TeraFire® Cryptographic Applications Library (CAL) for Microsemi PolarFire® FPGAs

User's Manual

Revision 2.4g
December, 2016

The Athena Group, Inc.
408 W. University Ave.
Suite 306
Gainesville, FL 32601

Phone: (352) 371-2567
FAX: (352) 373-5182

www.athena-group.com

CHAPTER 1 Introduction

The Microsemi PolarFire™ FPGA is offered with an available TeraFire® EXP-F5200B cryptography microprocessor, provided as hard IP on the device. Athena's portfolio of TeraFire® cryptographic cores is also available as soft IP for all Microsemi FPGA and SoC devices, including SmartFusion™2, and IGLOO™2.

The EXP-F5200B provides complete support for the Commercial National Security Algorithm (CNSA) Suite[1] and beyond, and also includes side-channel analysis (SCA) resistant cryptography using proprietary, patent pending leakage reduction countermeasures. These countermeasures provide strong resistance against SCA attacks such as differential power analysis (DPA) and simple power analysis (SPA).

The TeraFire cryptographic applications library (CAL) is a C language library that executes on the user's soft core processor and provides functions that access symmetric key, elliptic curve, public key, hash, random number generation, and message authentication code algorithms. The CAL functions interface with Athena cryptographic IP cores, including the EXP-F5200B, to provide high-performance cryptography that is easily integrated into SoC software designs.

The TeraFire CAL on MIcrosemi PolarFire FPGAs has support for numerous cryptographic algorithms, including the following:

- AES with 128-, 192-, and 256-bit key sizes in ECB, CBC, CFB, OFB, CTR, and GCM modes;
- AES key wrap and unwrap;
- SHA1, SHA2-224, SHA2-256, SHA2-384, and SHA2-512;
- HMAC-SHA;
- true random number generation (non-deterministic random bit generator plus NIST SP800-90A deterministic random bit generator);

1. The CNSA Suite is the successor to NSA Suite B. See www.iad.gov/iad/programs/iad-initiatives/cnsa-suite.cfm for additional information.

- RSA, DSA, and modular exponentiation (Diffie-Hellman) with key sizes to 4096-bits; and
- ECDSA and EC point multiplication (EC Diffie-Hellman).

All supported secret/private key algorithms are protected with strong leakage reduction SCA countermeasures, and built-in support for protocol SCA countermeasures for applicable algorithms is included. For more information on countermeasures, and their strength for a given algorithm, please contact your local Microsemi representative.

Many of the cryptographic algorithms implemented on the EXP-F5200B on the Microsemi PolarFire FPGA have NIST cryptographic algorithm validation program (CAVP) certificates; see *NIST Certifications* on page 3 for a complete listing. TeraFire CAL allows developers to meet the cryptographic requirements for many standards such as, but not limited to, IEEE 802.11-2007, IEEE 802.15, IEEE 802.16 (WiMax), IEEE 802.1AE (MACsec), IPSEC, SSL/TLS, and IKEv2.

For applications requiring higher performance, additional algorithms, or other cryptographic functionality, Athena maintains a broad portfolio of cryptographic IP cores that address a spectrum of applications. Athena is a pioneer in the field of SCA-resistant cryptography using leakage reduction countermeasures, and Athena's entire portfolio of cryptographic IP cores is available either with or without strong SCA countermeasures.

In Microsemi PolarFire FPGAs, higher performance and/or additional functionality may be added using a combination of firmware upgrades and reduced area-cost TeraFire ecosystem IP cores that operate in tandem with the built-in EXP-F5200B. For more information, please contact Athena or your local Microsemi representative.

This document is specific to the Athena drivers and cryptographic core configured as a hard macro on the Microsemi PolarFire FPGAs. Other Athena product configurations may vary from this document.

1.1 NIST Certifications

Many of the algoirthms implemented on the EXP-F5200B have NIST CAVP certificates. The specific certificates for the EXP-F5200B core on PolarFire FPGAs are the following:

- AES, including GCM, certificate #3950;
- DSA, certificate #1077;
- RSA, certificate #2018;
- ECDSA, certificate #867;
- SHA, certificate #3258;
- DRBG, certificate #1153; and
- HMAC, certificate #2573.

Not all key sizes, modes, and other options are covered by these certificates. Refer to the specific NIST validation lists at www.csrc.nist.gov for detailed information on key sizes, modes, and other options covered by these certificates.

1.2 TeraFire EXP-F5200B Cryptography Microprocessor

Athena's TeraFire EXP-F5200B cryptography microprocessor is available as hard IP on Microsemi PolarFire FPGAs and as soft IP for any Microsemi FPGA. An interface block diagram for the EXP-F5200B hard IP block on the PolarFire FPGAs is shown in Figure 1-1. Refer to Microsemi documentation for instructions on how to instantiate the interface to the EXP-F5200B hard IP block.

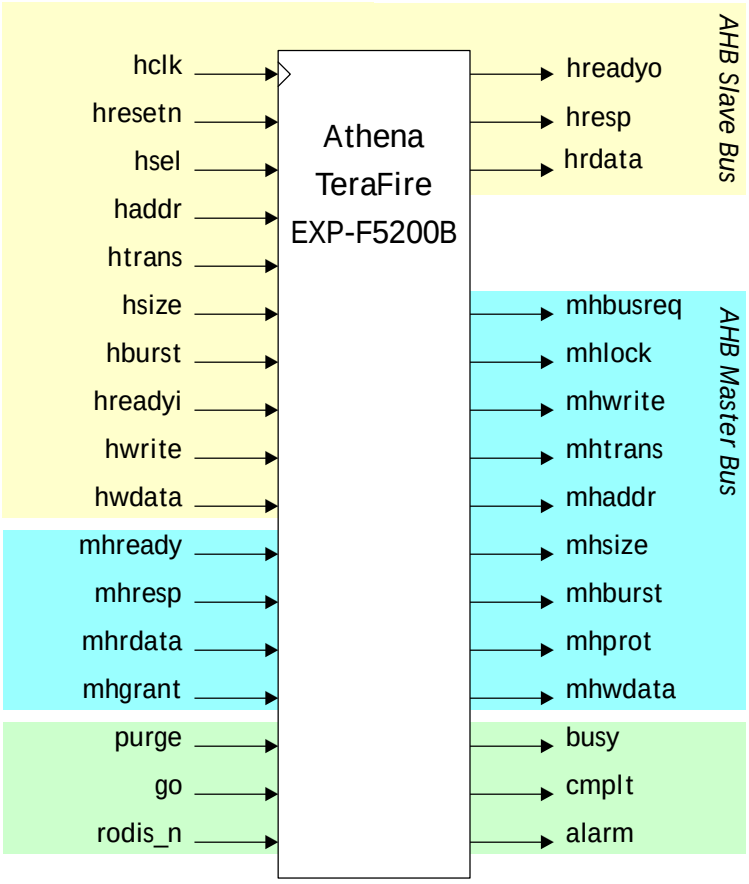

Figure 1-1: TeraFire EXP-F5200B Interface Block Diagram

The EXP-F5200B interface comprises the following major elements:

- core control and status signals (see *Core Control and Status Interface* on page 5);
- a 32-bit AMBA AHB bus slave interface (see *AHB Slave Interface* on page 7), which is used for control, programming, and primary data input and output; and
- a 32-bit AMBA AHB bus master direct memory access (DMA) interface (see *AHB Master Direct Memory Access Interface* on page 7), which may optionally be used for data input and output.

These elements are described in the following sections.

Note that all interface pins are active high, except where otherwise noted.

1.2.1 Core Control and Status Interface

The core control and status interface signals are listed in Table 1-1.

Table 1-1: Core Control and Status Interface Signal Definitions

Pin Name	*Size*	*Direction*	*Description*
go	1	I	External execution initiation input.
purge	1	I	On-demand purge input pin.
rodis_n	1	I	Ring oscillator disable input. Active low.
cmplt	1	O	Computation complete flag.
busy	1	O	Execution status.
alarm	1	O	Alarm output.

The core control and status interface signals are used to initiate action and obtain status, and equivalent signals are accessible via the AHB slave interface and embedded within the CAL.

Note that reset is provided by the AHB **hresetn** pin, see *AHB Slave Interface* on page 7.

1.2.1.1 Control Inputs

For normal operation, Athena recommends tying the input pins as listed in Table 1-2.

Table 1-2: Core Control and Status Interface Recommended Inputs

Pin Name	*Value*
go	low
purge	low
rodis_n	high

Additional information regarding the control and status input pins is provided as follows.

- The **rodis_n** pin is described in *Ring Oscillators* on page 8.
- The **purge** pin is described in *Purge* on page 9.

The **go** pin should always be tied low. Contact support for additional information.

1.2.1.2 Status Outputs

The status outputs listed in Table 1-1 on page 5 are described below.

- The **busy** pin is asserted to indicate that the EXP-F5200B is busy performing an operation.
- The **cmplt** pin is asserted to indicate that the EXP-F5200B has completed an operation. When the EXP-F5200B is connected to a host microprocessor, this pin will usually be connected to the microprocessor as an interrupt request signal, enabling the EXP-F5200B to interrupt the processor when it completes an operation.
- The **alarm** pin is asserted to indicate an uncorrectable memory error condition. An uncorrectable memory error will cause the core to perform a reset and purge. Any in-progress operation will be terminated by this reset. For most CAL operations, the CALPKTrfRes function (see *CALPKTrfRes* on page 27) is used to complete the operation, and will generate a hardware fault code in the event of an alarm.

1.2.2 AHB Slave Interface

The EXP-F5200B implements a subset of the AHB interface. The AHB interface signals are listed in Table 1-3.

Table 1-3: Interface Signal Definitions

Pin Name	*Size*	*Direction*	*Description*
hclk	1	I	Clock pin.
hresetn	1	I	Reset pin. Active low.
hsel	1	I	Select pin.
hwrite	1	I	Read/write select pin.
haddr	17	I	Address input.
hsize	3	I	Transfer size.
hburst	3	I	Burst type.
htrans	2	I	AHB transfer type.
hreadyi	1	I	hready input.
hwdata	32	I	Data input (write) bus.
hreadyo	1	O	hready output.
hrdata	32	O	Data output (read) bus.
hresp	2	O	Transfer response.

The EXP-F5200B is compatible with AHB-Lite[1] bus masters, supports the hreadyi/hreadyo protocol, and may use wait states, depending upon the implementation technology and frequency.

For more information regarding the AHB bus, refer to the ARM AMBA 2.0 specification, document number IHI0011A.

1.2.3 AHB Master Direct Memory Access Interface

The optional AHB master DMA interface enables the EXP-F5200B cores to directly read and write host memory. The AHB master signals are listed in Table 1-4. Where signals such as **hclk** are common to both the slave and master interfaces, they are not re-listed here. All AHB bus

1. See ARM document *AHB-Lite: Overview*, document DVI0044A.

master signals are prefixed with "**m**" to differentiate them from the slave interface.

Table 1-4: AHB Master Interface Signal Definitions

Pin Name	*Size*	*Direction*	*Description*
mhready	1	I	hready input.
mhresp	2	I	Transfer response.
mhrdata	32	I	Data input (read) bus.
mhgrant	1	I	Bus grant.
mhbusreq	1	O	Bus request.
mhlock	1	O	Bus lock.
mhwrite	1	O	Read/write select.
mhtrans	2	O	AHB transfer type.
mhaddr	32	O	Address output.
mhsize	3	O	Transfer size.
mhburst	3	O	Burst type.
mhprot	4	O	Protection control.
mhwdata	32	O	Data output (write) bus.

The optional DMA interface conforms with the AHB-Lite specification.

1.2.4 Ring Oscillators

The ring oscillators are used as the entropy source for the non-determinstc random bit generator (NRBG) and are implemented with a functional disable control, **rodis_n**, which is provided at the top-level interface to the core. This pin is intended to allow the user to disable the ring oscillators for certain use cases, such as when the core is in a sleep state and no clock is available. If such a state is not contemplated, it may be tied high.

The **rodis_n** pin overrides the internal ring oscillator controls. If this pin is tied low, the oscillators will not oscillate and any attempts to use the NRBG will trigger NRBG health errors, since the oscillators will be in a stuck state. It is recommended that the ring oscillators only be disabled in this way when the core is in reset (**hresetn** low) and/or has no active clock.

1.2.5 Purge

The operational state diagram for the EXP-F5200B in the typical use case is shown in Figure 1-2.

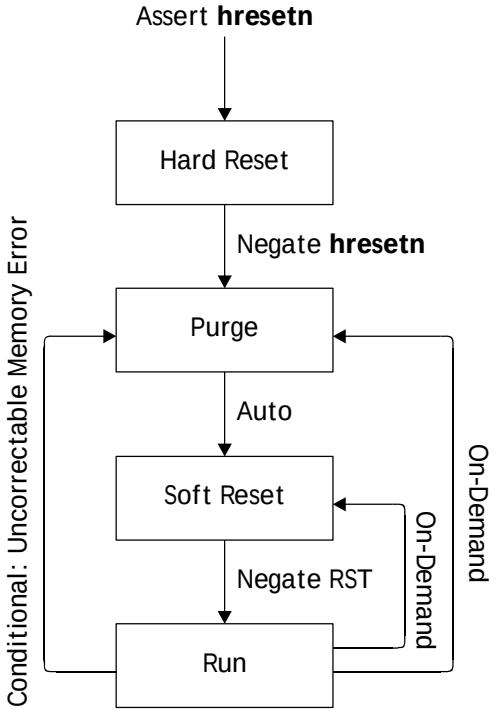

Figure 1-2: Nominal State Diagram

The EXP-F5200B supports on-demand and conditional purge (zeroization) of the core. A purge may be initiated by asserting the **purge** pin or via the AHB bus slave interface.

The purge operation performs the following functions on the EXP-F5200B core:

- **Data memory zeroization**. All internal data memories are zeroized. If verification is required, this must be performed by the user; see *CALPurge52* on page 26.

- **Program memory zeroization**. The internal program memory RAM is zeroized. If verification is required, this must be performed by the user; see *CALPurge52* on page 26.
- **Datapath zeroization**. All EXP-F5200B data registers in the datapath are zeroized.
- **Instruction fetch, decode, and dispatch/execution purge**. Registers associated with instruction fetch, decode, dispatch, and execution are purged. This includes the call stack, instruction registers, length registers (ALEN, BLEN), location registers (MLOC, PLOC), and other internal registers that are used during the execution of instructions. Since some registers are actively used to perform the purge operation, their state after completion of the purge may not be zero; however, their residual value is unrelated to any data stored or computation underway prior to the purge.

The purge operation requires an operating clock (**hclk**) and will require approximately 2K-cycles to complete.

Contact support for additional information regarding the purge operation.

1.3 Support

Support requests for the Microsemi PolarFire FPGA with the Athena EXP-F5200B hard macro, and CAL driver software, should be directed to your Microsemi support representative.

1.4 Intellectual Property Rights

1.4.1 Open Source Software

No open source software is used in Athena's products.

CHAPTER 2 CAL Public Key/Elliptic Curve Cryptography Functions

The TeraFire CAL functions for public key and elliptic curve cryptography are documented in this chapter. These functions work with Athena's line of cryptography microprocessors and dedicated elliptic curve cryptography accelerators to perform these operations. Because these operations are relatively costly to execute, CAL has been designed so that operations can be executed while your application performs other operations by providing support for non-blocking operation initiation and completion (fork and join).

Most of the work associated with any particular CAL function is actually performed on the EXP-F5200B cryptography microprocessor. The CAL functions load data onto the EXP-F5200B microprocessor, initiates execution of the operation, and retrieves results when the operation is complete. Even complex operations such as RSA with CRT and EC-DSA sign and verify execute in their entirety on the EXP-F5200B microprocessor.

2.1 Public Key and Elliptic Curve Cryptography Algorithm Support

2.1.1 Cryptography Algorithm Support

Typical uses of public key and elliptic curve cryptography include key agreement protocols, cryptographic signatures of messages, and cryptographic confidentiality of messages. The CAL includes support for the following public key cryptography operations:

- RSA public key cryptography;
- digital signature standard (DSS) message signature generation and signature verification;
- Diffie-Hellman (DH) key agreement;
- elliptic curve digital signature algorithm (EC-DSA) for message signature generation and signature verification; and
- elliptic curve Diffie-Hellman key agreement.

For key agreement protocols (DH and EC-DH), a network transport or other communications are necessary and are outside of the scope of the CAL. However, the CAL does provide the necessary functions to realize these algorithms, namely modular exponentiation and EC point multiplication.

2.1.2 Elliptic Curve Support

The EXP-F5200B's elliptic curve cryptography support includes NIST P-curves, and other curves such as Brainpool[1]. The future-proof, fully-programmable EXP-F5200B cryptography microprocessors may be programmed to support virtually any curve or algorithm: contact support for additonal information.

2.1.3 Twisted Elliptic Curves

For elliptic curves of the form $y^2 = x^3 + Ax + B$, if there exists a solution to $AZ^4 = -3 \pmod{p}$, then there exists an isomorphism with the curve ${y_0}^2 = {x_0}^3 - 3x_0 + b$, where $(x_0, y_0) = (xZ^2, yZ^3)$, and $b = BZ^6$. This mapping is referred to as a quadratic twist, and may be used to efficiently per-

1. See www.ecc-brainpool.org.

form operations on elliptic curve for curves of the form $y^2 = x^3 + Ax + B$. CAL functions that support such operations require the Z twist factor to perform mapping/unmapping, and the twisted b, which is used in elliptic curve computations for point validation.

2.2 Operation

2.2.1 Blocking and Non-Blocking Operation

Since many public key cryptography operations take microseconds, or sometimes even milliseconds, to execute on the TeraFire public key cryptography microprocessors, the CAL has been designed to support non-blocking execution of operations; however, blocking execution can also be supported. Function calls that initiate operations accept the location(s) for the storage of results and save that location for use when the selected operation is complete. The function for transferring results accepts an argument that controls whether it blocks or not (see *CALPK-TrfRes* on page 27 for more information). An example of blocking execution is shown below.

Examples given below are intended to illustrate concepts and do *not* include robust error checking and handling.

Example 2-1 Blocking CAL-PK Execution

```
/* Initiate modular exponentiation operation */
/* Note: modulus and pre-compute are already loaded. */
rStatus=CALExpo(ipBase, ipExpo, SAT_NULL, SAT_NULL, iExpLen,
                iModLen, ipResult);

/* Wait for completion to recover results */
if (rStatus==SAT_SUCCESS) CALPKTrfRes(SAT_TRUE);
```

In some cases, non-blocking execution simply means that an operation will be initiated, a finite set of other tasks will be performed, and then a blocking result retrieval will be called, effectively a join operation. This is illustrated below.

Example 2-2 CAL-PK Operation with Join

```
/* Initiate modular exponentiation operation */
/* Note: modulus and pre-compute are already loaded. */
rStatus=CALExpo(ipBase, ipExpo, SAT_NULL, SAT_NULL, iExpLen,
                iModLen, ipResult);

if (rStatus==SAT_SUCCESS) {

  /* Perform a useful function here. */
  UserFunction(iArg1,szArg2);

  /* Wait for completion to recover results */
  CALPKTrfRes(SAT_TRUE);
}
```

Finally, in some cases it may be desirable to poll for the result while doing other useful work in the polling loop. This is illustrated below.

Example 2-3 CAL-PK Operation with Polling Loop

```
/* Initiate modular exponentiation operation */
/* Note: modulus and pre-compute are already loaded. */
rStatus=CALExpo(puiBase, puiExpo, SAT_NULL, SAT_NULL,
                uiExpLen, uiModLen, puiResult);

/* Poll for completion of operation. */
if (rStatus==SAT_SUCCESS)
  while (CALPKTrfRes(SAT_FALSE)==SATR_BUSY) {

    /* Perform a useful function here. */
    UserFunction(iParameter1,szParameter2);
  }
```

2.2.2 Pre-Compute Values

In general, high-performance modular arithmetic implementations, such as in the TeraFire public key microprocessors, require the use of a pre-computed value that is derived from the modulus. Calculation of this pre-computed value is a relatively costly operation, and therefore it is desirable that the value is computed once and then reused as much as possible. CAL provides a standard function to compute this value (see *CALPreCompute* on page 28 for details on this function). An example of the use of this function is shown below.

Example 2-4 Pre-Compute Computation and Subsequent Operation

```
/* Initiate generation of pre-compute value. */
rStatus=CALPrecompute(ipMod, ipMu, iModLen);

/* Wait for completion to recover results (if CALPrecompute
   initiated successfully. */
if (rStatus==SAT_SUCCESS) {

  /* Recover result. */
  CALPKTrfRes(SAT_TRUE);

  /* Initiate modular exponentiation operation */
  rStatus=CALExpo(ipBase, ipExpo, ipMod, ipMu,
                  iExpLen, iModLen, ipResult);

  /* Wait for completion to recover results -- if expo
     initiated successfully. */
  if (rStatus==SAT_SUCCESS) CALPKTrfRes(SAT_TRUE);
}
```

2.2.3 NIST P-Curve ROM

The CAL provides specific support for the EXP-F5200B's P-curve ROM. The EXP-F5200B microprocessor is configured to reserve internal ROM that contains the modulus and pre-compute parameters for the following NIST P-curves: P-192, P-224, P-256, P-384, and P-521.

In order to reference the ROM P-curve locations, the CAL provides macro definitions for the ROM'd P-curves. The macros, defined in Table 2-1, should be used to reference the ROM'd P-curves when calling CAL elliptic curve functions. The P-curve macro should be used as the function's EC modulus parameter and the pre-compute parameter should be set to SAT_NULL; the use of the P-curve modulus macro will automatically cause the associated ROM'd pre-compute value to be used. Examples of using the ROM P-256 P-curve with EC Multiply, ECDSA Sign, and ECDSA Verify are shown below.

Example 2-5 EC Point Multiply with ROM Constants

```
/* Initiate EC multiply with P-256 ROM */
CALECMult(puiMul, puiPx, puiPy, puiB, P256_MOD,
SAT_NULL, uiLen, uiPtCompress, puiRx, puiRy)

/* Wait for completion to get result (puiResult) */
CALPKTrfRes(SAT_TRUE);
```

Example 2-6 EC-DSA Sign with ROM Constants

```
/* Initiate ECDSA sign with P-256 ROM */
CALECDSASign(puiHash, puiGx, puiGy, puiK, puiD,
puiB, P256_MOD, SAT_NULL, puiN, puiNMu, uiLen, puiSigR,
puiSigS)

/* Wait for completion to get result (puiResult) */
CALPKTrfRes(SAT_TRUE);
```

Example 2-7 EC-DSA Verify with ROM Constants

```
/* Initiate ECDSA verify with P-256 ROM */
CALECDSAVerify(puiHash, puiGx, puiGy, puiQx,
puiQy, puiSigR, puiSigS, puiB, P256_MOD, SAT_NULL,
puiN, puiNMu, uiLen, uiPtCompress)

/* Wait for completion to get result (puiResult) */
CALPKTrfRes(SAT_TRUE);
```

Table 2-1: CAL P-Curve Modulus Macro References

Macro	*Description*
P192_MOD	P-192 curve modulus reference
P224_MOD	P-224 curve modulus reference
P256_MOD	P-256 curve modulus reference
P384_MOD	P-384 curve modulus reference
P521_MOD	P-521 curve modulus reference

2.3 High-Level Operations

In public key cryptography, some operations will require multiple steps to execute. In some cases, the underlying protocol requires a network transport (*e.g.*, Diffie-Hellman key agreement). In other cases, there are multiple computed values required (*e.g.*, pre-compute and possibly derived values for RSA with CRT). This section provides an outline of some of these common multi-step operations.

Examples given below are intended to illustrate concepts and do *not* include robust error checking and handling.

2.3.1 Diffie-Hellman Key Agreement

The Diffie-Hellman key agreement protocol requires the following parameters that are known to both parties participating in the key agreement process:

- a base value (`puiBase`); and
- a modulus (`puiMod`).

Additionally, each party must generate a random exponent (`puiExp`), typically of the same size as the modulus and base, so `uiModLen` should be used for the operation lengths throughout. A code segment that illustrates the entire Diffie-Hellman key agreement process, including the pre-compute generation, is shown below.

Example 2-8 Diffie-Hellman Key Agreement

```
/* Initiate generation of pre-compute value. */
CALPrecompute(puiMod, puiMu, puiModLen);

/* Wait for completion to recover result (puiMu) */
CALPKTrfRes(SAT_TRUE);

/* Initiate modular exponentiation operation */
CALExpo(puiBase, puiExp, puiMod, puiMu,
        uiModLen, uiModLen, puiResult);

/* Wait for completion to get result (puiResult) */
CALPKTrfRes(SAT_TRUE);

/* Exchange my result with other party's result */
/* This function should send puiResult to the other party
   and store the other party's result at puiBase */
EndUserExchangeDHValues(puiBase,puiResult);
```

```
/* Initiate modular exponentiation operation */
/* Modulus and pre-compute already loaded from previous
   CALExpo above. */
CALExpo(puiBase, puiExp, SAT_NULL, SAT_NULL,
        uiModLen, uiModLen, puiResult);

/* Wait for completion to get result (puiResult) */
CALPKTrfRes(SAT_TRUE);
```

2.3.2 RSA Private Key Operation with CRT

RSA private key operations may be executed more efficiently using the Chinese remainder theorem (CRT). The RSA with CRT algorithm requires two modular exponentiations, as well as other modular arithmetic operations; the CAL performs the entire process in a single function call, with no processing on the host processor. Unlike other operations in CAL, there are two moduli involved, which means that the RSA with CRT function (see *CALRSACRTSign* on page 40) operates a little differently than other CAL functions with respect to moduli and pre-compute values.

An RSA with CRT decryption requires the following input parameters:

- a ciphertext value (`puiCipher`); and
- an RSA private key (see PKCS#1), which includes *p* (`puiP`), *q* (`puiQ`), *d*, and generally also includes the derived values *d* mod *p*-1 (`puiDP`), *d* mod *q*-1 (`puiDQ`), and the CRT coefficient *q*-1 mod *p* (`puiQInv`).

In the event that the derived parameters are not provided, they may be readily computed, although that will not be illustrated here. A code segment that illustrates the RSA private key operation with CRT, including pre-compute generation, is shown below.

Example 2-9 RSA Private Key Operation with CRT

```
/* Initiate generation of pre-compute value for Q. */
CALPrecompute(puiQ, puiQMu, uiModLen);

/* Wait for completion to recover result (puiQMu) */
CALPKTrfRes(SAT_TRUE);

/* Derived puiDQ could be computed here, if needed. */

/* Initiate generation of pre-compute value for P. */
CALPrecompute(puiP, puiPMu, uiModLen);
```

```
/* Wait for completion to recover result (puiPMu) */
CALPKTrfRes(SAT_TRUE);

/* puiDP and puiQInv could be computed here, if needed. */

/* Initiate RSA with CRT. */
CALRSACRT(puiCipher, puiQInv, puiDP, puiDQ, puiP, puiPMu,
          puiQ, puiQMu,  uiLen, puiPlain)

/* Wait for completion to get result (puiPlain) */
CALPKTrfRes(SAT_TRUE);
```

2.4 Public Key and Elliptic Curve Cryptography Library Organization

TeraFire CAL for public key and elliptic curve cryptography functions are described in the following sections.

- *General Functions* on page 24, see Table 2-3.
- *Conventional Public Key Cryptography Functions* on page 29, see Table 2-3.
- *Elliptic Curve Cryptography Functions* on page 60, see Table 2-4.
- *Functions with SCA Countermeasures* on page 96, see Table 2-5.

Function descriptions include syntax, parameters, return values, and a detailed description of the operation of the function.

2.5 General Functions

Each of the functions listed in Table 2-2 is described in this section.

Table 2-2: TeraFire CAL-PK General Functions

Function	*Description*	*Page*
CALIni	CAL initialization	25
CALPurge52	Performs an on-demand purge of a cryptography microprocessor core	26
CALPKTrfRes	Transfers the results once processing is finished	27
CALPreCompute	Calculates the pre-compute value given a modulus	28

CALIni

Syntax `rReturn=CALIni(void)`

Parameters CALIni has no parameters.

Returns SATR rReturn values resulting from this function.

Return Value	*Causes*
SATR_SUCCESS	Successful initialization.
SATR_FAIL	This indicates that the initialization failed.

Description The CALIni function initializes CAL and the EXP-F5200B core driven by CAL. The CALIni function must be called prior to calling any other CAL functions.

CALPurge52

Syntax `rReturn=CALPurge52(bVerify)`

Parameters

Type	*Identifier*	*Description*
SATBOOL	bVerify	Flag to verify purge.

Returns SATR rReturn values resulting from this function.

Return Value	*Causes*
SATR_SUCCESS	Successful purge.
SATR_FAIL	This indicates that the purge failed.

Description This function purges the EXP-F5200B microprocessor core using the on-demand purge capability. If bVerify is set to SAT_TRUE, the function will verify that the contents of the program and data RAMs in the EXP-F5200B microprocessor core were zeroized. If the verification fails, then the function will return SATR_FAIL; otherwise, the function will return SATR_SUCCESS.

CALPKTrfRes

Syntax `rReturn=CALPKTrfRes(bBlock)`

Parameters

Type	*Identifier*	*Description*
SATBOOL	bBlock	Blocking or non-blocking result transfer.

Returns Return values produced by this function depend upon the preceeding initiation function (*e.g.*, CALExpo). This section documents return values that arise directly from this function, not as a result of the initiation function.

SATR rReturn values resulting from this function. For other return values, refer to the initiating function.

Return Value	*Causes*
SATR_BUSY	For non-blocking operation this return indicates that the computation is still being performed and that a result is not yet ready.
SATR_PARITYFLUSH	This indicates that the hardware performing the computation entered an alarm state, typically as a result of an uncorrectable memory error. The hardware was successfully re-initialized; however, the operation was not completed as a result.
SATR_HFAULT	This indicates that the hardware performing the computation entered an alarm state and CAL was unable to re-initialize the hardware.
SATR_NOPEND	There is no pending or completed operation awaiting transfer of results.
SATR_FNP	Function referenced by the initiating function is not populated.

Description All the CAL functions initiate an operation and then return. This function is used to transfer the results of an operation once it has completed. The function can block until the operation has completed by setting bBlock to SAT_TRUE. If bBlock is set to SAT_FALSE, then the function will return SAT_BUSY if the processor is still busy; otherwise, it will transfer the results.

CALPreCompute

Syntax

```
rReturn=CALPreCompute(puiMod, puiMu, uiModLen)
```

Parameters

Type	*Identifier*	*Description*
const uint32_t*	puiMod	Pointer to modulus.
uint32_t*	puiMu	Pointer to pre-compute.
uint32_t	uiModLen	Modulus length in 32-bit words.

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus length is less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADPARAM	Most significant word of the modulus is zero.

CALPKTrfRes SATR rReturn values resulting from this function.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description

This function calculates the pre-compute of the modulus specified by puiMod with word length uiModLen for use in modular arithmetic. The modulus must be non-zero in the most significant word. The resulting pre-compute value will be stored at the location pointed at by puiMu with word length uiModLen+1.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

2.6 Conventional Public Key Cryptography Functions

Each of the functions listed in Table 2-3 is described in this section.

Table 2-3: TeraFire CAL Conventional Public Key Cryptography Functions

Function	*Description*	*Page*
CALExpo	Performs modular exponentiation	30
CALModRed	Performs modular reduction	32
CALMMult	Performs modular multiplication	34
CALMMultAdd	Performs a modular multiply and add	36
CALRSACRT	Performs RSA CRT	38
CALRSACRTSign	Performs RSA private key operation with CRT	40
CALRSACRTSign-Hash	Performs RSA CRT with a hash	42
CALDSASign	Calculates a DSA signature	52
CALDSAVerify	Verifies a DSA signature	54
CALRSASign	Performs RSA Sign	44
CALRSAVerify	Performs RSA Verify	46
CALRSASignHash	Performs RSA Sign with a hash	48
CALRSAVerifyHash	Performs RSA Verify with a hash	50
CALDSASignHash	Performs DSA Sign with a hash	56
CALDSAVerifyHash	Performs DSA Verify with a hash	58

CALExpo

Syntax

```
rReturn=CALExpo(puiBase, puiExpo, puiMod, puiMu,
uiExpLen, uiModLen, puiResult)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiBase	Pointer to base.
const uint32_t *	puiExpo	Pointer to exponent (SAT_NULL).
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL).
const uint32_t *	puiMu	Pointer to pre-compute (SAT_NULL).
uint32_t	uiExpLen	Exponent length in words.
uint32_t	uiModLen	Modulus length in words.
uint32_t *	puiResult	Pointer to modular exponentiation result.

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus length is less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADLEN	The exponent length is zero or greater than the maximum size supported by the implementation.

CALPKTrfRes will place the result from this operation in the location specified by puiResult, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description

This function performs modular exponentiation on the base specified by puiBase raised to the exponent specified by puiExpo performed over the modulus specified by puiMod with a pre-compute specified by puiMu. The exponent word length is uiExpLen and the modulus word

length is uiModLen. The modulus length must be at least two words, and the exponent length must be non-zero. The length of the pre-compute is uiModLen+1. The result is stored at the location pointed at by puiResult and is uiModLen words.

For performance reasons, the exponent length (uiExpLen) should be set to the actual length of the exponent in words (*i.e.*, the least value such that the most significant word is non-zero). For example, if the exponent is $2^{16}+1$, then uiExpLen should be set to 1; setting it to a larger value will produce the same result but with unnecessary additional processing.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALModRed

Syntax

```
rReturn=CALModRed(puiA, puiMod, puiMu, uiALen,
uiModLen, puiResult)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiA	Pointer to operand to be reduced.
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL).
const uint32_t *	puiMu	Pointer to pre-compute (SAT_NULL).
uint32_t	uiALen	Operand length in words.
uint32_t	uiModLen	Modulus length in words.
uint32_t *	puiResult	Pointer to modular reduction result.

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus length is less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADLEN	The operand length is zero, greater than twice the modulus length, or greater than the maximum size supported by the implementation.

CALPKTrfRes will place the result from this operation in the location specified by puiResult, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description

This function performs modular reduction on the operand specified by puiA over the modulus specified by puiMod with a pre-compute specified by puiMu. The operand word length is uiALen, and must be non-zero and less than or equal to twice the modulus length. The modulus word length is uiModLen, and must be at least two words. The length of

the pre-compute is uiModLen+1. The result is stored at the location pointed at by puiResult and is uiModLen words.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALMMult

Syntax

```
rReturn=CALMMult(puiA, puiB, puiMod, puiMu,
uiModLen, puiResult)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiA	Pointer to modular operand A.
const uint32_t *	puiB	Pointer to modular operand B.
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL).
const uint32_t *	puiMu	Pointer to pre-compute (SAT_NULL).
uint32_t	uiModLen	Modulus and operand lengths in words.
uint32_t *	puiResult	Pointer to modular multiplication result.

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus length is less than 2 words or greater than the maximum size supported by the implementation.

CALPKTrfRes will place the result from this operation in the location specified by puiResult, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description

This function performs modular multiplication on the operands specified by puiA and puiB over the modulus specified by puiMod with a pre-compute specified by puiMu. Both the operand word lengths and the modulus word length are uiModLen. The length of the pre-compute is uiModLen+1. The result is stored at the location pointed at by puiResult and is uiModLen words.

The operands puiA and puiB do not have to be modularly reduced prior to initiating the computation; however, they may not be larger in words, than the modulus.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALMMultAdd

Syntax

```
rReturn=CALMMultAdd(puiA, puiB, puiC, puiMod,
puiMu, uiModLen, puiResult)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiA	Pointer to modular operand A.
const uint32_t *	puiB	Pointer to modular operand B.
const uint32_t *	puiC	Pointer to modular operand C.
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL).
const uint32_t *	puiMu	Pointer to pre-compute (SAT_NULL).
uint32_t	uiModLen	Modulus and operand lengths in words.
uint32_t *	puiResult	Pointer to modular multiply add result.

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus length is less than 2 words or greater than the maximum size supported by the implementation.

CALPKTrfRes will place the result from this operation in the location specified by puiResult, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description

This function performs a modular multiply and add of the form (A*B + C (mod M)). The input operands are specified by puiA, puiB, and puiC over the modulus specified by puiMod with a pre-compute specified by puiMu. This function requires that the operand, puiC, be reduced by the modulus, puiMod, prior to performing modular multiply and add. Both the operand word lengths and the modulus word length are uiModLen.

The length of the pre-compute is uiModLen+1. The result is stored at the location pointed at by puiResult and is uiModLen words.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALRSACRT

Syntax

```
rReturn=CALRSACRT(puiCipher, puiQInv, puiDP,
puiDQ, puiP, puiPMu, puiQ, puiQMu,  uiLen,
puiPlain)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiCipher	Pointer to ciphertext input.
const uint32_t *	puiQInv	Pointer to Q inverse parameter.
const uint32_t *	puiDP	Pointer to DP parameter.
const uint32_t *	puiDQ	Pointer to DQ parameter.
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL).
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL).
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL).
const uint32_t *	puiQMu	Pointer to Q modulus pre-compute (SAT_NULL).
uint32_t	uiLen	Modulus and operand lengths in words.
uint32_t *	puiPlain	Pointer to plaintext result.

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

CALPKTrfRes will place the result from this operation in the location specified by puiPlain, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description This function performs an RSA private key decryption operation with CRT on the data buffer pointed at by puiCipher and stores the result at the location pointed at by puiPlain. The RSA CRT parameters are specified by puiQInv, puiDP, and puiDQ. The prime moduli are specified by puiP and puiQ with corresponding pre-compute values specified by puiPMu and puiQMu. The word lengths of puiQInv, puiDP, puiDQ, puiP, and puiQ are uiLen, and uiLen must be at least two. The word lengths of puiPMu and puiQMu are uiLen+1. The word length of puiCipher and puiPlain are twice uiLen (2×uiLen). For example, uiLen is 32 for a 2048-bit RSA decryption.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALRSACRTSign

Syntax

```
rReturn=CALRSACRTSign(eRSAEncod, eHashType,
puiHash, puiQInv, puiDP, puiDQ, puiP, puiPMu,
puiQ, puiQMu,  uiLen, puiSig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATENCODING-TYPE	eRSAEncod	Type of RSA encoding. See Table 4-2 for valid values.
SATHASHTYPE	eHashType	Hash type. See Table 4-3 for defined values.
const uint32_t *	puiHash	Pointer to hash value.
const uint32_t *	puiQInv	Pointer to Q inverse parameter.
const uint32_t *	puiDP	Pointer to DP parameter.
const uint32_t *	puiDQ	Pointer to DQ parameter.
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL).
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL).
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL).
const uint32_t *	puiQMu	Pointer to Q modulus pre-compute (SAT_NULL).
const uint32_t *	puiN	Pointer to public modulus N.
uint32_t	uiLen	Modulus and operand lengths in words.
uint32_t *	puiSig	Pointer to signature result.

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADRSAENC	Invalid RSA encoding type.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

If the function executes successfully, CALPKTrfRes will place the signature in the location specified by puiSig. SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_BADPARAM	For PKCS encoding, padding length is less than 3 words.

Description

This function computes an RSA-CRT signature using a user-supplied hash pointed at by puiHash using the RSA encoding type provided by eRsaEncod. The hash type used is given by eHashType, and the size of the vector pointed at by puiHash is automatically determined by the hash type (eHashType) parameter. The RSA-CRT parameters are specified by puiQInv, puiDP, and puiDQ. The prime moduli are specified by puiP and puiQ with corresponding pre-compute values specified by puiPMu and puiQMu. The public modulus is specified by puiN. The result is stored at the location pointed at by puiSig. The word lengths of puiQInv, puiDP, puiDQ, puiP, and puiQ are uiLen, and the word length of puiN is 2×uiLen+1. The word lengths of puiPMu and puiQMu are uiLen+1. The word length of puiSig is twice uiLen (2×uiLen).

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALRSACRTSignHash

Syntax

```
rReturn=CALRSACRTSignHash(eRsaEncod, eHashType,
puiMsg, puiQInv, puiDP, puiDQ, puiP, puiPMu,
puiQ, puiQMu, uiModLen, uiMsgLen, puiSig, bDMA,
uiDMAChConfig)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATRSAENC-TYPE	eRsaEncod	Type of RSA encoding.
SATHASHTYPE	eHashType	Hash type
const uint32_t *	puiMsg	Pointer to message value
const uint32_t *	puiQInv	Pointer to Q inverse parameter
const uint32_t *	puiDP	Pointer to DP parameter
const uint32_t *	puiDQ	Pointer to DQ parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL)
const uint32_t *	puiQMu	Pointer to Q modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to public modulus N.
uint32_t	uiMsgLen	Message length in bytes
uint32_t	uiModLen	Modulus length in words
uint32_t *	puiSig	Pointer to signature result
SATBOOL	bDMA	DMA select flag
uint32_t	uiDMAChCon-fig	DMA channel configuration word

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

Return Value	*Causes*
SATR_BADRSAENC	Invalid RSA encoding type.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

If the function executes successfully, CALPKTrfRes will place the signature in the location specified by puiSig. SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_BADPARAM	For PKCS encoding, padding length is less than 3 words.

Description

This function computes an RSA-CRT signature on the message hash produced from puiMsg using the hash type given by eHashType and the RSA encoding type provided by eRsaEncod. The result is stored at the location pointed at by puiSig. The RSA-CRT parameters are specified by puiQInv, puiDP, and puiDQ. The prime moduli are specified by puiP and puiQ with corresponding pre-compute values specified by puiPMu and puiQMu. The public modulus is specified by puiN. The result is stored at the location pointed at by puiSig. The word lengths of puiQInv, puiDP, puiDQ, puiP, and puiQ are uiLen, and the word length of puiN is 2×uiLen+1. The word lengths of puiPMu and puiQMu are uiLen+1. The word length of puiSig is twice uiLen (2×uiLen).

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALRSASign

Syntax

```
rReturn=CALRSASign(eRsaEncod, eHashType, puiHash,
puiE, puiN, puiNMu, uiModLen, puiS)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATRSAENC-TYPE	eRsaEncod	Type of RSA encoding
SATHASHTYPE	eHashType	Hash type
const uint32_t *	puiHash	Pointer to hash value
const uint32_t *	puiD	Pointer to private key exponent value
const uint32_t *	puiN	Pointer to modulus value
const uint32_t *	puiNMu	Pointer to modulus pre-compute value
uint32_t	uiModLen	Modulus length in words
uint32_t *	puiSig	Pointer to signature value

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADRSAENC	Invalid RSA encoding type.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

If the function executes successfully, CALPKTrfRes will place the signature in the location specified by puiSig. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_BADPARAM	For PKCS encoding, padding length is less than 3 words.

Description

Note: this function performs the same function as *CALRSACRTSign* on page 40, except without the performance benefits of the CRT optimization.

This function computes an RSA signature using a user-supplied hash pointed at by puiHash using the RSA encoding type provided by eRsaEncod. The hash type used is given by eHashType, and the size of the vector pointed at by puiHash is automatically determined by the hash type (eHashType) parameter. The public modulus is specified by puiN, and the pre-compute associated with the public modulus is specified by puiNMu. The private exponent is given by puiD. The result is stored at the location pointed at by puiSig. The word lengths of puiN and puiD are uiModLen, and the word length of puiNMu is uiModLen+1. The word length of puiSig is uiModLen+1.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALRSAVerify

Syntax

```
rReturn=CALRSAVerify(eRsaEncod, eHashType,
puiHash, puiE, uiExpLen, puiN, puiNMu, uiModLen,
puiS)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATENCODING-TYPE	eRsaEncod	Type of RSA encoding
SATHASHTYPE	eHashType	Hash type
const uint32_t *	puiHash	Pointer to hash value
const uint32_t *	puiE	Pointer to public exponent value
const uint32_t	uiExpLen	Public exponent length in words
const uint32_t *	puiN	Pointer to modulus value
const uint32_t *	puiNMu	Pointer to modulus pre-compute value
uint32_t	uiModLen	Modulus length in words
uint32_t *	puiSig	Pointer to signature value

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADLEN	Exponent length is zero or greater than the modulus/operand lengths.
SATR_BADRSAENC	Invalid RSA encoding type.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

This function does not cause CALPKTrfRes to return a value. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Return Value	*Causes*
SATR_BADPARAM	For PKCS encoding, padding length is less than 3 words.
SATR_VERIFYFAIL	The signature verify failed.

Description This function performs signature verification for RSA public-key cryptography on the hash pointed to by puiHash using the RSA encoding type provided by eRsaEncod. The hash type is given by eHashType, and the size of the hash is implied by the hash type. The RSA parameters are specified by the public exponent, puiE, with word length uiExpLen, the modulus, puiN, with word length uiModLen, and modulus pre-compute value, puiNMu, with word length uiModLen+1. The signature input is pointed to by puiS.

CALRSASignHash

Syntax

```
rReturn=CALRSASignHash(eRsaEncod, eHashType,
puiMsg, puiD, puiN, puiNMu, uiMsgLen, uiModLen,
puiS, bDMA, uiDMAChConfig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATRSAENC-TYPE	eRsaEncod	Type of RSA encoding. See Table 4-2 for valid values.
SATHASHTYPE	eHashType	Hash type. See Table 4-3 for defined values.
const uint32_t *	puiMsg	Pointer to message value.
const uint32_t *	puiD	Pointer to private key exponent value.
const uint32_t *	puiN	Pointer to modulus value.
const uint32_t *	puiNMu	Pointer to modulus pre-compute value.
uint32_t	uiMsgLen	Message length in bytes.
uint32_t	uiModLen	Modulus length in words.
uint32_t *	puiS	Pointer to signature value.
SATBOOL	bDMA	DMA select flag.
uint32_t	uiDMACh-Config	Channel configuration word.

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADRSAENC	Invalid RSA encoding type.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

If the function executes successfully, CALPKTrfRes will place the signature in the location specified by puiSig. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_BADPARAM	For PKCS encoding, padding length is less than 3 words.

Description

This function performs signature generation for RSA public-key cryptography on the hash produced from puiMsg using the RSA encoding type provided by eRsaEncod. The hash type is given by eHashType. The RSA parameters are specified by the private exponent, puiD, the modulus, puiN, and modulus pre-compute value, puiNMu. The modulus length is specified by uiModLen, and the message length is specified by uiMsgLen. The final signature result is pointed to by puiS.

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALRSAVerifyHash

Syntax

```
rReturn=CALRSAVerifyHash(eRsaEncod, eHashType,
puiMsg, puiE, uiExpLen, puiN, puiNMu, uiMsgLen,
uiModLen, puiS, bDMA, uiDMAChConfig)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATENCODING-TYPE	eRsaEncod	Type of RSA encoding
SATHASHTYPE	eHashType	Hash type
const uint32_t *	puiMsg	Pointer to message value
const uint32_t *	puiE	Pointer to public exponent value
const uint32_t	uiExpLen	Public exponent length in words
const uint32_t *	puiN	Pointer to modulus value
const uint32_t *	puiNmu	Pointer to modulus pre-compute value
uint32_t	uiMsgLen	Message length in bytes
uint32_t	uiModLen	Modulus length in words
uint32_t *	puiS	Pointer to signature value
SATBOOL	bDMA	DMA select flag
uint32_t	uiDMACh-Config	Channel configuration word

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADLEN	Exponent length is zero or greater than the modulus/operand lengths.
SATR_BADRSAENC	Invalid RSA encoding type.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

This function does not cause CALPKTrfRes to return a value. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_BADPARAM	For PKCS encoding, padding length is less than 3 words.
SATR_VERIFYFAIL	The signature verify failed.

Description

This function performs signature verification for RSA public-key cryptography on the hash produced from puiMsg using the RSA encoding type provided by eRsaEncod. The hash type is given by eHashType. The RSA parameters are specified by the public exponent, puiE, with word length uiExpLen, the modulus, puiN, with word length uiModLen, and modulus pre-compute value, puiNMu, with word length uiModLen+1. The signature input is pointed to by puiS.

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALDSASign

Syntax

```
rReturn=CALDSASign(puiHash, puiG, puiK, puiX,
puiP, puiPMu, puiQ, puiQMu, uiN, uiL, puiSigR,
puiSigS)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiHash	Pointer to non-zero hash value
const uint32_t *	puiG	Pointer to G parameter
const uint32_t *	puiK	Pointer to K parameter
const uint32_t *	puiX	Pointer to X parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL)
const uint32_t *	puiQMu	Pointer to Q modulus pre-compute (SAT_NULL)
uint32_t	uiN	Q length parameter
uint32_t	uiL	P length parameter
uint32_t *	puiSigR	Pointer to signature R value
uint32_t*	puiSigS	Pointer to signature S value

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADLEN	uiN>uiL or uiN==uiL and P<=Q.

If the function executes successfully, CALPKTrfRes will place the signature in the locations specified by puiSigR and puiSigS. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_SIGNPARMK	K parameter must be in the range [1,Q-1].
SATR_SIGNFAIL	Signature fails if computed signature R value is 0 or signature S value is 0.

Description

This function calculates a DSA signature given the hash value of a message specified by puiHash. The resulting signature is stored at the locations pointed at by puiSigR and puiSigS. The DSA primes are specified by puiP and puiQ with corresponding pre-compute values specified by puiPMu and puiQMu. The DSA value G is specified by puiG. The private key is specified by puiX. The random per-message value is specified by puiK and must be in the range [1, Q-1]. The word length of the Q, X, and K, paramters is uiN and the word length of the P and G parameters is uiL. The size of the hash value must be greater than or equal to uiN, and the hash value in the uiN words used by this function must be non-zero. The value of uiL must be greater than or equal to uiN.

In the event of a failure due to the K parameter being out of range (SATR_SIGNPARMK) or a signature failure (SATR_SIGNFAIL), a new K parameter should be selected and the computation re-attempted.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALDSAVerify

Syntax

```
rReturn=CALDSAVerify(puiHash, puiG, puiY,
puiSigR, puiSigS, puiP, puiPMu, puiQ, puiQMu,
uiN, uiL)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiHash	Pointer to hash value
const uint32_t *	puiG	Pointer to G parameter
const uint32_t *	puiY	Pointer to Y parameter
const uint32_t *	puiSigR	Pointer to signature R value
const uint32_t *	puiSigS	Pointer to signature S value
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL)
const uint32_t *	puiQMu	Pointer to Q modulus pre-compute (SAT_NULL)
uint32_t	uiN	Q length parameter
uint32_t	uiL	P length parameter

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADLEN	uiN>uiL or uiN==uiL and P<=Q.

This function does not cause CALPKTrfRes to return a value. CALPkTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion with a verified signature.

Return Value	*Causes*
SATR_VERPARMR	The signature R parameter must be in the range [1,Q-1].
SATR_VERPARMS	The signature parameter S must be in the range [1,Q-1].
SATR_VERIFYFAIL	The signature verify failed.

Description This function verifies a DSA signature given the hash value of a message specified by puiHash. The signature to be verified is specified by puiSigR and puiSigS. The DSA primes are specified by puiP and puiQ with corresponding pre-compute values specified by puiPMu and puiQMu. The DSA value G is specified by puiG. The public key is specified by puiY. The word length of Q, SigR, and SigS is uiN and the word length of P, G, and Y is uiL. The size of the hash value must be greater than or equal to uiN. The value of uiL must be greater than or equal to uiN.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results. For this function the only return value is a return code.

CALDSASignHash

Syntax

```
rReturn=CALDSASignHash(puiMsg, eHashType,
uiMsgLen, puiG, puiK, puiX, puiP, puiPMu, puiQ,
puiQMu, uiN, uiL, puiSigR, puiSigS, bDMA,
uiDMAChConfig)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMsg	Pointer to message input
SATHASHTYPE	eHashType	Hash algorithm
uint32_t	uiMsgLen	Message length, in bytes
const uint32_t *	puiG	Pointer to G parameter
const uint32_t *	puiK	Pointer to K parameter
const uint32_t *	puiX	Pointer to X parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL)
const uint32_t *	puiQMu	Pointer to Q modulus pre-compute (SAT_NULL)
uint32_t	uiN	Q length parameter
uint32_t	uiL	P length parameter
uint32_t *	puiSigR	Pointer to signature R value
uint32_t*	puiSigS	Pointer to signature S value
SATBOOL	bDMA	DMA select flag
uint32_t	uiDMAChConfig	Channel configuration word

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

Return Value	*Causes*
SATR_BADLEN	uiN>uiL or uiN==uiL and P<=Q.
SATR_BADHASHTYPE	Invalid/unsupported hash type, including hash size less than N parameter.

If the function executes successfully, CALPKTrfRes will place the signature in the locations specified by puiSigR and puiSigS. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_SIGNPARMK	K parameter must be in the range [1,Q-1].
SATR_SIGNFAIL	Signature fails if computed signature R value is 0 or signature S value is 0.

Description

This function calculates a DSA signature for a message at puiMsg using the hash type eHashType. The resulting signature is stored at the locations pointed at by puiSigR and puiSigS. The DSA primes are specified by puiP and puiQ with corresponding pre-compute values specified by puiPMu and puiQMu. The DSA value G is specified by puiG. The private key is specified by puiX. The random per-message value is specified by puiK and must be in the range [1, Q-1]. The word length of the Q, X, and K, paramters is uiN and the word length of the P and G parameters is uiL. The size of the hash value must be greater than or equal to uiN. The value of uiL must be greater than or equal to uiN.

In the event of a failure due to the K parameter being out of range (SATR_SIGNPARMK) or a signature failure (SATR_SIGNFAIL), a new K parameter should be selected and the computation re-attempted.

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALDSAVerifyHash

Syntax

```
rReturn=CALDSAVerifyHash(puiMsg, eHashType,
uiMsgLen, puiG, puiY, puiSigR, puiSigS, puiP,
puiPMu, puiQ, puiQMu, uiN, uiL, bDMA,
uiDMAChConfig)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMsg	Pointer to message value
SATHASHTYPE	eHashType	Hash algorithm
uint32_t	uiMsgLen	Message length in bytes
const uint32_t *	puiG	Pointer to G parameter
const uint32_t *	puiY	Pointer to K parameter
const uint32_t *	puiSigR	Pointer to signature R value
const uint32_t *	puiSigS	Pointer to signature S value
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL)
const uint32_t *	puiQMu	Pointer to Q modulus pre-compute (SAT_NULL)
uint32_t	uiN	N length parameter
uint32_t	uiL	L length parameter
SATBOOL	bDMA	DMA select flag
uint32_t	uiDMACh-Config	Channel configuration word

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADLEN	uiN>uiL or uiN==uiL and P<=Q.

Return Value	*Causes*
SATR_BADHASHTYPE	Invalid/unsupported hash type.

This function does not cause CALPKTrfRes to return a value. CALPkTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion with a verified signature.
SATR_VERPARMR	The signature R parameter must be in the range [1,Q-1].
SATR_VERPARMS	The signature parameter S must be in the range [1,Q-1].
SATR_VERIFYFAIL	The signature verify failed.

Description

This function verifies a DSA signature based on the hash value calculated from the message, puiMsg. The hash type is given by eHashType. The signature to be verified is specified by puiSigR and puiSigS. The DSA primes are specified by puiP and puiQ, with corresponding precompute values specified by puiPMu and puiQMu. The DSA value G is specified by puiG. The public key is specified by puiY. The word length of Q, Hash, SigR, and SigS is uiN, and the word length of P, G, and Y is uiL.

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results. For this function the only return value is a return code.

2.7 Elliptic Curve Cryptography Functions

Each of the functions listed in Table 2-4 is described in this section.

Table 2-4: TeraFire CAL Elliptic Curve Cryptography Functions

Function	*Description*	*Page*
CALECMult	Performs elliptic curve multiplication	61
CALECMultTwist	Performs twisted elliptic curve multiplication	63
CALECMultAdd	Performs an elliptic curve multiply and add	65
CALECPtValidate	Performs elliptic curve point validation	67
CALECKeyPairGen	Generates elliptic curve public/private key pair	69
CALECDHC	Performs ECDHC primitive	72
CALECDSASign	Calculates an ECDSA signature	74
CALECDSAVerify	Verifies an ECDSA signature	76
CALECDSASign-Hash	Performs ECDSA Sign with a hash	79
CALECDSAVerify-Hash	Performs ECDSA Verify with a hash	82
CALECD-SASignTwist	Calculates an ECDSA signature on a twisted elliptic curve	85
CALECDSAVeri-fyTwist	Verifies an ECDSA signature on a twisted elliptic curve	87
CALECD-SASignTwistHash	Calculates an ECDSA signature on a twisted elliptic curve combined with a hash	90
CALECDSAVeri-fyTwistHash	Verifies an ECDSA signature on a twisted elliptic curve combined with a hash	93

CALECMult

Syntax

```
rReturn=CALECMult(puiMul, puiPx, puiPy, puiB,
puiMod, puiMu, uiLen, uiPtCompress, puiRx, puiRy)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMul	Pointer to multiplier (SAT_NULL)
const uint32_t *	puiPx	Pointer to base point x-coordinate
const uint32_t *	puiPy	Pointer to base point y-coordinate
const uint32_t *	puiB	Pointer to b curve parameter
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL)
const uint32_t *	puiMu	Pointer to modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t	uiPtCompress	Point compression control
uint32_t *	puiRx	Pointer to product x-coordinate
uint32_t *	puiRy	Pointer to product y-coordinate

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

If the function executes successfully, CALPKTrfRes will place the product components in the locations specified by puiRx and puiRy. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.

Return Value	*Causes*
SATR_DCMPPARMX	Point decompression with X parameter not in range [0, modulus-1].
SATR_DCMPPARMB	Point decompression with b parameter greater than modulus.
SATR_DCMPPARMP	Point decompression with modulus not of the form 4n+3.
SATR_VALIDATEFAIL	Point not on curve.
SATR_PAF	Point at infinity result generated.

Description

This function performs an elliptic curve point multiply over a curve $y^2 = x^3 - 3x + b$. The multiplier is specified by puiMul. The affine point to be multiplied is specified by puiPx and puiPy. The field's prime modulus is specified by puiMod, with corresponding pre-compute value puiMu. The *b* curve parameter is specified by puiB. The word length of all operands except puiMu is uiLen; the word length of puiMu is uiLen+1. The resulting point product will be stored at the location pointed at by puiRx and puiRy.

This function supports point compression of point P. When bit 1 of uiPtCompress is set, this indicates that P is compressed, and bit 0 of uiPtCompress is the LSB of Py. When point compression is used, puiPy (Py) should point to the EC curve parameter *b*.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECMultTwist

Syntax

```
rReturn=CALECMultTwist(puiMul, puiPx, puiPy,
puiB, puiZ, puiMod, puiMu, puiN, uiLen, puiRx,
puiRy)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMul	Pointer to multiplier (SAT_NULL)
const uint32_t *	puiPx	Pointer to base point x-coordinate
const uint32_t *	puiPy	Pointer to base point y-coordinate
const uint32_t *	puiB	Pointer to twisted b curve parameter
const uint32_t *	puiZ	Pointer to twist factor
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL)
const uint32_t *	puiMu	Pointer to modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t *	puiRx	Pointer to product x-coordinate
uint32_t *	puiRy	Pointer to product y-coordinate

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

If the function executes successfully, CALPKTrfRes will place the product components in the locations specified by puiRx and puiRy. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].

Return Value	*Causes*
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALIDATEFAIL	Point not on curve.
SATR_PAF	Point at infinity result generated.

Description This function performs an elliptic curve point multiply over a twisted curve, see *Twisted Elliptic Curves* on page 14. The multiplier is specified by puiMul. The affine point to be multiplied is specified by puiPx and puiPy. The field's prime modulus is specified by puiMod, with corresponding pre-compute value puiMu. The *b* curve parameter is specified by puiB, and the twist factor is specified by puiZ. The order of the group is specified by puiN. The word length of all operands except puiMu is uiLen; the word length of puiMu is uiLen+1. The resulting point product will be stored at the location pointed at by puiRx and puiRy.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECMultAdd

Syntax

```
rReturn=CALECMultAdd(puiMul, puiPx, puiPy, puiQx,
puiQy, puiB, puiMod, puiMu, uiLen, uiPtPCompress,
uiPtQCompress, puiRx, puiRy)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMul	Pointer to multiplier (SAT_NULL)
const uint32_t *	puiPx	Pointer to multiplicand point x-coordinate
const uint32_t *	puiPy	Pointer to multiplicand point y-coordinate
const uint32_t *	puiQx	Pointer to addition point x-coordinate
const uint32_t *	puiQy	Pointer to addition point y-coordinate
const uint32_t *	puiB	Pointer to b curve parameter
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL)
const uint32_t *	puiMu	Pointer to modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t	uiPtPCompress	Point compression control for point P
uint32_t	uiPtQCompress	Point compression control for point Q
uint32_t *	puiRx	Pointer to result x-coordinate
uint32_t *	puiRy	Pointer to result y-coordinate

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

If the function executes successfully, CALPKTrfRes will place the result components in the locations specified by puiRx and puiRy. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_DCMPPARMX	Point decompression with X parameter not in range [0, modulus-1].
SATR_DCMPPARMB	Point decompression with b parameter greater than modulus.
SATR_DCMPPARMP	Point decompression with modulus not of the form 4n+3.
SATR_VALIDATEFAIL	Point not on curve.
SATR_PAF	Point at infinity result generated.

Description

This function performs an elliptic curve point multiply and add of the form Mul*P+Q over a curve $y^2 = x^3 - 3x + b$. The multiplier is specified by puiMul. The affine point to be multiplied is specified by puiPx and puiPy. The affine point to be added is specified by puiQx and puiQy. The *b* curve parameter is specified by puiB, and the field's prime modulus is specified by puiMod with corresponding pre-compute value puiMu. The word length of all operands is uiLen. The resulting point will be stored at the location pointed at by puiRx and puiRy.

This function supports point compression of P and Q. When bit 1 of uiPtPCompress is set, this indicates that P is compressed, and bit 0 of uiPtPCompress is the LSB of Py. When point P is compressed, puiPy (pY) should point to the EC curve parameter *b*. When bit 1 of uiPtQCompress is set, this indicates that Q is compressed, and bit 0 of uiPtQCompress is the LSB of Qy. When point Q is compressed, puiQy (Qy) should point to the EC curve parameter *b*.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECPtValidate

Syntax

```
rReturn=CALECPtValidate(puiPx, puiPy, puiB,
puiMod, puiMu, uiLen)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation

Type	*Identifier*	*Description*
const uint32_t *	puiPx	Pointer to P x-coordinate
const uint32_t *	puiPy	Pointer to P y-coordinate
const uint32_t *	puiB	Pointer to b parameter
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL)
const uint32_t *	puiMu	Pointer to modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

This function does not cause CALPKTrfRes to return a value. CALPkTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALIDATEFAIL	Point not on curve.

Description

This function validates whether the point specified by puiPx and puiPy is on the curve $y^2 = x^3 - 3x + b$. The b curve parameter is specified by puiB, and the field's prime modulus is specified by puiMod with corre-

sponding pre-compute value puiMu. The word length of all operands is uiLen.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results. For this function the only return value is a return code.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECKeyPairGen

Syntax

```
rReturn=CALECKeyPairGen(puiC, puiPx, puiPy,
puiMod, puiMu, puiNM1, puiNM1Mu, puiB, puiD,
puiQx, puiQy, uiLen)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
const uint32_t *	puiC	Pointer to random value *c*, or SAT_NULL
const uint32_t *	puiPx	Pointer to P x-coordinate
const uint32_t *	puiPy	Pointer to P y-coordinate
const uint32_t *	puiMod	Pointer to modulus
const uint32_t *	puiMu	Pointer to modulus pre-compute
const uint32_t *	puiNM1	Pointer to curve order minus 1
const uint32_t *	puiNM1Mu	Pointer to curve order minus 1 precompute
const uint32_t *	puiB	Pointer to b parameter
uint32_t *	puiD	Pointer to private key result *d*
uint32_t *	puiQx	Pointer to public key Q x-coordinate result
uint32_t *	puiQy	Pointer to public key Q y-coordinate result
uint32_t	uiLen	Modulus length in words

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 5 words or greater than the maximum size supported by the implementation.
SATR_FAIL	puiC is SAT_NULL and the implementation lacks a DRBG implementation, or has a DRBG implementation but it is not instantiated.

CALPkTrfRes SATR rReturn values resulting from this function are listed below. If the function is called with puiC set to SAT_NULL, then the

return values associated with CALDRBGGenerate (see *CALDRBGGenerate* on page 194) may also be returned.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALIDATEFAIL	Point not on curve.

Description This function generates a public/private elliptic curve cryptography key pair using the key pair generation using extra random bits as specified in FIPS 186-4, Section B.4.1. There are two modes of operation:

- The random value is supplied as an input by the user using puiC. The random value supplied must be 64-bits longer than the actual size of the curve modulus. For example, if the curve used is P-521, the the modulus must be 585-bits. Since data is transfered in units of 32-bit words, any additional bits must be zero for conforming operation.
- The random value is generated internally by the implementation by setting puiC to SAT_NULL. If this mode is used, the pre-requisite is that the implementation has an appropriate random number generation capability, and that the random number generator is in an instantiated state with the correct security strength.

Except for the puiC parameter, the remaining parameters are the same regardless of mode. The base point for the curve is supplied by the puiPx and puiPy parameters, the *b* curve parameter is supplied by the puiB parameter, and the curve modulus is supplied by the puiMod parameter. The pre-compute for the curve modulus is supplied by the puiMu parameter. The function also requires the order of the curve, minus one (puiNM1), as well as the associated pre-compute for that value. Upon completion of execution, the private key is stored at the location pointed at by puiD, and the public key is stored at the locations pointed at by puiQx and puiQy. The length of most parameters and results is provided by the uiLen parameter, except the pre-computes (puiMu and puiNM1Mu), which are uiLen+1 words, and puiC (if not SAT_NULL), which is uiLen+3 words.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDHC

Syntax

```
rReturn=CALECDHC(puiS, puiWx, puiWy, puiB,
puiMod, puiMu, puiK, puiR, puiRMu, uiLen,
uiPtCompress, puiZ)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiS	Pointer to secret key
const uint32_t *	puiWx	Pointer to public key x-coordinate (SAT_NULL)
const uint32_t *	puiWy	Pointer to public key x-coordinate (SAT_NULL)
const uint32_t *	puiB	Pointer to b parameter
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL)
const uint32_t *	puiMu	Pointer to modulus pre-compute (SAT_NULL)
const uint32_t *	puiK	Pointer to cofactor
const uint32_t *	puiR	Pointer to prime divisor of the order of the curve
const uint32_t *	puiRMu	Pointer to R modulus pre-compute
uint32_t	uiLen	Modulus length in words
uint32_t	uiPtCompress	Point compression control
uint32_t *	puiZ	Pointer to shared secret result location

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

If the function executes successfully, CALPKTrfRes will place the result in the location specified by puiZ. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_DCMPPARMX	Point decompression with X parameter not in range [0, modulus-1].
SATR_DCMPPARMB	Point decompression with b parameter greater than modulus.
SATR_DCMPPARMP	Point decompression with modulus not of the form 4n+3.
SATR_VALIDATEFAIL	Point not on curve.
SATR_PAF	Point at infinity result generated.

Description This function implements the ECDHC primitive used in IEEE 1363-2000 and other specifications. The function operates by performing a modular multiply on the user's secret key puiS (s) with the cofactor puiK (k). An EC multiply is then performed using the result and the other party's public key, which is specified by puiWx and puiWy (W'). The field's prime modulus is specified by puiMod, with corresponding pre-compute value puiMu. The order of the curve is specified by puiR, with corresponding pre-compute value puiRMu. If the field has a cofactor==1, then puiK, puiR, and puiRMu should be set to SAT_NULL for efficient operation. The word length of all operands is uiLen. The resulting shared secret (z) will be stored at the location pointed at by puiZ.

This function supports point compression of point W'. When bit 1 of uiPtCompress is set, this indicates that W' is compressed, and bit 0 of uiPtCompress is the LSB of Wy. When point compression is used, puiWy (Wy) should point to the EC curve parameter *b*.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDSASign

This function may be subject to third party IP claims (see *Intellectual Property Rights* on page 12).

Syntax

```
rReturn=CALECDSASign(puiHash, puiGx, puiGy, puiK,
puiD, puiB, puiP, puiPMu, puiN, puiNMu, uiLen,
puiSigR, puiSigS)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiHash	Pointer to hash value
const uint32_t *	puiGx	Pointer to G x-coordinate parameter
const uint32_t *	puiGy	Pointer to G y-coordinate parameter
const uint32_t *	puiK	Pointer to K parameter
const uint32_t *	puiD	Pointer to D parameter
const uint32_t *	puiB	Pointer to b curve parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
const uint32_t *	puiNMu	Pointer to N modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t *	puiSigR	Pointer to signature R value
uint32_t *	puiSigS	Pointer to signature S value

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

If the function executes successfully, CALPKTrfRes will place the result components in the locations specified by puiSigR and puiSigS. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALIDATEFAIL	Point not on curve.
SATR_SIGNPARMD	D parameter not in range [1, N-1].
SATR_SIGNPARMK	K parameter not in range [1, N-1].
SATR_SIGNFAIL	Signature failure: puiSigR=0 or puiSigS=0.

Description

This function calculates an ECDSA signature given the hash value of a message specified by puiHash. The resulting signature is stored at the locations pointed at by puiSigR and puiSigS. The EC base point G is specified in affine coordinates by puiGx and puiGy. The b parameter of the curve $y^2 = x^3 - 3x + b$ is specified by puiB. The EC field's prime modulus is specified by puiP with corresponding pre-compute value specified by puiPMu. The order of the group is specified by puiN with corresponding pre-compute value specified by puiNMu. The private key is specified by puiD. The random per-message value is specified by puiK. The word length of all operands, except puiPMu and puiNMu is uiLen; the word length of puiPMu and puiNMu is uiLen+1.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDSAVerify

This function may be subject to third party IP claims (see *Intellectual Property Rights* on page 12).

Syntax

```
rReturn=CALECDSAVerify(puiHash, puiGx, puiGy,
puiQx, puiQy, puiSigR, puiSigS, puiB, puiP,
puiPMu, puiN, puiNMu, uiLen, uiPtCompress)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiHash	Pointer to hash value
const uint32_t *	puiGx	Pointer to G x-coordinate
const uint32_t *	puiGy	Pointer to G y-coordinate
const uint32_t *	puiQx	Pointer to Q x-coordiante
const uint32_t *	puiQy	Pointer to Q y-coordinate
const uint32_t *	puiSigR	Pointer to signature R value
const uint32_t *	puiSigS	Pointer to signature S value
const uint32_t *	puiB	Pointer to b curve parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
const uint32_t *	puiNMu	Pointer to N modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t	uiPtCompress	Point compression control

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

This function does not cause CALPKTrfRes to return a value. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_DCMPPARMX	Point decompression with X parameter not in range [0, modulus-1].
SATR_DCMPPARMB	Point decompression with b parameter greater than modulus.
SATR_DCMPPARMP	Point decompression with modulus not of the form 4n+3.
SATR_VERPARMR	SigR parameter not in [1, modulus-1].
SATR_VERPARMS	SigS parameter not in [1, modulus-1].
SATR_VALIDATEFAIL	Point not on curve.
SATR_PAF	Point at infinity result generated.
SATR_VERIFYFAIL	The signature verify failed.

Description

This function verifies an ECDSA signature given the hash value of a message specified by puiHash. The signature to be verified is specified by puiSigR and puiSigS. The EC base point G is specified in affine coordinates by puiGx and puiGy. For the curve $y^2 = x^3 - 3x + b$, b is specified by puiB. The EC field's prime modulus is specified by puiP with corresponding pre-compute value specified by puiPMu. The order N of the group is specified by puiN with corresponding pre-compute value specified by puiNMu. The public key Q is specified in affine coordinates by puiQx and puiQy. The word length of all operands is uiLen.

This function supports point compression of Q. When bit 1 of uiPtCompress is set, this indicates that Q is compressed, and bit 0 of uiPtCompress is the LSB of Qy. When point compression is used, puiQy (Qy) should point to the EC curve parameter *b*.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results. For this function the only return value is a return code.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDSASignHash

Syntax

```
rReturn=CALECDSASignHash(puiMsg, eHashType,
uiMsgLen, puiGx, puiGy, puiK, puiD, puiB, puiP,
puiPMu, puiN, puiNMu, uiLen, puiSigR, puiSigS,
bDMA, uiDMAChConfig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMsg	Pointer to message value
SATHASHTYPE	eHashType	Hash algorithm
uint32_t	uiMsgLen	Message length in bytes
const uint32_t *	puiGx	Pointer to G x-coordinate parameter
const uint32_t *	puiGy	Pointer to G y-coordinate parameter
const uint32_t *	puiK	Pointer to K parameter
const uint32_t *	puiD	Pointer to D parameter
const uint32_t *	puiB	Pointer to B parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
const uint32_t *	puiNMu	Pointer to N modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t *	puiSigR	Pointer to signature R value
uint32_t *	puiSigS	Pointer to signature S value
SATBOOL	bDMA	DMA select flag
uint32_t	uiDMACh-Config	Channel configuration word

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.

Return Value	*Causes*
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

If the function executes successfully, CALPKTrfRes will place the result components in the locations specified by puiSigR and puiSigS. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALIDATEFAIL	Point not on curve.
SATR_SIGNPARMD	D parameter not in range [1, N-1].
SATR_SIGNPARMK	K parameter not in range [1, N-1].
SATR_SIGNFAIL	Signature failure: puiSigR=0 or puiSigS=0.

Description

This function calculates an ECDSA signature based on the hash value calculated from the message, puiMsg. The hash is given by eHashType. The resulting signature is stored at the locations pointed at by puiSigR and puiSigS. The EC base point G is specified in affine coordinates by puiGx and puiGy. The b parameter of the curve $y^2 = x^3 - 3x + b$ is specified by puiB. The EC field's prime modulus is specified by puiP with corresponding pre-compute value specified by puiPMu. The order of the group is specified by puiN with corresponding pre-compute value specified by puiNMu. The private key is specified by puiD. The random per-message value is specified by puiK. The word length of all operands is uiLen.

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDSAVerifyHash

Syntax

```
rReturn=CALECDSAVerifyHash(puiMsg, eHashType,
uiMsgLen, puiGx, puiGy, puiQx, puiQy, puiSigR,
puiSigS, puiB, puiP, puiPMu, puiN, puiNMu, uiLen,
uiPtCompress, bDMA, uiDMAChConfig)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMsg	Pointer to message value
SATHASHTYPE	eHashType	Hash algorithm
uint32_t	uiMsgLen	Message length in bytes
const uint32_t *	puiGx	Pointer to G x-coordinate parameter
const uint32_t *	puiGy	Pointer to G y-coordinate parameter
const uint32_t *	puiQx	Pointer to Q x-coordinate
const uint32_t *	puiQy	Pointer to Q y-coordinate
const uint32_t *	puiSigR	Pointer to signature R value
const uint32_t *	puiSigS	Pointer to signature S value
const uint32_t *	puiB	Pointer to B parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
const uint32_t *	puiNMu	Pointer to N modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t	uiPtCompress	Point compression control
SATBOOL	bDMA	DMA select flag
uint32_t	uiDMACh-Config	Channel configuration word

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.

Return Value	*Causes*
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

This function does not cause CALPKTrfRes to return a value. CALPkTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_DCMPPARMX	Point decompression with X parameter not in range [0, modulus-1].
SATR_DCMPPARMB	Point decompression with b parameter greater than modulus.
SATR_DCMPPARMP	Point decompression with modulus not of the form 4n+3.
SATR_VERPARMR	SigR parameter not in [1, modulus-1].
SATR_VERPARMS	SigS parameter not in [1, modulus-1].
SATR_VALIDATEFAIL	Point not on curve.
SATR_PAF	Point at infinity result generated.
SATR_VERIFYFAIL	The signature verify failed.

Description

This function verifies an ECDSA signature based on the hash value calculated from the message, puiMsg. The hash is given by eHashType. The signature to be verified is specified by puiSigR and puiSigS. The EC base point G is specified in affine coordinates by puiGx and puiGy. The b parameter of the curve $y^2 = x^3 - 3x + b$ is specified by puiB. The EC field's prime modulus is specified by puiP with corresponding pre-compute value specified by puiPMu. The order N of the group is specified by puiN with corresponding pre-compute value specified by puiNMu. The public key Q is specified in affine coordinates by puiQx and puiQy. The word length of all operands is uiLen.

This function supports point compression of Q. When bit 1 of uiPtCompress is set, this indicates that Q is compressed, and bit 0 of uiPtCom-

press is the LSB of Qy. When point compression is used, puiQy (Qy) should point to the EC curve parameter b.

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDSASignTwist

Syntax

```
rReturn=CALECDSASignTwist(puiHash, puiGx, puiGy,
puiK, puiD, puiB, puiZ, puiP, puiPMu, puiN,
puiNMu, uiLen, puiSigR, puiSigS)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiHash	Pointer to hash value
const uint32_t *	puiGx	Pointer to G x-coordinate parameter
const uint32_t *	puiGy	Pointer to G y-coordinate parameter
const uint32_t *	puiK	Pointer to K parameter
const uint32_t *	puiD	Pointer to D parameter
const uint32_t *	puiB	Pointer to twisted b curve parameter
const uint32_t*	puiZ	Pointer to twist factor
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
const uint32_t *	puiNMu	Pointer to N modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t *	puiSigR	Pointer to signature R value
uint32_t *	puiSigS	Pointer to signature S value

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

If the function executes successfully, CALPKTrfRes will place the result components in the locations specified by puiSigR and puiSigS. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALIDATEFAIL	Point not on curve.
SATR_SIGNPARMD	D parameter not in range [1, N-1].
SATR_SIGNPARMK	K parameter not in range [1, N-1].
SATR_SIGNFAIL	Signature failure: puiSigR=0 or puiSigS=0.

Description

This function calculates an ECDSA signature on a twisted curve (see *Twisted Elliptic Curves* on page 14) given the hash value of a message specified by puiHash. The resulting signature is stored at the locations pointed at by puiSigR and puiSigS. The EC base point G is specified in affine coordinates by puiGx and puiGy. The *b* parameter of the curve is specified by puiB, and the curve twist is specified by puiZ. The EC field's prime modulus is specified by puiP with corresponding pre-compute value specified by puiPMu. The order of the group is specified by puiN with corresponding pre-compute value specified by puiNMu. The private key is specified by puiD. The random per-message value is specified by puiK. The word length of all operands, except puiPMu and puiNMu is uiLen; the word length of puiPMu and puiNMu is uiLen+1.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDSAVerifyTwist

Syntax

```
rReturn=CALECDSAVerifyTwist(puiHash, puiGx,
puiGy, puiQx, puiQy, puiSigR, puiSigS, puiB,
puiP, puiPMu, puiN, puiNMu, uiLen, uiPtCompress)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiHash	Pointer to hash value
const uint32_t *	puiGx	Pointer to G x-coordinate
const uint32_t *	puiGy	Pointer to G y-coordinate
const uint32_t *	puiQx	Pointer to Q x-coordiante
const uint32_t *	puiQy	Pointer to Q y-coordinate
const uint32_t *	puiSigR	Pointer to signature R value
const uint32_t *	puiSigS	Pointer to signature S value
const uint32_t *	puiB	Pointer to twisted b curve parameter
const uint32_t*	puiZ	Pointer to twist factor
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
const uint32_t *	puiNMu	Pointer to N modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t	uiPtCompress	Point compression control

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

This function does not cause CALPKTrfRes to return a value. CALPkTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_DCMPPARMX	Point decompression with X parameter not in range [0, modulus-1].
SATR_DCMPPARMB	Point decompression with b parameter greater than modulus.
SATR_DCMPPARMP	Point decompression with modulus not of the form 4n+3.
SATR_VERPARMR	SigR parameter not in [1, modulus-1].
SATR_VERPARMS	SigS parameter not in [1, modulus-1].
SATR_VALIDATEFAIL	Point not on curve.
SATR_PAF	Point at infinity result generated.
SATR_VERIFYFAIL	The signature verify failed.

Description

This function verifies an ECDSA signature on a twisted curve (see *Twisted Elliptic Curves* on page 14) given the hash value of a message specified by puiHash. The signature to be verified is specified by puiSigR and puiSigS. The EC base point G is specified in affine coordinates by puiGx and puiGy. The *b* parameter of the curve is specified by puiB, and the curve twist is specified by puiZ. The EC field's prime modulus is specified by puiP with corresponding pre-compute value specified by puiPMu. The order N of the group is specified by puiN with corresponding pre-compute value specified by puiNMu. The public key Q is specified in affine coordinates by puiQx and puiQy. The word length of all operands is uiLen.

This function supports point compression of Q. When bit 1 of uiPtCompress is set, this indicates that Q is compressed, and bit 0 of uiPtCompress is the LSB of Qy. When point compression is used, puiQy (Qy) should point to the EC curve parameter *b*.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results. For this function the only return value is a return code.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDSASignTwistHash

Syntax

```
rReturn=CALECDSASignTwistHash(puiMsg, eHashType,
uiMsgLen, puiGx, puiGy, puiK, puiD, puiB, puiP,
puiPMu, puiN, puiNMu, uiLen, puiSigR, puiSigS,
bDMA, uiDMAChConfig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMsg	Pointer to message value
SATHASHTYPE	eHashType	Hash algorithm
uint32_t	uiMsgLen	Message length in bytes
const uint32_t *	puiGx	Pointer to G x-coordinate parameter
const uint32_t *	puiGy	Pointer to G y-coordinate parameter
const uint32_t *	puiK	Pointer to K parameter
const uint32_t *	puiD	Pointer to D parameter
const uint32_t *	puiB	Pointer to twisted b parameter
const uint32_t *	puiZ	Pointer to twist factor.
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
const uint32_t *	puiNMu	Pointer to N modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t *	puiSigR	Pointer to signature R value
uint32_t *	puiSigS	Pointer to signature S value
SATBOOL	bDMA	DMA select flag
uint32_t	uiDMACh-Config	Channel configuration word

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.

Return Value	*Causes*
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

If the function executes successfully, CALPKTrfRes will place the result components in the locations specified by puiSigR and puiSigS. CALPK-TrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALIDATEFAIL	Point not on curve.
SATR_SIGNPARMD	D parameter not in range [1, N-1].
SATR_SIGNPARMK	K parameter not in range [1, N-1].
SATR_SIGNFAIL	Signature failure: puiSigR=0 or puiSigS=0.

Description

This function calculates an ECDSA signature on a twisted curve (see *Twisted Elliptic Curves* on page 14) based on the hash value calculated from the message, puiMsg. The hash is given by eHashType. The resulting signature is stored at the locations pointed at by puiSigR and puiSigS. The EC base point G is specified in affine coordinates by puiGx and puiGy. The *b* parameter of the curve is specified by puiB, and the curve twist is specified by puiZ. The EC field's prime modulus is specified by puiP with corresponding pre-compute value specified by puiPMu. The order of the group is specified by puiN with corresponding pre-compute value specified by puiNMu. The private key is specified by puiD. The random per-message value is specified by puiK. The word length of all operands is uiLen.

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDSAVerifyTwistHash

Syntax

```
rReturn=CALECDSAVerifyTwistHash(puiMsg,
eHashType, uiMsgLen, puiGx, puiGy, puiQx, puiQy,
puiSigR, puiSigS, puiB, puiP, puiPMu, puiN,
puiNMu, uiLen, uiPtCompress, bDMA, uiDMAChConfig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMsg	Pointer to message value
SATHASHTYPE	eHashType	Hash algorithm
uint32_t	uiMsgLen	Message length in bytes
const uint32_t *	puiGx	Pointer to G x-coordinate parameter
const uint32_t *	puiGy	Pointer to G y-coordinate parameter
const uint32_t *	puiQx	Pointer to Q x-coordinate
const uint32_t *	puiQy	Pointer to Q y-coordinate
const uint32_t *	puiSigR	Pointer to signature R value
const uint32_t *	puiSigS	Pointer to signature S value
const uint32_t *	puiB	Pointer to twisted b parameter
const uint32_t *	puiZ	Pointer to twist factor.
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
const uint32_t *	puiNMu	Pointer to N modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t	uiPtCompress	Point compression control
SATBOOL	bDMA	DMA select flag
uint32_t	uiDMACh-Config	Channel configuration word

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.

Return Value	*Causes*
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

This function does not cause CALPKTrfRes to return a value. CALPkTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_DCMPPARMX	Point decompression with X parameter not in range [0, modulus-1].
SATR_DCMPPARMB	Point decompression with b parameter greater than modulus.
SATR_DCMPPARMP	Point decompression with modulus not of the form 4n+3.
SATR_VERPARMR	SigR parameter not in [1, modulus-1].
SATR_VERPARMS	SigS parameter not in [1, modulus-1].
SATR_VALIDATEFAIL	Point not on curve.
SATR_PAF	Point at infinity result generated.
SATR_VERIFYFAIL	The signature verify failed.

Description This function verifies an ECDSA signature on a twisted curve (see *Twisted Elliptic Curves* on page 14) based on the hash value calculated from the message, puiMsg. The hash is given by eHashType. The signature to be verified is specified by puiSigR and puiSigS. The EC base point G is specified in affine coordinates by puiGx and puiGy. The *b* parameter of the curve is specified by puiB, and the curve twist is specified by puiZ. The EC field's prime modulus is specified by puiP with corresponding pre-compute value specified by puiPMu. The order N of the group is specified by puiN with corresponding pre-compute value specified by puiNMu. The public key Q is specified in affine coordinates by puiQx and puiQy. The word length of all operands is uiLen.

This function supports point compression of Q. When bit 1 of uiPtCompress is set, this indicates that Q is compressed, and bit 0 of uiPtCompress is the LSB of Qy. When point compression is used, puiQy (Qy) should point to the EC curve parameter b.

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

2.8 Functions with SCA Countermeasures

This section documents CAL functions for public key and elliptic curve cryptography that employ SCA countermeasures. Public key and elliptic curve cryptography functions not listed in this section do not employ SCA countermeasures.

The performance of public key and elliptic curve cryptography functions that employ SCA countermeasures is generally slower than those that do not employ SCA countermeasures.

Each of the functions listed in Table 2-5 is described in this section.

Table 2-5: TeraFire CAL PK and EC Cryptography Functions with SCA Countermeasures

Function	*Description*	*Page*
CALExpoCM	Performs module exponentiation using SCA countermeasures	97
CALRSACRTCM	Performs RSA private key operation with CRT using SCA countermeasures	99
CALRSACRTSignCM	Performs RSA signature operation with CRT using SCA countermeasures	101
CALRSACRTSign-HashCM	Performs RSA signature operation with CRT on a message using SCA countermeasures	103
CALDSASignCM	Calculates a DSA signature on a provided hash value using SCA countermeasures	105
CALDSASign-HashCM	Calculates a DSA signature on a message using SCA countermeasures	107
CALECMultCM	Performs elliptic curve multiplication using SCA countermeasures	110
CALECMultTwistCM	Performs twisted elliptic curve multiplication using SCA countermeasures	112
CALECDSASignCM	Calculates an ECDSA signature on a provided hash value using SCA countermeasures	114
CALECDSASign-HashCM	Calculates an ECDSA signature on a message using SCA countermeasures	116
CALECD-SASignTwistCM	Calculates an ECDSA signature on a twisted elliptic curve using SCA countermeasures	119

CALExpoCM

Syntax

```
rReturn=CALExpoCM(puiBase, puiExpo, puiMod,
puiMu, uiExpLen, uiModLen, puiResult)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value.

Type	*Identifier*	*Description*
const uint32_t *	puiBase	Pointer to base
const uint32_t *	puiExpo	Pointer to exponent (SAT_NULL)
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL)
const uint32_t *	puiMu	Pointer to pre-compute (SAT_NULL)
uint32_t	uiExpLen	Exponent length in words
uint32_t	uiModLen	Modulus length in words
uint32_t *	puiResult	Pointer to modular exponentiation result

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus length is less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADLEN	The exponent length is zero or greater than the maximum size supported by the implementation.

CALPKTrfRes will place the result from this operation in the location specified by puiResult, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description This function performs modular exponentiation with SCA countermeasures on the base specified by puiBase raised to the exponent specified by puiExpo performed over the *prime* modulus specified by puiMod with a pre-compute specified by puiMu. The exponent word length is uiExpLen and the modulus word length is uiModLen. The modulus

length must be at least two words, and the exponent length must be non-zero. The length of the pre-compute is uiModLen+1. The result is stored at the location pointed at by puiResult and is uiModLen words.

For performance reasons, the exponent length (uiExpLen) should be set to the actual length of the exponent in words (*i.e.*, the least value such that the most significant word is non-zero). For example, if the exponent is $2^{16}+1$, then uiExpLen should be set to 1; setting it to a larger value will produce the same result but with unnecessary additional processing.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

This function requires that the modulus, puiMod, is prime. If the modulus is not prime, then the function may not produce the correct result.

CALRSACRTCM

Syntax

```
rReturn=CALRSACRTCM(puiCipher, puiQInv, puiDP,
puiDQ, puiE, puiP, puiQ, puiN, puiNMu,  uiLen,
uiELen, puiPlain)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiCipher	Type of RSA encoding
const uint32_t *	puiQInv	Pointer to Q inverse parameter
const uint32_t *	puiDP	Pointer to DP parameter
const uint32_t *	puiDQ	Pointer to DQ parameter
const uint32_t *	puiE	Pointer to public key exponent
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL)
const uint32_t *	puiN	Pointer to public modulus N
const uint32_t *	puiNMu	Pointer to N modulus pre-compute
uint32_t	uiLen	Modulus and operand lengths in words
uint32_t	uiELen	Public exponent length in words
uint32_t *	puiPlain	Pointer to plaintext result

This function has different parameters than the non-SCA countermeasure version of this function; see *CALRSACRT* on page 38.

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADLEN	The public exponent length is zero or greater than the maximum size supported by the implementation.

CALPKTrfRes will place the result from this operation in the location specified by puiPlain, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description This function performs an RSA private key decryption operation with CRT and SCA countermeasures on the data buffer pointed at by puiCipher and stores the result at the location pointed at by puiPlain. The RSA CRT parameters are specified by puiQInv, puiDP, and puiDQ. The private prime moduli are specified by puiP and puiQ, while the public exponent is specified by puiE, and the public modulus by puiN, with corresponding pre-compute value specified by puiNMu. The word lengths of puiQInv, puiDP, puiDQ, puiP, and puiQ are uiLen, and uiLen must be at least two. The word length of puiE is uiELen, and must be at least 1. The word length of puiCipher, puiPlain, and puiN are twice uiLen (2×uiLen), and the word length of puiNMu is 2×uiLen+1.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALRSACRTSignCM

Syntax

```
rReturn=CALRSACRTSignCM(eRSAEncod, eHashType,
puiHash, puiE, puiELen, puiQInv, puiDP, puiDQ,
puiP, puiQ, puiN, puiNMu, uiLen, puiSig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATENCODING-TYPE	eRSAEncod	Type of RSA encoding. See Table 4-2 for valid values.
SATHASHTYPE	eHashType	Hash type. See Table 4-3 for defined values.
const uint32_t *	puiHash	Pointer to hash value.
const uint32_t *	puiE	Pointer to public key exponent.
uint32_t	uiELen	Public exponent length in words
const uint32_t *	puiQInv	Pointer to Q inverse parameter.
const uint32_t *	puiDP	Pointer to DP parameter.
const uint32_t *	puiDQ	Pointer to DQ parameter.
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL).
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL).
const uint32_t *	puiN	Pointer to public modulus N.
const uint32_t *	puiNMu	Pointer to N modulus pre-compute
uint32_t	uiLen	Modulus and operand lengths in words.
uint32_t *	puiSig	Pointer to signature result.

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADLEN	The public exponent length is zero or greater than the maximum size supported by the implementation.
SATR_BADRSAENC	Invalid RSA encoding type.

Return Value	*Causes*
SATR_BADHASHTYPE	Invalid/unsupported hash type.

If the function executes successfully, CALPKTrfRes will place the signature in the location specified by puiSig. SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_BADPARAM	For PKCS encoding, padding length is less than 3 words.

Description

This function computes an RSA-CRT signature with SCA countermeasures using a user-supplied hash pointed at by puiHash using the RSA encoding type provided by eRsaEncod. The hash type used is given by eHashType, and the size of the vector pointed at by puiHash is automatically determined by the hash type (eHashType) parameter. The RSA-CRT parameters are specified by puiQInv, puiDP, and puiDQ. The prime moduli are specified by puiP and puiQ. The public modulus is specified by puiN with corresponding pre-compute value specified by puiNMu, and the public exponent is specified by puiE with word length uiELen. The result is stored at the location pointed at by puiSig. The word lengths of puiQInv, puiDP, puiDQ, puiP, and puiQ are uiLen. The word length of puiN is 2×uiLen+1, and the word length of puiNMu is 2×uiLen+2. The word length of puiSig is twice uiLen (2×uiLen).

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALRSACRTSignHashCM

Syntax

```
rReturn=CALRSACRTSignHashCM(eRsaEncod, eHashType,
puiMsg, puiE, uiELen, puiQInv, puiDP, puiDQ,
puiP, puiQ, puiN, puiNMu, uiMsgLen, uiModLen,
puiSig, bDMA, uiDMAChConfig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATRSAENC-TYPE	eRsaEncod	Type of RSA encoding.
SATHASHTYPE	eHashType	Hash type
const uint32_t *	puiMsg	Pointer to message value
const uint32_t *	puiE	Pointer to public key exponent.
uint32_t	uiELen	Public exponent length in words
const uint32_t *	puiQInv	Pointer to Q inverse parameter
const uint32_t *	puiDP	Pointer to DP parameter
const uint32_t *	puiDQ	Pointer to DQ parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL)
const uint32_t *	puiN	Pointer to public modulus N.
const uint32_t *	puiNMu	Pointer to N modulus pre-compute
uint32_t	uiMsgLen	Message length in bytes
uint32_t	uiModLen	Modulus length in words
uint32_t *	puiSig	Pointer to signature result
SATBOOL	bDMA	DMA select flag
uint32_t	uiDMAChCon-fig	DMA channel configuration word

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

Return Value	*Causes*
SATR_BADLEN	The public exponent length is zero or greater than the maximum size supported by the implementation.
SATR_BADRSAENC	Invalid RSA encoding type.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

If the function executes successfully, CALPKTrfRes will place the signature in the location specified by puiSig. SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_BADPARAM	For PKCS encoding, padding length is less than 3 words.

Description

This function computes an RSA-CRT signature with SCA countermeasures on the message hash produced from puiMsg using the hash type given by eHashType and the RSA encoding type provided by eRsaEncod. The result is stored at the location pointed at by puiSig. The RSA-CRT parameters are specified by puiQInv, puiDP, and puiDQ. The prime moduli are specified by puiP and puiQ. The public modulus is specified by puiN with corresponding pre-compute value specified by puiNMu, and the public exponent is specified by puiE with word length uiELen. The result is stored at the location pointed at by puiSig. The word lengths of puiQInv, puiDP, puiDQ, puiP, and puiQ are uiLen. The word length of puiN is 2×uiLen+1, and the word length of puiNMu is 2×uiLen+2. The word length of puiSig is twice uiLen (2×uiLen).

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALDSASignCM

Syntax

```
rReturn=CALDSASignCM(puiHash, puiG, puiK, puiX,
puiP, puiPMu, puiQ, puiQMu, uiN, uiL, puiSigR,
puiSigS)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiHash	Pointer to hash value
const uint32_t *	puiG	Pointer to G parameter
const uint32_t *	puiK	Pointer to K parameter
const uint32_t *	puiX	Pointer to X parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL)
const uint32_t *	puiQMu	Pointer to Q modulus pre-compute (SAT_NULL)
uint32_t	uiN	Q length parameter
uint32_t	uiL	P length parameter
uint32_t *	puiSigR	Pointer to signature R value
uint32_t*	puiSigS	Pointer to signature S value

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADLEN	uiN>uiL or uiN==uiL and P<=Q.

If the function executes successfully, CALPKTrfRes will place the signature in the locations specified by puiSigR and puiSigS. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_SIGNPARMK	K parameter must be in the range [1,Q-1].
SATR_SIGNFAIL	Signature fails if computed signature R value is 0 or signature S value is 0.

Description

This function calculates a DSA signature with SCA countermeasures given the hash value of a message specified by puiHash. The resulting signature is stored at the locations pointed at by puiSigR and puiSigS. The DSA primes are specified by puiP and puiQ with corresponding pre-compute values specified by puiPMu and puiQMu. The DSA value G is specified by puiG. The private key is specified by puiX. The random per-message value is specified by puiK and must be in the range [1, Q-1]. The word length of the Q, X, and K, paramters is uiN and the word length of the P and G parameters is uiL. The size of the hash value must be greater than or equal to uiN, and the hash value in the uiN words used by this function must be non-zero. The value of uiL must be greater than or equal to uiN.

In the event of a failure due to the K parameter being out of range (SATR_SIGNPARMK) or a signature failure (SATR_SIGNFAIL), a new K parameter should be selected and the computation re-attempted.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALDSASignHashCM

Syntax

```
rReturn=CALDSASignHashCM(puiMsg, eHashType,
uiMsgLen, puiG, puiK, puiX, puiP, puiPMu, puiQ,
puiQMu, uiN, uiL, puiSigR, puiSigS, bDMA,
uiDMAChConfig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMsg	Pointer to message input
SATHASHTYPE	eHashType	Hash algorithm
uint32_t	uiMsgLen	Message length, in bytes
const uint32_t *	puiG	Pointer to G parameter
const uint32_t *	puiK	Pointer to K parameter
const uint32_t *	puiX	Pointer to X parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiQ	Pointer to Q modulus (SAT_NULL)
const uint32_t *	puiQMu	Pointer to Q modulus pre-compute (SAT_NULL)
uint32_t	uiN	Q length parameter
uint32_t	uiL	P length parameter
uint32_t *	puiSigR	Pointer to signature R value
uint32_t*	puiSigS	Pointer to signature S value
SATBOOL	bDMA	DMA select flag
uint32_t	uiDMAChConfig	Channel configuration word

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

Return Value	*Causes*
SATR_BADLEN	uiN>uiL or uiN==uiL and P<=Q.
SATR_BADHASHTYPE	Invalid/unsupported hash type, including hash size less than N parameter.

If the function executes successfully, CALPKTrfRes will place the signature in the locations specified by puiSigR and puiSigS. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_SIGNPARMK	K parameter must be in the range [1,Q-1].
SATR_SIGNFAIL	Signature fails if computed signature R value is 0 or signature S value is 0.

Description

This function calculates a DSA signature with SCA countermeasures for a message at puiMsg using the hash type eHashType. The resulting signature is stored at the locations pointed at by puiSigR and puiSigS. The DSA primes are specified by puiP and puiQ with corresponding precompute values specified by puiPMu and puiQMu. The DSA value G is specified by puiG. The private key is specified by puiX. The random per-message value is specified by puiK and must be in the range [1, Q-1]. The word length of the Q, X, and K, paramters is uiN and the word length of the P and G parameters is uiL. The size of the hash value must be greater than or equal to uiN. The value of uiL must be greater than or equal to uiN.

In the event of a failure due to the K parameter being out of range (SATR_SIGNPARMK) or a signature failure (SATR_SIGNFAIL), a new K parameter should be selected and the computation re-attempted.

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

CALECMultCM

Syntax

```
rReturn=CALECMultCM(puiMul, puiPx, puiPy, puiB,
puiMod, puiMu, puiN, uiLen, uiPtCompress, puiRx,
puiRy)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMul	Pointer to multiplier (SAT_NULL)
const uint32_t *	puiPx	Pointer to base point x-coordinate
const uint32_t *	puiPy	Pointer to base point y-coordinate
const uint32_t *	puiB	Pointer to b curve parameter
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL)
const uint32_t *	puiMu	Pointer to modulus pre-compute (SAT_NULL)
const uint32_t*	puiN	Pointer to curve order N modulus
uint32_t	uiLen	Modulus length in words
uint32_t	uiPtCompress	Point compression control
uint32_t *	puiRx	Pointer to product x-coordinate
uint32_t *	puiRy	Pointer to product y-coordinate

This function has different parameters than the non-SCA countermeasure version of this function; see *CALECMult* on page 61.

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

If the function executes successfully, CALPKTrfRes will place the product components in the locations specified by puiRx and puiRy. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_DCMPPARMX	Point decompression with X parameter not in range [0, modulus-1].
SATR_DCMPPARMB	Point decompression with b parameter greater than modulus.
SATR_DCMPPARMP	Point decompression with modulus not of the form 4n+3.
SATR_VALIDATEFAIL	Point not on curve.
SATR_PAF	Point at infinity result generated.

Description

This function performs an elliptic curve point multiply with SCA countermeasures over a curve $y^2 = x^3 - 3x + b$. The multiplier is specified by puiMul. The affine point to be multiplied is specified by puiPx and puiPy. The field's prime modulus is specified by puiMod, with corresponding pre-compute value puiMu. The *b* curve parameter is specified by puiB, and the order of the curve is specified by puiN. The word length of all operands except puiMu is uiLen; the word length of puiMu is uiLen+1. The resulting point product will be stored at the location pointed at by puiRx and puiRy.

This function supports point compression of point P. When bit 1 of uiPtCompress is set, this indicates that P is compressed, and bit 0 of uiPtCompress is the LSB of Py. When point compression is used, puiPy (Py) should point to the EC curve parameter *b*.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXPF5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECMultTwistCM

Syntax

```
rReturn=CALECMultTwistCM(puiMul, puiPx, puiPy,
puiB, puiZ, puiMod, puiMu, puiN, uiLen, puiRx,
puiRy)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMul	Pointer to multiplier (SAT_NULL)
const uint32_t *	puiPx	Pointer to base point x-coordinate
const uint32_t *	puiPy	Pointer to base point y-coordinate
const uint32_t *	puiB	Pointer to twisted b curve parameter
const uint32_t *	puiZ	Pointer to twist factor
const uint32_t *	puiMod	Pointer to modulus (SAT_NULL)
const uint32_t *	puiMu	Pointer to modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t *	puiRx	Pointer to product x-coordinate
uint32_t *	puiRy	Pointer to product y-coordinate

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

If the function executes successfully, CALPKTrfRes will place the product components in the locations specified by puiRx and puiRy. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].

Return Value	*Causes*
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALIDATEFAIL	Point not on curve.
SATR_PAF	Point at infinity result generated.

Description

This function performs an elliptic curve point multiply over a twisted curve, see *Twisted Elliptic Curves* on page 14, with SCA countermeasures. The multiplier is specified by puiMul. The affine point to be multiplied is specified by puiPx and puiPy. The field's prime modulus is specified by puiMod, with corresponding pre-compute value puiMu. The *b* curve parameter is specified by puiB, and the curve twist factor is specified by puiZ. The order of the group is specified by puiN. The word length of all operands except puiMu is uiLen; the word length of puiMu is uiLen+1. The resulting point product will be stored at the location pointed at by puiRx and puiRy.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDSASignCM

Syntax

```
rReturn=CALECDSASignCM(puiHash, puiGx, puiGy,
puiK, puiD, puiB, puiP, puiPMu, puiN, puiNMu,
uiLen, puiSigR, puiSigS)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiHash	Pointer to hash value
const uint32_t *	puiGx	Pointer to G x-coordinate parameter
const uint32_t *	puiGy	Pointer to G y-coordinate parameter
const uint32_t *	puiK	Pointer to K parameter
const uint32_t *	puiD	Pointer to D parameter
const uint32_t *	puiB	Pointer to b curve parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
const uint32_t *	puiNMu	Pointer to Q modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t *	puiSigR	Pointer to signature R value
uint32_t *	puiSigS	Pointer to signature S value

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

If the function executes successfully, CALPKTrfRes will place the result components in the locations specified by puiSigR and puiSigS. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALIDATEFAIL	Point not on curve.
SATR_SIGNPARMD	D parameter not in range [1, N-1].
SATR_SIGNPARMK	K parameter not in range [1, N-1].
SATR_SIGNFAIL	Signature failure: puiSigR=0 or puiSigS=0.

Description

This function calculates an ECDSA signature with SCA countermeasures given the hash value of a message specified by puiHash. The resulting signature is stored at the locations pointed at by puiSigR and puiSigS. The EC base point G is specified in affine coordinates by puiGx and puiGy. The b parameter of the curve $y^2 = x^3 - 3x + b$ is specified by puiB. The EC field's prime modulus is specified by puiP with corresponding pre-compute value specified by puiPMu. The order of the group is specified by puiN with corresponding pre-compute value specified by puiNMu. The private key is specified by puiD. The random per-message value is specified by puiK. The word length of all operands, except puiPMu and puiNMu is uiLen; the word length of puiPMu and puiNMu is uiLen+1.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDSASignHashCM

Syntax

```
rReturn=CALECDSASignHash(puiMsg, eHashType,
uiMsgLen, puiGx, puiGy, puiK, puiD, puiB, puiP,
puiPMu, puiN, puiNMu, uiLen, puiSigR, puiSigS,
bDMA, uiDMAChConfig)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiMsg	Pointer to message value
SATHASHTYPE	eHashType	Hash algorithm
uint32_t	uiMsgLen	Message length in bytes
const uint32_t *	puiGx	Pointer to G x-coordinate parameter
const uint32_t *	puiGy	Pointer to G y-coordinate parameter
const uint32_t *	puiK	Pointer to K parameter
const uint32_t *	puiD	Pointer to D parameter
const uint32_t *	puiB	Pointer to B parameter
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
const uint32_t *	puiNMu	Pointer to N modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t *	puiSigR	Pointer to signature R value
uint32_t *	puiSigS	Pointer to signature S value
SATBOOL	bDMA	DMA select flag
uint32_t	uiDMACh-Config	Channel configuration word

Returns Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.

Return Value	*Causes*
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.
SATR_BADHASHTYPE	Invalid/unsupported hash type.

If the function executes successfully, CALPKTrfRes will place the result components in the locations specified by puiSigR and puiSigS. CALPK-TrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALIDATEFAIL	Point not on curve.
SATR_SIGNPARMD	D parameter not in range [1, N-1].
SATR_SIGNPARMK	K parameter not in range [1, N-1].
SATR_SIGNFAIL	Signature failure: puiSigR=0 or puiSigS=0.

Description

This function calculates an ECDSA signature with SCA countermeasures based on the hash value calculated from the message, puiMsg. The hash is given by eHashType. The resulting signature is stored at the locations pointed at by puiSigR and puiSigS. The EC base point G is specified in affine coordinates by puiGx and puiGy. The b parameter of the curve $y^2 = x^3 - 3x + b$ is specified by puiB. The EC field's prime modulus is specified by puiP with corresponding pre-compute value specified by puiPMu. The order of the group is specified by puiN with corresponding pre-compute value specified by puiNMu. The private key is specified by puiD. The random per-message value is specified by puiK. The word length of all operands is uiLen.

DMA may be used for message input. This is accomplished by setting bDMA to SAT_TRUE, placing the channel configuration word in uiDMAChconfig, and providing the message location with puiMsg. See *CAL DMA Configuration* on page 121 for more information on DMA configuration.

This implementation only supports FIPS approved hash types as input, see Table 4-3. Non-FIPS hash types are not supported.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

CALECDSASignTwistCM

Syntax

```
rReturn=CALECDSASignTwistCM(puiHash, puiGx,
puiGy, puiK, puiD, puiB, puiZ, puiP, puiPMu,
puiN, puiNMu, uiLen, puiSigR, puiSigS)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
const uint32_t *	puiHash	Pointer to hash value
const uint32_t *	puiGx	Pointer to G x-coordinate parameter
const uint32_t *	puiGy	Pointer to G y-coordinate parameter
const uint32_t *	puiK	Pointer to K parameter
const uint32_t *	puiD	Pointer to D parameter
const uint32_t *	puiB	Pointer to twisted b curve parameter
const uint32_t*	puiZ	Pointer to twist factor
const uint32_t *	puiP	Pointer to P modulus (SAT_NULL)
const uint32_t *	puiPMu	Pointer to P modulus pre-compute (SAT_NULL)
const uint32_t *	puiN	Pointer to N modulus (SAT_NULL)
const uint32_t *	puiNMu	Pointer to N modulus pre-compute (SAT_NULL)
uint32_t	uiLen	Modulus length in words
uint32_t *	puiSigR	Pointer to signature R value
uint32_t *	puiSigS	Pointer to signature S value

Returns

Initiation: SATR rReturn

Return Value	*Causes*
SATR_SUCCESS	Successful initiation.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Modulus/operand lengths are less than 2 words or greater than the maximum size supported by the implementation.

If the function executes successfully, CALPKTrfRes will place the result components in the locations specified by puiSigR and puiSigS. CALPKTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALIDATEFAIL	Point not on curve.
SATR_SIGNPARMD	D parameter not in range [1, N-1].
SATR_SIGNPARMK	K parameter not in range [1, N-1].
SATR_SIGNFAIL	Signature failure: puiSigR=0 or puiSigS=0.

Description

This function calculates an ECDSA signature over a twisted curve, see *Twisted Elliptic Curves* on page 14, with SCA countermeasures, given the hash value of a message specified by puiHash. The resulting signature is stored at the locations pointed at by puiSigR and puiSigS. The EC base point G is specified in affine coordinates by puiGx and puiGy. The *b* parameter of the curve is specified by puiB, and the curve twist factor is specified by puiZ. The EC field's prime modulus is specified by puiP with corresponding pre-compute value specified by puiPMu. The order of the group is specified by puiN with corresponding pre-compute value specified by puiNMu. The private key is specified by puiD. The random per-message value is specified by puiK. The word length of all operands, except puiPMu and puiNMu is uiLen; the word length of puiPMu and puiNMu is uiLen+1.

This function initiates the operation and then returns. Use CALPKTrfRes to transfer the results.

For information on the use of ROM'd NIST P-curve values with the EXP-F5200B processor, see Section 2.2.3, 'NIST P-Curve ROM,' on page 18.

2.9 CAL DMA Configuration

Certain CAL functions support the use of DMA for data transfer, if it is connected in a specific hardware implementation; PolarFire FPGA users may choose whether to connect the AHB bus master port on the EXP-F5200B core. This section discusses DMA configuration for these operations.

2.9.1 AHB Master DMA

For implementations with AHB-hosted DMA a user supplied channel configuration value is passed as a parameter to DMA-capable functions. This channel configuration may be configured by bitwise ORing the values listed in Table 2-6, provided that only one of each type of configuration parameters is selected: beat size (BSIZE), byte swapping (ESWP), protection (PROT), and incrementing/non-incrementing address.

Table 2-6: CAL AHB DMA Configuration Parameters

Value	*Description*
X52CCR_DEFAULT	Default value, equivalent to auto-size, no byte swap, user priviledged transfer, and incrementing address.
X52CCR_BSIZEAUTO	Auto-size transfer based on address alignment.
X52CCR_BSIZEBYTE	Force one byte per beat transfer.
X52CCR_BSIZEHWORD	Force one half-word per beat transfer.
X52CCR_BSIZEWORD	Force one word per beat transfer.
X52CCR_ESWPNONE	No byte swapping.
X52CCR_ESWPWORD	Swap bytes within words.
X52CCR_PROTUSER	Set hprot[1]=0 for user priviledged transfer.
X52CCR_PROTPRIV	Set hprot[1]=1 for priviledged transfer.
X52CCR_INCADDR	Use incrementing address for transfers.
X52CCR_NOINCADDR	Use non-incrementing address - generally used when reading/writing a hardware port.

CHAPTER 3 CAL Symmetric Cryptography Functions

The TeraFire CAL functions for symmetric cryptography are documented in this chapter. These functions include symmetric (secret key) ciphers, hashes, message authentication codes, and random number generation.

Most of the work associated with any particular CAL function is actually performed on the EXP-F5200B cryptography microprocessor. The CAL functions load data onto the EXP-F5200B microprocessor, initiate execution of the operation, and retrieve results when the operation is complete. Even complex operations execute the actual data processing in their entirety on the EXP-F5200B microprocessor.

3.1 SCA Countermeasures

3.1.1 Leakage Reduction Countermeasures

CAL symmetric cryptography functions may operate with hardware implementations that perform leakage reduction SCA countermeasures. Implementations with leakage reduction SCA countermeasures may be employed in any application calling for cryptographic processing, without changing the cryptographic protocol. In hardware implementations with leakage reduction countermeasures there is not a version of the implementation without countermeasures unless explicitly noted.

In the EXP-F5200B configuration present in Microsemi PolarFire FPGAs, leakage reduction countermeasures are present for AES (including GCM), and SHA, and apply to all algorithms dependent upon these core operations.

3.1.2 Protocol Countermeasures

Some CAL symmetric cryptography functions implement *protocol countermeasures* against SCAs. The most common protocol countermeasure is key rolling, which involves changing the key between individual encrypt or decrypt operations. While these are powerful countermeasures, protocol countermeasures may not be suitable for standards-based applications that do not include explicit support for such countermeasures.

3.2 Symmetric Cryptography Library Organization

Symmetric cryptography functions are broken into groups by function class. These are listed below.

- *General Functions* on page 126.
- *Encryption Functions* on page 128.
- *Combined Encryption-Authentication Functions* on page 139.
- *Hashes* on page 157.
- *Multiple Call Hash Functions* on page 160.
- *Hash Functions with Context Switching* on page 164.
- *Message Authentication Codes* on page 168.
- *Multiple Call MAC Functions* on page 172.
- *MAC Functions with Context Switching* on page 176.
- *Random Number Generation* on page 180.
- *Key Wrap and Unwrap* on page 199.
- *Context Management Functions* on page 204.
- *Key Derivation Functions* on page 209.

Function descriptions include syntax, parameters, return values, and a detailed description of the operation of the function.

3.3 General Functions

Each of the functions listed in Table 3-1 is described in this section.

Table 3-1: TeraFire CAL-SYM General Functions

Function	*Description*	*Page*
CALSymTrfRes	Transfers the results once processing is finished	127

CALSymTrfRes

Syntax `rReturn=CALSymTrfRes(bBlock)`

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATBOOL	bBlock	Blocking or non-blocking result transfer.

Returns Return values produced by this function depend upon the preceding initiation function (*e.g.*, CALSymEncrypt). This section documents return values that arise directly from this function, not as a result of the initiation function.

SATR rReturn values resulting from this function. For other return values, refer to the initiating function.

Return Value	*Causes*
SATR_BUSY	For non-blocking operation this return indicates that the computation is still being performed and that a result is not yet ready.
SATR_PARITYFLUSH	This indicates that the hardware performing the computation entered an alarm state, typically as a result of an uncorrectable memory error. The hardware was successfully re-initialized; however, the operation was not completed as a result.
SATR_HFAULT	This indicates that the hardware performing the computation entered an alarm state and CAL was unable to re-initialize the hardware.
SATR_NOPEND	There is no pending or completed operation awaiting transfer of results.
SATR_FNP	Function referenced by the initiating function is not populated.

Description All the CAL functions initiate an operation and then return. This function is used to transfer the results of an operation once it has completed. The function can block until the operation has completed by setting bBlock to SAT_TRUE. If bBlock is set to SAT_FALSE, then the function will return SAT_BUSY if the processor is still busy; otherwise, it will transfer the results.

3.4 Encryption Functions

Symmetric key cryptography uses a single private key to both encrypt and decrypt data. Symmetric cryptography uses various modes of operation to perform encryption and decryption. The CAL library has support for multiple modes of operation (e.g., ECB, CBC, CTR), and all of these modes except for ECB require an initialization vector (IV), which is used in combination with the input data to perform encryption and decryption. The CAL symmetric key functions load IVs and store the resulting IVs, which allows for easy context switching when using CBC, CFB, OFB and CTR modes.

Each of the functions listed in Table 3-2 is described in this section.

Table 3-2: TeraFire CAL Symmetric Encryption Functions

Function	*Description*	*Page*
CALSymEncrypt	Symmetric encryption	131
CALSymDecrypt	Symmetric decryption	133
CALSymEncryptDMA	Symmetric encryption with DMA	135
CALSymDecryptDMA	Symmetric decryption with DMA	137

3.4.1 Symmetric Encryption/Decryption Algorithms

Symmetric encryption/decryption algorithms defined for use with CAL are tabulated in Table 3-3. Whether a particular algorithm is supported in a particular implementation depends on the specific implementation. Some algorithms, such as AES, have multiple modes of operation. For these algorithms the modes are listed in Table 3-4. Like algorithms, whether a particular mode is supported in an implementation depends on the specific implementation.

Table 3-3: TeraFire CAL Encryption/Decryption Algorithms

Identifier[a]	*Algorithm*
SATSYMTYPE_AES128	AES with 128-bit key
SATSYMTYPE_AES192	AES with 192-bit key
SATSYMTYPE_AES256	AES with 256-bit key
SATSYMTYPE_AESKS128	AES with 128-bit split key

Table 3-3: TeraFire CAL Encryption/Decryption Algorithms (Continued)

Identifier[a]	*Algorithm*
SATSYMTYPE_AESKS192	AES with 192-bit split key
SATSYMTYPE_AESKS256	AES with 256-bit split key

a. Identifier is SATSYMTYPE type.

Table 3-4: TeraFire CAL Encryption/Decryption Modes

Identifier[a]	*Algorithm*
SATSYMMODE_ECB	Electronic code book
SATSYMMODE_CBC	Cipher block chaining
SATSYMMODE_CFB	Cipher feedback
SATSYMMODE_OFB	Output feedback
SATSYMMODE_CTR	Counter mode
SATSYMMODE_GCM	Galois counter mode
SATSYMMODE_GHASH	Galois hash mode

a. Identifier is SATSYMMODE type.

3.4.2 Split Keys

Certain implementations of functions employing symmetric keys support handling of keys in split form. Refer to Table 3-3 on page 128 for listing of symmetric algorithms with split key formats, and refer to product configuration information to determine whether your implementation supports split keys.

Split keys comprise two fields, which, when XOR'd together reconstitutes the subject key. In typical use cases, a split key is created by generating random data of the same size as the key and XOR'ing the key with the random data, which is illustrated in the example below.

Example 3-1 Key Split for 128-bit AES

```
SATUINT32_t uiKey[4];      /* Original 128-bit key. */
SATUINT32_t uiSplitKey[8]; /* Split key. */
SATUINT32_t uiN;

/* Generate 128-bits of random data. */
CALDRBGGenerate(SAT_NULL, 0, SAT_FALSE, uiSplitKey, 1);

/* Split the key. */
for (uiN=0; uiN<4; ++uiN) {
  uiSplitKey[uiN+4]=uiSplitKey[uiN] ^ uiKey[uiN];
```

```
}

/* Use the split key. Note use of split key algorithm. */
CALSymEncrypt(SATSYMTYPE_AESKS128, uiSplitKey,
  SATSYMMODE_ECB, SAT_NULL, SAT_FALSE, pPlainText,
  pCipherText, uiLen);
```

Note that in typical use cases, the two portions of the split keys are stored separately as neither reveals the key without the other part of the split; however, in order for the functions that support the use of split keys to operate the split portions of the keys must be stored sequentially as illustrated in the example.

CALSymEncrypt

Syntax

```
rReturn=CALSymEncrypt(eSymType, puiKey, eMode,
pIV, bLoadIV, pSrc, pDest, uiLen)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key (SAT_NULL)
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
SATBOOL	bLoadIV	IV load control
const void *	pSrc	Pointer to the plaintext data buffer
void *	pDest	Pointer to the ciphertext data buffer
SATUINT32_t	uiLen	Length of plaintext in bytes to encrypt. Must be a non-zero multiple of the block size of the cipher.

Returns

SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Plaintext length is not a non-zero multiple of the cipher block size.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description

This function encrypts the data buffer pointed at by pSrc of specified byte length, uiLen, and stores the result at the location pointed at by

pDest. The symmetric encryption method is given by eSymType, and the mode of encryption is given by eMode. The data will be encrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded, and the previously loaded key will be used.

For algorithm/mode combinations that use an initialization vector, the IV is pointed at by pIV. If bLoadIV is SAT_TRUE, the IV will be loaded. Otherwise, the previous result's IV will be used. The resulting IV after encryption is stored at the location pointed at by pIV when pIV is not SAT_NULL and the mode is CBC, CFB, OFB or CTR.

Since symmetric encryption is performed in blocks, the byte length, uiLen, of the encryption must be a non-zero value divisible by the block size of the given symmetric method, eSymType.

CALSymDecrypt

Syntax

```
rReturn=CALSymDecrypt(eSymType, puiKey, eMode,
pIV, bLoadIV, pSrc, pDest, uiLen)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key (SAT_NULL)
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
SATBOOL	bLoadIV	IV load control
const void *	pSrc	Pointer to the ciphertext data buffer
void *	pDest	Pointer to the plaintext data buffer
SATUINT32_t	uiLen	Length of ciphertext in bytes to decrypt. Must be a non-zero multiple of the block size of the cipher.

Returns

SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Ciphertext length is not a non-zero multiple of the cipher block size.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description

This function decrypts the data buffer pointed at by pSrc of specified byte length, uiLen, and stores the result at the location pointed at by

pDest. The symmetric decryption method is given by eSymType, and the mode of decryption is given by eMode. The data will be decrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded and the previously loaded key will be used.

The initialization vector is pointed at by pIV. If bLoadIV is SAT_TRUE, the IV will be loaded. Otherwise, the previous result's IV will be used. The resulting IV after decryption is stored at the location pointed at by pIV when pIV is not SAT_NULL and the mode is CBC, CFB, OFB or CTR.

Since symmetric encryption is performed in blocks, the byte length, uiLen, of the encryption must be a non-zero value divisible by the block size of the given symmetric method, eSymType.

CALSymEncryptDMA

Syntax

```
rReturn=CALSymEncryptDMA(eSymType, puiKey, eMode,
pIV, bLoadIV, pSrc, pDest, uiLen, uiDMAChConfig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key (SAT_NULL)
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
SATBOOL	bLoadIV	IV load control
const void *	pSrc	Pointer to the plaintext data buffer
void *	pDest	Pointer to the ciphertext data buffer
SATUINT32_t	uiLen	Length of plaintext in bytes to encrypt. Must be a non-zero multiple of the block size of the cipher.
SATUINT32_t	uiDMAChConfig	DMA channel configuration

Returns

SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Plaintext length is not a non-zero multiple of the cipher block size.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description This function acts as CALSymEncrypt, but uses DMA to transfer plaintext and ciphertext. The DMA channel configuration is provided by uiDMAChConfig; refer to *CAL DMA Configuration* on page 121 for additional information on the channel configuration.

This function encrypts the data buffer pointed at by pSrc of specified byte length, uiLen, and stores the result at the location pointed at by pDest. The symmetric encryption method is given by eSymType, and the mode of encryption is given by eMode. The data will be encrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded, and the previously loaded key will be used.

For algorithm/mode combinations that use an initialization vector, the IV is pointed at by pIV. If bLoadIV is SAT_TRUE, the IV will be loaded. Otherwise, the previous result's IV will be used. The resulting IV after encryption is stored at the location pointed at by pIV when pIV is not SAT_NULL and the mode is CBC, CFB, OFB or CTR.

Since symmetric encryption is performed in blocks, the byte length, uiLen, of the encryption must be a non-zero value divisible by the block size of the given symmetric method, eSymType.

CALSymDecryptDMA

Syntax

```
rReturn=CALSymDecryptDMA(eSymType, puiKey, eMode,
pIV, bLoadIV, pSrc, pDest, uiLen, uiDMAChConfig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key (SAT_NULL)
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
SATBOOL	bLoadIV	IV load control
const void *	pSrc	Pointer to the ciphertext data buffer
void *	pDest	Pointer to the plaintext data buffer
SATUINT32_t	uiLen	Length of plaintext in bytes to decrypt
SATUINT32_t	uiDMAChConfig	DMA channel configuration

Returns

SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Ciphertext length is not a non-zero multiple of the cipher block size.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description

This function acts as CALSymDecrypt, but uses DMA to transfer plaintext and ciphertext. The DMA channel configuration is provided by uiD-

MAChConfig; refer to *CAL DMA Configuration* on page 121 for additional information on the channel configuration.

This function decrypts the data buffer pointed at by pSrc of specified byte length, uiLen, and stores the result at the location pointed at by pDest. The symmetric decryption method is given by eSymType, and the mode of decryption is given by eMode. The data will be decrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded and the previously loaded key will be used.

The initialization vector is pointed at by pIV. If bLoadIV is SAT_TRUE, the IV will be loaded. Otherwise, the previous result's IV will be used. The resulting IV after decryption is stored at the location pointed at by pIV when pIV is not SAT_NULL and the mode is CBC, CFB, OFB or CTR.

Since symmetric encryption is performed in blocks, the byte length, uiLen, of the encryption must be a non-zero value divisible by the block size of the given symmetric method, eSymType.

3.5 Combined Encryption-Authentication Functions

Each of the functions listed in Table 3-5 is described in this section.

Table 3-5: TeraFire CAL Combined Encryption/Authentication Functions

Function	*Description*	*Page*
CALSymEncAuth	Combined symmetric encryption and authentication	140
CALSymDecVerify	Combined symmetric decryption and verification	142
CALSymEncAuthDMA	Combined symmetric encryption and authentication with DMA	144
CALSymDecVerifyDMA	Combined symmetric decryption and verification with DMA	146

CALSymEncAuth

Syntax

```
rReturn=CALSymEncAuth(eSymType, puiKey, eMode,
pIV, pSrc, pDest, uiEncLen, pAuth, uiAuthLen,
pMAC)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key (SAT_NULL)
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
const void *	pSrc	Pointer to the plaintext/authenticate data buffer
void *	pDest	Pointer to the ciphertext data buffer
SATUINT32_t	uiEncLen	Length of plaintext in bytes to encrypt/authenticate
void *	pAuth	Pointer to the authenticate-only data buffer
SATUINT32_t	uiAuthLen	Length of authenticate-only data in bytes
void *	pMAC	Pointer to the message authentication code
SATUINT32_t	uiMACLen	Length of message authentication code in bytes

Returns SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Plaintext length plus authentication data length is zero.
SATR_BADMACLEN	Unsupported message authentication code length.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description This function performs authenticated encryption with associated data, for confidentiality and integrity. The associated data, pointed at by pAuth, with byte length uiAuthLen, is used in the computation of the message authentication code, but is not encrypted. The data pointed at by pSrc, with byte length uiEncLen, is both encrypted and used in the computation of the message authentication code according to the selected encryption algorithm, specified by eSymType, and mode, specified by eMode. The encrypted data will be stored at the location pointed at by pDest, and the message authentication code will be stored at the location pointed at by pMAC; the byte length of the message authentication code is specified by uiMACLen. The data will be encrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded and the previously loaded key will be used. The initialization vector is pointed at by pIV.

For SATSYMMODE_GCM, the initialization vector is J_0 as defined in Section 7.1 of NIST SP800-38D[1].

1. Dworkin, M., NIST Special Publication 800-38D, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, National Institute of Standards and Technology, November 2007.

CALSymDecVerify

Syntax

```
rReturn=CALSymDecVerify (eSymType, puiKey, eMode,
pIV, pSrc, pDest, uiEncLen, pAuth, uiAuthLen,
pMAC)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
const void *	pSrc	Pointer to the ciphertext/verify data buffer
void *	pDest	Pointer to the plaintext data buffer
SATUINT32_t	uiEncLen	Length of ciphertext in bytes to decrypt/verify
void *	pAuth	Pointer to the verify-only data buffer
SATUINT32_t	uiAuthLen	Length of verify-only data in bytes
void *	pMAC	Pointer to the message authentication code
SATUINT32_t	uiMACLen	Length of message authentication code in bytes

Returns SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Plaintext length plus authentication data length is zero.
SATR_BADMACLEN	Unsupported message authentication code length.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VERIFYFAIL	Message authentication code verification failure.

Description

This function performs authenticated decryption with associated data, for confidentiality and integrity. The associated data, pointed at by pAuth, with byte length uiAuthLen, is used in the computation of the message authentication code, but is not decrypted. The data pointed at by pSrc, with byte length uiEncLen, is both decrypted and used in the computation of the message authentication code according to the selected decryption algorithm, specified by eSymType, and mode, specified by eMode. The decrypted data will be stored at the location pointed at by pDest, and the message authentication code will be compared to the value stored at the location pointed at by pMAC; the byte length of the message authentication code is specified by uiMACLen. The data will be decrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded and the previously loaded key will be used. The initialization vector is pointed at by pIV.

For SATSYMMODE_GCM, the initialization vector is J_0 as defined in Section 7.1 of NIST SP800-38D[1].

1. Dworkin, M., NIST Special Publication 800-38D, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, National Institute of Standards and Technology, November 2007.

ATHENA

CALSymEncAuthDMA

Syntax

```
rReturn=CALSymEncAuthDMA(eSymType, puiKey, eMode,
pIV, pSrc, pDest, uiEncLen, pAuth, uiAuthLen,
pMAC, uiDMAChConfig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key (SAT_NULL)
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
const void *	pSrc	Pointer to the plaintext/authenticate data buffer
void *	pDest	Pointer to the ciphertext data buffer
SATUINT32_t	uiEncLen	Length of plaintext in bytes to encrypt/authenticate
void *	pAuth	Pointer to the authenticate-only data buffer
SATUINT32_t	uiAuthLen	Length of authenticate-only data in bytes
void *	pMAC	Pointer to the message authentication code
SATUINT32_t	uiMACLen	Length of message authentication code in bytes
SATUINT32_t	uiDMAChConfig	DMA channel configuration

Returns

SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Plaintext length plus authentication data length is zero.
SATR_BADMACLEN	Unsupported message authentication code length.

Return Value	*Causes*
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description This function is similar to CALSymEncAuth, but uses DMA to transfer plaintext, authenticate-only data, and ciphertext. The DMA channel configuration is provided by uiDMAChConfig; refer to *CAL DMA Configuration* on page 121 for additional information on the channel configuration.

This function performs authenticated encryption with associated data, for confidentiality and integrity. The associated data, pointed at by pAuth, with byte length uiAuthLen, is used in the computation of the message authentication code, but is not encrypted. The data pointed at by pSrc, with byte length uiEncLen, is both encrypted and used in the computation of the message authentication code according to the selected encryption algorithm, specified by eSymType, and mode, specified by eMode. The encrypted data will be stored at the location pointed at by pDest, and the message authentication code will be stored at the location pointed at by pMAC; the byte length of the message authentication code is specified by uiMACLen. The data will be encrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded and the previously loaded key will be used. The initialization vector is pointed at by pIV.

For SATSYMMODE_GCM, the initialization vector is J_0 as defined in Section 7.1 of NIST SP800-38D[1].

1. Dworkin, M., NIST Special Publication 800-38D, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, National Institute of Standards and Technology, November 2007.

CALSymDecVerifyDMA

Syntax

```
rReturn=CALSymDecVerify (eSymType, puiKey, eMode,
pIV, pSrc, pDest, uiEncLen, pAuth, uiAuthLen,
pMAC, uiDMAChConfig)
```

Parameters Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
const void *	pSrc	Pointer to the ciphertext/verify data buffer
void *	pDest	Pointer to the plaintext data buffer
SATUINT32_t	uiEncLen	Length of ciphertext in bytes to decrypt/verify
void *	pAuth	Pointer to the verify-only data buffer
SATUINT32_t	uiAuthLen	Length of verify-only data in bytes
void *	pMAC	Pointer to the message authentication code
SATUINT32_t	uiMACLen	Length of message authentication code in bytes
SATUINT32_t	uiDMAChConfig	DMA channel configuration

Returns SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Plaintext length plus authentication data length is zero.
SATR_BADMACLEN	Unsupported message authentication code length.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_VERIFYFAIL	Message authentication code verification failure.

Description

This function is similar to CALSymDecVerify, but uses DMA to transfer plaintext, authenticate-only data, and ciphertext. The DMA channel configuration is provided by uiDMAChConfig; refer to *CAL DMA Configuration* on page 121 for additional information on the channel configuration.

This function performs authenticated decryption with associated data, for confidentiality and integrity. The associated data, pointed at by pAuth, with byte length uiAuthLen, is used in the computation of the message authentication code, but is not decrypted. The data pointed at by pSrc, with byte length uiEncLen, is both decrypted and used in the computation of the message authentication code according to the selected decryption algorithm, specified by eSymType, and mode, specified by eMode. The decrypted data will be stored at the location pointed at by pDest, and the message authentication code will be compared to the value stored at the location pointed at by pMAC; the byte length of the message authentication code is specified by uiMACLen. The data will be decrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded and the previously loaded key will be used. The initialization vector is pointed at by pIV.

For SATSYMMODE_GCM, the initialization vector is J_0 as defined in Section 7.1 of NIST SP800-38D[1].

1. Dworkin, M., NIST Special Publication 800-38D, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, National Institute of Standards and Technology, November 2007.

3.6 Encryption with SCA Countermeasures Function Descriptions

Each of the functions listed in Table 3-6 is described in this section. The functions described in this section employ protocol countermeasures. The underlying hardware implementations may also include leakage reduction countermeasures; however, the presence and use of such countermeasures is not controllable via the CAL API. For additional information regarding countermeasures refer to *SCA Countermeasures* on page 124.

Table 3-6: TeraFire CAL Encryption Functions with Protocol SCA Countermeasures

Function	*Description*	*Page*
CALSymEncryptKR	Symmetric encryption with key roll	149
CALSymDecryptKR	Symmetric decryption with key roll	151
CALSymEncryptKRDMA	Symmetric encryption with key roll and DMA	153
CALSymDecryptKRDMA	Symmetric decryption with key roll DMA	155

CALSymEncryptKR

Syntax

```
rReturn=CALSymEncryptKR(eSymType, puiKey, eMode,
pIV, bLoadIV, pSrc, pDest, uiLen, uiKRF)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key (SAT_NULL)
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
SATBOOL	bLoadIV	IV load control
const void *	pSrc	Pointer to the plaintext data buffer
void *	pDest	Pointer to the ciphertext data buffer
SATUINT32_t	uiLen	Length of plaintext in bytes to encrypt
SATUINT32_t	uiKRF	Key roll factor

Returns

SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Plaintext length is not a non-zero multiple of the cipher block size.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description

This function acts as CALSymEncrypt with key rolling. Key update frequency is controlled by the key roll factor parameter, uiKRF. The key roll

factor is the number of blocks that will be processed before the key is updated. If the key roll factor is zero then the key will not be updated.

This function encrypts the data buffer pointed at by pSrc of specified byte length, uiLen, and stores the result at the location pointed at by pDest. The symmetric encryption method is given by eSymType, and the mode of encryption is given by eMode. The data will be encrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded, and the previously loaded key will be used.

For algorithm/mode combinations that use an initialization vector, the IV is pointed at by pIV. If bLoadIV is SAT_TRUE, the IV will be loaded. Otherwise, the previous result's IV will be used. The resulting IV after encryption is stored at the location pointed at by pIV when pIV is not SAT_NULL and the mode is CBC, CFB, OFB or CTR.

Since symmetric encryption is performed in blocks, the byte length, uiLen, of the encryption must be a non-zero value divisible by the block size of the given symmetric method, eSymType.

CALSymDecryptKR

Syntax

```
rReturn=CALSymDecryptKR(eSymType, puiKey, eMode,
pIV, bLoadIV, pSrc, pDest, uiLen, uiKRF)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key (SAT_NULL)
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
SATBOOL	bLoadIV	IV load control
const void *	pSrc	Pointer to the ciphertext data buffer
void *	pDest	Pointer to the plaintext data buffer
SATUINT32_t	uiLen	Length of plaintext in bytes to decrypt
SATUINT32_t	uiKRF	Key roll factor

Returns

SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Plaintext length is not a non-zero multiple of the cipher block size.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description

This function acts as CALSymDecrypt with key rolling. Key update frequency is controlled by the key roll factor parameter, uiKRF. The key roll

factor is the number of blocks that will be processed before the key is updated. If the key roll factor is zero then the key will not be updated.

This function decrypts the data buffer pointed at by pSrc of specified byte length, uiLen, and stores the result at the location pointed at by pDest. The symmetric decryption method is given by eSymType, and the mode of decryption is given by eMode. The data will be decrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded and the previously loaded key will be used.

The initialization vector is pointed at by pIV. If bLoadIV is SAT_TRUE, the IV will be loaded. Otherwise, the previous result's IV will be used. The resulting IV after decryption is stored at the location pointed at by pIV when pIV is not SAT_NULL and the mode is CBC, CFB, OFB or CTR.

Since symmetric encryption is performed in blocks, the byte length, uiLen, of the encryption must be a non-zero value divisible by the block size of the given symmetric method, eSymType.

CALSymEncryptKRDMA

Syntax

```
rReturn=CALSymEncryptKRDMA(eSymType, puiKey,
eMode, pIV, bLoadIV, pSrc, pDest, uiLen, uiKRF,
uiDMAChConfig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key (SAT_NULL)
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
SATBOOL	bLoadIV	IV load control
const void *	pSrc	Pointer to the plaintext data buffer
void *	pDest	Pointer to the ciphertext data buffer
SATUINT32_t	uiLen	Length of plaintext in bytes to encrypt
SATUINT32_t	uiKRF	Key roll factor
SATUINT32_t	uiDMAChConfig	DMA channel configuration

Returns

SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Plaintext length is not a non-zero multiple of the cipher block size.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description This function is similar to CALSymEncryptKR, but uses DMA to transfer plaintext and ciphertext. The DMA channel configuration is provided by uiDMAChConfig; refer to *CAL DMA Configuration* on page 121 for additional information on the channel configuration.

This function acts as CALSymEncryptDMA with key rolling. Key update frequency is controlled by the key roll factor parameter, uiKRF. The key roll factor is the number of blocks that will be processed before the key is updated. If the key roll factor is zero then the key will not be updated.

This function encrypts the data buffer pointed at by pSrc of specified byte length, uiLen, and stores the result at the location pointed at by pDest. The symmetric encryption method is given by eSymType, and the mode of encryption is given by eMode. The data will be encrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded, and the previously loaded key will be used.

For algorithm/mode combinations that use an initialization vector, the IV is pointed at by pIV. If bLoadIV is SAT_TRUE, the IV will be loaded. Otherwise, the previous result's IV will be used. The resulting IV after encryption is stored at the location pointed at by pIV when pIV is not SAT_NULL and the mode is CBC, CFB, OFB or CTR.

Since symmetric encryption is performed in blocks, the byte length, uiLen, of the encryption must be a non-zero value divisible by the block size of the given symmetric method, eSymType.

CALSymDecryptKRDMA

Syntax

```
rReturn=CALSymDecryptKRDMA(eSymType, puiKey,
eMode, pIV, bLoadIV, pSrc, pDest, uiLen, uiKRF,
uiDMAChConfig)
```

Parameters

Where indicated, SAT_NULL will cause reuse of the prior value, provided that the last operation initiated was the *same* as this operation.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm
const SATUINT32_t *	puiKey	Pointer to key (SAT_NULL)
SATSYMMODE	eMode	Encryption mode
void *	pIV	Pointer to the initialization vector
SATBOOL	bLoadIV	IV load control
const void *	pSrc	Pointer to the ciphertext data buffer
void *	pDest	Pointer to the plaintext data buffer
SATUINT32_t	uiLen	Length of plaintext in bytes to decrypt
SATUINT32_t	uiKRF	Key roll factor
SATUINT32_t	uiDMAChConfig	DMA channel configuration

Returns

SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Plaintext length is not a non-zero multiple of the cipher block size.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_BADMODE	Unsupported encryption mode.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description This function is similar to CALSymDecryptKR, but uses DMA to transfer plaintext and ciphertext. The DMA channel configuration is provided by uiDMAChConfig; refer to *CAL DMA Configuration* on page 121 for additional information on the channel configuration.

This function acts as CALSymDecryptDMA with key rolling. Key update frequency is controlled by the key roll factor parameter, uiKRF. The key roll factor is the number of blocks that will be processed before the key is updated. If the key roll factor is zero then the key will not be updated.

This function decrypts the data buffer pointed at by pSrc of specified byte length, uiLen, and stores the result at the location pointed at by pDest. The symmetric decryption method is given by eSymType, and the mode of decryption is given by eMode. The data will be decrypted using the key pointed at by puiKey. If puiKey is SAT_NULL, no key will be loaded and the previously loaded key will be used.

The initialization vector is pointed at by pIV. If bLoadIV is SAT_TRUE, the IV will be loaded. Otherwise, the previous result's IV will be used. The resulting IV after decryption is stored at the location pointed at by pIV when pIV is not SAT_NULL and the mode is CBC, CFB, OFB or CTR.

Since symmetric encryption is performed in blocks, the byte length, uiLen, of the encryption must be a non-zero value divisible by the block size of the given symmetric method, eSymType.

3.7 Hashes

Cryptographic hash functions are one-way functions that take an arbitrary length data input and output a fixed length result. CAL hash functions include support for a number of common hash algorithms, which are listed in Table 3-7. CAL can either perform a hash in a single call or separately in parts. The advantage to performing the hash in parts is that all the data is not required at once. When performing hashes in parts, context switching between messages may be performed.

Table 3-7: TeraFire CAL Hash Algorithms

Identifier[a]	*Algorithm*	*Block Size (32-bit Words)*	*Block Size (Bytes)*
SATHASHTYPE_SHA1	SHA-1	16	64
SATHASHTYPE_SHA224	SHA-224	16	64
SATHASHTYPE_SHA256	SHA-256	16	64
SATHASHTYPE_SHA384	SHA-384	32	128
SATHASHTYPE_SHA512	SHA-512	32	128
SATHASHTYPE_SHA512_224	SHA-512/224	32	128
SATHASHTYPE_SHA512_256	SHA-512/256	32	128

a. Identifier is SATHASHTYPE type.

This section describes integrated CAL hash functions that hash a message in a single call. Each of the functions listed in Table 3-8 is described in this section.

Table 3-8: TeraFire CAL Hash Functions

Function	*Description*	*Page*
CALHash	Single hash function	158
CALHashDMA	Single hash function with DMA	159

See also *Hash Functions with Context Switching* on page 164, and *Multiple Call Hash Functions* on page 160 for alternative hash functions.

CALHash

Syntax

```
rReturn=CALHash(eHashType, pMsg, uiMsgLen, pHash)
```

Parameters

Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATHASHTYPE	eHashType	Hash algorithm
const void *	pMsg	Pointer to message input buffer
SATUINT32_t	uiMsgLen	Message buffer length in bytes
void *	pHash	Pointer to hash result

Returns

SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADHASHTYPE	Unsupported hash algorithm.

This function does not use CALSymTrfRes.

Description

This function performs the hash operation specified by eHashType on the data stored at the location specified by pMsg of byte length uiMsgLen. The hash results are stored at location pointed at by pHash. This function requires the all data to be hashed to be in the buffer as this function does not support intermediate hash states.

CALHashDMA

Syntax

```
rReturn=CALHashDMA(eHashType, pMsg, uiMsgLen,
pHash, uiDMAChConfig)
```

Parameters

Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATHASHTYPE	eHashType	Hash algorithm
const void *	pMsg	Pointer to message input buffer
SATUINT32_t	uiMsgLen	Message buffer length in bytes
void *	pHash	Pointer to hash result
SATUINT32_t	uiDMAChConfig	DMA channel configuration

Returns

SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADHASHTYPE	Unsupported hash algorithm.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description

This function is similar to CALHash, but uses DMA to transfer the data and result. The DMA channel configuration is provided by uiDMAChConfig; refer to *CAL DMA Configuration* on page 121 for additional information on the channel configuration.

This function performs the hash operation specified by eHashType on the data stored at the location specified by pMsg of byte length uiMsgLen. The hash results are stored at location pointed at by pHash. This function requires the all data to be hashed to be in the buffer as this function does not support intermediate hash states.

3.8 Multiple Call Hash Functions

This section describes hash functions that support hashing of large messages that are not, or cannot be, loaded in a single block of memory and processed in an single operation. Each of the functions listed in Table 3-9 is described in this section.

Table 3-9: TeraFire CAL Multiple Call Hash Functions

Function	*Description*	*Page*
CALHashIni	Hash initialization function	161
CALHashWrite	Writes to and/or finalizes a hash function	162
CALHashRead	Reads the final or intermediate hash value	163

A multiple call hash function that processes a block at a time is shown in Example 3-2, Multiple Call Hash, below.

Example 3-2 Multiple Call Hash

```
SATUINT32_t ui32Message[16]; /* 16-w/64-bytes/512-bits */
SATUINT32_t ui32MsgLen;      /* Message length in bytes. */
SATUINT32_t ui32Hash[8];     /* 256-bits */

/* Initialize context for SHA-256 operation */
rStatus=CALHashIni(SATHASHTYPE_SHA256);

/* Read message a block at a time until we're done. */
/* ReadBlock returns the length read into the passed
   buffer -- 64-bytes (16-words) -- unless there's
   less than that left. */
ui32MsgLen=ReadBlock(ui32Message);
while (ui32MsgLen>0) {
  rStatus=CALHashWrite(ui32Message, ui32MsgLen);
  ui32MsgLen=ReadBlock(ui32Message);
}

/* Finalize */
rStatus=CALHashRead(ui32Hash);
```

CALHashIni

Syntax `rReturn=CALHashIni(eHashType, uiMsgLen)`

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATHASHTYPE	eHashType	Hash algorithm
SATUINT32_t	uiMsgLen	Message length in bytes

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADHASHTYPE	Unsupported hash algorithm.

This function does not use CALSymTrfRes.

Description This function initializes a hash operation specified by eHashType with message length, in bytes, specified by uiMsgLen. All data from previous hashing will be lost when this function is performed. This function must be called prior to call(s) to CALHashWrite to input message data into the hash processing resource.

This function requires the message length. If the length of the message is not known *a priori*, refer to *Hash Functions with Context Switching* on page 164, which supports this use case.

CALHashWrite

Syntax

```
rReturn=CALHashWrite(pBuffer, uiBufLen)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
const void *	pBuffer	Pointer to message input buffer
SATUINT32_t	uiBufLen	Message buffer length in bytes

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADHASHLEN	If the write is not the final write of the hash operation, then the length of the buffer must be a multiple of the word size of the selected hash type. If the write is the final write of the hash operation, then the total length of the message written must match the length provided in the CALHashIni call.
SATR_BADHASHTYPE	Unsupported hash algorithm type state, most likely subsequent to not calling CALHashIni successfully.

This function does not use CALSymTrfRes.

Description Before using this function the hash must be initialized using CALHashIni. This function processes data pointed at by pBuffer of size uiBufLen in bytes, using the hash operation initialized in CALHashIni. The value of uiBufLen must be less than or equal to the initialized message length. If uiBufLen is less than the initialized message length, multiple calls to CALHashWrite must be made. Unless the CALHashWrite call is the only or last call of the hash, uiBufLen must be divisible by the internal word size which is four bytes for SHA-1, SHA-224, and SHA-256 and eight bytes for SHA-384 and SHA-512 (including SHA-512/224 and SHA-512/256).

CALHashRead

Syntax `rReturn=CALHashRead(pHash)`

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
void *	pHash	Pointer to hash result

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADPARAM	pHash parameter is SAT_NULL
SATR_BADHASHTYPE	Unsupported hash algorithm type state, most likely subsequent to not calling CALHashIni successfully.

This function does not use CALSymTrfRes.

Description This function stores the processed hash value in the location pointed to by pHash.

3.9 Hash Functions with Context Switching

This section describes CAL hash functions that support context switching. These functions are intended to support hashing of large messages with timesharing of the hash processing resource to enable precedence for higher priority messages. Note that CAL does not provide a priority queueing facility. Each of the functions listed in Table 3-10 is described in this section.

Table 3-10: TeraFire CAL Hash Functions with Context Switching

Function	*Description*	*Page*
CALHashCtx	Hash with context support	165
CALHashCtxIni	Context initialization for hash with context	167

In order to support context switching, additional memory must be provided to support on-demand offloading of contexts from the computational resource.

CALHashCtx

Syntax

```
rReturn=CALHashCtx(hResource, pContext, pMsg,
uiMsgLen, pHash, bFinal)
```

Parameters

Parameters to this function are listed below.

Type	*Identifier*	*Description*
const SATRESHANDLE	hResource	Computational resource handle
SATRESCON-TEXTPTR const	pContext	Pointer to a context
const void *	pMsg	Pointer to input buffer
SATUINT32_t	uiMsgLen	Buffer length in bytes
void *	pHash	Pointer to hash result
const SATBOOL	bFinal	Flag to indicate final input to the hash computation

Returns

SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADPARAM	Resulting hash pointer is NULL for the final block.
SATR_BADCONTEXT	The provided context is SAT_NULL.

This function does not use CALSymTrfRes.

Description

This function performs the hash operation specified in the context on the supplied message. The computation is performed using the specified computational resource. If the resource is not currently loaded with the context of the specified computation, then the context associated with the resource will be unloaded, and the specified context will be loaded before performing the computation.

Prior to calling this function, the context must be initialized using the CALHashCtxIni function.

This function may be used to compute the hash of a message in a single call (see Example 3-3, Single Call Hash with Context) or multiple calls (see Example 3-4, Multiple Call Hash with Context). The latter use

case is useful when a message cannot be supplied in a single block in memory, such as when processing the hash of a large file. When using multiple calls to supply the message to the hash function, the message must be supplied in block sizes equal to an integer multiple of the native block size of the selected hash (*e.g.*, an integer multiple of 64-bytes for SHA-1, SHA-224, and SHA-256).

Example 3-3 Single Call Hash with Context

```
SATRESCONTEXT srcContext;
SATUINT32_t ui32Message[]={12,34,56,78}; /* 4-w/16-bytes */
SATUINT32_t ui32Hash[8];                 /* 256-bits */

/* Initialize context for SHA-256 operation */
rStatus=CALHashCtxIni(&srcContext, SATHASHTYPE_SHA256);

/* Perform SHA-256 on 4-word/16-byte message. */
rStatus=CALHashCtx(SATRES_DEFAULT, &srcContext, ui32Message,
                   16, ui32Hash, SAT_TRUE);
```

Example 3-4 Multiple Call Hash with Context

```
SATRESCONTEXT srcContext;
SATUINT32_t ui32Message[16]; /* 16-w/64-bytes/512-bits */
SATUINT32_t ui32MsgLen;
SATUINT32_t ui32Hash[8];     /* 256-bits */

/* Initialize context for SHA-256 operation */
rStatus=CALHashCtxIni(&srcContext, SATHASHTYPE_SHA256);

/* Read message a block at a time until we're done. */
/* ReadBlock returns the length read into the passed
   buffer -- 64-bytes (16-words) -- unless there's
   less than that left. */
ui32MsgLen=ReadBlock(ui32Message);
while (ui32MsgLen>0) {
  rStatus=CALHashCtx(SATRES_DEFAULT, &srcContext,
            ui32Message, ui32MsgLen, NULL, SAT_FALSE);
  ui32MsgLen=ReadBlock(ui32Message);
}

/* Finalize */
rStatus=CALHashCtx(SATRES_DEFAULT, &srcContext, ui32Message,
                   ui32MsgLen, ui32Hash, SAT_TRUE);
```

See Also *CALHashCtxIni* on page 167

CALHashCtxIni

Syntax

```
rReturn=CALHashCtxIni(pContext, eHashType)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATRESCON-TEXTPTR const	pContext	Pointer to a context
const SATHASHTYPE	eHashType	Hash algorithm

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADHASHTYPE	Unsupported hash algorithm.
SATR_BADCONTEXT	The provided context is SAT_NULL.

This function does not use CALSymTrfRes.

Description This function initializes a context for hashing with the specified algorithm. This is required before CALHashCtx can be used to compute a hash.

See Also *CALHashCtx* on page 165

3.10 Message Authentication Codes

Message authentication code (MAC) functions are one-way functions that take a secret key and an arbitrary length data input and output a fixed length result. The MAC value protects both integrity and authenticity of the message. CAL supports the HMACs in the SHA family. The supported algorithms are listed in Table 3-11.

Table 3-11: TeraFire CAL Message Authentication Algorithms

Identifier[a]	*Algorithm*
SATMACTYPE_SHA1	HMAC SHA-1
SATMACTYPE_SHA224	HMAC SHA-224
SATMACTYPE_SHA256	HMAC SHA-256
SATMACTYPE_SHA384	HMAC SHA-384
SATMACTYPE_SHA512	HMAC SHA-512
SATMACTYPE_AESCMAC128	AES-CMAC-128
SATMACTYPE_AESCMAC192	AES-CMAC-192
SATMACTYPE_AESCMAC256	AES-CMAC-256

a. Identifier is SATMACTYPE type.

This section describes integrated CAL MAC functions that compute a MAC for a message in a single call. Each of the functions listed in Table 3-12 is described in this section.

Table 3-12: TeraFire CAL Message Authentication Functions

Function	*Description*	*Page*
CALMAC	Single call MAC computation	158
CALMACDMA	Single call MAC computation with DMA	170

See also *Multiple Call MAC Functions* on page 172, and *MAC Functions with Context Switching* on page 176.

CALMAC

Syntax

```
rReturn=CALMAC(eMACType, puiKey, uiKeyLen, pMsg,
uiMsgLen, pMAC)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATMACTYPE	eMacType	MAC algorithm
const SATUINT32_t *	puiKey	Key
SATUINT32_t	uiKeyLen	Length of MAC key in bytes. Must be a non-zero value, unless the key length is implied by the MAC type.
const void *	pMsg	Pointer to message input buffer
SATUINT32_t	uiMsgLen	Message buffer length in bytes
void *	pMAC	Pointer to MAC results

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Key length is zero or a value not supported by the implementation.
SATR_BADMACTYPE	Unsupported hash algorithm.

This function does not use CALSymTrfRes.

Description This function calculates the MAC specified by eMACType on the data stored at the location specified by pMsg of byte length uiMsgLen with the secret key specified by puiKey of byte length uiKeyLen. Note that for some MAC types the key length is implicit in the MAC type (*e.g.*, SATMACTYPE_AESCMAC128/AES-CMAC-128, which uses a 128-bit/16-byte key). The MAC results are stored at location pointed at by pMAC. This functions requires the all data to be in the buffer as this function does not support intermediate states.

CALMACDMA

Syntax

```
rReturn=CALMACDMA(eMACType, puiKey, uiKeyLen,
pMsg, uiMsgLen, pMAC, uiDMAChConfig)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATMACTYPE	eMacType	MAC algorithm
const SATUINT32_t *	puiKey	Key
SATUINT32_t	uiKeyLen	Length of MAC key in bytes. Must be a non-zero value, unless the key length is implied by the MAC type.
const void *	pMsg	Pointer to message input buffer
SATUINT32_t	uiMsgLen	Message buffer length in bytes
void *	pMAC	Pointer to MAC results
SATUINT32_t	uiDMAChConfig	DMA channel configuration

Returns SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Key length is zero or a value not supported by the implementation.
SATR_BADMACTYPE	Unsupported hash algorithm.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description This function is similar to CALMAC, but uses DMA to transfer the data and result. The DMA channel configuration is provided by uiDMAChConfig; refer to *CAL DMA Configuration* on page 121 for additional information on the channel configuration.

This function calculates the MAC specified by eMACType on the data stored at the location specified by pMsg of byte length uiMsgLen with the secret key specified by puiKey of byte length uiKeyLen. Note that for some MAC types the key length is implicit in the MAC type (*e.g.*, SATMACTYPE_AESCMAC128/AES-CMAC-128, which uses a 128-bit/16-byte key). The MAC results are stored at location pointed at by pMAC.

3.11 Multiple Call MAC Functions

This section describes MAC functions that support computation of a MAC for large messages that are not, or cannot be, loaded in a single block of memory and processed in an single operation. Each of the functions listed in Table 3-12 is described in this section.

Table 3-13: TeraFire CAL Message Authentication Functions

Function	*Description*	*Page*
CALMACIni	Multiple call MAC computation initialization	173
CALMACWrite	Multiple call MAC computation message date input	174
CALMACRead	Multiple call MAC computation MAC output	175

CALMACIni

Syntax

```
rReturn=CALMACIni(eMACType, puiKey, uiKeyLen,
uiMsgLen)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATMACTYPE	eMacType	MAC algorithm
const SATUINT32_t *	puiKey	Key
SATUINT32_t	uiKeyLen	Length of MAC key in bytes. Must be a non-zero value, unless the key length is implied by the MAC type.
SATUINT32_t	uiMsgLen	Message buffer length in bytes

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Key length is zero or a value not supported by the implementation.
SATR_BADMACTYPE	Unsupported MAC algorithm.

This function does not use CALSymTrfRes.

Description This function initializes a MAC operation specified by eMACType for a message of length uiMsgLen and a MAC key specified by puiKey of byte length uiKeyLen. Note that for some MAC types the key length is implicit in the MAC type (*e.g.*, SATMACTYPE_AESCMAC128/AES-CMAC-128, which uses a 128-bit/16-byte key).

CALMACWrite

Syntax

```
rReturn=CALMACWrite(pBuffer, uiBufLen)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
const void *	pBuffer	Pointer to message input buffer
SATUINT32_t	uiBufLen	Buffer length in bytes

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADMACLEN	If the write is not the final write of the MAC operation, then the length of the buffer must be a multiple of the data word size of the selected MAC type. If the write is the final write of the MAC operation, then the total length of the message written must match the length provided in the CALMACIni call.
SATR_BADMACTYPE	Unsupported MAC algorithm type state, most likely subsequent to not calling CALMACIni successfully.

This function does not use CALSymTrfRes.

Description This function processes data pointed at by pBuffer of size uiBufLen using the MAC operation initialized in CALMACIni. uiBufLen must be less than or equal to the initialized message length. If uiBufLen is less than the initialized message length, multiple calls to CALMACWrite must be made. Unless the CALMACWrite call is the only or last call of the MAC, uiBufLen must be divisible by the internal word size which is four bytes for SHA-1, SHA-224, and SHA-256; eight bytes for SHA-384 and SHA-512; and sixteen bytes for AES-CMAC.

CALMACRead

Syntax `rReturn=CALMACRead(puiKey, uiKeyLen, pMAC)`

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
const SATUINT32_t *	puiKey	Key
SATUINT32_t	uiKeyLen	Length of MAC key in bytes
void *	pMAC	Pointer to MAC results.

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADPARAM	pMAC parameter is SAT_NULL
SATR_BADMACTYPE	Unsupported MAC algorithm type state, most likely subsequent to not calling CALMACIni successfully.

This function does not use CALSymTrfRes.

Description This function stores the resulting MAC in the location pointed to by pMAC, with a MAC key pointed to by puiKey, of byte length uiKeyLen.

3.12 MAC Functions with Context Switching

This section describes CAL MAC functions that support context switching. These functions are intended to support HMAC-SHA (only) MAC computation for large messages with timesharing of the processing resource to enable precedence for higher priority messages. Note that CAL does not provide a priority queueing facility. Each of the functions listed in Table 3-14 is described in this section.

Table 3-14: TeraFire CAL Message Authentication Functions

Function	*Description*	*Page*
CALMACCtx	MAC computation with context switching support	177
CALMACCtxIni	MAC computation with context switching support initialization function	179

In order to support context switching, additional memory must be provided to support on-demand offloading of contexts from the computational resource.

MAC functions with context switching support are an optional feature, and may not be present in some configurations. Refer to configuration documentation for more information.

CALMACCtx

Syntax

```
rReturn=CALMACCtx(hResource, pContext, pMsg,
uiMsgLen, pHash, bFinal)
```

Parameters

Parameters to this function are listed below.

Type	*Identifier*	*Description*
const SATRESHANDLE	hResource	Computational resource handle
SATRESCON-TEXTPTR const	pContext	Pointer to a context
const void *	pMsg	Pointer to input buffer
SATUINT32_t	uiMsgLen	Buffer length in bytes
void *	pHash	Pointer to hash result
const SATBOOL	bFinal	Flag to indicate final input to the hash computation

Returns

SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADPARAM	Resulting hash pointer is NULL for the final block.
SATR_BADCONTEXT	The provided context is SAT_NULL.

This function does not use CALSymTrfRes.

Description

This function performs the MAC computation operation specified in the context on the supplied message. The computation is performed using the specified computational resource. If the resource is not currently loaded with the context of the specified computation, then the context associated with the resource will be unloaded, and the specified context will be loaded before performing the computation.

Prior to calling this function, the context must be initialized using the CALMACCtxIni function.

This function may be used to compute the hash of a message in a single call (see Example 3-5, Single Call MAC with Context) or multiple calls (see Example 3-6, Multiple Call MAC with Context). The latter use case is

useful when a message cannot be supplied in a single block in memory, such as when processing the hash of a large file. When using multiple calls to supply the message to the hash function, the message must be supplied in block sizes equal to an integer multiple of the native block size of the selected hash (*e.g.*, an integer multiple of 64-bytes for SHA-1, SHA-224, and SHA-256).

Example 3-5 Single Call MAC with Context

```
SATRESCONTEXT srcContext;
SATUINT32_t ui32Message[]={12,34,56,78}; /* 4-w/16-bytes */
SATUINT32_t ui32Hash[8];                 /* 256-bits */
SATUINT32_t uiKey[8];                    /* 256-bit key. */

/* Initialize context for HMAC-SHA-256 operation */
rStatus=CALMACCtxIni(&srcContext, SATHASHTYPE_SHA256,
                     uiKey, 32);

/* Perform HMAC-SHA-256 on 4-word/16-byte message. */
rStatus=CALMACCtx(SATRES_DEFAULT, &srcContext, ui32Message,
                  16, ui32Hash, SAT_TRUE);
```

Example 3-6 Multiple Call MAC with Context

```
SATRESCONTEXT srcContext;
SATUINT32_t ui32Message[16]; /* 16-w/64-bytes/512-bits */
SATUINT32_t ui32MsgLen;
SATUINT32_t ui32Hash[8];     /* 256-bits */
SATUINT32_t uiKey[8];        /* 256-bit key. */

/* Initialize context for HMAC-SHA-256 operation */
rStatus=CALMACCtxIni(&srcContext, SATHASHTYPE_SHA256,
                     uiKey, 32);

/* Read message a block at a time until we're done. */
/* ReadBlock returns the length read into the passed
   buffer -- 64-bytes (16-words) -- unless there's
   less than that left. */
ui32MsgLen=ReadBlock(ui32Message);
while (ui32MsgLen>0) {
  rStatus=CALMACCtx(SATRES_DEFAULT, &srcContext,
            ui32Message, ui32MsgLen, NULL, SAT_FALSE);
  ui32MsgLen=ReadBlock(ui32Message);
}

/* Finalize */
rStatus=CALMACCtx(SATRES_DEFAULT, &srcContext, ui32Message,
                  ui32MsgLen, ui32Hash, SAT_TRUE);
```

CALMACCtxIni

Syntax

```
rReturn=CALMACCtxIni(pContext, eHashType, puiKey,
uiKey)
```

Parameters

Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATRESCON-TEXTPTR const	pContext	Pointer to a context
const SATHASHTYPE	eHashType	Hash algorithm
const SATUINT32_t *	puiKey	Pointer to MAC key
SATUINT32_t	uiKeyLen	Length of MAC key in bytes. Must be a non-zero value, unless the key length is implied by the MAC type.

Returns

SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADHASHTYPE	Unsupported hash algorithm.
SATR_BADCONTEXT	The provided context is SAT_NULL.

This function does not use CALSymTrfRes.

Description

This function initializes a context for computing a MAC with the specified algorithm. This is required before CALMACCtx can be used to compute a hash.

3.13 Random Number Generation

Cryptographic random number generators are random number generators that are suitably random for cryptographic applications and have the most stringent requirements for randomness of any application. Random number generation functions are broken into groups by function class. These are listed below.

- *Non-Deterministic Random Bit Generation* on page 181.
- *Deterministic Random Bit Generation (SP800-90A)* on page 190.

CAL random number generation combines a non-deterministic random bit generator (NRBG) with a deterministic random bit generator (DRBG) to implement a true random number generator (TRNG). The EXP-F5200B specifically supports an NRBG combined with an AES counter mode-based DRBG (AES CTR_DRBG), compliant with NIST SP800-90A[1]. For this TRNG solution, the DRBG accesses the NRBG to provide the entropy necessary for the TRNG solution, and continuous health monitoring of the NRBG built into the hardware and access to this data is provided in CAL.

1. Barker, E., Kelsey, J., NIST Special Publication 800-90A, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, National Institute of Standards and Technology, January 2012.

3.13.1 Non-Deterministic Random Bit Generation

This section describes the CAL non-deterministic random bit generator (CALNRBG) functions. Each of the functions listed in Table 3-19 is described in this section.

Table 3-15: TeraFire CAL NRBG Functions

Function	*Description*	*Page*
CALNRBGSetTestEntropy	Sets NRBG test entropy	182
CALNRBGAddTestEntropy	Adds NRBG test entropy	183
CALNRBGGetEntropy	Gets NRBG entropy	184
CALNRBGConfig	Configures NRBG for EXP-F5200B	186
CALNRBGHealthStatus	Provides NRBG health status	188

In most use cases, these functions will not be used by end users. These functions are provided primarily for test and assessment purposes. Users seeking RNG functions for end applications should use the functions described in *Deterministic Random Bit Generation (SP800-90A)* on page 190.

CALNRBGSetTestEntropy

Syntax

```
rReturn=CALNRBGSetTestEntropy(puiEntropy,
uiEntLen32)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATUINT32_t *	puiEntropy	Pointer to test entropy data
SATUINT32_t	uiEntLen32	Number of 32-bit words of test entropy

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADPARAM	The provided pointer to entropy is SAT_NULL, or the buffer length is zero.

This function does not use CALSymTrfRes.

Description This function sets the location of the test entropy buffer. This buffer is used as an entropy source when operating RNG functions in test mode, which enables deterministic testing of functions that consume entropy. The buffer is pointed at by puiEntropy, and the length of the buffer in 32-bit words is given by uiEntLen32.

CALNRBGAddTestEntropy

Syntax

```
rReturn=CALNRBGAddTestEntropy(puiEntropy,
uiEntLen32)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATUINT32_t *	puiEntropy	Pointer to buffer of test entropy to add
SATUINT32_t	uiEntLen32	Length of entropy to add in 32-bit words

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADPARAM	The provided pointer to entropy is SAT_NULL, or the buffer length is zero.

This function does not use CALSymTrfRes.

Description This function appends test entropy to a test entropy buffer previously established by CALNRBGSetTestEntropy. The data that is appended to the test entropy buffer is copied from the location pointed at by puiEntropy, with length uiEntLen32 words.

CALNRBGGetEntropy

Syntax

```
rReturn=CALNRBGGetEntropy(puiEntropy, uiEntLen32,
bTesting)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATUINT32_t *	puiEntropy	Pointer to memory buffer for entropy data
SATUINT32_t	uiEntLen32	Number of 32-bit words of entropy to fill
SATBOOL	bTesting	Boolean indicating whether or not to use test entropy

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_ROFATAL	NRBG fatal health failure.
SATR_FAIL	bTesting is SAT_TRUE and there is insufficient test entropy available.
SATR_FNP	Function implementation not populated.
SATR_BADPARAM	The provided pointer to entropy is SAT_NULL.
SATR_BADLEN	The number of words of entropy to fill is zero or a value unsupported by the implementation.

This function does not use CALSymTrfRes.

Description This function fills the data buffer pointed at by puiEntropy of specified 32-bit word length, uiEntLen32, with entropy. If bTesting is SAT_FALSE, then the entropy is obtained from the hardware NRBG. If bTesting is SAT_TRUE, then the entropy is test entropy, and it is obtained from the test entropy data buffer using the test entropy set or added by CALNRBGSetTestEntropy and CALNRBGAddTestEntropy, respectively. This function will return SATR_FAIL when bTesting is SAT_TRUE and there is an insufficient amount of test entropy available.

CALNRBGGetEntropy is intended primarily for NRBG assessment purposes. The output of this function should *not* be used directly in applications requiring cryptographic-grade random numbers. See

Deterministic Random Bit Generation (SP800-90A) on page 190 for functions that are suitable for generating random numbers for cryptographic applications.

CALNRBGConfig

Syntax

```
rReturn=CALNRBGConfig(uiWriteEn, uiCSR, uiCntLim,
uiVoTimer, uiFMsk, puiStatus)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATUINT32_t	uiWriteEn	Write enable control
SATUINT32_t	uiCSR	RNG_CSR value to write
SATUINT32_t	uiCntLim	RNG_CNTLIM value to write
SATUINT32_t	uiVoTimer	RNG_VOTIMER
SATUINT32_t	uiFMsk	RNG_FMSK
SATUINT32_t *	puiStatus	Pointer to array to store status values

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.

This function does not use CALSymTrfRes.

Description This function is used with EXP-F5200B implementations to set the NRBG control and status register, and NRBG health monitoring configuration values, and to read the current values of these same registers. The RNG_CSR, RNG_CNTLIM, RNG_VOTIMER, and RNG_FMSK register values may be set using dedicated parameters and controlled by flags OR'd together in the uiWriteEn parameter. The values used with uiWriteEn are listed in Table 3-16; if SATNRBGCONFIG_NONE is used for the uiWriteEn argument, then no values will be written.

Table 3-16: CALNRBGConfig Register Write Enable Identifiers

Identifier	*Register*
SATNRBGCONFIG_NONE	none
SATNRBGCONFIG_RNG_CSR	RNG_CSR
SATNRBGCONFIG_RNG_CNTLIM	RNG_CNTLIM
SATNRBGCONFIG_RNG_VOTIMER	RNG_VOTIMER

Table 3-16: CALNRBGConfig Register Write Enable Identifiers (Continued)

Identifier	*Register*
SATNRBGCONFIG_RNG_FMSK	RNG_FMSK

In most use cases, only the RNG_CSR will be of interest to end users. For the RNG_CSR, field identifiers are provided in Table 3-17, and examples of typical operations are provided in Example 3-7, NRBG Disable, and Example 3-8, NRBG Enable with Fatal Error Clear and Status Output.

Table 3-17: CALNRBGConfig RNG_CSR Field Identifiers

Identifier	*Field*
SATNRBGCONFIG_CSR_RODIS	NRBG disable control (mutually exclusive with NRBG enable)
SATNRBGCONFIG_CSR_ROEN	NRBG enable control (mutually exclusive with NRBG disable)
SATNRBGCONFIG_CSR_ROFATAL	NRBG fatal error status
SATNRBGCONFIG_CSR_ROFATALCLR	NRBG fatal error status clear

After setting any NRBG control values, if the argument puiStatus is not SAT_NULL, then the function will copy the value of the subject registers, and RNG_ROHEALTH, to the location pointed at by puiStatus. The population of this array is listed in Table 3-18.

Table 3-18: CALNRBGConfig Status Array Values

Array Offset	*Register*
0	RNG_CSR
1	RNG_CNTLIM
2	RNG_VOTIMER
3	RNG_ROHEALTH
4	RNG_FMSK

Example 3-7 NRBG Disable

```
/* Disable NRBG to save power. */
rStatus=CALNRBGConfig(SATNRBGCONFIG_RNG_CSR,
  SATNRBGCONFIG_CSR_RODIS,
  0, 0, 0,    /* Unused parameters. */
  SAT_NULL);  /* No status output. */
```

Example 3-8 NRBG Enable with Fatal Error Clear and Status Output

```
SATUINT32_t uiStatus[5];  /* Status output array. */

/* Enable NRBG and clear fatal error status (if any). */
rStatus=CALNRBGConfig(SATNRBGCONFIG_RNG_CSR,
  SATNRBGCONFIG_CSR_ROEN | SATNRBGCOFNIG_CSR_ROFATALCLR,
  0, 0, 0,    /* Unused parameters. */
  uiStatus);  /* Status output. */
```

CALNRBGHealthStatus

Syntax `uiStatus=CALNRBGHealthStatus()`

Parameters None.

Returns SATUINT32_t uiStatus, NRBG health fault status word. If the function is not populated then the function will return zero.

Description This function returns the status register value of the continuous NRBG health monitoring tests. See the RNG_ROHEALTH register definition for the subject core model for more information.

3.13.2 Deterministic Random Bit Generation (SP800-90A)

This section describes the CAL deterministic random bit generator (CALDRBG) functions. Each of the functions listed in Table 3-19 is described in this section.

Table 3-19: TeraFire CAL DRBG Functions

Function	*Description*	*Page*
CALDRBGInstantiate	Instantiates the DRBG	191
CALDRBGReseed	Reseeds the DRBG	193
CALDRBGGenerate	Generates DRBG output	194
CALDRBGUninstantiate	Uninstantiates the DRBG	196
CALDRBGGetCtx	Get the current context	197
CALDRBGLoadCtx	Load the current context	198

CALDRBG functions implement a full true random number generator combining NRBG output with an AES-based NIST SP800-90A CTR_DRBG. The CALDRBG includes functions to instantiate the DRBG, generate random bits from the DRBG, reseed the DRBG, and uninstantiate the DRBG.

When the DRBG is instantiated in normal mode, the DRBG accesses the hardware-based NRBG to provide the amount of entropy necessary for the instantiated security strength. For testing and validation purposes, the NRBG can be loaded with test entropy which is used by the DRBG when instantiated in test mode. Test entropy allows support for known answer testing on the DRBG, which is required by NIST SP800-90A.

If non-DRBG functions need to be performed while maintaining a DRBG instantiation, or multiple DRBG instantiations are desired, CALDRBG supports loading and unloading of DRBG contexts using the CALDRBGGetCtx and CALDRBGLoadCtx functions.

CALDRBGInstantiate

Syntax

```
rReturn=CALDRBGInstantiate(puiNonce, uiNonceLen,
puiPzStr, uiPzStrLen, eStrength, uiEntropyFactor,
uiReseedLim, bTesting)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
const SATUINT32_t *	puiNonce	Pointer to nonce
SATUINT32_t	uiNonceLen	Length, in words, of nonce
const SATUINT32_t *	puiPzStr	Pointer to personalization string
SATUINT32_t	uiPzStrLen	Length, in words, of personalization string
SATSYMKEYSIZE	eStrength	Security strength required
SATUINT32_t	uiEntropy-Factor	Entropy factor. Must be non-zero.
SATUINT32_t	uiReseedLim	Number of generates to perform before reseeding. Must be non-zero.
SATBOOL	bTesting	SAT_TRUE for testing, SAT_FALSE for normal operation

Returns SATR rReturn values produced by this initiation call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FAIL	Operation failed because the DRBG is already instantiated.
SATR_BADPARAM	Invalid security strength. In most configurations only 128-bit (SATSYMKEYSIZE_AES128) and 256-bit (SATSYMKEYSIZE_AES256) strengths are supported.
SATR_BADLEN	Zero entropy factor or zero reseed limit.
SATR_BADLEN	Nonce length and/or personalization string length exceeds capability of implementation.
SATR_FNP	Function implementation not populated.

This function does not cause CALSymTrfRes to return a value. CALSymTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_ROFATAL	NRBG fatal error during operation.

Description

This function instantiates the DRBG with a personalization string pointed at by puiPzStr of specified 32-bit word length, uiPzStrLen, and a nonce pointed at by puiNonce with the 32-bit word length defined by uiNonceLen. The nonce and/or personalization string may have zero length, in which case the respective pointers are ignored. The security strength of the instantiated DRBG is set to eStrength; in most implementations this must be either 128-bits (SATSYMKEYSIZE_AES128) or 256-bits (SATSYMKEYSIZE_AES256).

The entropy factor sets the ratio of raw NRBG data to entropy, and it is given by uiEntropyFactor. A default value for this is given by CALDRBGENTROPYFACTOR; it is strongly recommended that this default value is used unless otherwise instructed.

Using an entropy factor value other than the recommended default may result in an improperly seeded DRBG instantiation, or reduced performance.

The instantiation will automatically reseed after uiReseedLim number of generates. The boolean, bTesting, controls if the instantiation is a test instantiation, SAT_TRUE, or a normal instantiation, SAT_FALSE. Test instantiation should never be used in a production setting.

CALDRBGReseed

Syntax `rReturn=CALDRBGReseed(puiAddIn, uiAddInLen)`

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
const SATUINT32_t *	puiAddIn	Pointer to additional input
SATUINT32_t	uiAddInLen	Length, in words, of additional input

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FAIL	Operation failed because the DRBG is not instantiated.
SATR_BADLEN	Additional input length exceeds capability of implementation.
SATR_FNP	Function implementation not populated.

This function does not cause CALSymTrfRes to return a value. CALSymTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_ROFATAL	NRBG fatal error during operation.

Description This function performs an on-demand reseed of the instantiated DRBG with additional input pointed at by puiAddIn of specified 32-bit word length, uiAddInLen. If additional input is not used, set puiAddIn to SAT_NULL and uiAddInLen to zero.

CALDRBGGenerate

Syntax

```
rReturn=CALDRBGGenerate(puiAddIn, uiAddInLen,
bPredResist, puiOut, uiOutBlocks)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
const SATUINT32_t *	puiAddIn	Pointer to additional input
SATUINT32_t	uiAddInLen	Length, in words, of additional input
SATBOOL	bPredResist	If SAT_TRUE, requests prediction resistance
SATUINT32_t *	puiOut	Pointer to buffer to store random DRBG output
SATUINT32_t	uiOutBlocks	Non-zero number of 128-bit blocks to generate

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FAIL	Operation failed because the DRBG is not instantiated.
SATR_BADPARAM	puiOut is SAT_NULL
SATR_BADLEN	uiOutBlocks is either zero or exceeds the capability of the implementation.
SATR_BADLEN	Additional input length exceeds capability of implementation.
SATR_ROFATAL	NRBG fatal error during operation.
SATR_FNP	Function implementation not populated.

CALSymTrfRes will place the result from this operation in the specified location, and the SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.
SATR_ROFATAL	NRBG fatal error during operation.

Description This function generates random data using the instantiated DRBG. Additional input is pointed at by puiAddIn of the 32-bit word length specified by uiAddInLen. If additional input is not used, set puiAddIn to SAT_NULL and uiAddInLen to zero. The DRBG output of uiOutBlocks 128-bit blocks is stored at the location pointed at by puiOut. Prediction resistance can be requested by setting bPredResist to SAT_TRUE, which will initiate a reseed before the generate is performed.

CALDRBGUninstantiate

Syntax `rReturn=CALDRBGUninstantiate()`

Parameters None.

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FAIL	Operation failed because the DRBG is not instantiated.
SATR_FNP	Function implementation not populated.

This function does not cause CALSymTrfRes to return a value. CALSymTrfRes SATR rReturn values resulting from this function are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful completion.

Description This function uninstantiates the currently instantiated DRBG.

CALDRBGGetCtx

Syntax `rReturn=CALDRBGGetCtx(drbgCtxExt)`

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
DRBGCTXPTR	drbgCtxExt	Pointer to DRBG context

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.

This function does not use CALSymTrfRes.

Description This function stores the current DRBG context into the DRBG context pointed at by drbgCtxExt. Note that this function assumes that a DRBG is instantiated, and does not uninstantiate the DRBG from CAL. A full unload of the DRBG is shown in Example 3-9, DRBG Unload.

Example 3-9 DRBG Unload

```
DRBGCTX drbgCTX;

/* Unload currently instantiated DRBG. */
rStatus=CALDRBGGetCtx(&drbgCTX);
rStatus=CALDRBGUninstantiate();
```

CALDRBGLoadCtx

Syntax `rReturn=CALDRBGLoadCtx(drbgCtxExt)`

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
DRBGCTXPTR	drbgCtxExt	Pointer to DRBG context

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.

This function does not use CALSymTrfRes.

Description This function loads the CAL DRBG context with the DRBG context pointed at by drbgCtxExt. After executing the load, the function zeroizes the security relevant storage from the context.

3.14 Key Wrap and Unwrap

The CAL key wrap and unwrap functions implement NIST SP800-38F AES key wrap and unwrap functions, both for unpadded and padded keys. Each of the functions listed in Table 3-20 is described in this section. The key wrap function is used to encrypt a key using another key, referred to as a key encryption key (KEK), so that the encrypted key may be stored in an unprotected location or transmitted over an insecure channel. The unwrap function is used to decrypt a wrapped key. These functions are particularly useful when there is insufficient secure storage for the key material required for an application. .

Table 3-20: TeraFire CAL Key Wrap/Unwrap Functions

Function	*Description*	*Page*
CALSymKw	Wraps and unwraps keys using KW (no padding)	200
CALSymKwp	Wraps and unwraps keys using KWP (padded)	202

CALSymKw

Syntax

```
pReturn=CALSymKw(eSymType, puiKEK, puiInKey,
puiOutKey, uiLen, bWrap)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm/key size for KEK
const SATUINT32_t *	puiKEK	Pointer to key encryption key
const SATUINT32_t*	puiInKey	Key text input
SATUINT32_t*	puiOutKey	Key text output
SATUINT32_t	uiLen	Length of key to wrap/unwrap in semi-blocks
SATBOOL	bWrap	Wrap if SAT_FALSE, unwrap if SAT_TRUE

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Length of key to wrap/unwrap is either zero or a length that exceeds the capacity of the implementation.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_MSBICV1	For key unwrap, the ICV1 integrity check has failed, and the result of the unwrap should not be used.

This function does not use CALSymTrfRes.

Description This function performs and AES key wrap or unwrap without padding (NIST SP800-38F KW-AE/KW-AC), as specified by the parameter bWrap, using the key encryption key provided by puiKEK. The encryption algorithm selected with the eSymType parameter must be among those listed in Table 3-21; however, a specific implementation may only support a subset of the values listed in Table 3-21. The key text input is specified by puiInKey, and the key text output is specified by puiOut-

Key. The NIST SP800-38F specification defines the length of the key in semiblocks, which are 64-bit blocks. Therefore, the length of the key to be wrapped must be an integer multiple of 8-bytes (2-words). The uiLen parameter is the length of the key payload in semiblocks. For key wrap operations, the result of the wrap will be uiLen+1 semiblocks due to the inclusion of the integrity check value (ICV1). Likewise, for key unwrap operations the input text will be uiLen+1 semiblocks.

Table 3-21: CALSymKw/CALSymKwp eSymType Parameter Valid Values

Value	*Description*
SATSYMTYPE_AES128	AES 128-bit key
SATSYMTYPE_AES192	AES 192-bit key
SATSYMTYPE_AES256	AES 256-bit key

CALSymKwp

Syntax

```
pReturn=CALSymKwp(eSymType, puiKEK, puiInKey,
puiOutKey, uiLen, bWrap)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATSYMTYPE	eSymType	Encryption algorithm/key size for KEK
const SATUINT32_t *	puiKEK	Pointer to key encryption key
const SATUINT32_t*	puiInKey	Key text input
SATUINT32_t*	puiOutKey	Key text output
SATUINT32_t	uiLen	Length of key to wrap in bytes for wrap, or semiblocks for unwrap
SATBOOL	bWrap	Wrap if SAT_FALSE, unwrap if SAT_TRUE

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADLEN	Length of key to wrap/unwrap is either zero or a length that exceeds the capacity of the implementation.
SATR_BADTYPE	Unsupported encryption algorithm.
SATR_MSBICV2	For key unwrap, the ICV2 integrity check has failed, and the result of the unwrap should not be used.

This function does not use CALSymTrfRes.

Description This function performs an AES key wrap or unwrap with padding (NIST SP800-38F KWP-AE/KWP-AC), as specified by the parameter bWrap, using the key encryption key provided by puiKEK. The encryption algorithm selected with the eSymType parameter must be among those listed in Table 3-21; however, a specific implementation may only support a subset of the values listed in Table 3-21. The key text input is specified by puiInKey, and the key text output is specified by puiOut-

Key. For wrap operations, the length of the key payload to be wrapped is specified in bytes by uiLen; the resulting wrapped key size will be a number of semiblocks (8-byte/2-word blocks) given by $\lceil uiLen/8 \rceil + 1$. For key unwrap operations, the length of the wrapped key input text is specified in semiblocks by uiLen.

3.15 Context Management Functions

Starting with CAL v2.1, CAL provides context management functions that allow context switching for some CAL operations. Automatic context management is integrated with those CAL functions that support context switching. Contexts may also be managed manually using the functions documented in this section.

Table 3-22: TeraFire CAL Context Management Functions

Function	*Description*	*Page*
CALContextCurrent	Retrieves current resource context	205
CALContextLoad	Loads resource with context	206
CALContextRemove	Removes context association from resource	207
CALContextUnload	Unloads context from resource	208

CALContextCurrent

Syntax `pContext=CALContextCurrent(hResource)`

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
const SATRESHANDLE	hResource	Computational resource handle

Returns SATRESCONTEXTPTR pContext

This function does not use CALSymTrfRes.

Description Returns pointer to current context associated with a resource. If the return value is SAT_NULL, then no context is currently associated with the resource.

CALContextLoad

Syntax `rReturn=CALContextLoad(hResource,pContext)`

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
const SATRESHANDLE	hResource	Computational resource handle
SATRESCON-TEXTPTR const	pContext	Pointer to a context

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_FAIL	Current context needs to be unloaded before loading a new context.
SATR_BADHANDLE	The handle is out of range.

This function does not use CALSymTrfRes.

Description This function associates a context with a computational resource and loads the context from the location pointed at by pContext into the resource. The computational resource is usually a hardware accelerator, but may also be a software implementation of an algorithm.

If the resource is currently in use by another context, then the context load will fail. Before loading a new context, the previous context should be unloaded or removed.

CALContextRemove

Syntax `rReturn=CALContextRemove(hResource)`

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
const SATRESHANDLE	hResource	Computational resource handle

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_BADHANDLE	The handle is out of range.

This function does not use CALSymTrfRes.

Description This function sets the current context of the computational resource handle to SAT_NULL.

CALContextUnload

Syntax `rReturn=CALContextUnload(hResource)`

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
const SATRESHANDLE	hResource	Computational resource handle

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.
SATR_FAIL	There is no context to unload or the handle is out of range.

This function does not use CALSymTrfRes.

Description This function unloads the current context from a computational resource selected by hResource to the location of the context provided when the context was loaded using CALContextLoad (see *CALContext-Load* on page 206). After unloading the context, the current context of the computational resource handle is set to SAT_NULL.

3.16 Key Derivation Functions

Key derivation functions (KDFs) derive a secret key from a secret value or values.

CALKeyTree

Syntax

```
rReturn=CALKeyTree(bPathSizeSel, puiKey,
uiOpType, puiPath, puiKeyOut)
```

Parameters Parameters to this function are listed below.

Type	*Identifier*	*Description*
SATBOOL	bPathSizeSel	256-bit path nonce if SAT_TRUE, 128-bit path nonce if SAT_FALSE
const SATUINT32_t *	puiKey	Pointer to 256-bit key input
const uint8_t	uiOpType	8-bit user operation type
const SATUINT32_t*	puiPath	Pointer to path nonce input
SATUINT32_t *	puiKeyOut	Pointer to key result

Returns SATR rReturn values produced by this call are listed below.

Return Value	*Causes*
SATR_SUCCESS	Successful execution.
SATR_FNP	Function implementation not populated.

This function does not use CALSymTrfRes.

Description This function allows two parties holding a 256-bit secret key production key, puiKey, to derive a session key using a 128-bit or 256-bit path nonce, puiPath, with size determined by bPathSizeSel, combined with a user-defined operation type, uiOpType. The output of this function is a 256-bit (eight 32-bit word) key stored at the location pointed at by puiKeyOut.

In typical use cases, the path nonce may be communicated in the clear, allowing two parties to establish a secret ephemeral session key without using more expensive public key cryptography techniques.

CHAPTER 4 CAL Data Types

This chapter defines the data types used by the CAL. In this chapter, types may be described by the following terms, which are described below.

- A *handle* is a reference to an object. It is not necessarily a pointer. In general, all handle types will have an associated named value to represent a null or invalid handle. This named value is created by taking the type name and appending "_NULL". For example, for the type SATSYMKEYHANDLE, the null value is SATSYMKEYHANDLE_NULL.
- An *enumerated type* is a type with a finite set of non-ordinal valid values.

In general, a named pointer to a particular data type is found in the library by appending "PTR" to the end of the type name. For example, the a pointer to a value of the type SATSYMKEYHANDLE is SATSYMKEYHANDLEPTR.

4.1 Type Descriptions

SATASYMKEYTYPE

An enumerated type specifying the key type for an asymmetric key encryption/decryption algorithm. Valid values for this type are specified in Table 4-1.

Table 4-1: SATASYMKEYTYPE Valid Values

Value	*Description*
SATASYMKEYTYPE_NULL	Invalid key type
SATASYMKEYTYPE_RSA_PUBLIC	RSA public key
SATASYMKEYTYPE_RSA_PRIVATE	RSA private key
SATASYMKEYTYPE_EC_PUBLIC	EC public key
SATASYMKEYTYPE_EC_PRIVATE	EC private key

SATBOOL

An enumerated type with values SAT_TRUE and SAT_FALSE.

SATRSAENCTYPE

An enumerated type specifying an RSA encoding, or encryption block formatting type (see FIPS 186-4, *Digital Signature Standard* for more information). Valid values for this type are specified in Table 4-2.

Table 4-2: SATRSAENCTYPE Valid Values

Value	*Description*
SATRSAENCTYPE_NULL	No encoding/padding
SATRSAENCTYPE_PKCS	RSASSA-PKCS1-v1.5
SATRSAENCTYPE_ANSI	ANSI X9.31
SATRSAENCTYPE_PSS	RSASSA-PSS

SATHASHTYPE

An enumerated type specifying a cryptographic hash algorithm. Valid values for this type are specified in Table 4-3.

Table 4-3: SATHASHTYPE Valid Values

Value	*Description*
SATHASHTYPE_NULL	Invalid hash algorithm type
SATHASHTYPE_SHA1	FIPS 180-4 SHA-1
SATHASHTYPE_SHA224	FIPS 180-4 SHA-224
SATHASHTYPE_SHA256	FIPS 180-4 SHA-256
SATHASHTYPE_SHA384	FIPS 180-4 SHA-384
SATHASHTYPE_SHA512	FIPS 180-4 SHA-512
SATHASHTYPE_SHA512_224	FIPS 180-4 SHA-512/224
SATHASHTYPE_SHA512_256	FIPS 180-4 SHA-512/256

SATMACTYPE

An enumerated type specifying a message authentication code algorithm. Valid values for this type are specified in Table 4-4.

Table 4-4: SATMACTYPE Valid Values

Identifier	*Algorithm*
SATMACTYPE_NULL	NULL type
SATMACTYPE_SHA1	HMAC SHA-1
SATMACTYPE_SHA224	HMAC SHA-224
SATMACTYPE_SHA256	HMAC SHA-256
SATMACTYPE_SHA384	HMAC SHA-384
SATMACTYPE_SHA512	HMAC SHA-512
SATMACTYPE_SHA512_224	HMAC SHA-512/224
SATMACTYPE_SHA512_256	HMAC SHA-512/256
SATMACTYPE_AESCMAC128	AES-CMAC-128
SATMACTYPE_AESCMAC192	AES-CMAC-192
SATMACTYPE_AESCMAC256	AES-CMAC-256
SATMACTYPE_AESGMAC	AES GMAC

SATR

An enumerated type specifying return values for CAL functions. Valid values for this type are specified in Table 4-5.

Table 4-5: SATR Valid Values

Value	*Description*
SATR_BADCONTEXT	Bad context.
SATR_BADHANDLE	Bad context handle.
SATR_BADHASHLEN	Bad hash length.
SATR_BADHASHTYPE	Bad hash type.
SATR_BADLEN	Bad length parameter.
SATR_BADMACLEN	Bad MAC length.
SATR_BADMACTYPE	Bad MAC type.
SATR_BADMACLEN	Bad MAC length.
SATR_BADMODE	Bad operating mode.
SATR_BADPARAM	Bad parameter.
SATR_BADTYPE	Bad type.
SATR_BUSY	Busy
SATR_DCMPARMB	Point decompression with b parameter greater than modulus.
SATR_DCMPARMP	Point decompression with modulus not of the form 4n+3.
SATR_DCMPARMX	Point decompression with X parameter not in range [0, modulus-1].
SATR_FAIL	Unspecified failure.
SATR_FNP	Function non present/populated in implementation.
SATR_KEYSFULL	Key store full
SATR_LSB0PAD	LS bytes of zero padding
SATR_MSBICV1	MS block of ICV1
SATR_MSBICV2	MS block of ICV2
SATR_PADLEN	Pad length.
SATR_PAF	Point at infinity.
SATR_PARITYFLUSH	Parity error flush.
SATR_ROFATAL	NRBG fatal error.
SATR_SIGNFAIL	Signature fail.

Table 4-5: SATR Valid Values

Value	*Description*
SATR_SIGNPARMD	Signature D parameter not in range.
SATR_SIGNPARMK	Signature K parameter not in range.
SATR_SUCCESS	Successful completion of operation.
SATR_VALIDATEFAIL	Point validate fail - point not on curve.
SATR_VALPARMB	b parameter greater than modulus.
SATR_VALPARMX	X parameter not in range [1, modulus-1].
SATR_VALPARMY	Y parameter not in range [1, modulus-1].
SATR_VERIFYFAIL	Signature verification failure
SATR_VERPARMR	Signature verification parameter R not in range [1, modulus-1].
SATR_VERPARMS	Signature verification parameter S not in range [1, modulus-1].

SATRESCONTEXT

A resource context structure.

SATRESHANDLE

A handle for a functional resource. A resource handle may correspond to a hardware or a software resource.

Value	*Description*
SATRES_DEFAULT	Default resource handle.
SATRES_CALSW	Resource handle for CAL-SW software.

SATSYMMODE

An enumerated type specifying the encryption/decryption mode for a symmetric key encryption/decryption algorithm. Valid values for this type are specified in Table 4-6.

Table 4-6: SATSYMMODE Valid Values

Identifier	*Algorithm*
SATSYMMODE_ECB	Electronic code book
SATSYMMODE_CBC	Cipher block chaining

Table 4-6: SATSYMMODE Valid Values (Continued)

Identifier	*Algorithm*
SATSYMMODE_CFB	Cipher feedback
SATSYMMODE_OFB	Output feedback
SATSYMMODE_CTR	Counter mode
SATSYMMODE_GCM	Galois counter mode
SATSYMMOD_GHASH	Galois hash mode

SATSYMTYPE

An enumerated type specifying the key type for a symmetric key encryption/decryption algorithm. Valid values for this type are specified in Table 4-7.

Table 4-7: SATSYMTYPE Valid Values

Value	*Description*
SATSYMTYPE_NULL	Invalid key type
SATSYMTYPE_AES128	AES 128-bit key
SATSYMTYPE_AES192	AES 192-bit key
SATSYMTYPE_AES256	AES 256-bit key
SATSYMTYPE_AESKS128	AES with 128-bit split key
SATSYMTYPE_AESKS192	AES with 192-bit split key
SATSYMTYPE_AESKS256	AES with 256-bit split key

SATUINT8_t

An 8-bit unsigned integer. Equivalent to `unsigned char` in C.

SATUINT16_t

A 16-bit unsigned integer. Equivalent to `unsigned short` in C.

SATUINT32_t

A 32-bit unsigned integer. Equivalent to `unsigned long` in C.

SATUINT64_t

A 64-bit unsigned integer. Equivalent to `unsigned long long` in C.


ATHENA

INDEX

C

D

E

H

I

K

M

N

P

R

S

Z