Quantum Secure Cryptographic Library

QSC Integration and Evaluation Guide

Revision: 1.0
Date: October 15, 2025

Introduction

Quantum-safe infrastructure demands more than just a set of cryptographic primitives; it requires a **cohesive toolkit** that unifies symmetric and asymmetric ciphers, hash and MAC functions, random number generators, key derivation, and even network and threading utilities into a single, portable codebase. The **Quantum Secure Cryptographic (QSC) Library,** developed by Quantum Resistant Cryptographic Solutions (QRCS), answers this need. Written entirely in MISRA-compliant C, QSC aims to provide **long-term secure primitives and infrastructure** for embedded devices, desktop applications, servers and cryptographic protocols. It is the foundation for QRCS protocols such as AERN, UDIF, QSTP, QSMP, SATP and MPDC, but is also designed to be consumed as a standalone library.

This guide is an introduction to QSC; there is an extensive help library included with the library, along with a complete technical specification that enumerates protocols and API. This document breaks the library down by functional area (symmetric ciphers, hashes and MACs, randomness and KDFs, asymmetric ciphers and signatures, system utilities, network stack) and analyses the **API style**, **test infrastructure** and **practical integration**. The goal is to equip engineers and researchers with a preliminary understanding of how QSC works, how to integrate it into diverse environments, and where its strengths and limitations lie.

Design Philosophy and Architecture

Several core design principles underpin QSC:

1. **Modular, self-contained components:** Each primitive or subsystem is implemented in a dedicated module. Functions take a pointer to a context/state as their first argument and return status codes where appropriate. This uniform design reduces accidental misuse and clarifies ownership of sensitive data.
2. **MISRA compliance:** The library is written according to MISRA guidelines for safety and security, making it suitable for regulated sectors such as military, automotive, and aerospace.

3. **Unified API patterns:** Functions share verb-based naming (initialize, update, transform, sign, verify, encapsulate, decapsulate) and consistent parameter ordering (context first, then input/output buffers, length, and options). Enumerations and macros define parameter sets (e.g., key sizes, security levels) to avoid magic numbers.
4. **Extensive platform support and SIMD optimizations:** QSC detects CPU features at runtime and can be configured to use AVX, AVX2 and AVX-512 intrinsics when available. The same API therefore scales from microcontrollers to x86-64 servers without code changes.
5. **Secure memory management:** Custom memory allocators, constant-time comparison and copy routines, and secure zeroization ensure sensitive data are not leaked or left in memory.
6. **Integrated system utilities:** Beyond cryptography, QSC includes a full IPv4/IPv6 socket library, asynchronous threading and event primitives, generic collections, memory and integer tools, file and folder utilities, and CPU feature detection. These enable building complete secure applications without external dependencies.

The remainder of this guide explores each of these functional areas in detail.

1 Symmetric Cipher Modules

QSC offers a wide range of stream and block cipher modes, many of which implement **authenticated encryption with associated data (AEAD)**. Each cipher follows a predictable pattern: a **state structure** to hold internal variables and a **key parameter structure** to hold keys and nonces; an initialize() function to set up state; optional set_associated() to bind associated data to the authentication tag; one or more transform() functions to encrypt or decrypt; and a dispose() function to securely erase sensitive material. These patterns mirror those of modern cryptographic libraries but adhere to MISRA naming and structure guidelines.

1.2 RCS (Wide-Block Rijndael)

RCS is QSC's flagship authenticated stream cipher. RCS transforms the Rijndael (AES) wide-block cipher into an **authenticated stream cipher**. Key changes include:

- **Wide-block design:** Instead of 128-bit blocks, RCS uses a 256-bit internal state. The cipher operates in counter mode with a cSHAKE-derived key schedule and a configurable number of rounds: **22 rounds for 256-bit keys** and **30 rounds for 512-bit keys**.
- **cSHAKE key expansion:** The Rijndael (AES) key schedule is replaced with a cSHAKE-based expansion, which feeds unique round keys into each round.

- **KMAC authentication:** As with CSX, RCS employs KMAC for AEAD. Each message includes an authentication tag bound to the nonce, associated data and ciphertext.

The qsc_rcs_state holds the round keys, Keccak state, nonce, counter and mode flag. Constants define block sizes and MAC sizes. The API pattern mirrors CSX: initialization, optional associated data, transformation and disposal.

Use cases: RCS offers AES-like security semantics but with a wide state, a secure and tweakable key-schedule (cSHAKE) and strong authentication. It is appropriate for high-throughput networks, and implementations requiring strong post-quantum grade symmetric security. The integration of cSHAKE and KMAC ensures robust key derivation and authentication in a future-proof design.

1.2 CSX-512 (Wide-Block ChaCha Variant)

CSX-512 builds on the ChaCha design but extends the key and state:

- **512-bit key and 1024-bit state:** CSX uses 64-bit words, doubling the key size relative to ChaCha and expanding the internal state to 1024 bits. This yields an enormous key space and resilience against quantum Grover-style search.
- **40 rounds:** To compensate for the increased state, the number of quarter-round operations is increased from ChaCha's 20 to 40.
- **KMAC-based authentication:** Instead of Poly1305, CSX integrates Keccak-based KMAC for authentication. Associated data and ciphertext are processed in KMAC's duplex construction to produce a MAC tag.

The qsc_csx_state stores the 1024-bit cipher state, a 1600-bit Keccak state for KMAC, and mode flags. Key parameters (key, nonce, info, lengths) are encapsulated in qsc_csx_keyparams. After initialization, the caller may supply associated data. The transform() routine processes plaintext or ciphertext, updating the Keccak state and producing or verifying the MAC. Finally, dispose() securely clears the state.

Use cases: CSX is intended for high-assurance environments where long-term confidentiality and integrity are paramount. Its wide block and long keys make it suitable for firmware images, disk encryption and high-throughput network streams.

1.3 ChaCha and Poly1305

QSC includes ChaCha20/Poly1305 with AVX2/AVX-512 optimizations. A qsc_chacha_state holds the key, length and nonce. Constants define block size (64 bytes), key size (128/256 bits), nonce size (96 bits) and rounds (20). Functions qsc_chacha_initialize(), qsc_chacha_transform() and qsc_chacha_dispose() follow the standard pattern.

1.4 Advanced Encryption Standard (AES)

QSC implements AES with several modes: **ECB**, **CBC**, **CTR**, **GCM** and **Hash-Based Authentication (HBA)**. The qsc_aes_state holds the expanded round keys and mode data. Functions include:

- qsc_aes_initialize(state, keyparams, encrypt, mode): Prepares state with key and IV/nonce.
- qsc_aes_cbc_encrypt() / qsc_aes_cbc_decrypt(): Block-wise CBC encryption and decryption; an HBA variant uses KMAC for authentication and is integrated into qsc_aes_hba_* functions.
- qsc_aes_ctrle_transform(): CTR mode (little-endian counter) processing.
- qsc_aes_gcm_initialize(), qsc_aes_gcm_update(), qsc_aes_gcm_finalize(): For AEAD using GCM.
- qsc_aes_dispose() – Wipes the state.

Constants define block size (16 bytes), key sizes (128/256 bits) and HBA tag lengths. The API also includes helpers for padding and HBA parameter selection. The AES implementation uses AES-NI instructions when available, offering high throughput on modern CPUs.

These algorithms share the same initialize → transform → dispose pattern, with key parameter structures and enumerated parameter sets.

1.6 Symmetric API Patterns and Example

A typical encryption flow using QSC looks like this (C-style pseudocode):

```
// prepare key and nonce
qsc_rcs_keyparams kp = {
   .key = user_key,      // 32 or 64 bytes
   .keylen = sizeof(user_key),
   .nonce = user_nonce,  // 32 bytes
   .noncelen = sizeof(user_nonce),
   .info = NULL,
   .infolen = 0
};


qsc_rcs_state state;
```

```
// initialize for encryption
qsc_rcs_initialize(&state, &kp, true);
// optionally add AAD
qsc_rcs_set_associated(&state, header, header_len);
// encrypt plaintext to ciphertext and compute MAC
qsc_rcs_transform(&state, ciphertext, plaintext, plaintext_len);
// finalise and wipe state
qsc_rcs_dispose(&state);
```

The same sequence applies to CSX and other AEAD modes, with different state and key parameter types.

2 Hash Functions and Message Authentication

Cryptographic hashes and MACs are central to key derivation, integrity and authentication. QSC implements both traditional and Keccak-based functions.

2.1 Keccak/SHA-3 Family

At the core is the **Keccak** sponge. QSC exposes a low-level API for absorbing, finalizing and squeezing from a 1600-bit state. Domain IDs differentiate variants like SHA-3-256, SHAKE256, cSHAKE and KMAC. Functions include:

- qsc_keccak_absorb(state, in, inlen): Injects input into the sponge.
- qsc_keccak_finalize(state, domain): Pads and permutes the state with a domain separation tag.
- qsc_keccak_squeeze(state, out, outlen): Extracts output bytes.
- qsc_keccak_incremental_absorb(state, in, inlen) – Allows streaming input.

Higher-level wrappers implement SHA-3 (fixed length), SHAKE (extendable output), cSHAKE (customized XOF) and KMAC. For example, qsc_sha3_256_compute(out, in, inlen) produces a 256-bit digest; qsc_kmac256_compute(mac, key, keylen, data, datalen, custom, customlen) computes a KMAC tag; qsc_shake256_extract(out, outlen, seed, seedlen) yields variable-length pseudorandom output.

2.2 KMAC and QMAC

KMAC is a Keccak-based MAC specified in SP 800-185. QSC defines key sizes (128 or 256 bits) and domain IDs for KMAC. The API comprises initialize(), update() and finalize(). Because KMAC uses the same sponge as SHAKE, it integrates seamlessly with CSX and RCS for AEAD.

QMAC is a QSC-specific Keccak MAC variant that adds a finite field multiplication to reduce correlation attacks. It offers strong collision resistance and is used internally by some KDFs.

2.3 HMAC

For backward compatibility, QSC implements **HMAC** using SHA-2. Functions are qsc_hmac_initialize(), qsc_hmac_update(), qsc_hmac_finalize(). Parameter macros define block sizes and key sizes.

2.4 Key Derivation Functions

Key derivation drives every protocol built on QSC. Several KDFs are available:

- **HKDF (HMAC-based):** Implements RFC 5869. Functions qsc_hkdf_extract() and qsc_hkdf_expand() derive keys from initial key material and context.
- **CSG (CSHAKE-based DRBG):** A Keccak-based deterministic random bit generator (DRBG) that wraps cSHAKE. Functions qsc_csg_initialize(), qsc_csg_update(), and qsc_csg_generate() provide pseudorandom bits. An optional info string customises output. CSG can be set to deterministic mode (seeded) or non-dterministic (seed is injected internally from the random provider ACP).
- **HCG (SHA2 HMAC Counter Generator):** A DRBG and KDF that uses the HMAC(SHA2) as the primary pseudo random function. The qsc_hcg_state contains the persistant state; qsc_hcg_initialize(), qsc_hcg_generate() and qsc_hcg_update() manage its operation. Like CSG, HCG has both deterministic and non-deterministic modes.
- **SCB (SHAKE Cost-Based KDF):** Derived from cSHAKE, SCB supports configurable time and memory costs, similar to Argon2. Functions include qsc_scb_initialize(), qsc_scb_generate(), qsc_scb_update() and qsc_scb_dispose().

By combining these KDFs with QSC’s random providers, developers can derive keys and seeds for protocols that require both post-quantum security and configurable hardness.

3 Randomness Providers and DRBGs

Secure randomness underpins all cryptographic operations. QSC provides multiple entropy sources:

3.1 Auto-Entropy Collection Provider (ACP)

ACP combines output from hardware random number generators, such as Intel **RDRAND** or AMD **RDSEED**, with the output from operating system entropy (a hash of timers, statistics, process handles et. al), and output from the system random provider, hashing the three with cSHAKE-512, making this a very strong random provider function, and the default and recommended seed provider. The function qsc_acp_generate(*output, length*) produces random bytes, and qsc_acp_uint16(), qsc_acp_uint32(), andqsc_acp_uint64(), can each provide a single random integer.

3.2 Cryptographic Service Provider (CSP) and Random Data Provider (RDP)

CSP wraps platform cryptographic APIs (e.g., Windows CryptGenRandom and Linux /dev/urandom) to supply secure random bytes. RDP provides a platform-agnostic fallback. Both follow the same initialize(), generate(), dispose() pattern. Both fallback providers CSP and RDP use the same generate and integer api as ACP.

3.3 Deterministic and Extendable Generators

- **CSG:** A custom SHAKE-based (Keccak) DRBG; it uses a seed and optional info string to produce arbitrarily long output, making it suitable for key and nonce derivation.
- **HCG:** A SHA2 HMAC based generator. Like the generator CSG it can be set to either deterministic (a starting input seed), or non-deterministic (random seed injection at initialization and pre-defined intervals) modes of operation.
- **SCB:** A cSHAKE-based KDF/DRBG with variable cost parameters; recommended for deriving session keys from user secrets in protocols like SIAP and SATP. It ensures that deriving an incorrect key is expensive for attackers.

4 Asymmetric Cryptography and Signatures

Modern security requires hybrid cryptography that combines classical and post-quantum primitives. QSC includes several asymmetric algorithms with consistent APIs.

4.1 Key Encapsulation Mechanisms (KEMs)

- **Kyber:** A lattice-based KEM selected by NIST (FIPS 203). QSC implements qsc_kyber_generate_keypair(), qsc_kyber_encapsulate(), and qsc_kyber_decapsulate(). Different parameter sets correspond to NIST security levels (512, 768, 1024 bits, and an additional experimental parameter 1280), with key and ciphertext sizes scaling accordingly.
- **McEliece:** A long-term security code-based KEM with large public keys (hundreds of kilobytes). Functions follow the same pattern: generate, encapsulate, decapsulate. It offers resistance to quantum and side-channel attacks.

- **Classical ECDH:** For backward compatibility, QSC provides Diffie–Hellman on Curve25519. Functions include qsc_ecdh_generate_keypair() and qsc_ecdh_compute_key().

4.2 Digital Signature Schemes

- **Dilithium:** A lattice-based signature scheme (FIPS 204). QSC functions qsc_dilithium_generate_keypair(), qsc_dilithium_sign() and qsc_dilithium_verify() encapsulate key generation, signing and verification. Parameter sets (NIST PQC levels 2/3/5) yield different security and performance trade-offs.
- **SPHINCS+:** A stateless hash-based signature (FIPS 205) that promises long-term security. Functions include key generation, signing and verification. SPHINCS+ signatures are large (tens of kilobytes) but provide strong security and minimal reliance on structure.
- **Classical ECDSA:** Provided for compatibility, using Curve25519. Functions include qsc_ecdsa_generate_keypair(), qsc_ecdsa_sign(), qsc_ecdsa_verify().

The consistent API pattern across KEMs and signature schemes allows swapping algorithms with minimal code changes, easing transition to PQC.

5 System Utilities: Networking, Threads and Storage

Cryptography rarely exists in isolation; real applications require networking, concurrency and file system support. QSC provides these through a set of utilities that work uniformly across Windows, Linux and embedded environments.

5.1 Networking

QSC's networking layer wraps IPv4 and IPv6 sockets with cross-platform abstractions. Functions include:

- qsc_socket_create(family, type, protocol) – Creates a socket with given family (AF_INET/AF_INET6), type (SOCK_STREAM/SOCK_DGRAM) and protocol.
- qsc_socket_bind(sock, address, port) – Binds a socket to a local address and port.
- qsc_socket_listen(sock, backlog) – Listens for incoming connections.
- qsc_socket_accept(sock, new_sock, client_addr) – Accepts a connection on a listening socket.
- qsc_socket_send(sock, data, len) and qsc_socket_receive(sock, data, len) – Blocking send and receive.

- qsc_socket_close(sock) – Closes a socket.

On Windows, these functions wrap Winsock; on Unix-like systems, they wrap Berkeley sockets. QSC also includes helper functions for retrieving local IP addresses, enumerating network interfaces and converting addresses to strings. A **thread-safe queue** facilitates asynchronous send/receive operations, and the network layer integrates with QSC's thread pool to scale across cores. The sockets library is fully featured including a multi-threaded server and extensive support functions.

5.2 Threading and Events

Building high-performance protocols requires concurrency. QSC offers:

- **Threads:** The threading API abstracts OS thread creation. Thread attributes control stack size and scheduling. The async API includes functions like; thread creation, suspend, resume, wait, sleep, and various mutex operations.
- **Events:** The event API implements cross-platform condition variables. They enable coordination between producer and consumer threads.
- **Thread pool:** A dynamic pool manages a collection of worker threads and a job queue, allowing tasks like encryption, signature verification or network handling to be dispatched concurrently. The pool automatically scales with CPU cores and can be paused or resumed.

These abstractions simplify building asynchronous network servers without tying developers to a particular OS.

5.3 File, Console and System Utilities

QSC includes cross-platform wrappers for file I/O (open, read, write, close), directory listing, creation and deletion. These functions ensure that file operations behave consistently on all supported platforms. Console utilities support interactive applications, including reading passwords without echo. System utilities provide CPU feature detection (CPUID), high-resolution timers and random seeding.

5.4 Secure Memory and SIMD

The secure memory API allocates protected pages, performs constant-time comparisons, XORs and copies, securely wipes memory, and a variety of other memory related functions. When compiled with AVX/AVX2/AVX-512 support, these operations leverage vector instructions to improve throughput while maintaining constant-time behavior.

6 API Patterns and Best Practices

Because QSC exposes a large number of functions, recognizing patterns reduces cognitive load:

1. **State-first signature:** All cryptographic operations with persistent state take a pointer to a state or context as their first parameter. This pointer must be initialized once and later disposed of. This pattern reduces the risk of accidentally reusing old keys or leaking state.
2. **Verb-based functions:** initialize, dispose, set_associated, transform, generate, update, sign, verify, encapsulate, decapsulate describe exactly what the function does.
3. **Enumerations for mode selection:** Ciphers, hash functions and signature schemes accept enumerations for parameter sets and modes (e.g., qsc_aes_cipher_128, qsc_keccak_rate_256). This prevents errors when specifying sizes.
4. **Key parameter structures:** For ciphers and KEMs, keys and nonces are passed via dedicated structs, which include lengths and optional info fields. This encourages correct length management and reduces pointer confusion.
5. **Separate initialization and processing:** The design emphasizes explicit separation between preparing a context and performing operations. Developers should not call transform functions on uninitialized contexts.
6. **Secure disposal:** Always call the dispose() function when finished with a context to wipe sensitive data. This is critical in regulated industries where residual keys are unacceptable.

By adhering to these practices, developers can integrate QSC securely and effectively.

7 Testing Infrastructure: CAVP, QSCTest and QSCNETCWTEST

QSC's reliability is underpinned by a comprehensive test suite. The library includes several projects specifically for **algorithm validation** and **functional testing**. These tests exercise all supported primitives using published test vectors and ensure correct implementation across platforms.

7.1 Cryptographic Algorithm Validation Program (CAVP) Tests

The **CAVP project** (under Source/CAVP) implements automated tests against NIST's Cryptographic Algorithm Validation Program (CAVP) vectors. The project consists of a main driver (cavp.c) and algorithm-specific modules (e.g., cavp_aes.c, cavp_sha2.c, cavp_sha3.c, cavp_kyber.c, cavp_dilithium.c, cavp_sphincsplus.c). Each module reads **.rsp files** containing known-answer test (KAT) vectors for different modes and parameter sets, computes results using QSC functions and compares them to the expected values.

The main() function in cavp.c prints system capabilities (e.g., AES-NI and AVX support) and then calls each algorithm's test harness. For example, cavp_aes_run() exercises AES encryption and decryption in CBC, CTR, GCM and other modes, iterating through hundreds of test vectors across key sizes. The AES test function parses lines such as KEY =, PLAINTEXT = and CIPHERTEXT = from response files and uses QSC functions (qsc_aes_initialize(), qsc_aes_cbc_encrypt_block(), qsc_aes_cbc_decrypt_block()) to verify correctness.

The CAVP suite also includes **signature and KEM validation**: cavp_kyber_run() encapsulates and decapsulates with known secrets; cavp_dilithium_run() signs and verifies messages; cavp_sphincsplus_run() validates hash-based signatures. Each module reports pass/fail status and halts on any discrepancy.

At the root of Source/CAVP/KAT are subdirectories for **AES**, **MLDSA** (Dilithium), **MLKEM** (Kyber), **SHA2**, **SHA3**, and **SLHDSA** (SPHINCS+). Each contains .rsp files for different test categories: Galois/Counter Mode (GCM), Counter (CTR), Cipher Block Chaining (CBC), GHASH, and message digest test vectors. By shipping these vectors, QSC allows developers to independently verify the correctness of algorithms and to produce new test reports if needed, and a provides fully prepared architecture for FIPS certification testing.

7.2 QSCTest - General Library Tests

The **QSCTest** project (under Source/QSCTest) exercises a broad cross-section of the library. It contains modules for AES algorithm validation (AESAVS), post-quantum cryptography (NPQC), random number generation and hash functions. The NPQC folder includes test vectors (*.rsp files) for **Dilithium** and **Kyber** across several parameter sets (e.g., dilithium-2544.rsp, kyber-1632.rsp). These files allow the test harness to feed known inputs into QSC and verify outputs.

An example from the .NET wrapper demonstrates how this test harness is structured. In CryptoTest.cs, test functions convert hex strings to byte arrays, call QSC's .NET API to encrypt or hash, and compare results to expected outputs. For instance, TestAES256CBC() sets up a key and IV, encrypts a four-block message with AES-CBC, and then decrypts it to verify roundtrip correctness. Similarly, TestChaCha256() tests the ChaCha stream cipher and TestSHA2256() verifies the SHA-256 implementation. These tests ensure the C library's correctness when wrapped for managed languages.

7.3 QSCNETCWTEST - .NET Cryptographic Wrapper Tests

For developers using QSC's C#/.NET wrappers, the **QSCNETCWTEST** project provides integration tests. Files such as CryptoTest.cs, Utilities.cs and Program.cs compile into a test executable that verifies AES, ChaCha, SHA-2 and other primitives against known vectors. These tests mirror those in the CAVP suite but run through the .NET bindings, ensuring that the API translation between C and C# does not introduce errors. The tests are run from Program.cs and report pass/fail for each algorithm.

7.4 Running the Test Suite

The test projects are organized to run independently. To build and run them:

1. **CAVP:** Compile the CAVP project (e.g., Visual Studio solution QSCCAVP.sln or GCC/Clang equivalent). Ensure the KAT files remain in their relative directories. Run the cavp executable. The program prints system features (AES-NI, AVX) and sequentially runs each algorithm's test harness.
2. **QSCTest:** Build the QSCTest.sln solution. The test harness will read response files from its data folders and validate AES, ChaCha, SHA, Kyber, Dilithium and random functions.
3. **QSCNETCWTEST:** Build the C# project. Running it executes unit tests for AES-CBC/CTR, ChaCha20 and SHA-2; results are printed to the console.

Running these suites regularly during integration is strongly recommended. They not only detect regressions but also serve as **executable documentation**, illustrating correct API usage and parameter setup.

8 Practical Integration Considerations

8.1 Selecting Algorithms and Parameters

QSC exposes a rich set of primitives. When integrating, consider:

- **Security level:** Post-quantum algorithms (Kyber, Dilithium, SPHINCS+, McEliece) offer at least 128-bit security against quantum adversaries. Classical algorithms (AES-256, ChaCha20, SHA-3) remain robust against classical attacks and may be used for hybrid schemes. Post quantum ciphers and protocols (RCS, CSX, SCB) are designed for long-term security in high-security enclaves.
- **Performance:** Wide-block ciphers (CSX, RCS) incur higher per-block overhead but offer stronger diffusion and simpler handling of variable-length inputs. AES with AES-NI offers very high throughput on x86 but may not be available on all embedded platforms. ChaCha is constant-time and efficient on general hardware.

- **Key/nonce management:** For AEAD modes, nonces must never repeat for the same key. Use QSC's random providers and KDFs to derive nonces and keys from high-entropy seeds. Ensure that key parameter structures carry the correct lengths.
- **Protocol composition:** For key exchange, choose a KEM that matches your threat model. Kyber is widely standardized; McEliece offers exceptional security at the cost of large public keys. Dilithium and SPHINCS+ are recommended for signatures; SPHINCS+ can be used for root certificates due to its minimal assumptions.

8.2 Language Bindings and Portability

The core library is written in C and can be compiled with GCC, Clang or MSVC. For other languages:

- **C++:** QSC's C headers can be included in C++ projects; functions are declared extern "C" for linkage. RAII wrappers can be written around the state structures to automate initialization and disposal.
- **.NET:** QSC includes a C# wrapper (QSCNETCW), which exposes classes like AES, CHACHA, SHA256, KYBER and DILITHIUM. The .NET API mirrors the C naming and parameter order. Example usage is found in CryptoTest.cs.
- **Python/Rust:** No official bindings are provided, but the uniform C API makes writing foreign function interface (FFI) wrappers straightforward. Careful memory management is required, ensure states are disposed of to avoid leaks.

8.3 Integrating with QRCS Protocols

QSC is the cryptographic engine for QRCS protocols such as QSMP (messaging), QSTP (tunnelling), MPDC (multi-party domain cryptosystem) and SATP (symmetric tunnelling). When integrating these protocols, the choice of primitives may be predetermined. For example, SKDP uses RCS and KMAC; QSTP uses Kyber, Dilithium and RCS; MPDC uses Kyber, Dilithium and RCS with distributed entropy. Consult each protocol's specification to align parameters.

8.4 Testing and Validation

After integrating QSC into an application, always run the test suites. Use the CAVP project to re-validate algorithms on new hardware or compiler settings. Use the QSCTest and QSCNETCWTEST projects to ensure correct linkage and behavior in your target environment. The test harnesses illustrate typical usage patterns and catch issues such as misaligned buffers or incorrect key lengths.

9 Evaluation and Conclusions

QSC stands out as a **comprehensive cryptographic framework** that extends far beyond the typical collection of algorithms. Its unified API patterns, MISRA adherence, post-quantum readiness and integrated system utilities make it appealing for organizations seeking long-term security.

Strengths:

- **Breadth and depth:** QSC implements a wide array of ciphers, hashes, MACs, KDFs, DRBGs, PQC schemes and system utilities. This reduces the need to stitch together disparate libraries and ensures consistent coding style and security assumptions.
- **Comprehensive documentation:** The specification and Doxygen pages detail constants, structures and call hierarchies. Developers can trace how each function should be used and what internal state it manages.
- **Uniform API and MISRA compliance**: Patterns such as initialize() → update() → finalize() promote safe usage. MISRA compliance adds confidence in safety-critical contexts.
- **Performance:** SIMD intrinsics and AES-NI provide high throughput; wide-block ciphers deliver strong diffusion at a moderate cost. QSC automatically adapts to hardware features.
- **Test coverage:** The inclusion of CAVP and test projects demonstrates QSC's commitment to correctness and standard compliance. Developers can run these tests to verify builds across architectures.

Conclusion

QSC's cohesive design and robust implementation make it a powerful building block for post-quantum and classical cryptography. Its use in QRCS protocols demonstrates real-world applicability.