss_det_verify_attr, PSA_ALG_RSA_PSS_BASE);
    psa_set_key_type(&pss_det_verify_attr, PSA_KEY_TYPE_RSA_PUBLIC_
↪KEY);

    status = psa_import_key(&pss_det_verify_attr, public_asn1_key,
                            public_asn1_key_len, &pub_key_handle);
    if (status != PSA_SUCCESS)
    {
```

(continues on next page)

(continued from previous page)

```
        printf("psa_import_key (PSS det verify) Failed. Status: %d\r\n
↪", status);
        return status;
    }
    printf("psa_import_key Successful\r\n");

    status = psa_verify_message_with_salt(pub_key_handle, PSA_ALG_RSA_
↪PSS_BASE,
                                          input, sizeof(input),
                                          PSS_det_signature, PSS_det_sig_
↪len,
                                          sizeof(pss_salt));
    if (status != PSA_SUCCESS)
    {
        printf("psa_verify_message_with_salt Failed. Status: %d\r\n",␣
↪status);
        return status;
    }

    printf("Output Status                 : PSA_SUCCESS\r\n");
    printf("psa_verify_message_with_salt Successful\r\n");

    status = psa_destroy_key(pub_key_handle);
    if (status != PSA_SUCCESS)
    {
        printf("psa_destroy_key Failed. Status: %d\r\n", status);
        return status;
    }

    return PSA_SUCCESS;
}
```

Output

```
Input Data        (128 bytes) :␣
↪5af283b1b76ab2a695d794c23b35ca7371fc779e92eb
```

(continues on next page)

(continued from previous page)

```
                                    ␣
↪f589e304c7f923d8cf976304c19818fcd89d6f07c8d8
                                    ␣
↪e08bf371068bdf28ae6ee83b2e02328af8c0e2f96e52
                                    ␣
↪8e16f852f1fc5455e4772e288a68f159ca6bdcf902b8
                                    ␣
↪58a1f94789b3163823e2d0717ff56689eec7d0e54d93
                                     f520d96e1eb04515abc70ae90578ff38d31b
Input Length                      : 128 bytes
Key Size                          : 2048 bits (256 bytes)
Public Exponent  (256 bytes)  :␣
↪0000000000000000000000000000000000000000000000
                                    ␣
↪0000000000000000000000000000000000000000000000
                                    ␣
↪0000000000000000000000000000000000000000000000
                                    ␣
↪0000000000000000000000000000000000000000000000
                                    ␣
↪0000000000000000000000000000000000000000000000
                                    ␣
↪0000000000000000000000000000000000000000000000
                                    ␣
↪0000000000000000000000000000000000000000000000
                                    ␣
↪0000000000000000000000000000000000000000000000
                                    ␣
↪0000000000000000000000000000000000000000000000
                                    ␣
↪0000000000000000000000000000000000000000000000
                                    ␣
↪0000000000000000000000000000000000000000000000
                                     00000000000000000000000000010001
Private Exponent (256 bytes)  :␣
↪0997634c477c1a039d44c810b2aaa3c7862b0b88d370
                                    ␣
```

(continues on next page)

(continued from previous page)

```
↪8272e1e15f66fc9389709f8a11f3ea6a5af7effa2d01
                                    ␣
↪c189c50f0d5bcbe3fa272e56cfc4a4e1d388a9dcd65d
                                    ␣
↪f8629802556c8b6bb6a641709b5a35dd2622c73d4640
                                    ␣
↪bfa1359d0e76e1f219f8e33eb9bd0b59ec198eb2fcca
                                    ␣
↪ae0346bd8b401e12e3c67cb629569c185a2e0f35a2f7
                                    ␣
↪41644c1cca5ebb139d77a89a2953fc5e30048c0e619f
                                    ␣
↪07c8d21d1e56b8af07193d0fdf3f49cd49f2ef3138b5
                                    ␣
↪138862f1470bd2d16e34a2b9e7777a6c8c8d4cb94b4e
                                    ␣
↪8b5d616cd5393753e7b0f31cc7da559ba8e98d888914
                                    ␣
↪e334773baf498ad88d9631eb5fe32e53a4145bf0ba54
                                      8bf2b0a50c63f67b14e398a34b0d
Modulus           (256 bytes)  :␣
↪cea80475324c1dc8347827818da58bac069d3419c614
                                    ␣
↪a6ea1ac6a3b510dcd72cc516954905e9fef908d45e13
                                    ␣
↪006adf27d467a7d83c111d1a5df15ef293771aefb920
                                    ␣
↪032a5bb989f8e4f5e1b05093d3f130f984c07a772a36
                                    ␣
↪83f4dc6fb28a96815b32123ccd13954f19d5b8b24a10
                                    ␣
↪3e771a34c328755c65ed64e1924ffd04d30b2142cc26
                                    ␣
↪2f6e0048fef6dbc652f21479ea1c4b1d66d28f4d46ef
                                    ␣
↪7185e390cbfa2e02380582f3188bb94ebbf05d31487a
                                    ␣
```

(continues on next page)

(continued from previous page)

```
↪09aff01fcbb4cd4bfd1f0a833b38c11813c84360bb53
                                  ␣
↪c7d4481031c40bad8713bb6b835cb08098ed15ba31ee
                                  ␣
↪4ba728a8c8e10f7294e1b4163b7aee57277bfd881a6f
                                    9d43e02c6925aa3a043fb7fb78d

mg_psa_rsa_export_key (private) Successful
mg_psa_rsa_export_key (public) Successful

Running PKCS#1 v1.5 Encryption...
psa_import_key Successful
Ciphertext       (256 bytes)  :␣
↪a0f3480ff75e42a6b86149abca2f1b2b91805ec28ce5
                                  ␣
↪0a24ac00cbc530beea4a4ade76767cc3f67ce9621a30
                                  ␣
↪dde4a673abfa4257d522a9bc1056fe20de96bc9a5553
                                  ␣
↪4648a10c081c64d8e8ea11dd5638b84d929b7c03cb7f
                                  ␣
↪486bf1589d59e32b209018cfea8ffc483b0c540dd2d1
                                  ␣
↪b319621285d09a7a0d09c77743e8dfe392f7f4021056
                                  ␣
↪9ae8bc6c993d354564b3898f41fc2e30aa5ceb4fdd55
                                  ␣
↪2f7c2d664ce8f7b03573dd07af78b51ffae25be07150
                                  ␣
↪f5faa007ad7a9f1aeb625bf2ca278cd8189c4dec9eb0
                                  ␣
↪c8c3c7794c30e5da31109b76ecefb865066c9ee68fbf
                                  ␣
↪34e92d83465caffb5d079668a44b3b1f943f222e9a93
                                    188d6748604c3f208f64a71b4dbf
Output Length                  : 256 bytes
psa_asymmetric_encrypt Successful
```

(continues on next page)

(continued from previous page)

```
Running PKCS#1 v1.5 Decryption...
psa_import_key Successful
Decrypted Output (128 bytes)  :␣
↪5af283b1b76ab2a695d794c23b35ca7371fc779e92eb
                              ␣
↪f589e304c7f923d8cf976304c19818fcd89d6f07c8d8
                              ␣
↪e08bf371068bdf28ae6ee83b2e02328af8c0e2f96e52
                              ␣
↪8e16f852f1fc5455e4772e288a68f159ca6bdcf902b8
                              ␣
↪58a1f94789b3163823e2d0717ff56689eec7d0e54d93
                                f520d96e1eb04515abc70ae90578ff38d31b
Output Length                 : 128 bytes
psa_asymmetric_decrypt Successful

Running OAEP Encryption...
psa_import_key Successful
Ciphertext       (256 bytes)  :␣
↪6cd5eb08fc01be2609e03bcf092f3a2ff6695fe146dd
                              ␣
↪beeffb406cdda7404be8df6cf9e1921ad83a32e95cc5
                              ␣
↪936efb0e1131c88636d64cb6c227bd36b65e2c6d9a80
                              ␣
↪d9d1101915818b7e32a4c7e2f49a1ee3565d34268d0b
                              ␣
↪7acb54ad376d9adcee84cced41956e02ca22f293ddb8
                              ␣
↪a7fb3ca1dfe58a5c8b51c8f34710fff734dd7078f0db
                              ␣
↪3dc313e06224be1b87c5a4c892466a1758d9101549c1
                              ␣
↪e9dd4828d436f50f550bdf45ccc9d271157f6d663a46
                              ␣
↪a43aac2b5ae9c6e58526c41f990e49f90555d6ca5d3e
```

(continues on next page)

(continued from previous page)

```
                                    ␣
↪7949388ebe0fb0c4359d74f2271f7a881e281baf0632
                                    ␣
↪9964a84e0a448bab361ba19643c5d351db85dfa332da
                                     cf66a4043d7799dfc4a8d4333084
Output Length                    : 256 bytes
psa_asymmetric_encrypt Successful

Running OAEP Decryption...
psa_import_key Successful
Decrypted Output (128 bytes)  :␣
↪5af283b1b76ab2a695d794c23b35ca7371fc779e92eb
                                    ␣
↪f589e304c7f923d8cf976304c19818fcd89d6f07c8d8
                                    ␣
↪e08bf371068bdf28ae6ee83b2e02328af8c0e2f96e52
                                    ␣
↪8e16f852f1fc5455e4772e288a68f159ca6bdcf902b8
                                    ␣
↪58a1f94789b3163823e2d0717ff56689eec7d0e54d93
                                     f520d96e1eb04515abc70ae90578ff38d31b
Output Length                    : 128 bytes
psa_asymmetric_decrypt Successful

Running PKCS#1 v1.5 Sign...
psa_import_key Successful
Signature        (256 bytes)  :␣
↪6b8be97d9e518a2ede746ff4a7d91a84a1fc665b52f1
                                    ␣
↪54a927650db6e7348c69f8c8881f7bcf9b1a6d3366ee
                                    ␣
↪d30c3aed4e93c203c43f5528a45de791895747ade9c5
                                     fa5eee81427edee020821␣
↪47aa311712a6ad5fb1732e
                                    ␣
↪93b3d6cd23ffd46a0b3caf62a8b69957cc68ae39f999
                                    ␣
```

(continues on next page)

(continued from previous page)

```
↪3c1a779599cdda949bdaababb77f248fcfeaa44059be
                                ␣
↪5459fb9b899278e929528ee130facd53372ecbc42f3e
                                ␣
↪8de2998425860406440f248d817432de687112e504d7
                                ␣
↪34028e6c5620fa282ca07647006cf0a2ff83e19a9165
                                ␣
↪54cc61810c2e855305db4e5cf893a6a9676736579455
                                ␣
↪6ff033359084d7e38a8456e68e21155b76151314a298
                                 75feee09557161cbc654541e89e42
Output Length                 : 256 bytes
psa_sign_message Successful

Running PKCS#1 v1.5 Verify...
psa_import_key Successful
Output Status                 : PSA_SUCCESS
psa_verify_message Successful

Running PSS Sign (Random Salt)...
psa_import_key Successful
Signature        (256 bytes)  :␣
↪becbf1feef53842339eb2d507397563580dd90e441e8
                                ␣
↪232ec19170edb6f83baec43af99530de870f9223379e
                                ␣
↪5bbee9118d8d99de0096327e31fe9988d81f21f70bf6
                                ␣
↪2abc800bc13d183ac212a08a0f26137b358fbb5ea576
                                ␣
↪146fe6b62846e8564b7d515dc80c2d33988b2e17e664
                                ␣
↪0758f47ac5cd76dca98d92475168716ae5452f58a8d0
                                ␣
↪cf36c670cda2fc8a8915cd064bdebab9bb0d4464fc8c
                                ␣
```

(continues on next page)

(continued from previous page)

```
↪f7df796c392afa256e10ca48ce9d1fae92f06bff2a6b
                                  ␣
↪8bcf362c6903f298c35a9592c76291dcacc67c9cdd05
                                  ␣
↪2109c16fb7a39920af851d2426e8464b81fd9a0aef19
                                  ␣
↪78682192bc8d912ef8a0c0f8100f988c9e4a0e819533
                                    dc236138efea8c14bd99eddee244
Output Length                   : 256 bytes
psa_sign_message Successful

Running PSS Verify (Random Salt)...
psa_import_key Successful
Output Status                   : PSA_SUCCESS
psa_verify_message Successful

Running PSS Sign (Deterministic Salt)...
Salt              (20 bytes)    :␣
↪e1256fcleeef81773fdd54657e4007fde6bcb9b1
Salt Length                     : 20 bytes
psa_import_key Successful
Signature         (256 bytes)   :␣
↪77fe3124f0b306b30be1ec1343f751b90de074cd8e6a
                                  ␣
↪614fda0cfa9ff55325b5d285e4aef50bf7579cd959a6
                                  ␣
↪2ec5e29dad208441b86ed282efc555cfa9b331c8d13a
                                  ␣
↪3d25d4e761f35c3e69bb5f7a8c97306ca86fdced1cdb
                                  ␣
↪431f52a3dc7545463124abae2dcd9e33a8db2531d235
                                  ␣
↪abb2f858acc389e9b24d65c1a9125bda3fc06f493cfe
                                  ␣
↪60938d63882ee3827d3b46ca8fb3094dfc2102645712
                                  ␣
↪ef6cf814c8bb1fca9cf0dae3100c0a4f9463e3c8ca03
```

(continues on next page)

(continued from previous page)

```
                                  ␣
↪2d9857c97eb67225f9cc2b2d4298dbd6efc4ef508709
                                  ␣
↪a40a48f7e112add7185dcfb2ae8e5095ec81d4687326
                                  ␣
↪d2931da4e8b11ac98ce8d383bb9afc974d448d4746e7
                                    cbe63d3408ed49ce1b5b67189cfe
Output Length                   : 256 bytes
psa_sign_message_with_salt Successful

Running PSS Verify (Deterministic Salt)...
psa_import_key Successful
Output Status                   : PSA_SUCCESS
psa_verify_message_with_salt Successful
```

Troubleshooting

1. Ensure that RSA engine is in correct state before loading input.
2. Ensure the correct endianess before loading input abnd while checking output.

7.1.23 Secure Hashing Algorithm 256 bits (SHA256)

Example Code

```c
#include "sha256.h"         /* Included to access SHA256 API's */
#include "crypto_defines.h" /* Included to access SHA-256 related␣
↪constants */
#include "io.h"             /* Included to access IO operations API's */
#include "errors.h"         /* Included to access errors codes */

/* Total input length in bytes */
#define SHA256_TEST_INPUT_LEN 129

int main(void)
{
```

(continues on next page)

(continued from previous page)

```
    /* 129-byte multi-block input */
    uint8_t input[SHA256_TEST_INPUT_LEN] = {
        0xAA, 0x55, 0xAA, 0x55, 0x00, 0x01, 0x02, 0x03,
        0x04, 0x05, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60,
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
        0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
        0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F,
        0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
        0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F,
        0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
        0x48, 0x49, 0x4A, 0x4B, 0x4C, 0x4D, 0x4E, 0x4F,
        0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57,
        0x58, 0x59, 0x5A, 0x5B, 0x5C, 0x5D, 0x5E, 0x5F,
        0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
        0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F,
        0x3F};

    /* Pre-computed expected SHA-256 hash for the input above */
    const uint8_t expected_hash[SHA256_DIGEST_SIZE] = {
        0xA6, 0x10, 0x6A, 0x20, 0x77, 0xC8, 0x8B, 0xAB,
        0xF9, 0xB8, 0x02, 0xA9, 0xEB, 0x72, 0x67, 0xB9,
        0x9E, 0x2C, 0x64, 0x3A, 0x50, 0x5A, 0x97, 0xA0,
        0xDD, 0x36, 0x19, 0xB5, 0x51, 0xC0, 0xE0, 0xA6};

    uint16_t input_length_bits   = SHA256_TEST_INPUT_LEN * 8;
    uint8_t  single_shot_hash[SHA256_DIGEST_SIZE] = {0};
    uint8_t  multi_shot_hash[SHA256_DIGEST_SIZE]  = {0};
    size_t   output_length        = 0;
    uint16_t iterated_length_bits = 0;
    uint16_t remaining_bits       = 0;
    uint16_t current_chunk_bits   = 0;
    uint8_t  iter                 = 0;
    uint16_t status               = 0;
```

(continues on next page)

(continued from previous page)

```
    printf("\r\nInput Data        (%d bytes) : ", SHA256_TEST_INPUT_
↪LEN);
    for (uint8_t i = 0; i < SHA256_TEST_INPUT_LEN; i++)
        printf("%02x", input[i]);
    printf("\r\n");
    printf("Expected Hash     (32 bytes) : ");
    for (uint8_t i = 0; i < SHA256_DIGEST_SIZE; i++)
        printf("%02x", expected_hash[i]);
    printf("\r\n");

    /* ========================== SINGLE SHOT␣
↪========================== */

    printf("\r\nRunning Single-Shot...\r\n");

    status = SHA256_Single_Run(single_shot_hash, input, input_length_
↪bits);
    if (status != SUCCESS)
    {
        printf("SHA256_Single_Run Failed. Status: %d\r\n", status);
        return status;
    }

    printf("Computed Hash     (32 bytes) : ");
    for (uint8_t i = 0; i < SHA256_DIGEST_SIZE; i++)
        printf("%02x", single_shot_hash[i]);
    printf("\nSHA256_Single_Run Successful\r\n");

    /* ========================== MULTI SHOT␣
↪========================== */

    printf("\r\nRunning Multi-Shot...\r\n");

    while (iterated_length_bits < input_length_bits)
    {
        remaining_bits     = input_length_bits - iterated_length_bits;
        current_chunk_bits = (remaining_bits > SHA256_BLOCK_SIZE_BITS)␣
```

(continues on next page)

(continued from previous page)

```
↪?
                            SHA256_BLOCK_SIZE_BITS : remaining_bits;

        printf("\r\nIteration %d:\r\n", iter);
        printf("  Chunk Data      (%d bytes) : ", current_chunk_bits /␣
↪8);
        for (uint16_t i = 0; i < (current_chunk_bits / 8); i++)
            printf("%02x", input[(iterated_length_bits / 8) + i]);
        printf("\r\n  Chunk Length               : %d bits (%d bytes)\
↪r\n",
            current_chunk_bits, current_chunk_bits / 8);
        printf("  Processed So Far           : %d bits\r\n", iterated_
↪length_bits);

        status = SHA256_Multi_Run(input + (iterated_length_bits / 8),␣
↪current_chunk_bits,
                                  input_length_bits, iterated_length_
↪bits);
        if (status != SUCCESS)
        {
            printf("SHA256_Multi_Run (iteration %d) Failed. Status: %d\
↪r\n", iter, status);
            return status;
        }
        printf("SHA256_Multi_Run (iteration %d) Successful\r\n", iter);

        iterated_length_bits += current_chunk_bits;
        iter++;
    }

    status = SHA256_Read_Output(multi_shot_hash, &output_length);
    if (status != SUCCESS)
    {
        printf("SHA256_Read_Output Failed. Status: %d\r\n", status);
        return status;
    }
```

(continues on next page)

(continued from previous page)

```
    printf("\r\nComputed Hash     (%d bytes) : ", output_length);
    for (uint8_t i = 0; i < output_length; i++)
        printf("%02x", multi_shot_hash[i]);
    printf("\r\nOutput Hash Length           : %d bytes\r\n", output_
↪length);
    printf("SHA256_Read_Output Successful\r\n");

    return SUCCESS;
}
```

Output

```
Running Single-Shot...
Input Data        (129 bytes) :␣
↪aa55aa550001020304051020304050600001020304050
                              ␣
↪60607080900a0b0c0d0e0f101112131415161718191a1
                              ␣
↪b1c1d1e1f20212223242526272829 2a2b2c2d2e2f303
                              ␣
↪1323334353637383 93a3b3c3d3e3f404142434445464
                              ␣
↪748494a4b4c4d4e4f505152535455565758595a5b5c5
                               d5e5f606162636465666768696a6b6c6d6e6f3f
Input Length                 : 1032 bits (129 bytes)
Expected Hash     (32 bytes)  :␣
↪a6106a2077c88babf9b802a9eb7267b99e2c643a505a97a0dd3619b551c0e0a6
Computed Hash     (32 bytes)  :␣
↪a6106a2077c88babf9b802a9eb7267b99e2c643a505a97a0dd3619b551c0e0a6
Output Hash Length             : 32 bytes
SHA256_Single_Run Successful

Running Multi-Shot...
Input Data        (129 bytes) :␣
↪aa55aa550001020304051020304050600001020304050
                              ␣
```

(continues on next page)

(continued from previous page)

```
↪0607080900a0b0c0d0e0f10111213141516171819 1a1
                                          ␣
↪b1c1d1e1f20212223242526272829 2a2b2c2d2e2f303
                                          ␣
↪13233343536373839 3a3b3c3d3e3f404142434445464
                                          ␣
↪748494a4b4c4d4e4f505152535455565758595 a5b5c5
                                           d5e5f606162636465666768696 a6b6c6d6e6f3f
Total Input Length              : 1032 bits (129 bytes)
Expected Hash     (32 bytes)  :␣
↪a6106a2077c88babf9b802a9eb7267b99e2c643a505a97a0dd3619b551c0e0a6

Iteration 0:
Chunk Data      (64 bytes)  :␣
↪aa55aa550001020304051020304050600001 0203040506
                                        ␣
↪070809000a0b0c0d0e0f10111213141516171 8191a1b1c
                                         1d1e1f20212223242526272 8292a2b2c2d2e2f
Chunk Length                  : 512 bits (64 bytes)
Total Input Length            : 1032 bits
Processed So Far              : 0 bits
SHA256_Multi_Run (iteration 0) Successful

Iteration 1:
Chunk Data      (64 bytes)  :␣
↪303132333435363738393a3b3c3d3e3f40414243444546
                                        ␣
↪4748494a4b4c4d4e4f505152535455565758595 a5b5c5d
                                         5e5f606162636465666768696 a6b6c6d6e6f
Chunk Length                  : 512 bits (64 bytes)
Total Input Length            : 1032 bits
Processed So Far              : 512 bits
SHA256_Multi_Run (iteration 1) Successful

Iteration 2:
Chunk Data      (1 bytes)   : 3f
Chunk Length                  : 8 bits (1 bytes)
```

(continues on next page)

(continued from previous page)

```
Total Input Length           : 1032 bits
Processed So Far             : 1024 bits
SHA256_Multi_Run (iteration 2) Successful

Computed Hash     (32 bytes)  :␣
↪a6106a2077c88babf9b802a9eb7267b99e2c643a505a97a0dd3619b551c0e0a6
Output Hash Length             : 32 bytes
SHA256_Read_Output Successful
```

Example Code with PSA APIs

```
#include "psa.h" /* Included to access PSA Crypto API's */
#include "io.h"  /* Included to access printf, etc */

/* Total input length in bytes */
#define SHA256_TEST_INPUT_LEN 129

int main(void)
{
    /* 129-byte multi-block input */
    uint8_t input[SHA256_TEST_INPUT_LEN] = {
        0xAA, 0x55, 0xAA, 0x55, 0x00, 0x01, 0x02, 0x03,
        0x04, 0x05, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60,
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
        0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
        0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F,
        0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
        0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F,
        0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
        0x48, 0x49, 0x4A, 0x4B, 0x4C, 0x4D, 0x4E, 0x4F,
        0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57,
        0x58, 0x59, 0x5A, 0x5B, 0x5C, 0x5D, 0x5E, 0x5F,
        0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
```

(continues on next page)

(continued from previous page)

```
        0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F,
        0x3F};

    /* Pre-computed expected SHA-256 hash for the input above */
    uint8_t expected_hash[PSA_MG_HASH_MAX_SIZE] = {
        0xA6, 0x10, 0x6A, 0x20, 0x77, 0xC8, 0x8B, 0xAB,
        0xF9, 0xB8, 0x02, 0xA9, 0xEB, 0x72, 0x67, 0xB9,
        0x9E, 0x2C, 0x64, 0x3A, 0x50, 0x5A, 0x97, 0xA0,
        0xDD, 0x36, 0x19, 0xB5, 0x51, 0xC0, 0xE0, 0xA6};

    /* Single-shot compute output */
    uint8_t compute_hash[PSA_MG_HASH_MAX_SIZE] = {0};
    size_t  compute_hash_length           = 0;

    /* Multi-shot (setup, update, finish) output */
    uint8_t multi_shot_hash[PSA_MG_HASH_MAX_SIZE] = {0};
    size_t  multi_shot_hash_length           = 0;

    psa_hash_operation_t hash_operation = PSA_HASH_OPERATION_INIT;
    psa_status_t status;

    psa_crypto_init();

    printf("\r\nInput Data        (%d bytes) : ", SHA256_TEST_INPUT_
↪LEN);
    for (uint8_t i = 0; i < SHA256_TEST_INPUT_LEN; i++)
        printf("%02x", input[i]);
    printf("\r\n");
    printf("Expected Hash     (32 bytes) : ");
    for (uint8_t i = 0; i < PSA_MG_HASH_MAX_SIZE; i++)
        printf("%02x", expected_hash[i]);
    printf("\r\n");

    /* ========================== PSA HASH COMPUTE - SINGLE SHOT␣
↪========================== */

    printf("\r\nRunning Single-Shot...\r\n");
```

(continues on next page)

(continued from previous page)

```
    status = psa_hash_compute(PSA_ALG_SHA_256, input, sizeof(input),
                              compute_hash, sizeof(compute_hash),
                              &compute_hash_length);
    if (status != PSA_SUCCESS)
    {
        printf("psa_hash_compute Failed. Status: %d\r\n", status);
        return status;
    }

    printf("Computed Hash     (%d bytes) : ", compute_hash_length);
    for (uint8_t i = 0; i < compute_hash_length; i++)
        printf("%02x", compute_hash[i]);
    printf("\r\nOutput Hash Length           : %d bytes\r\n", compute_
↪hash_length);
    printf("psa_hash_compute Successful\r\n");

    /* ============================ PSA HASH COMPUTE - MULTI SHOT␣
↪============================ */

    printf("\r\nRunning Multi-Shot...\r\n");

    status = psa_hash_setup(&hash_operation, PSA_ALG_SHA_256);
    if (status != PSA_SUCCESS)
    {
        printf("psa_hash_setup Failed. Status: %d\r\n", status);
        return status;
    }
    printf("psa_hash_setup Successful\r\n");

    status = psa_hash_update(&hash_operation, input, sizeof(input));
    if (status != PSA_SUCCESS)
    {
        printf("psa_hash_update Failed. Status: %d\r\n", status);
        return status;
    }
    printf("psa_hash_update Successful\r\n");
```

(continues on next page)

(continued from previous page)

```
    status = psa_hash_finish(&hash_operation, multi_shot_hash,␣
↪sizeof(multi_shot_hash),
                             &multi_shot_hash_length);
    if (status != PSA_SUCCESS)
    {
        printf("psa_hash_finish Failed. Status: %d\r\n", status);
        return status;
    }

    printf("Computed Hash     (%d bytes) : ", multi_shot_hash_length);
    for (uint8_t i = 0; i < multi_shot_hash_length; i++)
        printf("%02x", multi_shot_hash[i]);
    printf("\r\nOutput Hash Length           : %d bytes\r\n", multi_
↪shot_hash_length);
    printf("psa_hash_finish Successful\r\n");

    /* ============================== PSA HASH COMPARE␣
↪============================== */

    printf("\r\nRunning Hash Compare...\r\n");

    printf("Reference Hash    (%d bytes) : ", PSA_MG_HASH_MAX_SIZE);
    for (uint8_t i = 0; i < PSA_MG_HASH_MAX_SIZE; i++)
        printf("%02x", compute_hash[i]);
    printf("\r\nReference Hash Length        : %d bytes\r\n", PSA_MG_
↪HASH_MAX_SIZE);

    status = psa_hash_compare(PSA_ALG_SHA_256, input, sizeof(input),
                              compute_hash, PSA_MG_HASH_MAX_SIZE);
    if (status != PSA_SUCCESS)
    {
        printf("psa_hash_compare Failed. Status: %d\r\n", status);
        return status;
    }

    printf("Output Status                : PSA_SUCCESS\r\n");
```

(continues on next page)

(continued from previous page)

```
    printf("psa_hash_compare Successful\r\n");

    return PSA_SUCCESS;
}
```

Output

```
Input Data        (129 bytes) :␣
↪aa55aa55000102030405102030405060000102030405
                              ␣
↪0607080900a0b0c0d0e0f10111213141516171819 1a1
                              ␣
↪b1c1d1e1f20212223242526272829 2a2b2c2d2e2f303
                              ␣
↪132333435363738393a3b3c3d3e3f404142434445464
                              ␣
↪748494a4b4c4d4e4f505152535455565758595a5b5c5
                               d5e5f606162636465666768696a6b6c6d6e6f3f
Input Length                  : 129 bytes
Expected Hash     (32 bytes)  :␣
↪a6106a2077c88babf9b802a9eb7267b99e2c643a505a97a0dd3619b551c0e0a6

Running Single-Shot...
Computed Hash     (32 bytes)  :␣
↪a6106a2077c88babf9b802a9eb7267b99e2c643a505a97a0dd3619b551c0e0a6
Output Hash Length            : 32 bytes
psa_hash_compute Successful

Running Multi-Shot...
psa_hash_setup Successful
psa_hash_update Successful
Computed Hash     (32 bytes)  :␣
↪a6106a2077c88babf9b802a9eb7267b99e2c643a505a97a0dd3619b551c0e0a6
Output Hash Length            : 32 bytes
psa_hash_finish Successful
```

(continues on next page)

(continued from previous page)

```
Running Hash Compare...
Reference Hash    (32 bytes)  :␣
↪a6106a2077c88babf9b802a9eb7267b99e2c643a505a97a0dd3619b551c0e0a6
Reference Hash Length          : 32 bytes
Output Status                  : PSA_SUCCESS
psa_hash_compare Successful
```

Troubleshooting

1. Ensure that SHA engine is in correct state before loading input.
2. Ensure the correct endianess before loading input abnd while checking output.

7.1.24 TRNG (True Random Number Generator)

Example Code

```c
#include "io.h"         // Included to access functions for basic IO␣
↪operations such as printf, etc.
#include "trng.h"       // Included to access TRNG driver APIs

int main(void) {

    nist_trng_state state;              // To maintain internal␣
↪variables inside driver
    uint8_t random_string[10];
    uint16_t req_sec_strength = 256;    // Required security strength
    uint32_t no_of_bytes = 10, i;       // Number of bytes to be␣
↪generated
    int status;
    uint8_t flag = 1;

    TRNG_init(&state, req_sec_strength);  // Initialiazation of TRNG␣
↪(SEEDING)

    while (flag--) {
```

(continues on next page)

(continued from previous page)

```
        status = TRNG_Generate(&state, (void *)random_string, no_of_
↪bytes);   // Generating and storing in pointer

        if (status == 0) {

            printf("Generated random string:\n\r");

            for (i = 0; i < no_of_bytes; i++) {
                printf("Random Byte : %02x\n\r", random_string[i]);
            }
        }
        else {
            printf("An error occurred -> Error code : %d\n\r", status);
        }
    }

    TRNG_Uninstantiate(&state);       // Uninstantiating the TRNG state

    return 0;
}
```

Output

```
Generated random string:
Random Byte : 60
Random Byte : 2f
Random Byte : 5e
Random Byte : 68
Random Byte : 62
Random Byte : 6f
Random Byte : 0c
Random Byte : 80
Random Byte : 61
Random Byte : 92
```

7.1.25 Advanced Encryption Standard – Galois/Counter Mode (AES-GCM)

Functions Usage

1. **Encryption:** Feed the input text, key, IV, and AAD to the AES_GCM() function in encryption mode, which generates the corresponding cipher text and authentication tag.

```
/* AES-GCM encryption configuration */
AES_GCM_Config GCM_Config_encrypt = {
    .cipher_text = cipher_text,
    .input_text = input_text,
    .input_len_bits = input_len_bits,
    .aad = aad,
    .aad_len_bits = aad_len_bits,
    .key = key,
    .key_len_bits = key_len_bits,
    .iv = iv,
    .iv_len_bits = iv_len_bits,
    .tag = tag,
    .tag_len_bits = tag_len_bits,
    .mode = AES_GCM_ENCRYPT};

/* Perform AES-GCM authenticated encryption */
uint8_t encryption_result = AES_GCM(&GCM_Config_encrypt);
```

2. **Decryption:** Feed the cipher text, key, IV, AAD, and authentication tag to the AES_GCM() function in decryption mode, which recovers the original plaintext if the authentication tag is valid; otherwise, decryption fails with a tag mismatch error.

```
/* AES-GCM decryption configuration */
AES_GCM_Config GCM_Config_decrypt = {
    .cipher_text = cipher_text_input,
    .input_text = decrypted_output,
    .input_len_bits = cipher_text_len_bits,
    .aad = aad,
    .aad_len_bits = aad_len_bits,
```

(continues on next page)

(continued from previous page)

```
    .key = key,
    .key_len_bits = key_len_bits,
    .iv = iv,
    .iv_len_bits = iv_len_bits,
    .tag = decrypt_tag,
    .tag_len_bits = tag_len_bits,
    .mode = AES_GCM_DECRYPT};

/* Perform AES-GCM authenticated decryption */
uint8_t decryption_result = AES_GCM(&GCM_Config_decrypt);
```

Example Code for Usage

```
#include "aes_gcm.h" /*Included to access AES GCM API's*/
#include "io.h"      /*Included to access printf etc */

int main()
{
    /* AES-GCM Authenticated Encryption Example */

    uint8_t key[16] = {0x28, 0x86, 0xD9, 0x76, 0x44, 0x0E, 0xD2, 0x60,
                       0xFC, 0x64, 0x12, 0x01, 0xBE, 0x50, 0x76, 0x02};
    uint8_t key_len_bits = sizeof(key) * 8;

    uint8_t iv[12] = {0xF9, 0x48, 0x99, 0x85, 0x1D, 0x37,
                      0xAC, 0x54, 0x68, 0xD0, 0xB8, 0x53};
    uint8_t iv_len_bits = sizeof(iv) * 8;

    uint8_t input_text[16] = {0x93, 0x61, 0x3F, 0xF4, 0x1C, 0x80, 0x29,
↪ 0x5B,
                              0x18, 0x7C, 0x3B, 0x05, 0x7F, 0x0A, 0xCC,␣
↪0x71};
    uint8_t input_len_bits = sizeof(input_text) * 8;

    uint8_t aad[16] = {0x3C, 0x73, 0x6C, 0xB0, 0xF4, 0xD6, 0x0B, 0x5B,
                       0x8D, 0xF5, 0x0E, 0x41, 0xF0, 0xCE, 0x34, 0x56};
```

(continues on next page)

(continued from previous page)

```
    uint8_t aad_len_bits = sizeof(aad) * 8;

    uint8_t cipher_text[input_len_bits / 8];

    uint8_t tag[128];
    uint8_t tag_len_bits = 112; /* Tag length can be arbitrary but it␣
↪should be one of the supported values"
                                128, 120, 112, 104, 96, 64, or 32 bits.
↪ */

    AES_GCM_Config GCM_Config_encrypt = {
        .cipher_text = cipher_text,
        .input_text = input_text,
        .input_len_bits = input_len_bits,
        .aad = aad,
        .aad_len_bits = aad_len_bits,
        .key = key,
        .key_len_bits = key_len_bits,
        .iv = iv,
        .iv_len_bits = iv_len_bits,
        .tag = tag,
        .tag_len_bits = tag_len_bits,
        .mode = AES_GCM_ENCRYPT};

    uint8_t encrytion_result = AES_GCM(&GCM_Config_encrypt);
    if (encrytion_result != SUCCESS)
    {
        printf(" Authenticated Encryption Failed ! Error Code :%d\n",␣
↪encrytion_result);
    }

    printf("Encryption Result\n");
    printf("Tag : ");
    for (uint8_t i = 0; i < tag_len_bits / 8; i++)
    {
        printf("0x%02x ", tag[i]);
    }
```

(continues on next page)

(continued from previous page)

```
    /* Expected Tag
    0x03, 0xF0, 0xBD, 0x12, 0x20, 0x97, 0x51,
    0x66, 0x8E, 0x7B, 0x35, 0x46, 0x80, 0x14 */

    printf("\nCipher Text : ");
    for (uint8_t i = 0; i < input_len_bits / 8; i++)
    {
        printf("0x%02x ", cipher_text[i]);
    }
    printf("\n\r");

    /* Expected Cipher Text
    0xF7, 0x4E, 0xA3, 0x89, 0xC6, 0xB1, 0x02, 0xA5,
    0x5F, 0xC7, 0xA8, 0xD7, 0xB0, 0xD7, 0x7F, 0x13 */

    /* AES-GCM Authenticated Decryption Example */

    uint8_t cipher_text_input[16] = {0xF7, 0x4E, 0xA3, 0x89, 0xC6,␣
↪0xB1, 0x02, 0xA5,
                                     0x5F, 0xC7, 0xA8, 0xD7, 0xB0, 0xD7,
↪ 0x7F, 0x13};
    uint8_t cipher_text_len_bits = sizeof(cipher_text_input) * 8;

    uint8_t decrypted_output[16];

    /* Tag from encryption (Expected Tag ) */
    uint8_t decrypt_tag[14] = {
        0x03, 0xF0, 0xBD, 0x12, 0x20, 0x97, 0x51,
        0x66, 0x8E, 0x7B, 0x35, 0x46, 0x80, 0x14};

    AES_GCM_Config GCM_Config_decrypt = {
        .cipher_text = cipher_text_input, // input ciphertext
        .input_text = decrypted_output,   // output plaintext
        .input_len_bits = cipher_text_len_bits,
        .aad = aad,
        .aad_len_bits = aad_len_bits,
```

(continues on next page)

(continued from previous page)

```
        .key = key,
        .key_len_bits = key_len_bits,
        .iv = iv,
        .iv_len_bits = iv_len_bits,
        .tag = decrypt_tag,
        .tag_len_bits = tag_len_bits,
        .mode = AES_GCM_DECRYPT};

    uint8_t decryption_result = AES_GCM(&GCM_Config_decrypt);

    if (decryption_result == AUTH_TAG_MISMATCH)
    {
        printf("Authenticated Decryption Failed! Error Code: %d\n",␣
↪decryption_result);
    }

    else
    {
        printf("\nDecryption Successful\n");
        printf("Decrypted Plaintext: ");
        for (uint8_t i = 0; i < cipher_text_len_bits / 8; i++)
        {
            printf("0x%02x ", decrypted_output[i]);
        }
        printf("\n");
    }
}
```

Output

```
Encryption Result:
Tag : 0x03 0xf0 0xbd 0x12 0x20 0x97 0x51 0x66 0x8e 0x7b 0x35 0x46 0x80␣
↪0x14
Cipher Text : 0xf7 0x4e 0xa3 0x89 0xc6 0xb1 0x02 0xa5 0x5f 0xc7 0xa8␣
↪0xd7 0xb0 0xd7 0x7f 0x13
```

(continues on next page)

(continued from previous page)

```
Decryption Successful
Decrypted Plaintext: 0x93 0x61 0x3f 0xf4 0x1c 0x80 0x29 0x5b 0x18 0x7c␣
↪0x3b 0x05 0x7f 0x0a 0xcc 0x71
```

Troubleshooting

1. Ensure that AES engine is in correct state before loading input.
2. Ensure that all AES-GCM configuration parameters meet the expected requirements to prevent incorrect operation.

7.1.26 ASCON

Functions Usage

1. **Encryption:** Initialize the key, nonce, plaintext, and optional AAD, then call `crypto_aead_encrypt()` to generate the ciphertext with an appended authentication tag.

```c
/* Key and nonce setup */
uint8_t key[ASCON_KEYBYTES] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F
};

uint8_t nonce[ASCON_NPUBBYTES] = {
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F
};

/* Additional Authenticated Data (AAD) */
uint8_t aad[] = {0x30, 0x31, 0x32, 0x33, 0x34, 0x35};
size_t aad_len = sizeof(aad);

/* Plaintext to encrypt */
uint8_t plaintext[] = {0x20, 0x21, 0x22, 0x23};
```

(continues on next page)

(continued from previous page)

```
size_t plaintext_len = sizeof(plaintext);

/* Output buffer for ciphertext + tag */
uint8_t ciphertext[plaintext_len + ASCON_TAG_SIZE];
size_t ciphertext_len;

/* Perform ASCON-128a authenticated encryption */
ASCON_AEAD128a_Config aead_cfg = {
    .output     = ciphertext,
    .output_len = &ciphertext_len,
    .input      = plaintext,
    .input_len  = plaintext_len,
    .aad        = aad,
    .aad_len    = aad_len,
    .sec_nonce  = NULL,  // optional secret nonce (not used)
    .pub_nonce  = nonce,
    .key        = key,
    .mode       = ASCON_ENCRYPT
};

ASCON_AEAD128a(&aead_cfg);
```

2. **Decryption:** Provide the ciphertext (including tag), key, nonce, and AAD to `crypto_aead_decrypt()`, which verifies the authentication tag and recovers the original plaintext if authentication succeeds; otherwise, decryption fails with a non-zero return value.

```
/* Output buffer for decrypted plaintext */
uint8_t decrypted_text[plaintext_len];
size_t decrypted_text_len;

/* Perform ASCON-128a authenticated decryption */
aead_cfg.output     = decrypted_text;
aead_cfg.output_len = &decrypted_text_len;
aead_cfg.input      = ciphertext;
aead_cfg.input_len  = ciphertext_len;
aead_cfg.aad        = aad;
aead_cfg.aad_len    = aad_len;
```

(continues on next page)

(continued from previous page)

```
aead_cfg.sec_nonce  = NULL;  // optional secret nonce (not used)
aead_cfg.pub_nonce  = nonce;
aead_cfg.key        = key;
aead_cfg.mode       = ASCON_DECRYPT;

uint16_t result = ASCON_AEAD128a(&aead_cfg);

if (result != SUCCESS) {
    /* Authentication failed - ciphertext was tampered with */
    printf("Decryption failed\n");
}
```

3. **Hash:** Provide input data to `crypto_hash()` to generate a fixed 256-bit cryptographic hash.

```
/* Input data to hash */
uint8_t hash_input[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07}
 ↪;
size_t hash_input_len = sizeof(hash_input);

/* Output buffer for hash (32 bytes) */
uint8_t hash_output[ASCON_HASH_SIZE];

/* Compute ASCON-Hash256 */
ASCON_Hash256_Config hash256_cfg = {
    .hash_output = hash_output,
    .input       = hash_input,
    .input_len   = hash_input_len
};

ASCON_Hash256(&hash256_cfg);
```

4. **XOF128:** Provide input data and desired output length to `crypto_xof128()` to generate a variable-length hash output.

```
/* Output buffer for XOF (variable length) */
 uint8_t xof_output[64];
 size_t xof_output_len = sizeof(xof_output);
```

(continues on next page)

(continued from previous page)

```
/* Compute ASCON-XOF128 */
ASCON_XOF128_Config xof128_cfg = {
    .hash_output     = xof_output,
    .hash_output_len = xof_output_len,
    .input           = hash_input,
    .input_len       = hash_input_len
};

ASCON_XOF128(&xof128_cfg);
```

5. **cXOF128:** Provide input data, customization string, and desired output length to `crypto_cxof128()` to generate a customized variable-length hash output with domain separation.

```
/* Customization string for domain separation */
uint8_t customization_string[] = {0x10, 0x11, 0x12, 0x13, 0x14, 0x15,␣
↪0x16, 0x17};
size_t customization_string_len = sizeof(customization_string);

/* Output buffer for cXOF */
uint8_t cxof_output[64];
size_t cxof_output_len = sizeof(cxof_output);

/* Compute ASCON-cXOF128 with customization */
ASCON_CXOF128_Config cxof128_cfg = {
    .hash_output        = cxof_output,
    .hash_output_len    = cxof_output_len,
    .input              = hash_input,
    .input_len          = hash_input_len,
    .custom_string      = customization_string,
    .custom_string_len  = customization_string_len
};

ASCON_CXOF128(&cxof128_cfg);
```

Example Code for Usage

```
#include "ascon.h" /* Included to access ASCON APIs */
#include "io.h"    /* Included to access printf etc */
#include "errors.h" /* Included to access error codes */

#define ASCON_KEYBYTES 16
#define ASCON_NPUBBYTES 16
#define ASCON_TAG_SIZE 16
#define ASCON_HASH_SIZE 32

#define ASCON_128a_ENCRYPT_DECRYPT
#define ASCON_HASH256_variant
#define ASCON_cXOF128_variant
#define ASCON_XOF128_variant

uint8_t hash_input[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07}
 ↪;
size_t hash_input_len = sizeof(hash_input);

int main()
{
    /* ASCON-128a AEAD Encryption/Decryption Example */

#ifdef ASCON_128a_ENCRYPT_DECRYPT
    uint8_t key[ASCON_KEYBYTES] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F};

    uint8_t nonce[ASCON_NPUBBYTES] = {
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F};

    /* Associated data (AAD) that is authenticated but not encrypted */
    uint8_t aad[] = {0x30, 0x31, 0x32, 0x33, 0x34, 0x35};
    size_t aad_len = sizeof(aad);

    uint8_t plaintext[] = {0x20, 0x21, 0x22, 0x23};
```

(continues on next page)

(continued from previous page)

```
    size_t plaintext_len = sizeof(plaintext);

    uint8_t ciphertext[plaintext_len + ASCON_TAG_SIZE]; // ciphertext␣
↪+ tag
    size_t ciphertext_len;

    uint8_t decrypted_text[plaintext_len];
    size_t decrypted_text_len;

    uint16_t result = 0;

    /* Encryption */
    ASCON_AEAD128a_Config aead_cfg = {
        .output = ciphertext,
        .output_len = &ciphertext_len,
        .input = plaintext,
        .input_len = plaintext_len,
        .aad = aad,
        .aad_len = aad_len,
        .sec_nonce = NULL, // optional secret nonce (not used)
        .pub_nonce = nonce,
        .key = key,
        .mode = ASCON_ENCRYPT
    };

    ASCON_AEAD128a(&aead_cfg);

    printf("Encryption Result\n");

    /* Print ciphertext (without tag) */
    printf("Ciphertext:\n");
    for (uint64_t i = 0; i < plaintext_len; i++) {
        printf("0x%02x ", ciphertext[i]);
    }
    printf("\n");

    /* Print authentication tag */
```

(continues on next page)

(continued from previous page)

```
    printf("Tag:\n");
    for (uint64_t i = 0; i < ASCON_TAG_SIZE; i++) {
        printf("0x%02x ", ciphertext[plaintext_len + i]);
    }
    printf("\n");

    /* Decryption */
    aead_cfg.output = decrypted_text;
    aead_cfg.output_len = &decrypted_text_len;
    aead_cfg.input = ciphertext;
    aead_cfg.input_len = ciphertext_len;
    aead_cfg.aad = aad;
    aead_cfg.aad_len = aad_len;
    aead_cfg.sec_nonce = NULL; // optional secret nonce (not used)
    aead_cfg.pub_nonce = nonce;
    aead_cfg.key = key;
    aead_cfg.mode = ASCON_DECRYPT;

    result = ASCON_AEAD128a(&aead_cfg);

    if (result != SUCCESS) {
        printf("Decryption failed \n");
        return result;
    }

    printf("\n");
    printf("Decryption Result\n");

    /* Print decrypted plaintext */
    printf("Plaintext:\n");
    for (size_t i = 0; i < decrypted_text_len; i++) {
        printf("0x%02x ", decrypted_text[i]);
    }
    printf("\n");
#endif

    /* ASCON-Hash256 Example */
```

(continues on next page)

(continued from previous page)

```
#ifdef ASCON_HASH256_variant

    uint8_t hash_output[ASCON_HASH_SIZE];

    ASCON_Hash256_Config has256_cfg = {
        .hash_output = hash_output,
        .input = hash_input,
        .input_len = hash_input_len
    };

    ASCON_Hash256(&has256_cfg);

    /* Print ASCON-Hash256 result */
    printf("\n");
    printf("Hash256 Result\n");
    for (uint64_t i = 0; i < ASCON_HASH_SIZE; i++) {
        printf("0x%02x ", hash_output[i]);
    }
    printf("\n");

#endif

    /* ASCON-XOF128 Example */

#ifdef ASCON_XOF128_variant
    uint8_t xof_output[64];
    size_t xof_output_len = sizeof(xof_output);

    ASCON_XOF128_Config xof128_cfg = {
        .hash_output = xof_output,
        .hash_output_len = xof_output_len,
        .input = hash_input,
        .input_len = hash_input_len
    };

    ASCON_XOF128(&xof128_cfg);
```

(continues on next page)

(continued from previous page)

```
    /* Print ASCON-XOF128 result */
    printf("\n");
    printf("XOF128 Result\n");
    for (uint64_t i = 0; i < xof_output_len; i++) {
        printf("0x%02x ", xof_output[i]);
    }
    printf("\n");

#endif

    /* ASCON-cXOF128 (Customizable XOF) Example */

#ifdef ASCON_cXOF128_variant
    uint8_t customization_string[] = {0x10, 0x11, 0x12, 0x13, 0x14,␣
↪0x15, 0x16, 0x17};
    size_t customization_string_len = sizeof(customization_string);
    uint8_t cxof_output[64];
    size_t cxof_output_len = sizeof(cxof_output);

    ASCON_CXOF128_Config cxof128_cfg = {
        .hash_output = cxof_output,
        .hash_output_len = cxof_output_len,
        .input = hash_input,
        .input_len = hash_input_len,
        .custom_string = customization_string,
        .custom_string_len = customization_string_len
    };

    ASCON_CXOF128(&cxof128_cfg);

    /* Print ASCON-cXOF128 result */
    printf("\n");
    printf("cXOF128 Result\n");
    for (uint64_t i = 0; i < cxof_output_len; i++) {
        printf("0x%02x ", cxof_output[i]);
    }
```

(continues on next page)

(continued from previous page)

```
    printf("\n");

#endif
}
```

Output

```
Encryption Result
Ciphertext:
0x93 0x10 0xc6 0xdd
Tag:
0x7d 0xc7 0x7c 0x54 0xaf 0x7a 0xd4 0xae 0xbe 0x2f 0x29 0xbc 0x1a 0x93␣
↪0x9d 0x18

Decryption Result
Plaintext:
0x20 0x21 0x22 0x23

Hash256 Result
0xb8 0x8e 0x49 0x7a 0xe8 0xe6 0xfb 0x64 0x1b 0x87 0xef 0x62 0x2e 0xb8␣
↪0xf2 0xfc
0xa0 0xed 0x95 0x38 0x3f 0x7f 0xfe 0xbe 0x16 0x7a 0xcf 0x10 0x99 0xba␣
↪0x76 0x4f

XOF128 Result
0x8d 0x18 0x86 0xf5 0xd3 0xec 0x4a 0xf8 0xd1 0x5b 0x44 0xbc 0x62 0xb7␣
↪0x4d 0xa6
0xea 0x91 0xbc 0x28 0xfb 0x82 0xf9 0xc3 0x40 0x79 0xb5 0xed 0x6e 0x38␣
↪0xb6 0xc9
0x51 0x80 0x3d 0x7d 0xfb 0x3c 0x5e 0x51 0x2a 0x0e 0xf5 0xe4 0x06 0x00␣
↪0x62 0xa6
0xfd 0x06 0x7f 0x9c 0x73 0xef 0x9b 0xee 0x52 0x74 0x11 0xbd 0xa6 0x7f␣
↪0xc8 0x96

cXOF128 Result
0x7c 0x2f 0xc5 0x90 0x4c 0xc9 0xac 0x51 0x49 0x02 0xe5 0x07 0x47 0xe3␣
```

(continues on next page)

(continued from previous page)

```
↪0x6f 0x99
0x3d 0xbd 0xe0 0x34 0xcb 0x05 0x58 0x7a 0xf1 0x43 0x2b 0xf8 0x1c 0x74␣
↪0xb1 0xec
0x87 0xec 0xf6 0x17 0x97 0x01 0x06 0x44 0x94 0x48 0x74 0x76 0xf6 0x07␣
↪0x71 0x58
0x53 0xd7 0x4c 0x57 0x27 0x92 0x5e 0xbf 0x49 0x74 0xe2 0x5e 0xb8 0x87␣
↪0x89 0x19
```

Troubleshooting

1. Ensure the ciphertext buffer is large enough for encrypted data and authentication tag (plaintext_len + CRYPTO_ABYTES).
2. Verify that all ASCON configuration parameters meet the expected requirements:
   - Key length must be CRYPTO_KEYBYTE(16 bytes for ASCON-128a)
   - Nonce length must be CRYPTO_NPUBBYTES (16 bytes)
   - Authentication tag length must be CRYPTO_ABYTES (16 bytes)
   - Hash output length must be ASCON_HASH_SIZE (32 bytes for Hash256)
3. If decryption fails, verify the ciphertext integrity, correct key/nonce/AAD usage, and authentication tag validity.

7.1.27 Elliptic Curve Cryptography (ECC)

Functions Usage

1. **ECDSA Signature generation:** Compute the hash of a message and call `ECC_sign()` to generate the signature.

```
/* Example message hash (SHA-256 recommended) */
uint8_t message_hash[32] = {
    0xA1, 0xB2, 0xC3, 0xD4, 0xE5, 0xF6, 0x07, 0x18,
    0x29, 0x3A, 0x4B, 0x5C, 0x6D, 0x7E, 0x8F, 0x90,
    0x01, 0x12, 0x23, 0x34, 0x45, 0x56, 0x67, 0x78,
    0x89, 0x9A, 0xAB, 0xBC, 0xCD, 0xDE, 0xEF, 0xF0
};
```

(continues on next page)

(continued from previous page)

```
/* Private key for secp256r1 */
uint8_t private_key[32] = {
    0x1F, 0x2E, 0x3D, 0x4C, 0x5B, 0x6A, 0x79, 0x88,
    0x97, 0xA6, 0xB5, 0xC4, 0xD3, 0xE2, 0xF1, 0x00,
    0x10, 0x21, 0x32, 0x43, 0x54, 0x65, 0x76, 0x87,
    0x98, 0xA9, 0xBA, 0xCB, 0xDC, 0xED, 0xFE, 0x0F
};

/* Buffer to hold the signature (2 * curve size bytes) */
uint8_t signature[64];

/* Select the curve */
ECC_Curve curve = ECC_secp256r1();

/* Generate ECDSA signature */
int success = ECC_sign(private_key, message_hash, sizeof(message_hash),
↪ signature, curve);
```

2. **ECDSA Signature verification:** Verify a message hash against a signer's public key and signature using `ECC_verify()`.

```
/* Signer's public key for secp256r1 (64 bytes) */
uint8_t public_key[64] = {
    0x04, 0x5A, 0x6B, 0x7C, 0x8D, 0x9E, 0xAF, 0xB0,
    0xC1, 0xD2, 0xE3, 0xF4, 0x05, 0x16, 0x27, 0x38,
    0x49, 0x5A, 0x6B, 0x7C, 0x8D, 0x9E, 0xAF, 0xB0,
    0xC1, 0xD2, 0xE3, 0xF4, 0x05, 0x16, 0x27, 0x38,
    0x39, 0x4A, 0x5B, 0x6C, 0x7D, 0x8E, 0x9F, 0xA0,
    0xB1, 0xC2, 0xD3, 0xE4, 0xF5, 0x06, 0x17, 0x28,
    0x39, 0x4A, 0x5B, 0x6C, 0x7D, 0x8E, 0x9F, 0xA0,
    0xB1, 0xC2, 0xD3, 0xE4, 0xF5, 0x06, 0x17, 0x28
};

/* Hash of the signed message (SHA-256 recommended) */
uint8_t message_hash[32] = {
    0xA1, 0xB2, 0xC3, 0xD4, 0xE5, 0xF6, 0x07, 0x18,
    0x29, 0x3A, 0x4B, 0x5C, 0x6D, 0x7E, 0x8F, 0x90,
```

(continues on next page)

(continued from previous page)

```
    0x01, 0x12, 0x23, 0x34, 0x45, 0x56, 0x67, 0x78,
    0x89, 0x9A, 0xAB, 0xBC, 0xCD, 0xDE, 0xEF, 0xF0
};

/* Signature to verify (2 * curve size bytes, 64 for secp256r1) */
uint8_t signature[64] = {
    0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88,
    0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x00,
    0x10, 0x21, 0x32, 0x43, 0x54, 0x65, 0x76, 0x87,
    0x98, 0xA9, 0xBA, 0xCB, 0xDC, 0xED, 0xFE, 0x0F,
    0x1F, 0x2E, 0x3D, 0x4C, 0x5B, 0x6A, 0x79, 0x88,
    0x97, 0xA6, 0xB5, 0xC4, 0xD3, 0xE2, 0xF1, 0x00,
    0x0F, 0x1E, 0x2D, 0x3C, 0x4B, 0x5A, 0x69, 0x78,
    0x87, 0x96, 0xA5, 0xB4, 0xC3, 0xD2, 0xE1, 0xF0
};

/* Select the curve */
ECC_Curve curve = ECC_secp256r1();

/* Verify ECDSA signature */
int valid = ECC_verify(public_key, message_hash, sizeof(message_hash),␣
↪signature, curve);

if(valid){
 printf("Signature Verification failed\n");
}
```

3. **ECC Key Pair Generation:** Generate a public/private key pair using `ECC_make_key()`.

```
/* Select the curve */
ECC_Curve curve = ECC_secp256r1();

/* Buffers to hold the generated keys */
uint8_t public_key[64];   /* 2 * curve size bytes */
uint8_t private_key[32];  /* curve size bytes */

/* Generate the key pair */
```

(continues on next page)

(continued from previous page)

```
int success = ECC_make_key(public_key, private_key, curve);

if (success) {
    printf("Key pair generated successfully.\n");
} else {
    printf("Key pair generation failed.\n");
}
```

4. **ECDH Shared Secret Computation:** Compute a shared secret using your private key and a peer's public key with `ECC_shared_secret()`.

```
/* Select the curve */
ECC_Curve curve = ECC_secp256r1();

/* Buffers */
uint8_t my_private_key[32];      /* Your private key */
uint8_t peer_public_key[64];     /* Peer's public key (2 * curve size) */
uint8_t shared_secret[32];       /* Output buffer (curve size bytes) */

/* Compute the shared secret */
int success = ECC_shared_secret(peer_public_key, my_private_key,␣
↪shared_secret, curve);

if (success) {
    printf("Shared secret computed successfully.\n");
} else {
    printf("Failed to compute shared secret.\n");
}
```

Example usage for ECDSA

This is an example code for ECDSA signature generation and Verification.

```
#include "ECC.h"
#include "io.h"

void main() {
```

(continues on next page)

(continued from previous page)

```
    /* Example using secp256r1 test vector */
    ECC_Curve curve = ECC_secp256r1();

    /* Private key (32 bytes) */
    uint8_t private_key[32] = {
        0x51, 0x9b, 0x42, 0x3d, 0x71, 0x5f, 0x8b, 0x58,
        0x1f, 0x4f, 0xa8, 0xee, 0x59, 0xf4, 0x77, 0x1a,
        0x5b, 0x44, 0xc8, 0x13, 0x0b, 0x4e, 0x3e, 0xac,
        0xca, 0x54, 0xa5, 0x6d, 0xda, 0x72, 0xb4, 0x64
    };

    /* Corresponding public key (64 bytes) */
    uint8_t public_key[64] = {
        0x1c, 0xcb, 0xe9, 0x1c, 0x07, 0x5f, 0xc7, 0xf4,
        0xf0, 0x33, 0xbf, 0xa2, 0x48, 0xdb, 0x8f, 0xcc,
        0xd3, 0x56, 0x5d, 0xe9, 0x4b, 0xbf, 0xb1, 0x2f,
        0x3c, 0x59, 0xff, 0x46, 0xc2, 0x71, 0xbf, 0x83,
        0xce, 0x40, 0x14, 0xc6, 0x88, 0x11, 0xf9, 0xa2,
        0x1a, 0x1f, 0xdb, 0x2c, 0x0e, 0x61, 0x13, 0xe0,
        0x6d, 0xb7, 0xca, 0x93, 0xb7, 0x40, 0x4e, 0x78,
        0xdc, 0x7c, 0xcd, 0x5c, 0xa8, 0x9a, 0x4c, 0xa9
    };

    /* Message hash to sign (32 bytes) */
    uint8_t message_hash[32] = {
        0x44, 0xac, 0xf6, 0xb7, 0xe3, 0x6c, 0x13, 0x42,
        0xc2, 0xc5, 0x89, 0x72, 0x04, 0xfe, 0x09, 0x50,
        0x4e, 0x1e, 0x2e, 0xfb, 0x1a, 0x90, 0x03, 0x77,
        0xdb, 0xc4, 0xe7, 0xa6, 0xa1, 0x33, 0xec, 0x56
    };

    /* Signature buffer (64 bytes for r+s) */
    uint8_t signature[64];

    /* Sign the hash */
    if (ECC_sign(private_key, message_hash, sizeof(message_hash),␣
↪signature, curve)) {
```

(continues on next page)

(continued from previous page)

```
        printf("Signature generated successfully!\n");
        printf("r: ");
        for (int i = 0; i < 32; i++) {
            printf("%02x", signature[i]);
        }
        printf("\n");
        printf("s: ");
        for (int i = 0; i < 32; i++) {
            printf("%02x", signature[32 + i]);
        }
        printf("\n");
    } else {
        printf("Failed to generate signature.\n");
        return 1;
    }

    /* Verify the signature */
    if (ECC_verify(public_key, message_hash, sizeof(message_hash),␣
↪signature, curve)) {
        printf("Signature verification succeeded!\n");
    } else {
        printf("Signature verification failed!\n");
    }

    while(1);
}
```

Output for ECDSA

```
Signature generated successfully!

r: 6890a08a412b543c38cbd0ef20b1eb26c42930070545b0d3fe1abe5137820e6b
s: 6fa4c53a388c2ec962a91770aec29193997b2d5b8b07c28a4299dd909571e6c0

Signature verification succeeded!
```

Example usage for ECDH

This is an Example code for ECDH shared secret key agreement.

```
#include "ECC.h"
#include "io.h"

/* Helper to print bytes in hex */
void print_hex(const uint8_t *data, unsigned size) {
    for (unsigned i = 0; i < size; i++) {
        printf("%02x", data[i]);
    }
    printf("\n");
}

void main() {
    /* Use secp256r1 curve */
    ECC_Curve curve = ECC_secp256r1();

    uint8_t private1[32], private2[32];
    uint8_t public1[64], public2[64];
    uint8_t secret1[32], secret2[32];

    /* Generate key pairs */
    if (!ECC_make_key(public1, private1, curve) ||
        !ECC_make_key(public2, private2, curve)) {
        printf("Key generation failed\n");
        return 1;
    }

    /* Compute shared secrets */
    if (!ECC_shared_secret(public2, private1, secret1, curve) ||
        !ECC_shared_secret(public1, private2, secret2, curve)) {
        printf("Shared secret computation failed\n");
        return 1;
    }

    /* Check if shared secrets match */
    if (memcmp(secret1, secret2, sizeof(secret1)) == 0) {
```

(continues on next page)

(continued from previous page)

```
        printf("Shared secrets match!\n");
        printf("Shared secret 1: "); print_hex(secret1,␣
↪sizeof(secret1));
        printf("Shared secret 2: "); print_hex(secret2,␣
↪sizeof(secret2));
    } else {
        printf("Error: shared secrets do NOT match!\n");
    }

    while(1);
}
```

Output for ECDH

```
Shared secrets match!
Shared secret 1:␣
↪1275738263e4e1c9ef85e7648de315789a08c7a25b97bda503477741836ce0ab
Shared secret 2:␣
↪1275738263e4e1c9ef85e7648de315789a08c7a25b97bda503477741836ce0ab
```

7.1.28 SHA software driver example

Example Code for SHA512

This example explores how to generate has for the given message using SHA512.

```
#include <sha_software.h>
#include <stdio.h>

void main() {
  unsigned char hash[SHA512_DIGEST_SIZE];
  SHA512 sha512[1];

  static const unsigned char msg[] = {
      0xfd, 0x22, 0x03, 0xe4, 0x67, 0x57, 0x4e, 0x83, 0x4a, 0xb0, 0x7c,
```

(continues on next page)

(continued from previous page)

```
↪ 0x90,
      0x97, 0xae, 0x16, 0x45, 0x32, 0xf2, 0x4b, 0xe1, 0xeb, 0x5d, 0x88,
↪ 0xf1,
      0xaf, 0x77, 0x48, 0xce, 0xff, 0x0d, 0x2c, 0x67, 0xa2, 0x1f, 0x4e,
↪ 0x40,
      0x97, 0xf9, 0xd3, 0xbb, 0x4e, 0x9f, 0xbf, 0x97, 0x18, 0x6e, 0x0d,
↪ 0xb6,
      0xdb, 0x01, 0x00, 0x23, 0x0a, 0x52, 0xb4, 0x53, 0xd4, 0x21, 0xf8,
↪ 0xab,
      0x9c, 0x9a, 0x60, 0x43, 0xaa, 0x32, 0x95, 0xea, 0x20, 0xd2, 0xf0,
↪ 0x6a,
      0x2f, 0x37, 0x47, 0x0d, 0x8a, 0x99, 0x07, 0x5f, 0x1b, 0x8a, 0x83,
↪ 0x36,
      0xf6, 0x22, 0x8c, 0xf0, 0x8b, 0x59, 0x42, 0xfc, 0x1f, 0xb4, 0x29,
↪ 0x9c,
      0x7d, 0x24, 0x80, 0xe8, 0xe8, 0x2b, 0xce, 0x17, 0x55, 0x40, 0xbd,
↪ 0xfa,
      0xd7, 0x75, 0x2b, 0xc9, 0x5b, 0x57, 0x7f, 0x22, 0x95, 0x15, 0x39,
↪ 0x4f,
      0x3a, 0xe5, 0xce, 0xc8, 0x70, 0xa4, 0xb2, 0xf8};

  int len = sizeof(msg);
  if (SHA_InitSha512(sha512) != 0) {
    printf("\nInit failed");
  } else {
    SHA_Sha512Update(sha512, msg, len);
    SHA_Sha512Final(sha512, hash);
    printf("\nSize : %d", len);
    printf("\nHash : ", &msg);
    for (int i = 0; i < 64; i++)
      printf("%02x", (hash[i]));
    /*Expected hash value :
    *␣
↪a21b1077d52b27ac545af63b32746c6e3c51cb0cb9f281eb9f3580a6d4996d5c9917d2a6e484627a9
↪
  }
}
```

7.1.29 Hash-based Message Authentication Code (HMAC)

HMAC Sample Application

This example shows how to use HMAC driver to compute the final digest.

Example Code:

Note

To use the SHA-2 and SHA-3 digest size and block size macros, include the corresponding header files.

```
#include <stdint.h>
#include "io.h"
#include "hmac.h"
#include "errors.h"
#include "sha3.h"

/* Example key and message */
static const uint8_t key[] = {
    0x9f, 0x9f, 0xb2, 0x80, 0xad, 0xf1, 0x2e, 0x73,
    0x95, 0x48, 0xb1, 0xd6, 0x76, 0xcb, 0x79, 0x4d,
    0x68, 0x5b, 0x91, 0x04, 0xe6, 0x3b, 0x61, 0x9b,
    0x05, 0x5c, 0xb6, 0x0f
};

static const uint8_t message[] = {
    0x91, 0x51, 0x3d, 0xd6, 0xde, 0x40, 0xa1, 0xc2,
    0x3f, 0x8d, 0x1e, 0xb0, 0xab, 0x8f, 0x5e, 0xa6,
    0xf6, 0x83, 0x55, 0x06, 0xec, 0x75, 0x08, 0x94
};

static void print_hexa(const uint8_t *buf, size_t len) {
    for (size_t i = 0; i < len; i++) {
        printf("%02x", buf[i]);
    }
    printf("\n\r");
}
```

(continues on next page)

(continued from previous page)

```
void main() {
    uint16_t ret;
    uint8_t buffer[HMAC_Get_Context_Size()];
    Hmac *hmac = (Hmac*)buffer;
    uint8_t mac[SHA3_224_DIGEST_SIZE];

    /* Reset HMAC context */
    ret = HMAC_Reset(hmac);
    if (ret != SUCCESS) {
        printf("HMAC_Reset failed \n\r");
        return;
    }

    /* Set key with SHA-256 */
    ret = HMAC_Set_Key(hmac, SHA3_224, key, sizeof(key));
    if (ret != SUCCESS) {
        printf("HMAC_Set_Key failed \n\r");
        return;
    }

    /* Update with message */
    ret = HMAC_Update(hmac, message, sizeof(message));
    if (ret != SUCCESS) {
        printf("HMAC_Update failed \n\r");
        return;
    }

    /* Finalize HMAC */
    ret = HMAC_Final(hmac, mac);
    if (ret != SUCCESS) {
        printf("HMAC_Final failed \n\r");
        return;
    }

    printf("HMAC-SHA3_224: ");
    print_hexa(mac, SHA3_224_DIGEST_SIZE);
}
```

Output

```
HMAC-SHA3-224 :␣
↪c49f485f16bbc63695ee3e5221d8b3dfda5b85aa461dbe925e44d18d
```

7.1.30 Software USB Example (Mouse)

The following example demonstrates a USB HID mouse implemented using a GPIO bit-banged USB driver.

```
/*
GPIO Bit banged Mouse
Ideas Behind GPIO Bit Banged USB protocol.

1)Initially DPLUS and DMINUS are configured as input and interrupt␣
↪for DMINUS is configured which will enter ISR when pin is driven LOW,
2)In secureiot board,DMINUS is PULLED-UP and DPLUS is left in␣
↪floating.
3)After connecting to host,the PULLUP which is in DMINUS is detected␣
↪by host side to decide the speed of device.
4)Now the Host will start to send its packets.
5)First host will send GET DEVICE Descriptor to find the type of␣
↪DEVICE and how many configurations are possible.
6)GET DEVICE descriptor comes as 2 packets->Setup token packet and␣
↪DATA0 PID packet
7)When packet has arrived it will go into ISR which will sample bits␣
↪till SE0 is encountered and store in a buffer after performing NRZI␣
↪decoding.
8)After deciding in software that GET DEVICE Descriptor request has␣
↪arrived device will reply DATA max of 8 bytes per packet for every␣
↪PID IN token recieved from HOST.
*/

#pragma GCC push_options
#pragma GCC optimize ("O0")
#include"io.h"
#include "log.h"
```

(continues on next page)

(continued from previous page)

```
#include "gpio.h"
#include "plic.h"
#include"gptimer.h"
#include "usb_bitbang.h"

#define BUTTON 28

typedef enum {
    WAIT_FOR_SETUP_TOKEN,
    WAIT_FOR_SETUP_DATA,
    WAIT_FOR_IN_TOKEN_FOR_GET_REQUESTS,
    WAIT_FOR_OUT_TOKEN_FOR_GET_REQUESTS,
    WAIT_FOR_DATA_ZLP
} USB_CONTROL_STAGE;

enum {
    USB_HOST_DISCONNECTED,
    USB_HOST_CONNECTED,
};
enum {
    USB_DEVICE_WAITING_FOR_RESET,
    USB_DEVICE_ADDR_0,
    USB_DEVICE_ADDRESSED
};


#define MAX_PACKET_SIZE 8
#define DEVICE_DESC_LEN 18
#define CONFIG_DESC_LEN 34
#define HID_REPORT_DESC_LEN sizeof(hid_report_descriptor)

// USB request codes
#define USB_REQ_GET_DESCRIPTOR     0x06
#define USB_REQ_SET_CONFIGURATION  0x09
#define USB_REQ_SET_ADDRESS        0x05
#define USB_REQ_SET_IDLE           0x0A

```

(continues on next page)

(continued from previous page)

```
// USB descriptor types
#define USB_DEVICE_DESCRIPTOR_TYPE        0x01
#define USB_CONFIGURATION_DESCRIPTOR_TYPE 0x02
#define USB_STRING_DESCRIPTOR_TYPE        0x03
#define HID_REPORT_DESCRIPTOR_TYPE        0x22

typedef struct {
    uint8_t bmRequestType;
    uint8_t bRequest;
    uint16_t wValue;
    uint16_t wIndex;
    uint16_t wLength;
} USB_SetupPacket;

// Globals and state variables
volatile uint8_t state = 0;

Packet P;                 // Packet structure for transmission

uint8_t *descriptor_data = NULL;      // Pointer to current␣
↪descriptor being sent
volatile uint8_t descriptor_length = 0;           // Total length of␣
↪current descriptor
volatile uint8_t bytes_sent = 0;                  // Count of bytes␣
↪sent so far

USB_SetupPacket setup_packet;
volatile uint8_t request_type = 0;
// Standard Device Descriptor for a USB HID mouse (18 bytes)
uint8_t device_descriptor[18] = {
    0x12,                       // bLength
    0x01,                       // bDescriptorType: Device
    0x00, 0x02,                 // bcdUSB: USB 2.0
    0x00,                       // bDeviceClass: Defined at interface␣
↪level
    0x00,                       // bDeviceSubClass
    0x00,                       // bDeviceProtocol
```

(continues on next page)

(continued from previous page)

```
    MAX_PACKET_SIZE,            // bMaxPacketSize0
    0x5E, 0x04,                 // idVendor (example: 0x045E Microsoft)
    0x23, 0x09,                 // idProduct (example product ID)
    0x00, 0x01,                 // bcdDevice (1.00)
    0x01,                       // iManufacturer
    0x02,                       // iProduct
    0x00,                       // iSerialNumber
    0x01                        // bNumConfigurations
};



// Configuration Descriptor (example, 34 bytes total)
// Includes Configuration, Interface, HID, and Endpoint descriptors
uint8_t configuration_descriptor[34] = {
    // Configuration Descriptor
    0x09,                       // bLength
    0x02,                       // bDescriptorType: Configuration
    0x22, 0x00,                 // wTotalLength (34 bytes)
    0x01,                       // bNumInterfaces
    0x01,                       // bConfigurationValue
    0x00,                       // iConfiguration
    0xA0,                       // bmAttributes (Bus powered, remote wakeup)
    0x32,                       // bMaxPower (100 mA)

    // Interface Descriptor
    0x09,                       // bLength
    0x04,                       // bDescriptorType: Interface
    0x00,                       // bInterfaceNumber
    0x00,                       // bAlternateSetting
    0x01,                       // bNumEndpoints
    0x03,                       // bInterfaceClass: HID
    0x01,                       // bInterfaceSubClass: Boot Interface
    0x02,                       // bInterfaceProtocol: Mouse
    0x00,                       // iInterface
```

(continues on next page)

(continued from previous page)

```
124
125     // HID Descriptor
126     0x09,                       // bLength
127     0x21,                       // bDescriptorType: HID
128     0x11, 0x01,                 // bcdHID 1.11
129     0x00,                       // bCountryCode
130     0x01,                       // bNumDescriptors
131     0x22,                       // bDescriptorType (Report)
132     0x32, 0x00,                 // wDescriptorLength (50 bytes␣
    ↪hypothetical)
133
134     // Endpoint Descriptor
135     0x07,                       // bLength
136     0x05,                       // bDescriptorType: Endpoint
137     0x81,                       // bEndpointAddress: IN endpoint 1
138     0x03,                       // bmAttributes: Interrupt
139     0x08, 0x00,                 // wMaxPacketSize: 8 bytes
140     0x0A                        // bInterval: 10 ms
141 };
142
143
144
145
146
147 const uint8_t string_descriptor_0[] = {
148     0x04,                   // bLength
149     USB_STRING_DESCRIPTOR_TYPE,
150     0x09, 0x04              // LANGID: English (US)
151 };
152
153 const uint8_t string_descriptor_1[] = {
154     0x14,                   // bLength (20 bytes)
155     USB_STRING_DESCRIPTOR_TYPE,
156     'M', 0x00,
157     'i', 0x00,
158     'n', 0x00,
159     'd', 0x00,
```

(continues on next page)

(continued from previous page)

```
    'g', 0x00,
    'r', 0x00,
    'o', 0x00,
    'v', 0x00,
    'e', 0x00
};

const uint8_t string_descriptor_2[] = {
    0x14,                   // total length (20 bytes)
    USB_STRING_DESCRIPTOR_TYPE,
    'U', 0x00,
    'S', 0x00,
    'B', 0x00,
    ' ', 0x00,
    'M', 0x00,
    'o', 0x00,
    'u', 0x00,
    's', 0x00,
    'e', 0x00
};









const uint8_t hid_report_descriptor[] = {
    0x05, 0x01,       // USAGE_PAGE (Generic Desktop)
    0x09, 0x02,       // USAGE (Mouse)
    0xa1, 0x01,       // COLLECTION (Application)

    0x09, 0x01,       //   USAGE (Pointer)
    0xa1, 0x00,       //   COLLECTION (Physical)

    0x05, 0x09,       //     USAGE_PAGE (Button)
    0x19, 0x01,       //     USAGE_MINIMUM (Button 1)
    0x29, 0x03,       //     USAGE_MAXIMUM (Button 3)
```

(continues on next page)

(continued from previous page)

```
197     0x15, 0x00,         //     LOGICAL_MINIMUM (0)
198     0x25, 0x01,         //     LOGICAL_MAXIMUM (1)
199     0x95, 0x03,         //     REPORT_COUNT (3)
200     0x75, 0x01,         //     REPORT_SIZE (1)
201     0x81, 0x02,         //     INPUT (Data,Var,Abs)
202
203     0x95, 0x01,         //     REPORT_COUNT (1)
204     0x75, 0x05,         //     REPORT_SIZE (5)  // Padding
205     0x81, 0x03,         //     INPUT (Cnst,Var,Abs)
206
207     0x05, 0x01,         //     USAGE_PAGE (Generic Desktop)
208     0x09, 0x30,         //     USAGE (X)
209     0x09, 0x31,         //     USAGE (Y)
210     0x15, 0x81,         //     LOGICAL_MINIMUM (-127)
211     0x25, 0x7f,         //     LOGICAL_MAXIMUM (127)
212     0x75, 0x08,         //     REPORT_SIZE (8)
213     0x95, 0x02,         //     REPORT_COUNT (2)
214     0x81, 0x06,         //     INPUT (Data,Var,Rel)
215
216     0xc0,               //   END_COLLECTION
217     0xc0                // END_COLLECTION
218 };
219
220
221
222 /**
223 * 1.Sets the GPIO interrupt mode for d-, initialises the d+ and d- in␣
    ↪input mode and enables the interrupt.
224 *
225 * 2.Scans for flag == 1,flag will be 1 if a packet has arrived.
226 *
227 * 3.If in arrived packet endpoint is 0 then it will go through state␣
    ↪machines for different stages of control transfer else if packet␣
    ↪recieved is IN token
228 * then boot format mouse packets are sent after reading input from␣
    ↪button.
229 *
```

(continues on next page)

(continued from previous page)

```
230  * 4.State machine for Control transfer:-
231  *
232  * 1. WAIT_FOR_SETUP_TOKEN:
233  *    - Device waits for the SETUP token (PID 0x2D) from the host␣
     ↪indicating start of a control transfer.
234  *    - Once received, transitions to WAIT_FOR_SETUP_DATA.
235  *
236  * 2. WAIT_FOR_SETUP_DATA:
237  *    - Expects DATA0 (PID 0xC3) packet containing 8-byte setup data.
238  *    - Parses bmRequestType, bRequest, wValue, wIndex, and wLength␣
     ↪fields.
239  *    - Sends ACK, prepares descriptor data, and moves to WAIT_FOR_IN_
     ↪TOKEN_FOR_GET_REQUESTS.
240  *
241  * 3. WAIT_FOR_IN_TOKEN_FOR_GET_REQUESTS:
242  *    - On IN token (PID 0x69), sends requested descriptor or␣
     ↪acknowledgment.
243  *    - Alternates between DATA0 and DATA1 PIDs for correct USB␣
     ↪synchronization.
244  *    - After sending all data, transitions to WAIT_FOR_OUT_TOKEN_FOR_
     ↪GET_REQUESTS.
245  *
246  * 4. WAIT_FOR_OUT_TOKEN_FOR_GET_REQUESTS:
247  *    - Waits for host ACK (PID 0xE1) confirming successful data␣
     ↪reception.
248  *    - Then prepares for the final OUT zero-length packet stage.
249  *
250  * 5. WAIT_FOR_DATA_ZLP:
251  *    - Waits for OUT zero-length packet (PID 0xC3 or 0x4B) signaling␣
     ↪end of transfer.
252  *    - Sends final ACK, resets to WAIT_FOR_SETUP_TOKEN, ready for the␣
     ↪next transaction.
253  *
254  */
255 void main(void) {
256     asm volatile(
257         "li      t0, 0x800\t\n"         // Bit 11: MEIE
```

(continues on next page)

(continued from previous page)

```
        "csrrc   zero, mie, t0\t\n"     // Clear MEIE bit in mie
    );
    uint8_t next_data_toggle = 0;
    PLIC_Set_Handler(GPIO0_IRQn+DPLUS,usb_rx_irq_handler,NULL);
    GPIO_Config(GPIO_IN, GPIO_PINS(DMINUS)|GPIO_PINS(DPLUS)|GPIO_
↪PINS(BUTTON));
    GPIO_Interrupt_Config(GPIO_PINS(DPLUS),1);
    PLIC_Set_Interrupt_Priority(GPIO0_IRQn+DPLUS,PLIC_PRIORITY_7);/
↪*Setting interrupt priority*/
    PLIC_Interrupt_Enable(GPIO0_IRQn+DPLUS);/*Enabling Interrupt*/
    printf("Init over");
    asm volatile(
        "li      t0, 0x800\t\n"         // Bit 11: MEIE
        "csrrs   zero, mie, t0\t\n"     // Clear MEIE bit in mie
    );
    Packet P;
    volatile uint8_t state = 0;
    while (1) {
        if (flag == 1) {
            uint8_t endpoint = (buffer[3]>>7)|((buffer[4]&0b111)<<1);
            if(endpoint == 1 && buffer[2]==0x69) {
                // Prepare mouse report: buttons=0, x=1, y=0
                uint8_t mouse_report[3] = {0x00, 0x01, 0x00};
                if((*((uint32_t*)0x40208) & GPIO_PINS(BUTTON)) == 0)␣
↪mouse_report[1] = 1;
                else mouse_report[1] = 0;
                // Send DATA1 or DATA0 toggle packet with report data
                P.pid = next_data_toggle ? PID_DATA1 : PID_DATA0;
                P.data = mouse_report;
                P.data_length = sizeof(mouse_report);
                usb_tx_packet_send(&P);
                next_data_toggle = !next_data_toggle;
            }
            uint8_t pid = buffer[2];
            switch (state) {
            case WAIT_FOR_SETUP_TOKEN:
                // Wait for SETUP token packet from host
```

(continues on next page)

(continued from previous page)

```
292                 if (pid == 0x2D) {
293                     state = WAIT_FOR_SETUP_DATA; // Next expect DATA0␣
↪packet with setup data
294                 }
295                 break;
296 
297             case WAIT_FOR_SETUP_DATA:
298                 // Data stage: receive setup data in DATA0 packet
299                 if (pid == 0xC3) {
300                     // Parse the 8-byte setup packet from buffer
301                     setup_packet.bmRequestType = buffer[3];
302                     setup_packet.bRequest = buffer[4];
303                     setup_packet.wValue = ((uint16_t)buffer[6] << 8)␣
↪| buffer[5];
304                     setup_packet.wIndex = ((uint16_t)buffer[8] << 8)␣
↪| buffer[7];
305                     setup_packet.wLength = ((uint16_t)buffer[10] <<␣
↪8) | buffer[9];
306 
307                     // Immediately ACK the received DATA0 packet
308                     P.pid = PID_ACK;
309                     usb_tx_packet_send(&P);
310 
311                     request_type = setup_packet.bRequest;
312 
313                     if (request_type == USB_REQ_GET_DESCRIPTOR) {
314                         uint8_t desc_type = (setup_packet.wValue >>␣
↪8) & 0xFF;
315                         uint8_t desc_index = setup_packet.wValue &␣
↪0xFF;
316                         if (desc_type == USB_DEVICE_DESCRIPTOR_TYPE) {
317                             descriptor_data = device_descriptor;
318                             descriptor_length = (setup_packet.wLength
↪< DEVICE_DESC_LEN) ? setup_packet.wLength : DEVICE_DESC_LEN;
319                         } else if (desc_type == USB_CONFIGURATION_
↪DESCRIPTOR_TYPE) {
320                             descriptor_data = configuration_
```

(continues on next page)

(continued from previous page)

```
↪descriptor;
321                             descriptor_length = (setup_packet.wLength
↪< CONFIG_DESC_LEN) ? setup_packet.wLength : CONFIG_DESC_LEN;
322                         } else if (desc_type == USB_STRING_DESCRIPTOR_
↪TYPE) {
323                             switch(desc_index) {
324                             case 0:
325                                 descriptor_data = (uint8_t*)string_
↪descriptor_0;
326                                 descriptor_length = (setup_packet.
↪wLength < sizeof(string_descriptor_0)) ? setup_packet.wLength :␣
↪sizeof(string_descriptor_0);
327                                 break;
328                             case 1:
329                                 descriptor_data = (uint8_t*)string_
↪descriptor_1;
330                                 descriptor_length = (setup_packet.
↪wLength < sizeof(string_descriptor_1)) ? setup_packet.wLength :␣
↪sizeof(string_descriptor_1);
331                                 break;
332                             case 2:
333                                 descriptor_data = (uint8_t*)string_
↪descriptor_2;
334                                 descriptor_length = (setup_packet.
↪wLength < sizeof(string_descriptor_2)) ? setup_packet.wLength :␣
↪sizeof(string_descriptor_2);
335                                 break;
336                             default:
337                                 // Invalid string index
338                                 descriptor_data = NULL;
339                                 descriptor_length = 0;
340                                 break;
341                             }
342                         }
343                         // New addition for HID report descriptor
344                         else if (desc_type == HID_REPORT_DESCRIPTOR_
↪TYPE) {
```

(continues on next page)

(continued from previous page)

```
                                // Provide the HID report descriptor pointer and length
                                descriptor_data = (uint8_t*)hid_report_descriptor;
                                descriptor_length = (setup_packet.wLength < HID_REPORT_DESC_LEN) ? setup_packet.wLength : HID_REPORT_DESC_LEN;
                            }
                            bytes_sent = 0;
                        }
                        // For requests without data stage (SET_ADDRESS, SET_CONFIGURATION, SET_IDLE), no descriptor to send

                        state = WAIT_FOR_IN_TOKEN_FOR_GET_REQUESTS; // Move to respond to host IN tokens
                    }
                    break;

                case WAIT_FOR_IN_TOKEN_FOR_GET_REQUESTS:
                    // Respond to host IN tokens with data chunks or ACK for no data requests
                    if (pid == 0x69) {
                        switch (request_type) {
                        case USB_REQ_GET_DESCRIPTOR:
                            int bytes_remaining = descriptor_length - bytes_sent;
                            int chunk_size = (bytes_remaining > MAX_PACKET_SIZE) ? MAX_PACKET_SIZE : bytes_remaining;
                            // Alternate DATA1/DATA0 packet PID as per USB data toggle
                            P.pid = ((bytes_sent / MAX_PACKET_SIZE) % 2 == 0) ? PID_DATA1 : PID_DATA0;
                            P.data = descriptor_data + bytes_sent;
                            P.data_length = chunk_size;
                            usb_tx_packet_send(&P);
                            P.data_length = 0;
                            bytes_sent += chunk_size;

```

(continues on next page)

(continued from previous page)

```
372                         // If all bytes sent, move to wait for host␣
↪ACK (status stage)
373                         if (bytes_sent == descriptor_length) {
374                             state=3;
375                         }
376                         break;
377
378                     case USB_REQ_SET_CONFIGURATION:
379                     case USB_REQ_SET_ADDRESS:
380                     case USB_REQ_SET_IDLE:
381                         P.pid = PID_DATA1;  // ACK status stage␣
↪immediately
382                         usb_tx_packet_send(&P);
383                         state = WAIT_FOR_SETUP_TOKEN;        // Ready␣
↪for next transfer
384                         break;
385                     }
386                 }
387                 break;
388
389             case WAIT_FOR_OUT_TOKEN_FOR_GET_REQUESTS:
390                 // Wait for host ACK after last data packet sent
391                 if (pid == 0xE1) {
392                     // Now wait for host OUT zero-length packet (ZLP)␣
↪for status stage
393                     state = WAIT_FOR_DATA_ZLP;
394                 }
395                 break;
396
397             case WAIT_FOR_DATA_ZLP:
398                 // Wait for final OUT zero-length packet from host␣
↪signaling status stage completion
399                 if ((pid == 0xC3)||(pid == 0x4B)) {
400                     // ACK the OUT zero-length packet (ZLP)
401                     P.pid = PID_ACK;
402                     usb_tx_packet_send(&P);
403                     state = WAIT_FOR_SETUP_TOKEN; // Reset to wait␣
```

(continues on next page)

(continued from previous page)

```
↪for next SETUP
                    }
                    break;
                }

                flag = 0; // Clear flag after handling packet

            }
        }
    }

    #pragma GCC pop_options
```

7.2 Zephyr RTOS Port for Secure-IoT

This document describes the process of setting up the Zephyr RTOS environment for the Secure-IoT platform, including cloning the repository, building applications, and running them on the target environment.

7.2.1 Environment Setup

Required Packages

Install the following dependencies on a Debian/Ubuntu-based system:

```
sudo apt-get install --no-install-recommends git cmake ninja-build␣
↪gperf ccache doxygen dfu-util device-tree-compiler python3-ply␣
↪python3-pip python3-setuptools python3-wheel xz-utils file make gcc-
↪multilib autoconf automake libtool librsvg2-bin texlive-latex-base␣
↪texlive-latex-extra latexmk texlive-fonts-recommended
```

Cloning Required Repositories

Clone the required tools and Zephyr repository:

Install West :

```
pip3 install --user -U west
```

Initialize the workspace with the Zephyr repository and specified tag:

```
west init -m https://github.com/Mindgrove-Technologies/zephyr.git --mr
 ↪<branch-name> zephyrproject
cd zephyrproject/zephyr/
west update
```

Install Python Dependencies:

```
pip3 install --user -r scripts/requirements.txt
```

Environment Variables

Set the following environment variables:

```
unset GNUARMEMB_TOOLCHAIN_PATH
export ZEPHYR_BASE=$PWD
export ZEPHYR_TOOLCHAIN_VARIANT=cross-compile
export CROSS_COMPILE=</path/to>/opt/riscv/bin/riscv64-unknown-elf-
source zephyr-env.sh
```

Or Run the *Zephyr_run.sh* script

```
source zephyr_run.sh
```

Note

Make sure to replace `/path/to/opt/riscv/bin/riscv64-unknown-elf-` with the actual path of your RISC-V cross compiler toolchain.

7.2.2 Device Tree Configuration

This document explains how peripherals are described and configured using Device-tree (*.dts*, *.dtsi*) in a Zephyr RTOS environment.

- *UART Devicetree configuration*
  - *Secure-IoT UART Node (*.dtsi*)*
  - *UART Configuration in Board (*.dts*)*
  - *UART Integration in Zephyr*
  - *Overlay Example*
- *CLINT Device Tree Configuration*
  - *Secure-IoT CLINT Node (*.dtsi*)*
  - *CLINT Integration in Zephyr*
  - *Requirements*
- *SPI Devicetree configuration*
  - *Secure-IoT SPI Node (*.dtsi*)*
  - *SPI Configuration in Board (*.dts*)*
  - *SPI Integration in Zephyr*
- *GPIO Devicetree configuration*
  - *Secure-IoT GPIO Node (*.dtsi*)*
  - *GPIO Configuration in Board (*.dts*)*
  - *GPIO Integration in Zephyr*
  - *Devicetree Syntax for GPIO Specifier*
- *I2C Devicetree configuration*
  - *Secure-IoT I2C Node (*.dtsi*)*
  - *I2C Configuration in Board (*.dts*)*
  - *I2C Integration in Zephyr*
- *Watchdog Timer Devicetree configuration*
  - *Secure-IoT Watchdog Node (*.dtsi*)*
  - *Watchdog Timer Configuration in Board (*.dts*)*
  - *Watchdog Timer Integration in Zephyr*
- *PWM Devicetree configuration*
  - *Secure-IoT PWM Node (*.dtsi*)*
  - *PWM Configuration in Board (*.dts*)*
  - *PWM Integration in Zephyr*

- *PLIC Devicetree configuration*
  - *Secure-IoT PLIC Node (*.dtsi*)*
  - *PLIC Configuration in Board (*.dts*)*
  - *PLIC Integration in Zephyr*

UART Devicetree configuration

Zephyr uses Devicetree files to describe hardware topology and configuration. UART peripherals are defined at the Seciot-IoT (*.dtsi*) and customized at the board level (*.dts*) using node references and property overrides.

Secure-IoT UART Node (*.dtsi*)

The base UART hardware block is typically defined in the Seciot-IoT's *.dtsi* file.

```
uart0: serial@11300 {
    compatible = "mindgrove,uart";
    reg = <0x11300 0x100>;
};
```

Explanation:

- `uart0`: Node label used for referencing this UART elsewhere (e.g., overlays).
- `serial@11300`: Node name and base address in memory (0x11300).
- `compatible`: A string used by drivers to match and bind to the hardware. *"mindgrove,uart"* must match a driver in the Zephyr tree.
- `reg`: Specifies the address and size of the UART registers. *<0x11300 0x100>* means: - Base address: *0x11300* - Size: *0x100* bytes

This describes the hardware resource, but does not enable or configure it for use.

UART Configuration in Board (*.dts*)

The board-level device tree (*.dts*) is used to enable and configure this peripheral:

Example: Enabling uart0:

```
&uart0 {
    status = "okay";
    current-speed = <115200>;
    clock-frequency = <100000000>;
};
```

Explanation:

- `&uart0`: References the label defined in the *.dtsi* file.
- `status = "okay"`: Marks the peripheral as enabled and available to Zephyr.
- `current-speed`: Sets the baud rate, in this case 115200.
- `clock-frequency`: Input clock frequency to the UART peripheral, Input clock frequency in Hz.

Note

UART1 and UART2 can be configured in a similar way by referencing `&uart1` and `&uart2` respectively in your board *.dts* or overlay file. Make sure to provide appropriate `current-speed` and `clock-frequency` values.

UART Integration in Zephyr

To use *uart0* in a Zephyr application:

1. Ensure your board *.dts* or overlay enables the UART using *status = "okay"*.
2. Enable UART driver in *prj.conf*:

```
CONFIG_SERIAL=y  # Enables the UART subsystem in Zephyr.
CONFIG_UART_MINDGROVE=y  # Enables the SoC-specific UART hardware␣
↪driver.
CONFIG_UART_MINDGROVE_PORT=y  # Selects a specific UART port␣
↪implementation when the SoC has multiple UARTs.
CONFIG_UART_CONSOLE=y  # Routes the console output (e.g., `printk`,
↪ log) to a UART device.
```

3. Use standard Zephyr UART APIs (e.g., *uart_poll_in()*, *uart_poll_out()*) or enable

console/logging support.

```
const struct device *dev = DEVICE_DT_GET(DT_NODELABEL(uart2));
uart_poll_out(dev, msg);
```

Overlay Example

To override or modify UART configuration without editing the base *.dts*, create a *board.overlay* file in your application:

```
&uart0 {
    status = "okay";
    current-speed = <9600>;
};
```

This sets the UART speed to 9600 baud during build time.

Note

This overlay specification is same for all the other peripherals

CLINT Device Tree Configuration

The Core-Local Interruptor (CLINT) provides timer and software interrupt functionality to RISC-V cores. Each core typically has a memory-mapped interface to the CLINT, which supports:

- MSIP (Machine Software Interrupts)
- MTIMECMP / MTIME (Machine Timer Interrupts)

Secure-IoT CLINT Node (*.dtsi*)

The CLINT node in the Devicetree source (*.dtsi*) looks like this:

```
soc {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "mindgrove,spike-bare-soc", "simple-bus";
    ranges;
```

(continues on next page)

(continued from previous page)

```
    clint@2000000 {
        #address-cells = <0>;
        #size-cells = <2>;
        compatible = "sifive,clint0";
        interrupts-extended = <&CPU0_intc 3 &CPU0_intc 7>;
        reg = <0x02000000 0x00000008     // msip
               0x02004000 0x00000008     // mtimecmp
               0x0200bff8 0x00000008>;   // mtime
    };
};
```

Explanation:

- `soc`: The top-level bus in which peripherals are instantiated.
  - *#address-cells* and *#size-cells* specify address formatting.
  - *ranges*: Declares address translations, inherited from the parent bus.
- `clint@2000000`:
  - Base address: *0x02000000*
  - Represents the CLINT peripheral.
- `#address-cells = <0>`: CLINT does not have child nodes that need addressing.
- `#size-cells = <2>`: Address regions in *reg* are expressed using two 32-bit values (64-bit total).
- `compatible = "sifive,clint0"`: Identifies this CLINT device as compatible with the SiFive CLINT specification. Required for binding to the correct driver in Zephyr.
- `interrupts-extended`: Specifies interrupt lines this CLINT will signal.
  - *<&CPU0_intc 3>*: Machine software interrupt (MSIP) for CPU0.
  - *<&CPU0_intc 7>*: Machine timer interrupt (MTIP) for CPU0.
- `reg`: Specifies memory-mapped register regions:
  - *0x02000000* (msip): software interrupt control
  - *0x02004000* (mtimecmp): timer compare value
  - *0x0200bff8* (mtime): timer counter value

CLINT Integration in Zephyr

Zephyr does not require explicit user configuration of the CLINT node if:

- The node is defined in the *.dtsi*.
- It is correctly associated with the CPU interrupt controller.
- The *compatible* string matches the expected value.

The kernel will automatically configure the CLINT to provide:

- Periodic system tick using *mtime* and *mtimecmp*.
- Optional inter-CPU software interrupts (in SMP systems).

Requirements

Ensure the following Kconfig options are set:

```
CONFIG_RISCV_MACHINE_TIMER=y  # Enables support for the machine-level␣
↪timer (`mtime`/`mtimecmp`).
CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC=156250  # Sets the hardware clock␣
↪frequency in cycles per second. Value is platform-specific (e.g.,␣
↪156250 for 700 MHz system).
```

These allow the kernel to use *mtime* as the system timer source.

SPI Devicetree configuration

This document explains how SPI peripherals are described and configured using Devicetree (*.dts*, *.dtsi*) in a Zephyr RTOS project. It covers definitions at both SoC (*.dtsi*) and board (*.dts* or overlay) levels, including how to configure multiple instances (*spi0*, *spi1*, *spi2*).

Secure-IoT SPI Node (*.dtsi*)

The base definition of an SPI controller may look like the following in the *.dtsi*:

```
spi0: spi@20000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "mindgrove,spi";
```

(continues on next page)

(continued from previous page)

```
    reg = <0x0 0x20000 0x0 0x100>;
};
```

Explanation:

- `spi0`: Label to reference this SPI controller in other Devicetree files.
- `spi@20000`: Node name with memory-mapped base address *0x20000*.
- `#address-cells` / `#size-cells`: Define the format of child nodes (e.g., SPI devices).
- `compatible`: Must match a driver in Zephyr for the SPI controller.
- `reg`: Specifies the base address and size:
  - Base address: *0x20000*
  - Size: *0x100* bytes
  - Prefixed with 64-bit address support (*<0x0 0x20000>*)

SPI Configuration in Board (*.dts*)

To enable and configure an SPI controller, reference it in the board *.dts* or an overlay file.

Example: Enabling and Configuring *spi0*

```
&spi0 {
    status = "okay";
    sclk_configure = <0x0 0x0 0x10 0x0 0x0>;
    comm_configure = <0x01 0x00 0x03 0x08>;
    clock-frequency = <10000000>;
    cs-gpios = <&gpio0 10 0>;
};
```

Explanation:

- `status = "okay"`: Enables the SPI controller for use.
- `sclk_configure`: Custom property specific to the hardware implementation (may define polarity, phase, delay, etc.).
- `comm_configure`: Another platform-specific property defining communication parameters (e.g., master/slave mode, data width).
- `clock-frequency`: Sets the SPI bus speed to 10 MHz.

- `cs-gpios`: Defines the chip select line as GPIO10 from *gpio0*.

Note

SPI1 and SPI2 can be configured in the same manner by using `&spi1` and `&spi2` in place of `&spi0`.

SPI Integration in Zephyr

To use SPI in your application:

1. Ensure the desired SPI controller is enabled in the Devicetree (*status = "okay"*).
2. Define SPI slave devices as child nodes if needed.
3. Enable SPI driver in *prj.conf*:

```
CONFIG_SPI=y  # Enables the SPI subsystem to allow communication␣
↪with SPI peripherals.
CONFIG_SPI_MINDGROVE=y  # Enables the SoC-specific SPI controller␣
↪driver.
```

4. Use the Zephyr SPI API (e.g., *spi_transceive()*, *spi_write()*).

```
const struct device *const dev = DEVICE_DT_GET(DT_NODELABEL(spi0));
spi_transceive(dev, &config, &tx_bufs, &rx_bufs);
```

GPIO Devicetree configuration

In Zephyr, GPIO controllers are described in the SoC's *.dtsi* file, while their usage (enabled status and pin configuration) is typically handled at the board level (*.dts* or overlay).

Secure-IoT GPIO Node (*.dtsi*)

An example GPIO controller definition in the *.dtsi* file:

```
gpio0: gpio@40200 {
    #gpio-cells = <2>;
    compatible = "mindgrove,gpio";
    reg = <0x0 0x40200 0x0 0x08>;
```

(continues on next page)

(continued from previous page)

```
    config_gpio = <GPIO_OUTPUT>;
    interrupt-parent = <&plic>;
    interrupts = <1 1>, <2 1>, <3 1>, <4 1>,
                 <5 1>, <6 1>, <7 1>, <8 1>,
                 <9 1>, <10 1>, <11 1>, <12 1>,
                 <13 1>, <14 1>, <15 1>, <16 1>,
                 <17 1>, <18 1>, <19 1>, <20 1>,
                 <21 1>, <22 1>, <23 1>, <24 1>,
                 <25 1>, <26 1>, <27 1>, <28 1>,
                 <29 1>, <30 1>, <31 1>, <32 1>;
};
```

Explanation:

- `gpio0`: Node label for referencing the GPIO controller elsewhere.
- `gpio@40200`: Node name and base address (*0x40200*) of the GPIO controller.
- `#gpio-cells = <2>`: Each GPIO specifier must contain two cells—typically the pin number and flags.
- `compatible = "mindgrove,gpio"`: String used to match a compatible driver for this hardware.
- `reg = <0x0 0x40200 0x0 0x08>`: Specifies the base address and size of the GPIO controller's register space (64-bit capable addressing).
- `config_gpio = <GPIO_OUTPUT>`: Platform-specific property to indicate default pin direction (optional or custom binding).
- `interrupt-parent = <&plic>`: Links interrupt lines to the PLIC (Platform-Level Interrupt Controller).
- `interrupts`: Lists all the interrupt lines this controller supports — one for each GPIO pin.

GPIO Configuration in Board (*.dts*)

To enable the GPIO controller at the board level, use the node label reference:

```
&gpio0 {
    status = "okay";
};
```

Explanation:

- `&gpio0`: References the GPIO node label defined in the *.dtsi*.
- `status = "okay"`: Enables the GPIO controller for use.

Note

Can use multiple GPIO instances with the gpio0, using their respective gpio pins.

GPIO Integration in Zephyr

To use GPIOs in your Zephyr application:

1. Enable the GPIO controller in the board *.dts* or overlay (*status = "okay"*).
2. In your application's *prj.conf*, enable GPIO support:

```
CONFIG_GPIO=y  # Enables the generic GPIO API in Zephyr.
CONFIG_GPIO_MINDGROVE=y  # Enables the SoC-specific GPIO␣
↪controller driver.
```

3. Use Zephyr's GPIO API to configure and control GPIO pins:

```
const struct device *gpio_dev = DEVICE_DT_GET(DT_NODELABEL(gpio0));
gpio_pin_configure(gpio_dev, 10, GPIO_OUTPUT_ACTIVE);
gpio_pin_set(gpio_dev, 10, 1);
```

4. Alternatively, use Devicetree aliases and *DT_ALIAS()*/*DT_NODELABEL()* macros to access pins.

Devicetree Syntax for GPIO Specifier

```
<&gpioN pin flags>
```

Where: - *gpioN* is the GPIO controller (e.g., *&gpio0*) - *pin* is the pin number (e.g., *10*) - *flags* specify configuration like active high/low, input/output

I2C Devicetree configuration

Zephyr uses Devicetree to describe I2C hardware. The controller is typically declared in the SoC's *.dtsi* file, and enabled or configured in the board's *.dts* or an overlay.

Secure-IoT I2C Node (*.dtsi*)

An I2C controller node is defined in the *.dtsi* as follows:

```
i2c0: i2c@44000 {
    compatible = "mindgrove,i2c";
    reg = <0x0 0x44000 0x0 0x100>;
};
```

Explanation:

- `i2c0`: Node label, used to reference this controller elsewhere.
- `i2c@44000`: Node name with base address *0x44000*.
- `compatible = "mindgrove,i2c"`: Indicates the driver that should bind to this device.
- `reg`: 64-bit base address and size of the I2C controller's register space:
  - Base address: *0x44000*
  - Size: *0x100* bytes

I2C Configuration in Board (*.dts*)

To enable and configure an I2C controller, reference it in the board *.dts* or an overlay file.

```
&i2c0 {
    status = "okay";
    base = <0x44000>;
    clock_frequency = <700000000>;
    scl_frequency = <100000>;
};
```

Explanation:

- `status = "okay"`: Enables the I2C controller.
- `base`: base address in memory.
- `clock_frequency`: Input clock to the I2C peripheral (e.g., 700 MHz).
- `scl_frequency`: Desired SCL line frequency, e.g., 100 kHz for standard-mode I2C.

Note

I2C1 can be configured in a similar way by referencing `&i2c1` respectively in your board *.dts* or overlay file. Make sure to provide appropriate `scl_frequency` and `clock-frequency` values.

I2C Integration in Zephyr

To use I2C in your Zephyr application:

1. Enable the controller in your board *.dts* or overlay (*status = "okay"*).
2. In *prj.conf*, enable I2C subsystem:

```
CONFIG_I2C=y  # Enables the I2C subsystem for Zephyr applications.
CONFIG_I2C_MINDGROVE=y  # Enables the SoC-specific I2C driver to␣
↪interact with hardware controllers.
```

3. Use the Zephyr I2C API:

```
const struct device *i2c_dev = DEVICE_DT_GET(DT_NODELABEL(i2c0));
i2c_transfer(i2c_dev, data, len, address);
```

Watchdog Timer Devicetree configuration

The watchdog timer is a safety mechanism to automatically reset the system in case of software or hardware failure. It is defined in the SoC's *.dtsi* and enabled/configured at the board level using *.dts* or an overlay.

Secure-IoT Watchdog Node (*.dtsi*)

A typical watchdog node definition in the SoC-level *.dtsi* file:

```
watchdog: watchdog@40400 {
    compatible = "mindgrove,wdt";
    reg = <0x0 0x40400 0x0 0x1>;
    clock-frequency = <700000000>;
    status = "okay";
};
```

Explanation:

- `watchdog`: Label used to reference this watchdog elsewhere.
- `watchdog@40400`: Node name and base address (*0x40400*) of the watchdog controller.
- `compatible`: Matches the hardware with a suitable driver (e.g., *"mindgrove,wdt"*).
- `reg`: Defines the memory-mapped region:
  - Base address: *0x40400*
  - Size: *0x1* byte
  - 64-bit address format: *<0x0 0x40400 0x0 0x1>*
- `clock_frequency`: Input clock frequency (e.g., 700 MHz).
- `status = "okay"`: Enables the Watchdog timer.

Watchdog Timer Configuration in Board (*.dts*)

Watchdog timer doesn't need any enabling in dts. It Enables automatically.

Watchdog Timer Integration in Zephyr

1. Enable watchdog in your Devicetree (*status = "okay"*).
2. Enable WDT subsystem in your *prj.conf*:

```
CONFIG_WATCHDOG=y  # Enables the watchdog framework in Zephyr.
CONFIG_WDT_DISABLE_AT_BOOT=n  # Keeps the watchdog enabled on␣
↪system startup (`n` = disabled).
CONFIG_WDT_MINDGROVE=y  # Enables the SoC-specific watchdog driver.
CONFIG_WDT_LOG_LEVEL_DBG=y  # Sets the watchdog driver's log level␣
↪to debug (for verbose output).
```

3. Use the Zephyr Watchdog API in your application:

```
const struct device *wdt = DEVICE_DT_GET(DT_NODELABEL(watchdog));
```

4. Make your application matches the timing implied by *rcycles*.

PWM Devicetree configuration

In Zephyr, PWM controllers are declared in the SoC *.dtsi* file and enabled/configured per board using *.dts* or overlay files.

Secure-IoT PWM Node (*.dtsi*)

A sample PWM controller node in the *.dtsi*:

```
pwm0: pwm@30000 {
    compatible = "mindgrove,pwm";
    reg = <0x0 0x30000 0x0 0x100>;
};
```

Explanation:

- `pwm0`: Label used to reference the PWM node elsewhere.
- `pwm@30000`: Node name and base address of the PWM controller.
- `compatible = "mindgrove,pwm"`: String matched to the driver supporting this hardware.
- `reg`: Specifies the base address (*0x30000*) and register space size (*0x100* bytes). The 64-bit addressing format is used here.

PWM Configuration in Board (*.dts*)

To enable and configure a PWM instance at the board level:

```
&pwm0 {
    status = "okay";
    db_configure = <0 5000>;
};
```

Explanation:

- `status = "okay"`: Enables the PWM controller.
- `db_configure`: Platform-specific property for deadband or prescaler configuration. The meaning of these values is hardware-dependent; for example:
  - *0*: PWM mode or channel number
  - *5000*: prescaler or timing configuration

Note

pwm1 through *pwm7* can be enabled similarly by replacing *&pwm0* with *&pwm1*, *&pwm2*, etc., and setting appropriate *db_configure* values.

PWM Integration in Zephyr

To use a PWM controller in your Zephyr application:

1. Ensure it is enabled in the Devicetree with *status = "okay"*.
2. Enable PWM support in your *prj.conf*:

```
CONFIG_PWM=y   # Enables the Pulse-Width Modulation subsystem.
CONFIG_PWM_MINDGROVE=y  # Enables the SoC-specific PWM hardware␣
↪implementation.
```

3. In your application code:

```
#include <zephyr/drivers/pwm.h>

const struct device *pwm_dev = DEVICE_DT_GET(DT_NODELABEL(pwm0));
pwm_set(pwm_dev, 0, PWM_USEC(1000), PWM_USEC(500), 0); // 50% duty␣
↪at 1 kHz
```

4. The parameters to *pwm_set()* include:
   - PWM channel index
   - Period
   - Pulse width (high time)
   - Flags (optional, usually 0)

PLIC Devicetree configuration

In Zephyr, PLIC controllers are declared in the SoC *.dtsi* file and enabled/configured per board using *.dts* or overlay files.

Secure-IoT PLIC Node (*.dtsi*)

A typical PLIC definition at the SoC level looks like this:

```
plic: interrupt-controller@c000000 {
    #address-cells = <0>;
    #interrupt-cells = <2>;
    compatible = "mindgrove,plic";
    interrupt-controller;
```

(continues on next page)

(continued from previous page)

```
    reg = <0x0c000000 0x00000080   // priority registers
           0x0c002000 0x00000008   // enable registers
           0x0c200000 0x00001000>; // context-specific registers
    reg-names = "prio", "irq_en", "reg";
    interrupts-extended = <&CPU0_intc 11 &CPU0_intc 9>;
    riscv,max-priority = <7>;
    riscv,ndev = <8>;
};
```

Explanation:

- `plic`: Node label to reference this interrupt controller.
- `interrupt-controller@c000000`: Node name and base address.
- `#address-cells = <0>`: No address cells are used in children.
- `#interrupt-cells = <2>`: Interrupt specifiers are 2 cells wide (interrupt number + mode).
- `compatible = "mindgrove,plic"`: Matches this controller with a PLIC driver.
- `interrupt-controller`: Declares this node as an interrupt controller.
- `reg`: A list of physical address ranges and sizes for:
  - Priority register base
  - Interrupt enable register base
  - Context-specific control registers
- `reg-names`: Maps each region to a logical name.
- `interrupts-extended`: Maps the PLIC to interrupt lines on a CPU:
  - e.g., *<&CPU0_intc 11>* for machine mode, *<&CPU0_intc 9>* for supervisor mode
- `riscv,max-priority`: Maximum priority level supported (e.g., 7)
- `riscv,ndev`: Number of external interrupt sources (e.g., 8 devices)

PLIC Configuration in Board (*.dts*)

To enable and configure the PLIC at the board level:

```
&plic {
    status = "okay";
    base = <0xc000000>;
};
```

Explanation:

- `status = "okay"`: Enables the interrupt controller.
- `base`: base address in memory.

PLIC Integration in Zephyr

1. Ensure that PLIC is enabled in the board *.dts* file.
2. In *prj.conf*, enable the required PLIC configuration options:

```
CONFIG_PLIC=y  # Enables the PLIC subsystem in Zephyr.
CONFIG_MINDGROVE_PLIC=y  # Enables the SoC-specific PLIC support.
CONFIG_RISCV_HAS_PLIC=y  # Indicates that the RISC-V core uses a␣
↪PLIC.
CONFIG_GEN_ISR_TABLES=y  # Enables generation of interrupt vector␣
↪tables.
CONFIG_GEN_SW_ISR_TABLE=y  # Enables support for shared interrupt␣
↪handlers.
```

3. Write your interrupt handler and configure the peripheral to use interrupts.

```
const struct device *dev = DEVICE_DT_GET(DT_NODELABEL(gpio0));
printf("Entered main.c\n");

gpio_pin_configure(dev, GPIO_PIN, 0);
gpio_pin_interrupt_configure(dev, GPIO_PIN, 1);

plic_irq_enable(INT_ID);

isr_installer();
```

7.2.3 Configuration Guide

This explains the *prj.conf* configuration options used in the Secure IoT Zephyr-based platform. Core kernel and SoC configurations are explained in detail below. Peripheral configurations are listed separately and should be enabled per application as needed.

Core System Configuration

The following configuration options are common and essential for the Secure IoT SoC platform. These are included in the base configuration of all applications unless stated otherwise.

```
CONFIG_SOC_SERIES_RISCV64_SHAKTI=y  # Selects the 64-bit RISC-V SoC␣
↪family support
CONFIG_SOC_RISCV64_SHAKTI=y  # Enables the RISC-V architecture support␣
↪in the kernel.
CONFIG_BOARD_SECURE_IOT=y  # Sets the board definition to the custom
↪"Secure IoT" platform.
CONFIG_XIP=n  # Disables Execute-In-Place (XIP). The code is executed␣
↪from RAM.
CONFIG_SRAM_BASE_ADDRESS=0x80000000  # Defines the base address.
CONFIG_SRAM_SIZE=256000  # Defines the size of internal SRAM.
CONFIG_MAIN_STACK_SIZE=2048  # Stack size (in bytes) for the main␣
↪thread (e.g., `main()`).
CONFIG_CONSOLE=y  # Enables the console system for UART output.
CONFIG_PRINTK=y  # `printk()` and log messages will be printed to␣
↪serial.
CONFIG_STDOUT_CONSOLE=y  # Redirects standard output functions like␣
↪printf() and printk() to the system console (typically UART).
CONFIG_BOOT_BANNER=y  # Prints a welcome banner at startup, useful for␣
↪verification.
CONFIG_LOG=y  # Enables Zephyr's logging subsystem.
CONFIG_LOG_MODE_IMMEDIATE=y  # Forces immediate output (no buffering).
CONFIG_RISCV_MACHINE_TIMER=y  # Enables support for the machine-level␣
↪timer (`mtime`/`mtimecmp`).
CONFIG_TIMING_FUNCTIONS=y  # Provides `k_cycle_get_32()` and other time-
↪measurement APIs.
CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC=156250  # Sets the hardware clock␣
```

(continues on next page)

(continued from previous page)

```
↪frequency in cycles per second. Value is platform-specific (e.g.,␣
↪156250 for 700 MHz system).
CONFIG_SYS_CLOCK_TICKS_PER_SEC=10000  # Configures number of system␣
↪ticks per second (for `k_sleep`, scheduling, etc.).
CONFIG_POSIX_CLOCK=y  # Enables POSIX-compatible time APIs.
CONFIG_FPU=y  # Enables Floating Point Unit.
CONFIG_MULTITHREADING=y  # Enables support for multi-threaded kernel␣
↪applications.
CONFIG_OUTPUT_DISASSEMBLY=y  # Enables generation of disassembly␣
↪output during the build process.
CONFIG_OUTPUT_DISASSEMBLE_ALL=y  # Generates disassembly for all␣
↪object files in the project, not just the main application binary.
```

7.2.4 List of Sample Applications

This section lists the available sample applications that can be built and run on the Secure IoT SoC platform using Zephyr RTOS. Each application demonstrates the usage of a specific peripheral or feature supported by the SoC.

- **Hello World**
  - Verifies toolchain, board configuration, and basic output through UART.
  - path : *samples/hello_world*
- **GPIO Blink**
  - Toggles an LED connected to a GPIO pin.
  - path : *samples/gpio_blink*
- **UART Loopback**
  - Send and receives characters through a UART interface.
  - path : *samples/uart_loopback*
- **I2C RTC**
  - Reads data from a connected RTC sensor.
  - path : *samples/i2c_rtc*
- **SPI Loopback**
  - Send and receives characters through a SPI interface.
  - path : *samples/spi_loopback*

- **Watchdog Reset Demo**
  - Demonstrates watchdog timer by intentionally triggering a system reset.
  - path : *samples/drivers/watchdog_timer*
- **PWM Blink**
  - Controls LED brightness using PWM output.
  - path : *samples/pwm_blink*
- **Plic interrupt**
  - Demonstrates use of the RISC-V Platform-Level Interrupt Controller (PLIC) with a peripheral (e.g., GPIO or timer).
  - path : *samples/plic_interrupt*
- **Multi-tasking**
  - Demonstrates Zephyr's multithreading and task scheduling.
  - path : *samples/multitasking*
- **Matrix Multiplication**
  - Performs matrix multiplication using C code, showcasing computational load and benchmarking.
  - Path: *samples/matrix_multiply*

7.2.5 Application Build Process

Customize the kernel and board configuration using menuconfig.

Configurations settings

Using Make :

```
make menuconfig
make
```

Using West :

```
west build -b secure-iot -t menuconfig
west build
```

Building the Application

Zephyr applications are built using either the West or Make build systems.

Using Make :

```
cd samples/<application_dir>
mkdir build
cd build
cmake -DBOARD=secure-iot ..
make
```

Using West :

To clean the build:

```
rm -rf build/
```

To build a sample application:

```
cd samples/<application_dir>/
west build -b secure-iot
```

After building, the output ELF binary will be found at:

```
$ZEPHYR_BASE/samples/<application_dir>/build/zephyr/zephyr.elf
```

This ELF file can be run using the Secure-IoT hardware.

- Always use the *zephyr_run.sh* script to source environment variables in a new terminal.
- Clean the build directory (*rm -rf build*) if you switch boards or major configuration settings.

INDEX

Symbols

A

B

C

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

Y

Z