

# LibTomCrypt Reference Manual

## 1.15

Generated by Doxygen 1.4.6

Tue Nov 21 12:28:35 2006



# Contents

<b>1</b>	<b>LibTomCrypt Hierarchical Index</b>	<b>1</b>
1.1	LibTomCrypt Class Hierarchy . . . . .	1
<b>2</b>	<b>LibTomCrypt Data Structure Index</b>	<b>3</b>
2.1	LibTomCrypt Data Structures . . . . .	3
<b>3</b>	<b>LibTomCrypt File Index</b>	<b>5</b>
3.1	LibTomCrypt File List . . . . .	5
<b>4</b>	<b>LibTomCrypt Data Structure Documentation</b>	<b>13</b>
4.1	edge Struct Reference . . . . .	13
4.2	Hash_state Union Reference . . . . .	14
4.3	ltc_cipher_descriptor Struct Reference . . . . .	15
4.4	ltc_hash_descriptor Struct Reference . . . . .	25
4.5	ltc_math_descriptor Struct Reference . . . . .	28
4.6	ltc_prng_descriptor Struct Reference . . . . .	44
4.7	Prng_state Union Reference . . . . .	48
4.8	Symmetric_key Union Reference . . . . .	49
<b>5</b>	<b>LibTomCrypt File Documentation</b>	<b>51</b>
5.1	ciphers/aes/aes.c File Reference . . . . .	51
5.2	ciphers/aes/aes_tab.c File Reference . . . . .	64
5.3	ciphers/anubis.c File Reference . . . . .	68
5.4	ciphers/blowfish.c File Reference . . . . .	83
5.5	ciphers/cast5.c File Reference . . . . .	90
5.6	ciphers/des.c File Reference . . . . .	99
5.7	ciphers/kasumi.c File Reference . . . . .	118
5.8	ciphers/khazad.c File Reference . . . . .	125
5.9	ciphers/kseed.c File Reference . . . . .	134
5.10	ciphers/noekeon.c File Reference . . . . .	141

5.11	ciphers/rc2.c File Reference	149
5.12	ciphers/rc5.c File Reference	156
5.13	ciphers/rc6.c File Reference	163
5.14	ciphers/safer/safer.c File Reference	171
5.15	ciphers/safer/safer_tab.c File Reference	181
5.16	ciphers/safer/saferp.c File Reference	183
5.17	ciphers/skipjack.c File Reference	194
5.18	ciphers/twofish/twofish.c File Reference	202
5.19	ciphers/twofish/twofish_tab.c File Reference	213
5.20	ciphers/xtea.c File Reference	214
5.21	encauth/ccm/ccm_memory.c File Reference	219
5.22	encauth/ccm/ccm_test.c File Reference	225
5.23	encauth/eax/eax_addheader.c File Reference	228
5.24	encauth/eax/eax_decrypt.c File Reference	229
5.25	encauth/eax/eax_decrypt_verify_memory.c File Reference	230
5.26	encauth/eax/eax_done.c File Reference	232
5.27	encauth/eax/eax_encrypt.c File Reference	234
5.28	encauth/eax/eax_encrypt_authenticate_memory.c File Reference	235
5.29	encauth/eax/eax_init.c File Reference	237
5.30	encauth/eax/eax_test.c File Reference	240
5.31	encauth/gcm/gcm_add_aad.c File Reference	245
5.32	encauth/gcm/gcm_add_iv.c File Reference	248
5.33	encauth/gcm/gcm_done.c File Reference	250
5.34	encauth/gcm/gcm_gf_mult.c File Reference	252
5.35	encauth/gcm/gcm_init.c File Reference	254
5.36	encauth/gcm/gcm_memory.c File Reference	256
5.37	encauth/gcm/gcm_mult_h.c File Reference	259
5.38	encauth/gcm/gcm_process.c File Reference	261
5.39	encauth/gcm/gcm_reset.c File Reference	264
5.40	encauth/gcm/gcm_test.c File Reference	265
5.41	encauth/ocb/ocb_decrypt.c File Reference	272
5.42	encauth/ocb/ocb_decrypt_verify_memory.c File Reference	274
5.43	encauth/ocb/ocb_done_decrypt.c File Reference	276
5.44	encauth/ocb/ocb_done_encrypt.c File Reference	278
5.45	encauth/ocb/ocb_encrypt.c File Reference	279
5.46	encauth/ocb/ocb_encrypt_authenticate_memory.c File Reference	281

5.47	<a href="#">encauth/ocb/ocb_init.c File Reference</a>	283
5.48	<a href="#">encauth/ocb/ocb_ntz.c File Reference</a>	286
5.49	<a href="#">encauth/ocb/ocb_shift_xor.c File Reference</a>	287
5.50	<a href="#">encauth/ocb/ocb_test.c File Reference</a>	288
5.51	<a href="#">encauth/ocb/s_ocb_done.c File Reference</a>	292
5.52	<a href="#">hashes/chc/chc.c File Reference</a>	295
5.53	<a href="#">hashes/helper/hash_file.c File Reference</a>	300
5.54	<a href="#">hashes/helper/hash_filehandle.c File Reference</a>	302
5.55	<a href="#">hashes/helper/hash_memory.c File Reference</a>	304
5.56	<a href="#">hashes/helper/hash_memory_multi.c File Reference</a>	306
5.57	<a href="#">hashes/md2.c File Reference</a>	308
5.58	<a href="#">hashes/md4.c File Reference</a>	314
5.59	<a href="#">hashes/md5.c File Reference</a>	322
5.60	<a href="#">hashes/rmd128.c File Reference</a>	329
5.61	<a href="#">hashes/rmd160.c File Reference</a>	338
5.62	<a href="#">hashes/rmd256.c File Reference</a>	348
5.63	<a href="#">hashes/rmd320.c File Reference</a>	357
5.64	<a href="#">hashes/sha1.c File Reference</a>	368
5.65	<a href="#">hashes/sha2/sha224.c File Reference</a>	374
5.66	<a href="#">hashes/sha2/sha256.c File Reference</a>	377
5.67	<a href="#">hashes/sha2/sha384.c File Reference</a>	384
5.68	<a href="#">hashes/sha2/sha512.c File Reference</a>	388
5.69	<a href="#">hashes/tiger.c File Reference</a>	394
5.70	<a href="#">hashes/whirl/whirl.c File Reference</a>	401
5.71	<a href="#">hashes/whirl/whirltab.c File Reference</a>	407
5.72	<a href="#">headers/tomcrypt.h File Reference</a>	410
5.73	<a href="#">headers/tomcrypt_argchk.h File Reference</a>	414
5.74	<a href="#">headers/tomcrypt_cfg.h File Reference</a>	416
5.75	<a href="#">headers/tomcrypt_cipher.h File Reference</a>	418
5.76	<a href="#">headers/tomcrypt_custom.h File Reference</a>	423
5.77	<a href="#">headers/tomcrypt_hash.h File Reference</a>	435
5.78	<a href="#">headers/tomcrypt_mac.h File Reference</a>	445
5.79	<a href="#">headers/tomcrypt_macros.h File Reference</a>	446
5.80	<a href="#">headers/tomcrypt_math.h File Reference</a>	449
5.81	<a href="#">headers/tomcrypt_misc.h File Reference</a>	452
5.82	<a href="#">headers/tomcrypt_pk.h File Reference</a>	455

5.83 headers/tomcrypt_pkcs.h File Reference . . . . .	457
5.84 headers/tomcrypt_prng.h File Reference . . . . .	458
5.85 mac/f9/f9_done.c File Reference . . . . .	463
5.86 mac/f9/f9_file.c File Reference . . . . .	465
5.87 mac/f9/f9_init.c File Reference . . . . .	467
5.88 mac/f9/f9_memory.c File Reference . . . . .	469
5.89 mac/f9/f9_memory_multi.c File Reference . . . . .	471
5.90 mac/f9/f9_process.c File Reference . . . . .	473
5.91 mac/f9/f9_test.c File Reference . . . . .	475
5.92 mac/hmac/hmac_done.c File Reference . . . . .	477
5.93 mac/hmac/hmac_file.c File Reference . . . . .	480
5.94 mac/hmac/hmac_init.c File Reference . . . . .	482
5.95 mac/hmac/hmac_memory.c File Reference . . . . .	485
5.96 mac/hmac/hmac_memory_multi.c File Reference . . . . .	487
5.97 mac/hmac/hmac_process.c File Reference . . . . .	489
5.98 mac/hmac/hmac_test.c File Reference . . . . .	490
5.99 mac/omac/omac_done.c File Reference . . . . .	495
5.100mac/omac/omac_file.c File Reference . . . . .	497
5.101mac/omac/omac_init.c File Reference . . . . .	499
5.102mac/omac/omac_memory.c File Reference . . . . .	501
5.103mac/omac/omac_memory_multi.c File Reference . . . . .	503
5.104mac/omac/omac_process.c File Reference . . . . .	505
5.105mac/omac/omac_test.c File Reference . . . . .	507
5.106mac/pelican/pelican.c File Reference . . . . .	509
5.107mac/pelican/pelican_memory.c File Reference . . . . .	513
5.108mac/pelican/pelican_test.c File Reference . . . . .	515
5.109mac/pmac/pmac_done.c File Reference . . . . .	517
5.110mac/pmac/pmac_file.c File Reference . . . . .	519
5.111mac/pmac/pmac_init.c File Reference . . . . .	521
5.112mac/pmac/pmac_memory.c File Reference . . . . .	524
5.113mac/pmac/pmac_memory_multi.c File Reference . . . . .	526
5.114mac/pmac/pmac_ntz.c File Reference . . . . .	528
5.115mac/pmac/pmac_process.c File Reference . . . . .	529
5.116mac/pmac/pmac_shift_xor.c File Reference . . . . .	531
5.117mac/pmac/pmac_test.c File Reference . . . . .	532
5.118mac/xcbc/xcbc_done.c File Reference . . . . .	535

5.119	mac/xcbc/xcbc_file.c File Reference . . . . .	537
5.120	mac/xcbc/xcbc_init.c File Reference . . . . .	539
5.121	mac/xcbc/xcbc_memory.c File Reference . . . . .	541
5.122	mac/xcbc/xcbc_memory_multi.c File Reference . . . . .	543
5.123	mac/xcbc/xcbc_process.c File Reference . . . . .	545
5.124	mac/xcbc/xcbc_test.c File Reference . . . . .	547
5.125	math/fp/ltc_ecc_fp_mulmod.c File Reference . . . . .	549
5.126	math/gmp_desc.c File Reference . . . . .	550
5.127	math/ltn_desc.c File Reference . . . . .	551
5.128	math/multi.c File Reference . . . . .	552
5.129	math/rand_prime.c File Reference . . . . .	554
5.130	math/tfm_desc.c File Reference . . . . .	556
5.131	misc/base64/base64_decode.c File Reference . . . . .	557
5.132	misc/base64/base64_encode.c File Reference . . . . .	560
5.133	misc/burn_stack.c File Reference . . . . .	562
5.134	misc/crypt/crypt.c File Reference . . . . .	563
5.135	misc/crypt/crypt_argchk.c File Reference . . . . .	564
5.136	misc/crypt/crypt_cipher_descriptor.c File Reference . . . . .	565
5.137	misc/crypt/crypt_cipher_is_valid.c File Reference . . . . .	566
5.138	misc/crypt/crypt_find_cipher.c File Reference . . . . .	567
5.139	misc/crypt/crypt_find_cipher_any.c File Reference . . . . .	568
5.140	misc/crypt/crypt_find_cipher_id.c File Reference . . . . .	570
5.141	misc/crypt/crypt_find_hash.c File Reference . . . . .	571
5.142	misc/crypt/crypt_find_hash_any.c File Reference . . . . .	572
5.143	misc/crypt/crypt_find_hash_id.c File Reference . . . . .	574
5.144	misc/crypt/crypt_find_hash_oid.c File Reference . . . . .	575
5.145	misc/crypt/crypt_find_prng.c File Reference . . . . .	576
5.146	misc/crypt/crypt_fsa.c File Reference . . . . .	577
5.147	misc/crypt/crypt_hash_descriptor.c File Reference . . . . .	579
5.148	misc/crypt/crypt_hash_is_valid.c File Reference . . . . .	580
5.149	misc/crypt/crypt_ltc_mp_descriptor.c File Reference . . . . .	581
5.150	misc/crypt/crypt_prng_descriptor.c File Reference . . . . .	582
5.151	misc/crypt/crypt_prng_is_valid.c File Reference . . . . .	583
5.152	misc/crypt/crypt_register_cipher.c File Reference . . . . .	584
5.153	misc/crypt/crypt_register_hash.c File Reference . . . . .	586
5.154	misc/crypt/crypt_register_prng.c File Reference . . . . .	588

5.155misc/crypt/crypt_unregister_cipher.c File Reference . . . . .	590
5.156misc/crypt/crypt_unregister_hash.c File Reference . . . . .	591
5.157misc/crypt/crypt_unregister_prng.c File Reference . . . . .	592
5.158misc/error_to_string.c File Reference . . . . .	593
5.159misc/pkcs5/pkcs_5_1.c File Reference . . . . .	594
5.160misc/pkcs5/pkcs_5_2.c File Reference . . . . .	596
5.161misc/zeromem.c File Reference . . . . .	599
5.162modes/cbc/cbc_decrypt.c File Reference . . . . .	600
5.163modes/cbc/cbc_done.c File Reference . . . . .	602
5.164modes/cbc/cbc_encrypt.c File Reference . . . . .	603
5.165modes/cbc/cbc_getiv.c File Reference . . . . .	605
5.166modes/cbc/cbc_setiv.c File Reference . . . . .	606
5.167modes/cbc/cbc_start.c File Reference . . . . .	607
5.168modes/cfb/cfb_decrypt.c File Reference . . . . .	609
5.169modes/cfb/cfb_done.c File Reference . . . . .	611
5.170modes/cfb/cfb_encrypt.c File Reference . . . . .	612
5.171modes/cfb/cfb_getiv.c File Reference . . . . .	614
5.172modes/cfb/cfb_setiv.c File Reference . . . . .	615
5.173modes/cfb/cfb_start.c File Reference . . . . .	616
5.174modes/ctr/ctr_decrypt.c File Reference . . . . .	618
5.175modes/ctr/ctr_done.c File Reference . . . . .	619
5.176modes/ctr/ctr_encrypt.c File Reference . . . . .	620
5.177modes/ctr/ctr_getiv.c File Reference . . . . .	622
5.178modes/ctr/ctr_setiv.c File Reference . . . . .	623
5.179modes/ctr/ctr_start.c File Reference . . . . .	625
5.180modes/ctr/ctr_test.c File Reference . . . . .	627
5.181modes/ecb/ecb_decrypt.c File Reference . . . . .	629
5.182modes/ecb/ecb_done.c File Reference . . . . .	631
5.183modes/ecb/ecb_encrypt.c File Reference . . . . .	632
5.184modes/ecb/ecb_start.c File Reference . . . . .	634
5.185modes/f8/f8_decrypt.c File Reference . . . . .	635
5.186modes/f8/f8_done.c File Reference . . . . .	636
5.187modes/f8/f8_encrypt.c File Reference . . . . .	637
5.188modes/f8/f8_getiv.c File Reference . . . . .	639
5.189modes/f8/f8_setiv.c File Reference . . . . .	640
5.190modes/f8/f8_start.c File Reference . . . . .	641



5.191modes/f8/f8_test_mode.c File Reference . . . . .	643
5.192modes/lrw/lrw_decrypt.c File Reference . . . . .	645
5.193modes/lrw/lrw_done.c File Reference . . . . .	646
5.194modes/lrw/lrw_encrypt.c File Reference . . . . .	647
5.195modes/lrw/lrw_getiv.c File Reference . . . . .	648
5.196modes/lrw/lrw_process.c File Reference . . . . .	649
5.197modes/lrw/lrw_setiv.c File Reference . . . . .	652
5.198modes/lrw/lrw_start.c File Reference . . . . .	654
5.199modes/lrw/lrw_test.c File Reference . . . . .	656
5.200modes/ofb/ofb_decrypt.c File Reference . . . . .	659
5.201modes/ofb/ofb_done.c File Reference . . . . .	660
5.202modes/ofb/ofb_encrypt.c File Reference . . . . .	661
5.203modes/ofb/ofb_getiv.c File Reference . . . . .	663
5.204modes/ofb/ofb_setiv.c File Reference . . . . .	664
5.205modes/ofb/ofb_start.c File Reference . . . . .	665
5.206pk/asn1/der/bit/der_decode_bit_string.c File Reference . . . . .	667
5.207pk/asn1/der/bit/der_encode_bit_string.c File Reference . . . . .	669
5.208pk/asn1/der/bit/der_length_bit_string.c File Reference . . . . .	671
5.209pk/asn1/der/boolean/der_decode_boolean.c File Reference . . . . .	672
5.210pk/asn1/der/boolean/der_encode_boolean.c File Reference . . . . .	673
5.211pk/asn1/der/boolean/der_length_boolean.c File Reference . . . . .	674
5.212pk/asn1/der/choice/der_decode_choice.c File Reference . . . . .	675
5.213pk/asn1/der/ia5/der_decode_ia5_string.c File Reference . . . . .	678
5.214pk/asn1/der/ia5/der_encode_ia5_string.c File Reference . . . . .	680
5.215pk/asn1/der/ia5/der_length_ia5_string.c File Reference . . . . .	682
5.216pk/asn1/der/integer/der_decode_integer.c File Reference . . . . .	685
5.217pk/asn1/der/integer/der_encode_integer.c File Reference . . . . .	687
5.218pk/asn1/der/integer/der_length_integer.c File Reference . . . . .	690
5.219pk/asn1/der/object_identifier/der_decode_object_identifier.c File Reference . . . . .	692
5.220pk/asn1/der/object_identifier/der_encode_object_identifier.c File Reference . . . . .	694
5.221pk/asn1/der/object_identifier/der_length_object_identifier.c File Reference . . . . .	696
5.222pk/asn1/der/octet/der_decode_octet_string.c File Reference . . . . .	698
5.223pk/asn1/der/octet/der_encode_octet_string.c File Reference . . . . .	700
5.224pk/asn1/der/octet/der_length_octet_string.c File Reference . . . . .	702
5.225pk/asn1/der/printable_string/der_decode_printable_string.c File Reference . . . . .	703
5.226pk/asn1/der/printable_string/der_encode_printable_string.c File Reference . . . . .	705

5.227pk/asn1/der/printable_string/der_length_printable_string.c File Reference . . . . .	707
5.228pk/asn1/der/sequence/der_decode_sequence_ex.c File Reference . . . . .	710
5.229pk/asn1/der/sequence/der_decode_sequence_flexi.c File Reference . . . . .	715
5.230pk/asn1/der/sequence/der_decode_sequence_multi.c File Reference . . . . .	722
5.231pk/asn1/der/sequence/der_encode_sequence_ex.c File Reference . . . . .	725
5.232pk/asn1/der/sequence/der_encode_sequence_multi.c File Reference . . . . .	731
5.233pk/asn1/der/sequence/der_length_sequence.c File Reference . . . . .	734
5.234pk/asn1/der/sequence/der_sequence_free.c File Reference . . . . .	737
5.235pk/asn1/der/set/der_encode_set.c File Reference . . . . .	739
5.236pk/asn1/der/set/der_encode_setof.c File Reference . . . . .	741
5.237pk/asn1/der/short_integer/der_decode_short_integer.c File Reference . . . . .	744
5.238pk/asn1/der/short_integer/der_encode_short_integer.c File Reference . . . . .	746
5.239pk/asn1/der/short_integer/der_length_short_integer.c File Reference . . . . .	748
5.240pk/asn1/der/utctime/der_decode_utctime.c File Reference . . . . .	750
5.241pk/asn1/der/utctime/der_encode_utctime.c File Reference . . . . .	753
5.242pk/asn1/der/utctime/der_length_utctime.c File Reference . . . . .	755
5.243pk/dsa/dsa_decrypt_key.c File Reference . . . . .	756
5.244pk/dsa/dsa_encrypt_key.c File Reference . . . . .	759
5.245pk/dsa/dsa_export.c File Reference . . . . .	762
5.246pk/dsa/dsa_free.c File Reference . . . . .	764
5.247pk/dsa/dsa_import.c File Reference . . . . .	765
5.248pk/dsa/dsa_make_key.c File Reference . . . . .	767
5.249pk/dsa/dsa_shared_secret.c File Reference . . . . .	770
5.250pk/dsa/dsa_sign_hash.c File Reference . . . . .	772
5.251pk/dsa/dsa_verify_hash.c File Reference . . . . .	776
5.252pk/dsa/dsa_verify_key.c File Reference . . . . .	779
5.253pk/ecc/ecc.c File Reference . . . . .	781
5.254pk/ecc/ecc_ansi_x963_export.c File Reference . . . . .	782
5.255pk/ecc/ecc_ansi_x963_import.c File Reference . . . . .	784
5.256pk/ecc/ecc_decrypt_key.c File Reference . . . . .	786
5.257pk/ecc/ecc_encrypt_key.c File Reference . . . . .	789
5.258pk/ecc/ecc_export.c File Reference . . . . .	792
5.259pk/ecc/ecc_free.c File Reference . . . . .	794
5.260pk/ecc/ecc_get_size.c File Reference . . . . .	795
5.261pk/ecc/ecc_import.c File Reference . . . . .	796
5.262pk/ecc/ecc_make_key.c File Reference . . . . .	799

5.263pk/ecc/ecc_shared_secret.c File Reference . . . . .	801
5.264pk/ecc/ecc_sign_hash.c File Reference . . . . .	803
5.265pk/ecc/ecc_sizes.c File Reference . . . . .	805
5.266pk/ecc/ecc_test.c File Reference . . . . .	806
5.267pk/ecc/ecc_verify_hash.c File Reference . . . . .	808
5.268pk/ecc/lte_ecc_is_valid_idx.c File Reference . . . . .	811
5.269pk/ecc/lte_ecc_map.c File Reference . . . . .	812
5.270pk/ecc/lte_ecc_mulmod.c File Reference . . . . .	814
5.271pk/ecc/lte_ecc_mulmod_timing.c File Reference . . . . .	818
5.272pk/ecc/lte_ecc_points.c File Reference . . . . .	819
5.273pk/ecc/lte_ecc_projective_add_point.c File Reference . . . . .	821
5.274pk/ecc/lte_ecc_projective_dbl_point.c File Reference . . . . .	825
5.275pk/katja/katja_decrypt_key.c File Reference . . . . .	828
5.276pk/katja/katja_encrypt_key.c File Reference . . . . .	829
5.277pk/katja/katja_export.c File Reference . . . . .	830
5.278pk/katja/katja_exptmod.c File Reference . . . . .	831
5.279pk/katja/katja_free.c File Reference . . . . .	832
5.280pk/katja/katja_import.c File Reference . . . . .	833
5.281pk/katja/katja_make_key.c File Reference . . . . .	834
5.282pk/pkcs1/pkcs_1_i2osp.c File Reference . . . . .	835
5.283pk/pkcs1/pkcs_1_mgf1.c File Reference . . . . .	836
5.284pk/pkcs1/pkcs_1_oaep_decode.c File Reference . . . . .	838
5.285pk/pkcs1/pkcs_1_oaep_encode.c File Reference . . . . .	842
5.286pk/pkcs1/pkcs_1_os2ip.c File Reference . . . . .	845
5.287pk/pkcs1/pkcs_1_pss_decode.c File Reference . . . . .	846
5.288pk/pkcs1/pkcs_1_pss_encode.c File Reference . . . . .	850
5.289pk/pkcs1/pkcs_1_v1_5_decode.c File Reference . . . . .	853
5.290pk/pkcs1/pkcs_1_v1_5_encode.c File Reference . . . . .	855
5.291pk/rsa/rsa_decrypt_key.c File Reference . . . . .	857
5.292pk/rsa/rsa_encrypt_key.c File Reference . . . . .	859
5.293pk/rsa/rsa_export.c File Reference . . . . .	861
5.294pk/rsa/rsa_exptmod.c File Reference . . . . .	863
5.295pk/rsa/rsa_free.c File Reference . . . . .	866
5.296pk/rsa/rsa_import.c File Reference . . . . .	867
5.297pk/rsa/rsa_make_key.c File Reference . . . . .	870
5.298pk/rsa/rsa_sign_hash.c File Reference . . . . .	872

5.299pk/rsa/rsa_verify_hash.c File Reference . . . . .	875
5.300prngs/fortuna.c File Reference . . . . .	878
5.301prngs/rc4.c File Reference . . . . .	887
5.302prngs/rng_get_bytes.c File Reference . . . . .	894
5.303prngs/rng_make_prng.c File Reference . . . . .	896
5.304prngs/sober128.c File Reference . . . . .	898
5.305prngs/sober128tab.c File Reference . . . . .	911
5.306prngs/sprng.c File Reference . . . . .	912
5.307prngs/yarrow.c File Reference . . . . .	917

# Chapter 1

## LibTomCrypt Hierarchical Index

### 1.1 LibTomCrypt Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

edge . . . . .	13
Hash_state . . . . .	14
ltc_cipher_descriptor . . . . .	15
ltc_hash_descriptor . . . . .	25
ltc_math_descriptor . . . . .	28
ltc_prng_descriptor . . . . .	44
Prng_state . . . . .	48
Symmetric_key . . . . .	49



## Chapter 2

# LibTomCrypt Data Structure Index

### 2.1 LibTomCrypt Data Structures

Here are the data structures with brief descriptions:

<a href="#">edge</a> . . . . .	13
<a href="#">Hash_state</a> . . . . .	14
<a href="#">ltc_cipher_descriptor</a> (Cipher descriptor table, last entry has "name == NULL" to mark the end of table ) . . . . .	15
<a href="#">ltc_hash_descriptor</a> (Hash descriptor ) . . . . .	25
<a href="#">ltc_math_descriptor</a> (Math descriptor ) . . . . .	28
<a href="#">ltc_prng_descriptor</a> (PRNG descriptor ) . . . . .	44
<a href="#">Prng_state</a> . . . . .	48
<a href="#">Symmetric_key</a> . . . . .	49





## Chapter 3

# LibTomCrypt File Index

### 3.1 LibTomCrypt File List

Here is a list of all files with brief descriptions:

ciphers/ <a href="#">anubis.c</a> (Anubis implementation derived from public domain source Authors: Paulo S.L.M ) . . . . .	68
ciphers/ <a href="#">blowfish.c</a> (Implementation of the Blowfish block cipher, Tom St Denis ) . . . . .	83
ciphers/ <a href="#">cast5.c</a> (Implementation of CAST5 (RFC 2144) by Tom St Denis ) . . . . .	90
ciphers/ <a href="#">des.c</a> (DES code submitted by Dobes Vandermeer ) . . . . .	99
ciphers/ <a href="#">kasumi.c</a> (Implementation of the 3GPP Kasumi block cipher Derived from the 3GPP standard source code ) . . . . .	118
ciphers/ <a href="#">khazad.c</a> (Khazad implementation derived from public domain source Authors: Paulo S.L.M ) . . . . .	125
ciphers/ <a href="#">kseed.c</a> (Seed implementation of SEED derived from RFC4269 Tom St Denis ) . . . . .	134
ciphers/ <a href="#">noekeon.c</a> (Implementation of the Noekeon block cipher by Tom St Denis ) . . . . .	141
ciphers/ <a href="#">rc2.c</a> (Implementation of RC2 ) . . . . .	149
ciphers/ <a href="#">rc5.c</a> (RC5 code by Tom St Denis ) . . . . .	156
ciphers/ <a href="#">rc6.c</a> (RC6 code by Tom St Denis ) . . . . .	163
ciphers/ <a href="#">skipjack.c</a> (Skipjack Implementation by Tom St Denis ) . . . . .	194
ciphers/ <a href="#">xtea.c</a> (Implementation of XTEA, Tom St Denis ) . . . . .	214
ciphers/aes/ <a href="#">aes.c</a> (Implementation of AES ) . . . . .	51
ciphers/aes/ <a href="#">aes_tab.c</a> (AES tables ) . . . . .	64
ciphers/safer/ <a href="#">safer.c</a> . . . . .	171
ciphers/safer/ <a href="#">safer_tab.c</a> (Tables for SAFER block ciphers ) . . . . .	181
ciphers/safer/ <a href="#">saferp.c</a> (SAFER+ Implementation by Tom St Denis ) . . . . .	183
ciphers/twofish/ <a href="#">twofish.c</a> (Implementation of Twofish by Tom St Denis ) . . . . .	202
ciphers/twofish/ <a href="#">twofish_tab.c</a> (Twofish tables, Tom St Denis ) . . . . .	213
encauth/ccm/ <a href="#">ccm_memory.c</a> (CCM support, process a block of memory, Tom St Denis ) . . . . .	219
encauth/ccm/ <a href="#">ccm_test.c</a> (CCM support, process a block of memory, Tom St Denis ) . . . . .	225
encauth/eax/ <a href="#">eax_addheader.c</a> (EAX implementation, add meta-data, by Tom St Denis ) . . . . .	228
encauth/eax/ <a href="#">eax_decrypt.c</a> (EAX implementation, decrypt block, by Tom St Denis ) . . . . .	229
encauth/eax/ <a href="#">eax_decrypt_verify_memory.c</a> (EAX implementation, decrypt block of memory, by Tom St Denis ) . . . . .	230
encauth/eax/ <a href="#">eax_done.c</a> (EAX implementation, terminate session, by Tom St Denis ) . . . . .	232
encauth/eax/ <a href="#">eax_encrypt.c</a> (EAX implementation, encrypt block by Tom St Denis ) . . . . .	234
encauth/eax/ <a href="#">eax_encrypt_authenticate_memory.c</a> (EAX implementation, encrypt a block of memory, by Tom St Denis ) . . . . .	235

encauth/eax/eax_init.c (EAX implementation, initialized EAX state, by Tom St Denis ) . . . . .	237
encauth/eax/eax_test.c (EAX implementation, self-test, by Tom St Denis ) . . . . .	240
encauth/gcm/gcm_add_aad.c (GCM implementation, Add AAD data to the stream, by Tom St Denis ) . . . . .	245
encauth/gcm/gcm_add_iv.c (GCM implementation, add IV data to the state, by Tom St Denis ) . . . . .	248
encauth/gcm/gcm_done.c (GCM implementation, Terminate the stream, by Tom St Denis ) . . . . .	250
encauth/gcm/gcm_gf_mult.c (GCM implementation, do the GF mult, by Tom St Denis ) . . . . .	252
encauth/gcm/gcm_init.c (GCM implementation, initialize state, by Tom St Denis ) . . . . .	254
encauth/gcm/gcm_memory.c (GCM implementation, process a packet, by Tom St Denis ) . . . . .	256
encauth/gcm/gcm_mult_h.c (GCM implementation, do the GF mult, by Tom St Denis ) . . . . .	259
encauth/gcm/gcm_process.c (GCM implementation, process message data, by Tom St Denis ) . . . . .	261
encauth/gcm/gcm_reset.c (GCM implementation, reset a used state so it can accept IV data, by Tom St Denis ) . . . . .	264
encauth/gcm/gcm_test.c (GCM implementation, testing, by Tom St Denis ) . . . . .	265
encauth/ocb/ocb_decrypt.c (OCB implementation, decrypt data, by Tom St Denis ) . . . . .	272
encauth/ocb/ocb_decrypt_verify_memory.c (OCB implementation, helper to decrypt block of memory, by Tom St Denis ) . . . . .	274
encauth/ocb/ocb_done_decrypt.c (OCB implementation, terminate decryption, by Tom St Denis ) . . . . .	276
encauth/ocb/ocb_done_encrypt.c (OCB implementation, terminate encryption, by Tom St Denis ) . . . . .	278
encauth/ocb/ocb_encrypt.c (OCB implementation, encrypt data, by Tom St Denis ) . . . . .	279
encauth/ocb/ocb_encrypt_authenticate_memory.c (OCB implementation, encrypt block of memory, by Tom St Denis ) . . . . .	281
encauth/ocb/ocb_init.c (OCB implementation, initialize state, by Tom St Denis ) . . . . .	283
encauth/ocb/ocb_ntz.c (OCB implementation, internal function, by Tom St Denis ) . . . . .	286
encauth/ocb/ocb_shift_xor.c (OCB implementation, internal function, by Tom St Denis ) . . . . .	287
encauth/ocb/ocb_test.c (OCB implementation, self-test by Tom St Denis ) . . . . .	288
encauth/ocb/s_ocb_done.c (OCB implementation, internal helper, by Tom St Denis ) . . . . .	292
hashes/md2.c . . . . .	308
hashes/md4.c . . . . .	314
hashes/md5.c (MD5 hash function by Tom St Denis ) . . . . .	322
hashes/rmd128.c . . . . .	329
hashes/rmd160.c (RMD160 hash function ) . . . . .	338
hashes/rmd256.c . . . . .	348
hashes/rmd320.c (RMD320 hash function ) . . . . .	357
hashes/sha1.c (SHA1 code by Tom St Denis ) . . . . .	368
hashes/tiger.c (Tiger hash function, Tom St Denis ) . . . . .	394
hashes/chc/chc.c (CHC support ) . . . . .	295
hashes/helper/hash_file.c (Hash a file, Tom St Denis ) . . . . .	300
hashes/helper/hash_filehandle.c (Hash open files, Tom St Denis ) . . . . .	302
hashes/helper/hash_memory.c (Hash memory helper, Tom St Denis ) . . . . .	304
hashes/helper/hash_memory_multi.c (Hash (multiple buffers) memory helper, Tom St Denis ) . . . . .	306
hashes/sha2/sha224.c . . . . .	374
hashes/sha2/sha256.c (SHA256 by Tom St Denis ) . . . . .	377
hashes/sha2/sha384.c . . . . .	384
hashes/sha2/sha512.c . . . . .	388
hashes/whirl/whirl.c (WHIRLPOOL (using their new sbbox) hash function by Tom St Denis ) . . . . .	401
hashes/whirl/whirltab.c (WHIRLPOOL tables, Tom St Denis ) . . . . .	407
headers/tomcrypt.h . . . . .	410
headers/tomcrypt_argchk.h . . . . .	414
headers/tomcrypt_cfg.h . . . . .	416
headers/tomcrypt_cipher.h . . . . .	418
headers/tomcrypt_custom.h . . . . .	423
headers/tomcrypt_hash.h . . . . .	435
headers/tomcrypt_mac.h . . . . .	445

headers/tomcrypt_macros.h	446
headers/tomcrypt_math.h	449
headers/tomcrypt_misc.h	452
headers/tomcrypt_pk.h	455
headers/tomcrypt_pkcs.h	457
headers/tomcrypt_prng.h	458
mac/f9/f9_done.c (F9 Support, terminate the state)	463
mac/f9/f9_file.c (F9 support, process a file, Tom St Denis)	465
mac/f9/f9_init.c (F9 Support, start an F9 state)	467
mac/f9/f9_memory.c	469
mac/f9/f9_memory_multi.c (F9 support, process multiple blocks of memory, Tom St Denis)	471
mac/f9/f9_process.c (F9 Support, terminate the state)	473
mac/f9/f9_test.c (F9 Support, terminate the state)	475
mac/hmac/hmac_done.c (HMAC support, terminate stream, Tom St Denis/Dobes Vandermeer)	477
mac/hmac/hmac_file.c (HMAC support, process a file, Tom St Denis/Dobes Vandermeer)	480
mac/hmac/hmac_init.c (HMAC support, initialize state, Tom St Denis/Dobes Vandermeer)	482
mac/hmac/hmac_memory.c (HMAC support, process a block of memory, Tom St Denis/Dobes Vandermeer)	485
mac/hmac/hmac_memory_multi.c (HMAC support, process multiple blocks of memory, Tom St Denis/Dobes Vandermeer)	487
mac/hmac/hmac_process.c (HMAC support, process data, Tom St Denis/Dobes Vandermeer)	489
mac/hmac/hmac_test.c (HMAC support, self-test, Tom St Denis/Dobes Vandermeer)	490
mac/omac/omac_done.c (OMAC1 support, terminate a stream, Tom St Denis)	495
mac/omac/omac_file.c (OMAC1 support, process a file, Tom St Denis)	497
mac/omac/omac_init.c (OMAC1 support, initialize state, by Tom St Denis)	499
mac/omac/omac_memory.c (OMAC1 support, process a block of memory, Tom St Denis)	501
mac/omac/omac_memory_multi.c (OMAC1 support, process multiple blocks of memory, Tom St Denis)	503
mac/omac/omac_process.c (OMAC1 support, process data, Tom St Denis)	505
mac/omac/omac_test.c (OMAC1 support, self-test, by Tom St Denis)	507
mac/pelican/pelican.c (Pelican MAC, initialize state, by Tom St Denis)	509
mac/pelican/pelican_memory.c (Pelican MAC, MAC a block of memory, by Tom St Denis)	513
mac/pelican/pelican_test.c (Pelican MAC, test, by Tom St Denis)	515
mac/pmac/pmac_done.c (PMAC implementation, terminate a session, by Tom St Denis)	517
mac/pmac/pmac_file.c (PMAC implementation, process a file, by Tom St Denis)	519
mac/pmac/pmac_init.c (PMAC implementation, initialize state, by Tom St Denis)	521
mac/pmac/pmac_memory.c (PMAC implementation, process a block of memory, by Tom St Denis)	524
mac/pmac/pmac_memory_multi.c (PMAC implementation, process multiple blocks of memory, by Tom St Denis)	526
mac/pmac/pmac_ntz.c (PMAC implementation, internal function, by Tom St Denis)	528
mac/pmac/pmac_process.c (PMAC implementation, process data, by Tom St Denis)	529
mac/pmac/pmac_shift_xor.c (PMAC implementation, internal function, by Tom St Denis)	531
mac/pmac/pmac_test.c (PMAC implementation, self-test, by Tom St Denis)	532
mac/xcbc/xcbc_done.c (XCBC Support, terminate the state)	535
mac/xcbc/xcbc_file.c (XCBC support, process a file, Tom St Denis)	537
mac/xcbc/xcbc_init.c (XCBC Support, start an XCBC state)	539
mac/xcbc/xcbc_memory.c	541
mac/xcbc/xcbc_memory_multi.c (XCBC support, process multiple blocks of memory, Tom St Denis)	543
mac/xcbc/xcbc_process.c (XCBC Support, terminate the state)	545
mac/xcbc/xcbc_test.c (XCBC Support, terminate the state)	547
math/gmp_desc.c	550
math/itm_desc.c	551

math/multi.c	552
math/rand_prime.c (Generate a random prime, Tom St Denis)	554
math/tfm_desc.c	556
math/fp/lte_ecc_fp_mulmod.c (ECC Crypto, Tom St Denis)	549
misc/burn_stack.c (Burn stack, Tom St Denis)	562
misc/error_to_string.c (Convert error codes to ASCII strings, Tom St Denis)	593
misc/zeromem.c (Zero a block of memory, Tom St Denis)	599
misc/base64/base64_decode.c (Compliant base64 code donated by Wayne Scott (wscott@bitmover.com))	557
misc/base64/base64_encode.c (Compliant base64 encoder donated by Wayne Scott (wscott@bitmover.com))	560
misc/crypt/crypt.c (Build strings, Tom St Denis)	563
misc/crypt/crypt_argchk.c (Perform argument checking, Tom St Denis)	564
misc/crypt/crypt_cipher_descriptor.c (Stores the cipher descriptor table, Tom St Denis)	565
misc/crypt/crypt_cipher_is_valid.c (Determine if cipher is valid, Tom St Denis)	566
misc/crypt/crypt_find_cipher.c (Find a cipher in the descriptor tables, Tom St Denis)	567
misc/crypt/crypt_find_cipher_any.c (Find a cipher in the descriptor tables, Tom St Denis)	568
misc/crypt/crypt_find_cipher_id.c (Find cipher by ID, Tom St Denis)	570
misc/crypt/crypt_find_hash.c (Find a hash, Tom St Denis)	571
misc/crypt/crypt_find_hash_any.c (Find a hash, Tom St Denis)	572
misc/crypt/crypt_find_hash_id.c (Find hash by ID, Tom St Denis)	574
misc/crypt/crypt_find_hash_oid.c (Find a hash, Tom St Denis)	575
misc/crypt/crypt_find_prng.c (Find a PRNG, Tom St Denis)	576
misc/crypt/crypt_fsa.c (LibTomCrypt FULL SPEED AHEAD!, Tom St Denis)	577
misc/crypt/crypt_hash_descriptor.c (Stores the hash descriptor table, Tom St Denis)	579
misc/crypt/crypt_hash_is_valid.c (Determine if hash is valid, Tom St Denis)	580
misc/crypt/crypt_ltc_mp_descriptor.c	581
misc/crypt/crypt_prng_descriptor.c (Stores the PRNG descriptors, Tom St Denis)	582
misc/crypt/crypt_prng_is_valid.c (Determine if PRNG is valid, Tom St Denis)	583
misc/crypt/crypt_register_cipher.c (Register a cipher, Tom St Denis)	584
misc/crypt/crypt_register_hash.c (Register a HASH, Tom St Denis)	586
misc/crypt/crypt_register_prng.c (Register a PRNG, Tom St Denis)	588
misc/crypt/crypt_unregister_cipher.c (Unregister a cipher, Tom St Denis)	590
misc/crypt/crypt_unregister_hash.c (Unregister a hash, Tom St Denis)	591
misc/crypt/crypt_unregister_prng.c (Unregister a PRNG, Tom St Denis)	592
misc/pkcs5/pkcs_5_1.c (PKCS #5, Algorithm #1, Tom St Denis)	594
misc/pkcs5/pkcs_5_2.c (PKCS #5, Algorithm #2, Tom St Denis)	596
modes/cbc/cbc_decrypt.c (CBC implementation, encrypt block, Tom St Denis)	600
modes/cbc/cbc_done.c (CBC implementation, finish chain, Tom St Denis)	602
modes/cbc/cbc_encrypt.c (CBC implementation, encrypt block, Tom St Denis)	603
modes/cbc/cbc_getiv.c (CBC implementation, get IV, Tom St Denis)	605
modes/cbc/cbc_setiv.c (CBC implementation, set IV, Tom St Denis)	606
modes/cbc/cbc_start.c (CBC implementation, start chain, Tom St Denis)	607
modes/cfb/cfb_decrypt.c (CFB implementation, decrypt data, Tom St Denis)	609
modes/cfb/cfb_done.c (CFB implementation, finish chain, Tom St Denis)	611
modes/cfb/cfb_encrypt.c (CFB implementation, encrypt data, Tom St Denis)	612
modes/cfb/cfb_getiv.c (CFB implementation, get IV, Tom St Denis)	614
modes/cfb/cfb_setiv.c (CFB implementation, set IV, Tom St Denis)	615
modes/cfb/cfb_start.c (CFB implementation, start chain, Tom St Denis)	616
modes/ctr/ctr_decrypt.c (CTR implementation, decrypt data, Tom St Denis)	618
modes/ctr/ctr_done.c (CTR implementation, finish chain, Tom St Denis)	619
modes/ctr/ctr_encrypt.c (CTR implementation, encrypt data, Tom St Denis)	620
modes/ctr/ctr_getiv.c (CTR implementation, get IV, Tom St Denis)	622
modes/ctr/ctr_setiv.c (CTR implementation, set IV, Tom St Denis)	623

modes/ctr/ctr_start.c (CTR implementation, start chain, Tom St Denis ) . . . . .	625
modes/ctr/ctr_test.c (CTR implementation, Tests again RFC 3686, Tom St Denis ) . . . . .	627
modes/ecb/ecb_decrypt.c (ECB implementation, decrypt a block, Tom St Denis ) . . . . .	629
modes/ecb/ecb_done.c (ECB implementation, finish chain, Tom St Denis ) . . . . .	631
modes/ecb/ecb_encrypt.c (ECB implementation, encrypt a block, Tom St Denis ) . . . . .	632
modes/ecb/ecb_start.c (ECB implementation, start chain, Tom St Denis ) . . . . .	634
modes/f8/f8_decrypt.c (F8 implementation, decrypt data, Tom St Denis ) . . . . .	635
modes/f8/f8_done.c (F8 implementation, finish chain, Tom St Denis ) . . . . .	636
modes/f8/f8_encrypt.c (F8 implementation, encrypt data, Tom St Denis ) . . . . .	637
modes/f8/f8_getiv.c . . . . .	639
modes/f8/f8_setiv.c (F8 implementation, set IV, Tom St Denis ) . . . . .	640
modes/f8/f8_start.c (F8 implementation, start chain, Tom St Denis ) . . . . .	641
modes/f8/f8_test_mode.c (F8 implementation, test, Tom St Denis ) . . . . .	643
modes/lrw/lrw_decrypt.c (LRW_MODE implementation, Decrypt blocks, Tom St Denis ) . . . .	645
modes/lrw/lrw_done.c (LRW_MODE implementation, Free resources, Tom St Denis ) . . . .	646
modes/lrw/lrw_encrypt.c (LRW_MODE implementation, Encrypt blocks, Tom St Denis ) . . . .	647
modes/lrw/lrw_getiv.c (LRW_MODE implementation, Retrieve the current IV, Tom St Denis ) .	648
modes/lrw/lrw_process.c (LRW_MODE implementation, Encrypt/decrypt blocks, Tom St Denis )	649
modes/lrw/lrw_setiv.c (LRW_MODE implementation, Set the current IV, Tom St Denis ) . . . .	652
modes/lrw/lrw_start.c (LRW_MODE implementation, start mode, Tom St Denis ) . . . . .	654
modes/lrw/lrw_test.c (LRW_MODE implementation, test LRW, Tom St Denis ) . . . . .	656
modes/ofb/ofb_decrypt.c (OFB implementation, decrypt data, Tom St Denis ) . . . . .	659
modes/ofb/ofb_done.c (OFB implementation, finish chain, Tom St Denis ) . . . . .	660
modes/ofb/ofb_encrypt.c (OFB implementation, encrypt data, Tom St Denis ) . . . . .	661
modes/ofb/ofb_getiv.c (F8 implementation, get IV, Tom St Denis ) . . . . .	663
modes/ofb/ofb_setiv.c (OFB implementation, set IV, Tom St Denis ) . . . . .	664
modes/ofb/ofb_start.c (OFB implementation, start chain, Tom St Denis ) . . . . .	665
pk/asn1/der/bit/der_decode_bit_string.c (ASN.1 DER, encode a BIT STRING, Tom St Denis ) .	667
pk/asn1/der/bit/der_encode_bit_string.c (ASN.1 DER, encode a BIT STRING, Tom St Denis ) .	669
pk/asn1/der/bit/der_length_bit_string.c (ASN.1 DER, get length of BIT STRING, Tom St Denis )	671
pk/asn1/der/boolean/der_decode_boolean.c (ASN.1 DER, decode a BOOLEAN, Tom St Denis )	672
pk/asn1/der/boolean/der_encode_boolean.c (ASN.1 DER, encode a BOOLEAN, Tom St Denis )	673
pk/asn1/der/boolean/der_length_boolean.c (ASN.1 DER, get length of a BOOLEAN, Tom St Denis ) . . . . .	674
pk/asn1/der/choice/der_decode_choice.c (ASN.1 DER, decode a CHOICE, Tom St Denis ) . . .	675
pk/asn1/der/ia5/der_decode_ia5_string.c (ASN.1 DER, encode a IA5 STRING, Tom St Denis ) .	678
pk/asn1/der/ia5/der_encode_ia5_string.c (ASN.1 DER, encode a IA5 STRING, Tom St Denis ) .	680
pk/asn1/der/ia5/der_length_ia5_string.c (ASN.1 DER, get length of IA5 STRING, Tom St Denis )	682
pk/asn1/der/integer/der_decode_integer.c (ASN.1 DER, decode an integer, Tom St Denis ) . . .	685
pk/asn1/der/integer/der_encode_integer.c (ASN.1 DER, encode an integer, Tom St Denis ) . . .	687
pk/asn1/der/integer/der_length_integer.c (ASN.1 DER, get length of encoding, Tom St Denis ) .	690
pk/asn1/der/object_identifier/der_decode_object_identifier.c (ASN.1 DER, Decode Object Iden- tifier, Tom St Denis ) . . . . .	692
pk/asn1/der/object_identifier/der_encode_object_identifier.c (ASN.1 DER, Encode Object Iden- tifier, Tom St Denis ) . . . . .	694
pk/asn1/der/object_identifier/der_length_object_identifier.c (ASN.1 DER, get length of Object Identifier, Tom St Denis ) . . . . .	696
pk/asn1/der/octet/der_decode_octet_string.c (ASN.1 DER, encode a OCTET STRING, Tom St Denis ) . . . . .	698
pk/asn1/der/octet/der_encode_octet_string.c (ASN.1 DER, encode a OCTET STRING, Tom St Denis ) . . . . .	700
pk/asn1/der/octet/der_length_octet_string.c (ASN.1 DER, get length of OCTET STRING, Tom St Denis ) . . . . .	702



pk/asn1/der/printable_string/der_decode_printable_string.c (ASN.1 DER, encode a printable STRING, Tom St Denis ) . . . . .	703
pk/asn1/der/printable_string/der_encode_printable_string.c (ASN.1 DER, encode a printable STRING, Tom St Denis ) . . . . .	705
pk/asn1/der/printable_string/der_length_printable_string.c (ASN.1 DER, get length of Printable STRING, Tom St Denis ) . . . . .	707
pk/asn1/der/sequence/der_decode_sequence_ex.c (ASN.1 DER, decode a SEQUENCE, Tom St Denis ) . . . . .	710
pk/asn1/der/sequence/der_decode_sequence_flexi.c (ASN.1 DER, decode an array of ASN.1 types with a flexi parser, Tom St Denis ) . . . . .	715
pk/asn1/der/sequence/der_decode_sequence_multi.c (ASN.1 DER, decode a SEQUENCE, Tom St Denis ) . . . . .	722
pk/asn1/der/sequence/der_encode_sequence_ex.c (ASN.1 DER, encode a SEQUENCE, Tom St Denis ) . . . . .	725
pk/asn1/der/sequence/der_encode_sequence_multi.c (ASN.1 DER, encode a SEQUENCE, Tom St Denis ) . . . . .	731
pk/asn1/der/sequence/der_length_sequence.c (ASN.1 DER, length a SEQUENCE, Tom St Denis ) . . . . .	734
pk/asn1/der/sequence/der_sequence_free.c (ASN.1 DER, free's a structure allocated by der_decode_sequence_flexi(), Tom St Denis ) . . . . .	737
pk/asn1/der/set/der_encode_set.c (ASN.1 DER, Encode a SET, Tom St Denis ) . . . . .	739
pk/asn1/der/set/der_encode_setof.c (ASN.1 DER, Encode SET OF, Tom St Denis ) . . . . .	741
pk/asn1/der/short_integer/der_decode_short_integer.c (ASN.1 DER, decode an integer, Tom St Denis ) . . . . .	744
pk/asn1/der/short_integer/der_encode_short_integer.c (ASN.1 DER, encode an integer, Tom St Denis ) . . . . .	746
pk/asn1/der/short_integer/der_length_short_integer.c (ASN.1 DER, get length of encoding, Tom St Denis ) . . . . .	748
pk/asn1/der/utctime/der_decode_utctime.c (ASN.1 DER, decode a UTCTIME, Tom St Denis ) . . . . .	750
pk/asn1/der/utctime/der_encode_utctime.c (ASN.1 DER, encode a UTCTIME, Tom St Denis ) . . . . .	753
pk/asn1/der/utctime/der_length_utctime.c (ASN.1 DER, get length of UTCTIME, Tom St Denis ) . . . . .	755
pk/dsa/dsa_decrypt_key.c (DSA Crypto, Tom St Denis ) . . . . .	756
pk/dsa/dsa_encrypt_key.c (DSA Crypto, Tom St Denis ) . . . . .	759
pk/dsa/dsa_export.c (DSA implementation, export key, Tom St Denis ) . . . . .	762
pk/dsa/dsa_free.c (DSA implementation, free a DSA key, Tom St Denis ) . . . . .	764
pk/dsa/dsa_import.c (DSA implementation, import a DSA key, Tom St Denis ) . . . . .	765
pk/dsa/dsa_make_key.c (DSA implementation, generate a DSA key, Tom St Denis ) . . . . .	767
pk/dsa/dsa_shared_secret.c (DSA Crypto, Tom St Denis ) . . . . .	770
pk/dsa/dsa_sign_hash.c (DSA implementation, sign a hash, Tom St Denis ) . . . . .	772
pk/dsa/dsa_verify_hash.c (DSA implementation, verify a signature, Tom St Denis ) . . . . .	776
pk/dsa/dsa_verify_key.c (DSA implementation, verify a key, Tom St Denis ) . . . . .	779
pk/ecc/ecc.c (ECC Crypto, Tom St Denis ) . . . . .	781
pk/ecc/ecc_ansi_x963_export.c (ECC Crypto, Tom St Denis ) . . . . .	782
pk/ecc/ecc_ansi_x963_import.c (ECC Crypto, Tom St Denis ) . . . . .	784
pk/ecc/ecc_decrypt_key.c (ECC Crypto, Tom St Denis ) . . . . .	786
pk/ecc/ecc_encrypt_key.c (ECC Crypto, Tom St Denis ) . . . . .	789
pk/ecc/ecc_export.c (ECC Crypto, Tom St Denis ) . . . . .	792
pk/ecc/ecc_free.c (ECC Crypto, Tom St Denis ) . . . . .	794
pk/ecc/ecc_get_size.c (ECC Crypto, Tom St Denis ) . . . . .	795
pk/ecc/ecc_import.c (ECC Crypto, Tom St Denis ) . . . . .	796
pk/ecc/ecc_make_key.c (ECC Crypto, Tom St Denis ) . . . . .	799
pk/ecc/ecc_shared_secret.c (ECC Crypto, Tom St Denis ) . . . . .	801
pk/ecc/ecc_sign_hash.c (ECC Crypto, Tom St Denis ) . . . . .	803
pk/ecc/ecc_sizes.c (ECC Crypto, Tom St Denis ) . . . . .	805
pk/ecc/ecc_test.c (ECC Crypto, Tom St Denis ) . . . . .	806

pk/ecc/ecc_verify_hash.c (ECC Crypto, Tom St Denis ) . . . . .	808
pk/ecc/lte_ecc_is_valid_idx.c (ECC Crypto, Tom St Denis ) . . . . .	811
pk/ecc/lte_ecc_map.c (ECC Crypto, Tom St Denis ) . . . . .	812
pk/ecc/lte_ecc_mulmod.c (ECC Crypto, Tom St Denis ) . . . . .	814
pk/ecc/lte_ecc_mulmod_timing.c (ECC Crypto, Tom St Denis ) . . . . .	818
pk/ecc/lte_ecc_points.c (ECC Crypto, Tom St Denis ) . . . . .	819
pk/ecc/lte_ecc_projective_add_point.c (ECC Crypto, Tom St Denis ) . . . . .	821
pk/ecc/lte_ecc_projective_dbl_point.c (ECC Crypto, Tom St Denis ) . . . . .	825
pk/katja/katja_decrypt_key.c (Katja PKCS #1 OAEP Decryption, Tom St Denis ) . . . . .	828
pk/katja/katja_encrypt_key.c (Katja PKCS-style OAEP encryption, Tom St Denis ) . . . . .	829
pk/katja/katja_export.c (Export Katja PKCS-style keys, Tom St Denis ) . . . . .	830
pk/katja/katja_exptmod.c (Katja PKCS-style exptmod, Tom St Denis ) . . . . .	831
pk/katja/katja_free.c (Free an Katja key, Tom St Denis ) . . . . .	832
pk/katja/katja_import.c (Import a PKCS-style Katja key, Tom St Denis ) . . . . .	833
pk/katja/katja_make_key.c (Katja key generation, Tom St Denis ) . . . . .	834
pk/pkcs1/pkcs_1_i2osp.c (Integer to Octet I2OSP, Tom St Denis ) . . . . .	835
pk/pkcs1/pkcs_1_mgf1.c (The Mask Generation Function (MGF1) for PKCS #1, Tom St Denis ) . . . . .	836
pk/pkcs1/pkcs_1_oaep_decode.c (OAEP Padding for PKCS #1, Tom St Denis ) . . . . .	838
pk/pkcs1/pkcs_1_oaep_encode.c (OAEP Padding for PKCS #1, Tom St Denis ) . . . . .	842
pk/pkcs1/pkcs_1_os2ip.c (Octet to Integer OS2IP, Tom St Denis ) . . . . .	845
pk/pkcs1/pkcs_1_pss_decode.c (PKCS #1 PSS Signature Padding, Tom St Denis ) . . . . .	846
pk/pkcs1/pkcs_1_pss_encode.c (PKCS #1 PSS Signature Padding, Tom St Denis ) . . . . .	850
pk/pkcs1/pkcs_1_v1_5_decode.c . . . . .	853
pk/pkcs1/pkcs_1_v1_5_encode.c . . . . .	855
pk/rsa/rsa_decrypt_key.c (RSA PKCS #1 Decryption, Tom St Denis and Andreas Lange ) . . . . .	857
pk/rsa/rsa_encrypt_key.c (RSA PKCS #1 encryption, Tom St Denis and Andreas Lange ) . . . . .	859
pk/rsa/rsa_export.c (Export RSA PKCS keys, Tom St Denis ) . . . . .	861
pk/rsa/rsa_exptmod.c (RSA PKCS exptmod, Tom St Denis ) . . . . .	863
pk/rsa/rsa_free.c (Free an RSA key, Tom St Denis ) . . . . .	866
pk/rsa/rsa_import.c (Import a PKCS RSA key, Tom St Denis ) . . . . .	867
pk/rsa/rsa_make_key.c (RSA key generation, Tom St Denis ) . . . . .	870
pk/rsa/rsa_sign_hash.c (RSA PKCS #1 v1.5 and v2 PSS sign hash, Tom St Denis and Andreas Lange ) . . . . .	872
pk/rsa/rsa_verify_hash.c (RSA PKCS #1 v1.5 or v2 PSS signature verification, Tom St Denis and Andreas Lange ) . . . . .	875
prngs/fortuna.c (Fortuna PRNG, Tom St Denis ) . . . . .	878
prngs/rc4.c (RC4 PRNG, Tom St Denis ) . . . . .	887
prngs/rng_get_bytes.c (Portable way to get secure random bits to feed a PRNG (Tom St Denis) ) . . . . .	894
prngs/rng_make_prng.c (Portable way to get secure random bits to feed a PRNG (Tom St Denis) ) . . . . .	896
prngs/sober128.c (Implementation of SOBER-128 by Tom St Denis ) . . . . .	898
prngs/sober128tab.c (SOBER-128 Tables ) . . . . .	911
prngs/sprng.c (Secure PRNG, Tom St Denis ) . . . . .	912
prngs/yarrow.c (Yarrow PRNG, Tom St Denis ) . . . . .	917





## Chapter 4

# LibTomCrypt Data Structure Documentation

### 4.1 edge Struct Reference

#### 4.1.1 Detailed Description

Definition at line 20 of file `der_encode_setof.c`.

#### Data Fields

- unsigned char \* [start](#)
- unsigned long [size](#)

#### 4.1.2 Field Documentation

##### 4.1.2.1 unsigned long [edge::size](#)

Definition at line 22 of file `der_encode_setof.c`.

Referenced by `dsa_decrypt_key()`, `ecc_ansi_x963_import()`, `ecc_decrypt_key()`, `ecc_make_key()`, `ecc_sizes()`, `ecc_test()`, `pkcs_1_i2osp()`, `qsort_helper()`, and `rsa_verify_hash_ex()`.

##### 4.1.2.2 unsigned char\* [edge::start](#)

Definition at line 21 of file `der_encode_setof.c`.

Referenced by `qsort_helper()`.

The documentation for this struct was generated from the following file:

- `pk/asn1/der/set/der\_encode\_setof.c`

## 4.2 Hash\_state Union Reference

```
#include <tomcrypt_hash.h>
```

### 4.2.1 Detailed Description

Definition at line 105 of file `tomcrypt_hash.h`.

#### Data Fields

- char `dummy` [1]
- void \* `data`

### 4.2.2 Field Documentation

#### 4.2.2.1 void\* `Hash_state::data`

Definition at line 146 of file `tomcrypt_hash.h`.

#### 4.2.2.2 char `Hash_state::dummy`[1]

Definition at line 106 of file `tomcrypt_hash.h`.

The documentation for this union was generated from the following file:

- `headers/tomcrypt_hash.h`

## 4.3 ltc\_cipher\_descriptor Struct Reference

```
#include <tomcrypt_cipher.h>
```

### 4.3.1 Detailed Description

cipher descriptor table, last entry has "name == NULL" to mark the end of table

Definition at line 318 of file tomlcrypt\_cipher.h.

#### Data Fields

- char \* [name](#)  
*name of cipher*
- unsigned char [ID](#)  
*internal ID*
- int [min\\_key\\_length](#)  
*min keysize (octets)*
- int [max\\_key\\_length](#)  
*max keysize (octets)*
- int [block\\_length](#)  
*block size (octets)*
- int [default\\_rounds](#)  
*default number of rounds*
- int(\* [setup](#))(const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Setup the cipher.*
- int(\* [ecb\\_encrypt](#))(const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypt a block.*
- int(\* [ecb\\_decrypt](#))(const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypt a block.*
- int(\* [test](#))(void)  
*Test the block cipher.*
- void(\* [done](#))([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- int(\* [keysize](#))(int \*[keysize](#))  
*Determine a key size.*
- int(\* [accel\\_ecb\\_encrypt](#))(const unsigned char \*pt, unsigned char \*ct, unsigned long blocks, [symmetric\\_key](#) \*skey)

*Accelerated ECB encryption.*

- `int(* accel_ecb_decrypt )(const unsigned char *ct, unsigned char *pt, unsigned long blocks, symmetric_key *skey)`

*Accelerated ECB decryption.*

- `int(* accel_cbc_encrypt )(const unsigned char *pt, unsigned char *ct, unsigned long blocks, unsigned char *IV, symmetric_key *skey)`

*Accelerated CBC encryption.*

- `int(* accel_cbc_decrypt )(const unsigned char *ct, unsigned char *pt, unsigned long blocks, unsigned char *IV, symmetric_key *skey)`

*Accelerated CBC decryption.*

- `int(* accel_ctr_encrypt )(const unsigned char *pt, unsigned char *ct, unsigned long blocks, unsigned char *IV, int mode, symmetric_key *skey)`

*Accelerated CTR encryption.*

- `int(* accel_lrw_encrypt )(const unsigned char *pt, unsigned char *ct, unsigned long blocks, unsigned char *IV, const unsigned char *tweak, symmetric_key *skey)`

*Accelerated LRW.*

- `int(* accel_lrw_decrypt )(const unsigned char *ct, unsigned char *pt, unsigned long blocks, unsigned char *IV, const unsigned char *tweak, symmetric_key *skey)`

*Accelerated LRW.*

- `int(* accel_ccm_memory )(const unsigned char *key, unsigned long keylen, symmetric_key *uskey, const unsigned char *nonce, unsigned long noncelen, const unsigned char *header, unsigned long headerlen, unsigned char *pt, unsigned long ptlen, unsigned char *ct, unsigned char *tag, unsigned long *taglen, int direction)`

*Accelerated CCM packet (one-shot).*

- `int(* accel_gcm_memory )(const unsigned char *key, unsigned long keylen, const unsigned char *IV, unsigned long IVlen, const unsigned char *adata, unsigned long adatalen, unsigned char *pt, unsigned long ptlen, unsigned char *ct, unsigned char *tag, unsigned long *taglen, int direction)`

*Accelerated GCM packet (one shot).*

- `int(* omac_memory )(const unsigned char *key, unsigned long keylen, const unsigned char *in, unsigned long inlen, unsigned char *out, unsigned long *outlen)`

*Accelerated one shot OMAC.*

- `int(* xcbc_memory )(const unsigned char *key, unsigned long keylen, const unsigned char *in, unsigned long inlen, unsigned char *out, unsigned long *outlen)`

*Accelerated one shot XCBC.*

- `int(* f9_memory )(const unsigned char *key, unsigned long keylen, const unsigned char *in, unsigned long inlen, unsigned char *out, unsigned long *outlen)`

*Accelerated one shot F9.*

### 4.3.2 Field Documentation

**4.3.2.1** `int(* ltc_cipher_descriptor::accel_cbc_decrypt)(const unsigned char *ct, unsigned char *pt, unsigned long blocks, unsigned char *IV, symmetric_key *skey)`

Accelerated CBC decryption.

**Parameters:**

*pt* Plaintext  
*ct* Ciphertext  
*blocks* The number of complete blocks to process  
*IV* The initial value (input/output)  
*skey* The scheduled key context

**Returns:**

CRYPT\_OK if successful

Referenced by cbc\_decrypt().

**4.3.2.2** `int(* ltc_cipher_descriptor::accel_cbc_encrypt)(const unsigned char *pt, unsigned char *ct, unsigned long blocks, unsigned char *IV, symmetric_key *skey)`

Accelerated CBC encryption.

**Parameters:**

*pt* Plaintext  
*ct* Ciphertext  
*blocks* The number of complete blocks to process  
*IV* The initial value (input/output)  
*skey* The scheduled key context

**Returns:**

CRYPT\_OK if successful

Referenced by cbc\_encrypt().

**4.3.2.3** `int(* ltc_cipher_descriptor::accel_ccm_memory)(const unsigned char *key, unsigned long keylen, symmetric_key *uskey, const unsigned char *nonce, unsigned long noncelen, const unsigned char *header, unsigned long headerlen, unsigned char *pt, unsigned long ptlen, unsigned char *ct, unsigned char *tag, unsigned long *taglen, int direction)`

Accelerated CCM packet (one-shot).

**Parameters:**

*key* The secret key to use  
*keylen* The length of the secret key (octets)  
*uskey* A previously scheduled key [optional can be NULL]  
*nonce* The session nonce [use once]

*noncelen* The length of the nonce  
*header* The header for the session  
*headerlen* The length of the header (octets)  
*pt* [out] The plaintext  
*ptlen* The length of the plaintext (octets)  
*ct* [out] The ciphertext  
*tag* [out] The destination tag  
*taglen* [in/out] The max size and resulting size of the authentication tag  
*direction* Encrypt or Decrypt direction (0 or 1)

**Returns:**

CRYPT\_OK if successful

Referenced by ccm\_memory().

**4.3.2.4** `int(* ltc_cipher_descriptor::accel_ctr_encrypt)(const unsigned char *pt, unsigned char *ct, unsigned long blocks, unsigned char *IV, int mode, symmetric_key *skey)`

Accelerated CTR encryption.

**Parameters:**

*pt* Plaintext  
*ct* Ciphertext  
*blocks* The number of complete blocks to process  
*IV* The initial value (input/output)  
*mode* little or big endian counter (mode=0 or mode=1)  
*skey* The scheduled key context

**Returns:**

CRYPT\_OK if successful

Referenced by ctr\_encrypt().

**4.3.2.5** `int(* ltc_cipher_descriptor::accel_ecb_decrypt)(const unsigned char *ct, unsigned char *pt, unsigned long blocks, symmetric_key *skey)`

Accelerated ECB decryption.

**Parameters:**

*pt* Plaintext  
*ct* Ciphertext  
*blocks* The number of complete blocks to process  
*skey* The scheduled key context

**Returns:**

CRYPT\_OK if successful

Referenced by ecb\_decrypt().

**4.3.2.6** `int(* ltc_cipher_descriptor::accel_ecb_encrypt)(const unsigned char *pt, unsigned char *ct, unsigned long blocks, symmetric_key *skey)`

Accelerated ECB encryption.

**Parameters:**

*pt* Plaintext  
*ct* Ciphertext  
*blocks* The number of complete blocks to process  
*skey* The scheduled key context

**Returns:**

CRYPT\_OK if successful

Referenced by ecb\_encrypt().

**4.3.2.7** `int(* ltc_cipher_descriptor::accel_gcm_memory)(const unsigned char *key, unsigned long keylen, const unsigned char *IV, unsigned long IVlen, const unsigned char *adata, unsigned long adatalen, unsigned char *pt, unsigned long ptlen, unsigned char *ct, unsigned char *tag, unsigned long *taglen, int direction)`

Accelerated GCM packet (one shot).

**Parameters:**

*key* The secret key  
*keylen* The length of the secret key  
*IV* The initial vector  
*IVlen* The length of the initial vector  
*adata* The additional authentication data (header)  
*adatalen* The length of the adata  
*pt* The plaintext  
*ptlen* The length of the plaintext (ciphertext length is the same)  
*ct* The ciphertext  
*tag* [out] The MAC tag  
*taglen* [in/out] The MAC tag length  
*direction* Encrypt or Decrypt mode (GCM\_ENCRYPT or GCM\_DECRYPT)

**Returns:**

CRYPT\_OK on success

Referenced by gcm\_memory().

**4.3.2.8** `int(* ltc_cipher_descriptor::accel_lrw_decrypt)(const unsigned char *ct, unsigned char *pt, unsigned long blocks, unsigned char *IV, const unsigned char *tweak, symmetric_key *skey)`

Accelerated LRW.

**Parameters:**

*ct* Ciphertext  
*pt* Plaintext  
*blocks* The number of complete blocks to process  
*IV* The initial value (input/output)  
*tweak* The LRW tweak  
*skey* The scheduled key context

**Returns:**

CRYPT\_OK if successful

Referenced by `lrw_decrypt()`, and `lrw_setiv()`.

**4.3.2.9** `int(* ltc_cipher_descriptor::accel_lrw_encrypt)(const unsigned char *pt, unsigned char *ct, unsigned long blocks, unsigned char *IV, const unsigned char *tweak, symmetric_key *skey)`

Accelerated LRW.

**Parameters:**

*pt* Plaintext  
*ct* Ciphertext  
*blocks* The number of complete blocks to process  
*IV* The initial value (input/output)  
*tweak* The LRW tweak  
*skey* The scheduled key context

**Returns:**

CRYPT\_OK if successful

Referenced by `lrw_encrypt()`, and `lrw_setiv()`.

**4.3.2.10** `int ltc_cipher_descriptor::block_length`

block size (octets)

Definition at line 324 of file `tomcrypt_cipher.h`.

Referenced by `cbc_start()`, `ccm_memory()`, `cfb_start()`, `chc_register()`, `ctr_start()`, `eax_init()`, `ecb_decrypt()`, `ecb_encrypt()`, `ecb_start()`, `f8_start()`, `f9_done()`, `f9_process()`, `ocb_decrypt()`, `ocb_encrypt()`, `ocb_init()`, `ofb_start()`, `omac_init()`, `pmac_init()`, `s_ocb_done()`, `xcbc_done()`, `xcbc_init()`, and `xcbc_process()`.

**4.3.2.11** `int ltc_cipher_descriptor::default_rounds`

default number of rounds

Definition at line 324 of file `tomcrypt_cipher.h`.

Referenced by `rc5_setup()`.



**4.3.2.12** void(\* [ltc\\_cipher\\_descriptor::done](#))([symmetric\\_key](#) \*skey)

Terminate the context.

**Parameters:**

*skey* The scheduled key

Referenced by cbc\_done(), ccm\_test(), cfb\_done(), ctr\_done(), ecb\_done(), f8\_done(), lrw\_done(), ofb\_done(), and omac\_done().

**4.3.2.13** int(\* [ltc\\_cipher\\_descriptor::ecb\\_decrypt](#))(const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)

Decrypt a block.

**Parameters:**

*ct* The ciphertext

*pt* [out] The plaintext

*skey* The scheduled key

**Returns:**

CRYPT\_OK if successful

Referenced by cbc\_decrypt(), ecb\_decrypt(), and ocb\_decrypt().

**4.3.2.14** int(\* [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#))(const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)

Encrypt a block.

**Parameters:**

*pt* The plaintext

*ct* [out] The ciphertext

*skey* The scheduled key

**Returns:**

CRYPT\_OK if successful

Referenced by cbc\_encrypt(), cfb\_decrypt(), cfb\_encrypt(), cfb\_setiv(), chc\_compress(), chc\_init(), ctr\_encrypt(), ctr\_setiv(), ecb\_encrypt(), f8\_encrypt(), f8\_setiv(), f9\_done(), f9\_process(), ofb\_encrypt(), ofb\_setiv(), omac\_done(), omac\_process(), pmac\_process(), and xcbc\_process().

**4.3.2.15** int(\* [ltc\\_cipher\\_descriptor::f9\\_memory](#))(const unsigned char \*key, unsigned long keylen, const unsigned char \*in, unsigned long inlen, unsigned char \*out, unsigned long outlen)

Accelerated one shot F9.

**Parameters:**

*key* The secret key

*keylen* The key length (octets)  
*in* The message  
*inlen* Length of message (octets)  
*out* [out] Destination for tag  
*outlen* [in/out] Initial and final size of out

**Returns:**

CRYPT\_OK on success

**Remarks:**

Requires manual padding

Referenced by f9\_memory().

**4.3.2.16 unsigned char [ltc\\_cipher\\_descriptor::ID](#)**

internal ID

Definition at line 322 of file tomcrypt\_cipher.h.

Referenced by register\_cipher(), and unregister\_cipher().

**4.3.2.17 int(\* [ltc\\_cipher\\_descriptor::keysize](#))(int \*[keysize](#))**

Determine a key size.

**Parameters:**

*keysize* [in/out] The size of the key desired and the suggested size

**Returns:**

CRYPT\_OK if successful

**4.3.2.18 int [ltc\\_cipher\\_descriptor::max\\_key\\_length](#)**

max keysize (octets)

Definition at line 324 of file tomcrypt\_cipher.h.

**4.3.2.19 int [ltc\\_cipher\\_descriptor::min\\_key\\_length](#)**

min keysize (octets)

Definition at line 324 of file tomcrypt\_cipher.h.

**4.3.2.20 char\* [ltc\\_cipher\\_descriptor::name](#)**

name of cipher

Definition at line 320 of file tomcrypt\_cipher.h.

Referenced by cipher\_is\_valid(), find\_cipher\_id(), find\_hash\_id(), and find\_hash\_oid().

**4.3.2.21** `int(* ltc_cipher_descriptor::omac_memory)(const unsigned char *key, unsigned long keylen, const unsigned char *in, unsigned long inlen, unsigned char *out, unsigned long *outlen)`

Accelerated one shot OMAC.

**Parameters:**

*key* The secret key  
*keylen* The key length (octets)  
*in* The message  
*inlen* Length of message (octets)  
*out* [out] Destination for tag  
*outlen* [in/out] Initial and final size of out

**Returns:**

CRYPT\_OK on success

Referenced by `omac_memory()`.

**4.3.2.22** `int(* ltc_cipher_descriptor::setup)(const unsigned char *key, int keylen, int num_rounds, symmetric_key *skey)`

Setup the cipher.

**Parameters:**

*key* The input symmetric key  
*keylen* The length of the input key (octets)  
*num\_rounds* The requested number of rounds (0==default)  
*skey* [out] The destination of the scheduled key

**Returns:**

CRYPT\_OK if successful

Referenced by `ccm_memory()`, and `ecb_start()`.

**4.3.2.23** `int(* ltc_cipher_descriptor::test)(void)`

Test the block cipher.

**Returns:**

CRYPT\_OK if successful, CRYPT\_NOP if self-testing has been disabled

**4.3.2.24** `int(* ltc_cipher_descriptor::xcbc_memory)(const unsigned char *key, unsigned long keylen, const unsigned char *in, unsigned long inlen, unsigned char *out, unsigned long *outlen)`

Accelerated one shot XCBC.

**Parameters:**

*key* The secret key  
*keylen* The key length (octets)  
*in* The message  
*inlen* Length of message (octets)  
*out* [out] Destination for tag  
*outlen* [in/out] Initial and final size of out

**Returns:**

CRYPT\_OK on success

Referenced by `xcbc_memory()`.

The documentation for this struct was generated from the following file:

- [headers/tomcrypt\\_cipher.h](#)

## 4.4 ltc\_hash\_descriptor Struct Reference

```
#include <tomcrypt_hash.h>
```

### 4.4.1 Detailed Description

hash descriptor

Definition at line 150 of file tomlcrypt\_hash.h.

### Data Fields

- char \* [name](#)  
*name of hash*
- unsigned char [ID](#)  
*internal ID*
- unsigned long [hashsize](#)  
*Size of digest in octets.*
- unsigned long [blocksize](#)  
*Input block size in octets.*
- unsigned long [OID](#) [16]  
*ASN.1 OID.*
- unsigned long [OIDlen](#)  
*Length of DER encoding.*
- int(\* [init](#))([hash\\_state](#) \*hash)  
*Init a hash state.*
- int(\* [process](#))([hash\\_state](#) \*hash, const unsigned char \*[in](#), unsigned long inlen)  
*Process a block of data.*
- int(\* [done](#))([hash\\_state](#) \*hash, unsigned char \*out)  
*Produce the digest and store it.*
- int(\* [test](#))(void)  
*Self-test.*
- int(\* [hmac\\_block](#))(const unsigned char \*key, unsigned long keylen, const unsigned char \*[in](#), unsigned long inlen, unsigned char \*out, unsigned long \*outlen)

## 4.4.2 Field Documentation

### 4.4.2.1 unsigned long `ltc_hash_descriptor::blocksize`

Input block size in octets.

Definition at line 158 of file `tomcrypt_hash.h`.

Referenced by `chc_register()`.

### 4.4.2.2 `int(* ltc_hash_descriptor::done)(hash_state *hash, unsigned char *out)`

Produce the digest and store it.

#### Parameters:

*hash* The hash state

*out* [out] The destination of the digest

#### Returns:

CRYPT\_OK if successful

Referenced by `hash_filehandle()`, and `hash_memory()`.

### 4.4.2.3 unsigned long `ltc_hash_descriptor::hashsize`

Size of digest in octets.

Definition at line 156 of file `tomcrypt_hash.h`.

Referenced by `chc_register()`, `hash_filehandle()`, `hash_memory()`, `hash_memory_multi()`, `hmac_done()`, `hmac_init()`, `pkcs_1_mgf1()`, `pkcs_1_oaep_decode()`, `pkcs_1_oaep_encode()`, `pkcs_1_pss_decode()`, `pkcs_1_pss_encode()`, and `pkcs_5_alg1()`.

### 4.4.2.4 `int(* ltc_hash_descriptor::hmac_block)(const unsigned char *key, unsigned long keylen, const unsigned char *in, unsigned long inlen, unsigned char *out, unsigned long *outlen)`

Referenced by `hmac_memory()`.

### 4.4.2.5 unsigned char `ltc_hash_descriptor::ID`

internal ID

Definition at line 154 of file `tomcrypt_hash.h`.

### 4.4.2.6 `int(* ltc_hash_descriptor::init)(hash_state *hash)`

Init a hash state.

#### Parameters:

*hash* The hash to initialize

#### Returns:

CRYPT\_OK if successful

**4.4.2.7 char\* [ltc\\_hash\\_descriptor::name](#)**

name of hash

Definition at line 152 of file [tomcrypt\\_hash.h](#).

Referenced by [hash\\_is\\_valid\(\)](#).

**4.4.2.8 unsigned long [ltc\\_hash\\_descriptor::OID](#)[16]**

ASN.1 OID.

Definition at line 160 of file [tomcrypt\\_hash.h](#).

**4.4.2.9 unsigned long [ltc\\_hash\\_descriptor::OIDlen](#)**

Length of DER encoding.

Definition at line 162 of file [tomcrypt\\_hash.h](#).

**4.4.2.10 int(\* [ltc\\_hash\\_descriptor::process](#))([hash\\_state](#) \*hash, const unsigned char \*[in](#), unsigned long [inlen](#))**

Process a block of data.

**Parameters:**

*hash* The hash state

*in* The data to hash

*inlen* The length of the data (octets)

**Returns:**

CRYPT\_OK if successful

Referenced by [hmac\\_process\(\)](#).

**4.4.2.11 int(\* [ltc\\_hash\\_descriptor::test](#))(void)**

Self-test.

**Returns:**

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

The documentation for this struct was generated from the following file:

- [headers/tomcrypt\\_hash.h](#)

## 4.5 ltc\_math\_descriptor Struct Reference

```
#include <tomcrypt_math.h>
```

### 4.5.1 Detailed Description

math descriptor

Definition at line 19 of file tomlcrypt\_math.h.

#### Data Fields

- char \* [name](#)  
*Name of the math provider.*
- int [bits\\_per\\_digit](#)  
*Bits per digit, amount of bits must fit in an unsigned long.*
- int(\* [init](#))(void \*\*a)  
*initialize a bignum*
- int(\* [init\\_copy](#))(void \*\*dst, void \*src)  
*init copy*
- void(\* [deinit](#))(void \*a)  
*deinit*
- int(\* [neg](#))(void \*src, void \*dst)  
*negate*
- int(\* [copy](#))(void \*src, void \*dst)  
*copy*
- int(\* [set\\_int](#))(void \*a, unsigned long n)  
*set small constant*
- unsigned long(\* [get\\_int](#))(void \*a)  
*get small constant*
- unsigned long(\* [get\\_digit](#))(void \*a, int n)  
*get digit n*
- int(\* [get\\_digit\\_count](#))(void \*a)  
*Get the number of digits that represent the number.*
- int(\* [compare](#))(void \*a, void \*b)  
*compare two integers*
- int(\* [compare\\_d](#))(void \*a, unsigned long n)



*compare against int*

- `int(* count\_bits )(void *a)`  
*Count the number of bits used to represent the integer.*
- `int(* count\_lsb\_bits )(void *a)`  
*Count the number of LSB bits which are zero.*
- `int(* twoexpt )(void *a, int n)`  
*Compute a power of two.*
- `int(* read\_radix )(void *a, const char *str, int radix)`  
*read ascii string*
- `int(* write\_radix )(void *a, char *str, int radix)`  
*write number to string*
- `unsigned long(* unsigned\_size )(void *a)`  
*get size as unsigned char string*
- `int(* unsigned\_write )(void *src, unsigned char *dst)`  
*store an integer as an array of octets*
- `int(* unsigned\_read )(void *dst, unsigned char *src, unsigned long len)`  
*read an array of octets and store as integer*
- `int(* add )(void *a, void *b, void *c)`  
*add two integers*
- `int(* addi )(void *a, unsigned long b, void *c)`  
*add two integers*
- `int(* sub )(void *a, void *b, void *c)`  
*subtract two integers*
- `int(* subi )(void *a, unsigned long b, void *c)`  
*subtract two integers*
- `int(* mul )(void *a, void *b, void *c)`  
*multiply two integers*
- `int(* muli )(void *a, unsigned long b, void *c)`  
*multiply two integers*
- `int(* sqr )(void *a, void *b)`  
*Square an integer.*
- `int(* mpdiv )(void *a, void *b, void *c, void *d)`  
*Divide an integer.*

- `int(* div_2)(void *a, void *b)`  
*divide by two*
- `int(* modi)(void *a, unsigned long b, unsigned long *c)`  
*Get remainder (small value).*
- `int(* gcd)(void *a, void *b, void *c)`  
*gcd*
- `int(* lcm)(void *a, void *b, void *c)`  
*lcm*
- `int(* mulmod)(void *a, void *b, void *c, void *d)`  
*Modular multiplication.*
- `int(* sqrmod)(void *a, void *b, void *c)`  
*Modular squaring.*
- `int(* invmod)(void *, void *, void *)`  
*Modular inversion.*
- `int(* montgomery_setup)(void *a, void **b)`  
*setup montgomery*
- `int(* montgomery_normalization)(void *a, void *b)`  
*get normalization value*
- `int(* montgomery_reduce)(void *a, void *b, void *c)`  
*reduce a number*
- `void(* montgomery_deinit)(void *a)`  
*clean up (frees memory)*
- `int(* exptmod)(void *a, void *b, void *c, void *d)`  
*Modular exponentiation.*
- `int(* isprime)(void *a, int *b)`  
*Primality testing.*
- `int(* ecc_ptmul)(void *k, ecc_point *G, ecc_point *R, void *modulus, int map)`  
*ECC GF(p) point multiplication (from the NIST curves).*
- `int(* ecc_ptadd)(ecc_point *P, ecc_point *Q, ecc_point *R, void *modulus, void *mp)`  
*ECC GF(p) point addition.*
- `int(* ecc_ptdbl)(ecc_point *P, ecc_point *R, void *modulus, void *mp)`  
*ECC GF(p) point double.*
- `int(* ecc_map)(ecc_point *P, void *modulus, void *mp)`  
*ECC mapping from projective to affine, currently uses  $(x,y,z) \Rightarrow (x/z^2, y/z^3, 1)$ .*

- `int(* rsa_keygen )(prng_state *prng, int wprng, int size, long e, rsa_key *key)`  
*RSA Key Generation.*
- `int(* rsa_me )(const unsigned char *in, unsigned long inlen, unsigned char *out, unsigned long *outlen, int which, rsa_key *key)`  
*RSA exponentiation.*

## 4.5.2 Field Documentation

### 4.5.2.1 `int(* ltc_math_descriptor::add)(void *a, void *b, void *c)`

add two integers

**Parameters:**

- a* The first source integer
- b* The second source integer
- c* The destination of "a + b"

**Returns:**

CRYPT\_OK on success

### 4.5.2.2 `int(* ltc_math_descriptor::addi)(void *a, unsigned long b, void *c)`

add two integers

**Parameters:**

- a* The first source integer
- b* The second source integer (single digit of upto bits\_per\_digit in length)
- c* The destination of "a + b"

**Returns:**

CRYPT\_OK on success

### 4.5.2.3 `int ltc_math_descriptor::bits_per_digit`

Bits per digit, amount of bits must fit in an unsigned long.

Definition at line 24 of file tomcrypt\_math.h.

### 4.5.2.4 `int(* ltc_math_descriptor::compare)(void *a, void *b)`

compare two integers

**Parameters:**

- a* The left side integer
- b* The right side integer

**Returns:**

LTC\_MP\_LT if  $a < b$ , LTC\_MP\_GT if  $a > b$  and LTC\_MP\_EQ otherwise. (signed comparison)

**4.5.2.5** `int(* ltc_math_descriptor::compare_d)(void *a, unsigned long n)`

compare against int

**Parameters:**

- a* The left side integer
- b* The right side integer (upto bits\_per\_digit)

**Returns:**

LTC\_MP\_LT if  $a < b$ , LTC\_MP\_GT if  $a > b$  and LTC\_MP\_EQ otherwise. (signed comparison)

**4.5.2.6** `int(* ltc_math_descriptor::copy)(void *src, void *dst)`

copy

**Parameters:**

- src* The number to copy from
- dst* The number to write to

**Returns:**

CRYPT\_OK on success

**4.5.2.7** `int(* ltc_math_descriptor::count_bits)(void *a)`

Count the number of bits used to represent the integer.

**Parameters:**

- a* The integer to count

**Returns:**

The number of bits required to represent the integer

**4.5.2.8** `int(* ltc_math_descriptor::count_lsb_bits)(void *a)`

Count the number of LSB bits which are zero.

**Parameters:**

- a* The integer to count

**Returns:**

The number of contiguous zero LSB bits

**4.5.2.9** `void(* ltc_math_descriptor::deinit)(void *a)`

deinit

**Parameters:**

- a* The number to free

**Returns:**

CRYPT\_OK on success

**4.5.2.10** `int(* ltc_math_descriptor::div_2)(void *a, void *b)`

divide by two

**Parameters:**

- a* The integer to divide (shift right)
- b* The destination

**Returns:**

CRYPT\_OK on success

**4.5.2.11** `int(* ltc_math_descriptor::ecc_map)(ecc_point *P, void *modulus, void *mp)`

ECC mapping from projective to affine, currently uses  $(x,y,z) \Rightarrow (x/z^2, y/z^3, 1)$ .

**Parameters:**

- P* The point to map
- modulus* The modulus
- mp* The "b" value from [montgomery\\_setup\(\)](#)

**Returns:**

CRYPT\_OK on success

**Remarks:**

The mapping can be different but keep in mind a `ecc_point` only has three integers (x,y,z) so if you use a different mapping you have to make it fit.

**4.5.2.12** `int(* ltc_math_descriptor::ecc_ptadd)(ecc_point *P, ecc_point *Q, ecc_point *R, void *modulus, void *mp)`

ECC GF(p) point addition.

**Parameters:**

- P* The first point
- Q* The second point
- R* The destination of  $P + Q$
- modulus* The modulus
- mp* The "b" value from [montgomery\\_setup\(\)](#)

**Returns:**

CRYPT\_OK on success

**4.5.2.13** `int(* ltc_math_descriptor::ecc_ptdbl)(ecc_point *P, ecc_point *R, void *modulus, void *mp)`

ECC GF(p) point double.

**Parameters:**

- P* The first point
- R* The destination of 2P
- modulus* The modulus
- mp* The "b" value from `montgomery_setup()`

**Returns:**

CRYPT\_OK on success

**4.5.2.14** `int(* ltc_math_descriptor::ecc_ptmul)(void *k, ecc_point *G, ecc_point *R, void *modulus, int map)`

ECC GF(p) point multiplication (from the NIST curves).

**Parameters:**

- k* The integer to multiply the point by
- G* The point to multiply
- R* The destination for kG
- modulus* The modulus for the field
- map* Boolean indicated whether to map back to affine or not (can be ignored if you work in affine only)

**Returns:**

CRYPT\_OK on success

Referenced by `ecc_shared_secret()`, and `ecc_verify_hash()`.

**4.5.2.15** `int(* ltc_math_descriptor::exptmod)(void *a, void *b, void *c, void *d)`

Modular exponentiation.

**Parameters:**

- a* The base integer
- b* The power (can be negative) integer
- c* The modulus integer
- d* The destination

**Returns:**

CRYPT\_OK on success

**4.5.2.16** `int(* ltc_math_descriptor::gcd)(void *a, void *b, void *c)`

gcd

**Parameters:**

- a* The first integer
- b* The second integer
- c* The destination for (a, b)

**Returns:**

CRYPT\_OK on success

**4.5.2.17** `unsigned long(* ltc_math_descriptor::get_digit)(void *a, int n)`

get digit n

**Parameters:**

- a* The number to read from
- n* The number of the digit to fetch

**Returns:**

The bits\_per\_digit sized n'th digit of a

**4.5.2.18** `int(* ltc_math_descriptor::get_digit_count)(void *a)`

Get the number of digits that represent the number.

**Parameters:**

- a* The number to count

**Returns:**

The number of digits used to represent the number

**4.5.2.19** `unsigned long(* ltc_math_descriptor::get_int)(void *a)`

get small constant

**Parameters:**

- a* Number to read, only fetches upto bits\_per\_digit from the number

**Returns:**

The lower bits\_per\_digit of the integer (unsigned)

**4.5.2.20** `int(* ltc_math_descriptor::init)(void **a)`

initialize a bignum

**Parameters:**

*a* The number to initialize

**Returns:**

CRYPT\_OK on success

**4.5.2.21** `int(* ltc_math_descriptor::init_copy)(void **dst, void *src)`

init copy

**Parameters:**

*dst* The number to initialize and write to

*src* The number to copy from

**Returns:**

CRYPT\_OK on success

**4.5.2.22** `int(* ltc_math_descriptor::invmod)(void *, void *, void *)`

Modular inversion.

**Parameters:**

*a* The value to invert

*b* The modulus

*c* The destination (1/a mod b)

**Returns:**

CRYPT\_OK on success

**4.5.2.23** `int(* ltc_math_descriptor::isprime)(void *a, int *b)`

Primality testing.

**Parameters:**

*a* The integer to test

*b* The destination of the result (FP\_YES if prime)

**Returns:**

CRYPT\_OK on success



**4.5.2.24** `int(* ltc_math_descriptor::lcm)(void *a, void *b, void *c)`

lcm

**Parameters:**

- a* The first integer
- b* The second integer
- c* The destination for [a, b]

**Returns:**

CRYPT\_OK on success

**4.5.2.25** `int(* ltc_math_descriptor::modi)(void *a, unsigned long b, unsigned long *c)`

Get remainder (small value).

**Parameters:**

- a* The integer to reduce
- b* The modulus (upto bits\_per\_digit in length)
- c* The destination for the residue

**Returns:**

CRYPT\_OK on success

**4.5.2.26** `void(* ltc_math_descriptor::montgomery_deinit)(void *a)`

clean up (frees memory)

**Parameters:**

- a* The value "b" from `montgomery_setup()`

**Returns:**

CRYPT\_OK on success

**4.5.2.27** `int(* ltc_math_descriptor::montgomery_normalization)(void *a, void *b)`

get normalization value

**Parameters:**

- a* The destination for the normalization value
- b* The modulus

**Returns:**

CRYPT\_OK on success

**4.5.2.28** `int(* ltc_math_descriptor::montgomery_reduce)(void *a, void *b, void *c)`

reduce a number

**Parameters:**

- a* The number [and dest] to reduce
- b* The modulus
- c* The value "b" from `montgomery_setup()`

**Returns:**

CRYPT\_OK on success

**4.5.2.29** `int(* ltc_math_descriptor::montgomery_setup)(void *a, void **b)`

setup montgomery

**Parameters:**

- a* The modulus
- b* The destination for the reduction digit

**Returns:**

CRYPT\_OK on success

**4.5.2.30** `int(* ltc_math_descriptor::mpdiv)(void *a, void *b, void *c, void *d)`

Divide an integer.

**Parameters:**

- a* The dividend
- b* The divisor
- c* The quotient (can be NULL to signify don't care)
- d* The remainder (can be NULL to signify don't care)

**Returns:**

CRYPT\_OK on success

**4.5.2.31** `int(* ltc_math_descriptor::mul)(void *a, void *b, void *c)`

multiply two integers

**Parameters:**

- a* The first source integer
- b* The second source integer (single digit of upto bits\_per\_digit in length)
- c* The destination of "a \* b"

**Returns:**

CRYPT\_OK on success

**4.5.2.32** `int(* ltc_math_descriptor::muli)(void *a, unsigned long b, void *c)`

multiply two integers

**Parameters:**

- a* The first source integer
- b* The second source integer (single digit of upto bits\_per\_digit in length)
- c* The destination of "a \* b"

**Returns:**

CRYPT\_OK on success

**4.5.2.33** `int(* ltc_math_descriptor::mulmod)(void *a, void *b, void *c, void *d)`

Modular multiplication.

**Parameters:**

- a* The first source
- b* The second source
- c* The modulus
- d* The destination (a\*b mod c)

**Returns:**

CRYPT\_OK on success

**4.5.2.34** `char* ltc_math_descriptor::name`

Name of the math provider.

Definition at line 21 of file tomcrypt\_math.h.

Referenced by dsa\_import(), dsa\_make\_key(), ecc\_import(), ecc\_make\_key(), rsa\_import(), and rsa\_make\_key().

**4.5.2.35** `int(* ltc_math_descriptor::neg)(void *src, void *dst)`

negate

**Parameters:**

- src* The number to negate
- dst* The destination

**Returns:**

CRYPT\_OK on success

**4.5.2.36** `int(* ltc_math_descriptor::read_radix)(void *a, const char *str, int radix)`

read ascii string

**Parameters:**

- a* The integer to store into
- str* The string to read
- radix* The radix the integer has been represented in (2-64)

**Returns:**

CRYPT\_OK on success

**4.5.2.37** `int(* ltc_math_descriptor::rsa_keygen)(prng_state *prng, int wprng, int size, long e, rsa_key *key)`

RSA Key Generation.

**Parameters:**

- prng* An active PRNG state
- wprng* The index of the PRNG desired
- size* The size of the modulus (key size) desired (octets)
- e* The "e" value (public key). e==65537 is a good choice
- key* [out] Destination of a newly created private key pair

**Returns:**

CRYPT\_OK if successful, upon error all allocated ram is freed

**4.5.2.38** `int(* ltc_math_descriptor::rsa_me)(const unsigned char *in, unsigned long inlen, unsigned char *out, unsigned long *outlen, int which, rsa_key *key)`

RSA exponentiation.

**Parameters:**

- in* The octet array representing the base
- inlen* The length of the input
- out* The destination (to be stored in an octet array format)
- outlen* The length of the output buffer and the resulting size (zero padded to the size of the modulus)
- which* PK\_PUBLIC for public RSA and PK\_PRIVATE for private RSA
- key* The RSA key to use

**Returns:**

CRYPT\_OK on success

Referenced by rsa\_decrypt\_key\_ex(), rsa\_encrypt\_key\_ex(), rsa\_sign\_hash\_ex(), and rsa\_verify\_hash\_ex().

**4.5.2.39** `int(* ltc_math_descriptor::set_int)(void *a, unsigned long n)`

set small constant

**Parameters:**

- a* Number to write to
- n* Source upto bits\_per\_digit (actually meant for very small constants)

**Returns:**

CRYPT\_OK on success

**4.5.2.40** `int(* ltc_math_descriptor::sqr)(void *a, void *b)`

Square an integer.

**Parameters:**

- a* The integer to square
- b* The destination

**Returns:**

CRYPT\_OK on success

**4.5.2.41** `int(* ltc_math_descriptor::sqrmmod)(void *a, void *b, void *c)`

Modular squaring.

**Parameters:**

- a* The first source
- b* The modulus
- c* The destination (a\*a mod b)

**Returns:**

CRYPT\_OK on success

**4.5.2.42** `int(* ltc_math_descriptor::sub)(void *a, void *b, void *c)`

subtract two integers

**Parameters:**

- a* The first source integer
- b* The second source integer
- c* The destination of "a - b"

**Returns:**

CRYPT\_OK on success

**4.5.2.43** `int(* ltc\_math\_descriptor::subi)(void *a, unsigned long b, void *c)`

subtract two integers

**Parameters:**

- a* The first source integer
- b* The second source integer (single digit of upto bits\_per\_digit in length)
- c* The destination of "a - b"

**Returns:**

CRYPT\_OK on success

**4.5.2.44** `int(* ltc\_math\_descriptor::twoexpt)(void *a, int n)`

Compute a power of two.

**Parameters:**

- a* The integer to store the power in
- n* The power of two you want to store ( $a = 2^n$ )

**Returns:**

CRYPT\_OK on success

**4.5.2.45** `int(* ltc\_math\_descriptor::unsigned\_read)(void *dst, unsigned char *src, unsigned long len)`

read an array of octets and store as integer

**Parameters:**

- dst* The integer to load
- src* The array of octets
- len* The number of octets

**Returns:**

CRYPT\_OK on success

**4.5.2.46** `unsigned long(* ltc\_math\_descriptor::unsigned\_size)(void *a)`

get size as unsigned char string

**Parameters:**

- a* The integer to get the size (when stored in array of octets)

**Returns:**

The length of the integer

**4.5.2.47** `int(* ltc_math_descriptor::unsigned_write)(void *src, unsigned char *dst)`

store an integer as an array of octets

**Parameters:**

*src* The integer to store

*dst* The buffer to store the integer in

**Returns:**

CRYPT\_OK on success

**4.5.2.48** `int(* ltc_math_descriptor::write_radix)(void *a, char *str, int radix)`

write number to string

**Parameters:**

*a* The integer to store

*str* The destination for the string

*radix* The radix the integer is to be represented in (2-64)

**Returns:**

CRYPT\_OK on success

The documentation for this struct was generated from the following file:

- [headers/tomcrypt\\_math.h](#)

## 4.6 ltc\_prng\_descriptor Struct Reference

```
#include <tomcrypt_prng.h>
```

### 4.6.1 Detailed Description

PRNG descriptor.

Definition at line 67 of file tomcrypt\_prng.h.

### Data Fields

- char \* [name](#)  
*Name of the PRNG.*
- int [export\\_size](#)  
*size in bytes of exported state*
- int(\* [start](#))([prng\\_state](#) \*prng)  
*Start a PRNG state.*
- int(\* [add\\_entropy](#))(const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)  
*Add entropy to the PRNG.*
- int(\* [ready](#))([prng\\_state](#) \*prng)  
*Ready a PRNG state to read from.*
- unsigned long(\* [read](#))(unsigned char \*out, unsigned long outlen, [prng\\_state](#) \*prng)  
*Read from the PRNG.*
- int(\* [done](#))([prng\\_state](#) \*prng)  
*Terminate a PRNG state.*
- int(\* [pexport](#))(unsigned char \*out, unsigned long \*outlen, [prng\\_state](#) \*prng)  
*Export a PRNG state.*
- int(\* [pimport](#))(const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)  
*Import a PRNG state.*
- int(\* [test](#))(void)  
*Self-test the PRNG.*

### 4.6.2 Field Documentation

**4.6.2.1** int(\* [ltc\\_prng\\_descriptor::add\\_entropy](#))(const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)

Add entropy to the PRNG.



**Parameters:**

*in* The entropy  
*inlen* Length of the entropy (octets)\n  
*prng* The PRNG state

**Returns:**

CRYPT\_OK if successful

**4.6.2.2** `int(* ltc_prng_descriptor::done)(prng_state *prng)`

Terminate a PRNG state.

**Parameters:**

*prng* The PRNG state to terminate

**Returns:**

CRYPT\_OK if successful

Referenced by f9\_init(), f9\_memory(), and pkcs\_5\_alg1().

**4.6.2.3** `int ltc_prng_descriptor::export_size`

size in bytes of exported state

Definition at line 71 of file tomcrypt\_prng.h.

**4.6.2.4** `char* ltc_prng_descriptor::name`

Name of the PRNG.

Definition at line 69 of file tomcrypt\_prng.h.

Referenced by prng\_is\_valid(), register\_cipher(), unregister\_cipher(), unregister\_hash(), and unregister\_prng().

**4.6.2.5** `int(* ltc_prng_descriptor::pexport)(unsigned char *out, unsigned long *outlen, prng_state *prng)`

Export a PRNG state.

**Parameters:**

*out* [out] The destination for the state  
*outlen* [in/out] The max size and resulting size of the PRNG state  
*prng* The PRNG to export

**Returns:**

CRYPT\_OK if successful

**4.6.2.6** `int(* ltc_prng_descriptor::pimport)(const unsigned char *in, unsigned long inlen, prng_state *prng)`

Import a PRNG state.

**Parameters:**

*in* The data to import  
*inlen* The length of the data to import (octets)  
*prng* The PRNG to initialize/import

**Returns:**

CRYPT\_OK if successful

**4.6.2.7** `unsigned long(* ltc_prng_descriptor::read)(unsigned char *out, unsigned long outlen, prng_state *prng)`

Read from the PRNG.

**Parameters:**

*out* [out] Where to store the data  
*outlen* Length of data desired (octets)  
*prng* The PRNG state to read from

**Returns:**

Number of octets read

**4.6.2.8** `int(* ltc_prng_descriptor::ready)(prng_state *prng)`

Ready a PRNG state to read from.

**Parameters:**

*prng* The PRNG state to ready

**Returns:**

CRYPT\_OK if successful

**4.6.2.9** `int(* ltc_prng_descriptor::start)(prng_state *prng)`

Start a PRNG state.

**Parameters:**

*prng* [out] The state to initialize

**Returns:**

CRYPT\_OK if successful

#### 4.6.2.10 int(\* [ltc\\_prng\\_descriptor::test](#))(void)

Self-test the PRNG.

**Returns:**

CRYPT\_OK if successful, CRYPT\_NOP if self-testing has been disabled

The documentation for this struct was generated from the following file:

- [headers/tomcrypt\\_prng.h](#)

## 4.7 Prng\_state Union Reference

```
#include <tomcrypt_prng.h>
```

### 4.7.1 Detailed Description

Definition at line 50 of file `tomcrypt_prng.h`.

#### Data Fields

- char `dummy` [1]

### 4.7.2 Field Documentation

#### 4.7.2.1 char `Prng_state::dummy`[1]

Definition at line 51 of file `tomcrypt_prng.h`.

The documentation for this union was generated from the following file:

- `headers/tomcrypt_prng.h`

## 4.8 Symmetric\_key Union Reference

```
#include <tomcrypt_cipher.h>
```

### 4.8.1 Detailed Description

Definition at line 134 of file `tomcrypt_cipher.h`.

#### Data Fields

- void \* [data](#)

### 4.8.2 Field Documentation

#### 4.8.2.1 void\* [Symmetric\\_key::data](#)

Definition at line 187 of file `tomcrypt_cipher.h`.

The documentation for this union was generated from the following file:

- `headers/tomcrypt_cipher.h`



## Chapter 5

# LibTomCrypt File Documentation

### 5.1 ciphers/aes/aes.c File Reference

#### 5.1.1 Detailed Description

Implementation of AES.

Definition in file [aes.c](#).

```
#include "tomcrypt.h"
```

```
#include "aes_tab.c"
```

Include dependency graph for aes.c:

#### Defines

- #define [SETUP](#) rijndael\_setup
- #define [ECB\\_ENC](#) rijndael\_ecb\_encrypt
- #define [ECB\\_DEC](#) rijndael\_ecb\_decrypt
- #define [ECB\\_DONE](#) rijndael\_done
- #define [ECB\\_TEST](#) rijndael\_test
- #define [ECB\\_KS](#) rijndael\_keysize

#### Functions

- static [ulong32](#) [setup\\_mix](#) ([ulong32](#) temp)
- int [SETUP](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the AES (Rijndael) block cipher.*
- int [ECB\\_ENC](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with AES.*
- int [ECB\\_DEC](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with AES.*
- int [ECB\\_TEST](#) (void)

*Performs a self-test of the AES block cipher.*

- void [ECB\\_DONE](#) ([symmetric\\_key](#) \*skey)

*Terminate the context.*

- int [ECB\\_KS](#) (int \*keysize)

*Gets suitable key size.*

## Variables

- const struct [ltc\\_cipher\\_descriptor](#) rijndael\_desc
- const struct [ltc\\_cipher\\_descriptor](#) aes\_desc

## 5.1.2 Define Documentation

### 5.1.2.1 #define ECB\_DEC rijndael\_ecb\_decrypt

Definition at line 41 of file aes.c.

### 5.1.2.2 #define ECB\_DONE rijndael\_done

Definition at line 42 of file aes.c.

### 5.1.2.3 #define ECB\_ENC rijndael\_ecb\_encrypt

Definition at line 40 of file aes.c.

### 5.1.2.4 #define ECB\_KS rijndael\_keysize

Definition at line 44 of file aes.c.

### 5.1.2.5 #define ECB\_TEST rijndael\_test

Definition at line 43 of file aes.c.

### 5.1.2.6 #define SETUP rijndael\_setup

Definition at line 39 of file aes.c.

## 5.1.3 Function Documentation

### 5.1.3.1 int ECB\_DEC (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)

Decrypts a block of text with AES.



**Parameters:**

- ct* The input ciphertext (16 bytes)  
*pt* The output plaintext (16 bytes)  
*skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 468 of file aes.c.

References byte, LTC\_ARGCHK, Td0, Td1, Td2, and Td3.

```

470 {
471     ulong32 s0, s1, s2, s3, t0, t1, t2, t3, *rk;
472     int Nr, r;
473
474     LTC_ARGCHK(pt != NULL);
475     LTC_ARGCHK(ct != NULL);
476     LTC_ARGCHK(skey != NULL);
477
478     Nr = skey->rijndael.Nr;
479     rk = skey->rijndael.dK;
480
481     /*
482      * map byte array block to cipher state
483      * and add initial round key:
484      */
485     LOAD32H(s0, ct); s0 ^= rk[0];
486     LOAD32H(s1, ct + 4); s1 ^= rk[1];
487     LOAD32H(s2, ct + 8); s2 ^= rk[2];
488     LOAD32H(s3, ct + 12); s3 ^= rk[3];
489
490 #ifdef LTC_SMALL_CODE
491     for (r = 0; ; r++) {
492         rk += 4;
493         t0 =
494             Td0(byte(s0, 3)) ^
495             Td1(byte(s3, 2)) ^
496             Td2(byte(s2, 1)) ^
497             Td3(byte(s1, 0)) ^
498             rk[0];
499         t1 =
500             Td0(byte(s1, 3)) ^
501             Td1(byte(s0, 2)) ^
502             Td2(byte(s3, 1)) ^
503             Td3(byte(s2, 0)) ^
504             rk[1];
505         t2 =
506             Td0(byte(s2, 3)) ^
507             Td1(byte(s1, 2)) ^
508             Td2(byte(s0, 1)) ^
509             Td3(byte(s3, 0)) ^
510             rk[2];
511         t3 =
512             Td0(byte(s3, 3)) ^
513             Td1(byte(s2, 2)) ^
514             Td2(byte(s1, 1)) ^
515             Td3(byte(s0, 0)) ^
516             rk[3];
517         if (r == Nr-2) {
518             break;
519         }
520         s0 = t0; s1 = t1; s2 = t2; s3 = t3;
521     }

```

```
522     rk += 4;
523
524 #else
525
526     /*
527     * Nr - 1 full rounds:
528     */
529     r = Nr >> 1;
530     for (;;) {
531
532         t0 =
533             Td0(byte(s0, 3)) ^
534             Td1(byte(s3, 2)) ^
535             Td2(byte(s2, 1)) ^
536             Td3(byte(s1, 0)) ^
537             rk[4];
538         t1 =
539             Td0(byte(s1, 3)) ^
540             Td1(byte(s0, 2)) ^
541             Td2(byte(s3, 1)) ^
542             Td3(byte(s2, 0)) ^
543             rk[5];
544         t2 =
545             Td0(byte(s2, 3)) ^
546             Td1(byte(s1, 2)) ^
547             Td2(byte(s0, 1)) ^
548             Td3(byte(s3, 0)) ^
549             rk[6];
550         t3 =
551             Td0(byte(s3, 3)) ^
552             Td1(byte(s2, 2)) ^
553             Td2(byte(s1, 1)) ^
554             Td3(byte(s0, 0)) ^
555             rk[7];
556
557         rk += 8;
558         if (--r == 0) {
559             break;
560         }
561
562
563         s0 =
564             Td0(byte(t0, 3)) ^
565             Td1(byte(t3, 2)) ^
566             Td2(byte(t2, 1)) ^
567             Td3(byte(t1, 0)) ^
568             rk[0];
569         s1 =
570             Td0(byte(t1, 3)) ^
571             Td1(byte(t0, 2)) ^
572             Td2(byte(t3, 1)) ^
573             Td3(byte(t2, 0)) ^
574             rk[1];
575         s2 =
576             Td0(byte(t2, 3)) ^
577             Td1(byte(t1, 2)) ^
578             Td2(byte(t0, 1)) ^
579             Td3(byte(t3, 0)) ^
580             rk[2];
581         s3 =
582             Td0(byte(t3, 3)) ^
583             Td1(byte(t2, 2)) ^
584             Td2(byte(t1, 1)) ^
585             Td3(byte(t0, 0)) ^
586             rk[3];
587     }
588 #endif
```

```

589
590  /*
591   * apply last round and
592   * map cipher state to byte array block:
593   */
594   s0 =
595       (Td4[byte(t0, 3)] & 0xff000000) ^
596       (Td4[byte(t3, 2)] & 0x00ff0000) ^
597       (Td4[byte(t2, 1)] & 0x0000ff00) ^
598       (Td4[byte(t1, 0)] & 0x000000ff) ^
599       rk[0];
600   STORE32H(s0, pt);
601   s1 =
602       (Td4[byte(t1, 3)] & 0xff000000) ^
603       (Td4[byte(t0, 2)] & 0x00ff0000) ^
604       (Td4[byte(t3, 1)] & 0x0000ff00) ^
605       (Td4[byte(t2, 0)] & 0x000000ff) ^
606       rk[1];
607   STORE32H(s1, pt+4);
608   s2 =
609       (Td4[byte(t2, 3)] & 0xff000000) ^
610       (Td4[byte(t1, 2)] & 0x00ff0000) ^
611       (Td4[byte(t0, 1)] & 0x0000ff00) ^
612       (Td4[byte(t3, 0)] & 0x000000ff) ^
613       rk[2];
614   STORE32H(s2, pt+8);
615   s3 =
616       (Td4[byte(t3, 3)] & 0xff000000) ^
617       (Td4[byte(t2, 2)] & 0x00ff0000) ^
618       (Td4[byte(t1, 1)] & 0x0000ff00) ^
619       (Td4[byte(t0, 0)] & 0x000000ff) ^
620       rk[3];
621   STORE32H(s3, pt+12);
622
623   return CRYPT_OK;
624 }

```

### 5.1.3.2 void ECB\_DONE (symmetric\_key \* skey)

Terminate the context.

#### Parameters:

*skey* The scheduled key

Definition at line 727 of file aes.c.

```

728 {
729 }

```

### 5.1.3.3 int ECB\_ENC (const unsigned char \* pt, unsigned char \* ct, symmetric\_key \* skey)

Encrypts a block of text with AES.

#### Parameters:

*pt* The input plaintext (16 bytes)

*ct* The output ciphertext (16 bytes)

*skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 289 of file aes.c.

References byte, LTC\_ARGCHK, t1, t2, t3, Te0, Te1, Te2, and Te3.

```

291 {
292     ulong32 s0, s1, s2, s3, t0, t1, t2, t3, *rk;
293     int Nr, r;
294
295     LTC_ARGCHK(pt != NULL);
296     LTC_ARGCHK(ct != NULL);
297     LTC_ARGCHK(skey != NULL);
298
299     Nr = skey->rijndael.Nr;
300     rk = skey->rijndael.eK;
301
302     /*
303      * map byte array block to cipher state
304      * and add initial round key:
305      */
306     LOAD32H(s0, pt); s0 ^= rk[0];
307     LOAD32H(s1, pt + 4); s1 ^= rk[1];
308     LOAD32H(s2, pt + 8); s2 ^= rk[2];
309     LOAD32H(s3, pt + 12); s3 ^= rk[3];
310
311 #ifdef LTC_SMALL_CODE
312
313     for (r = 0; ; r++) {
314         rk += 4;
315         t0 =
316             Te0(byte(s0, 3)) ^
317             Te1(byte(s1, 2)) ^
318             Te2(byte(s2, 1)) ^
319             Te3(byte(s3, 0)) ^
320             rk[0];
321         t1 =
322             Te0(byte(s1, 3)) ^
323             Te1(byte(s2, 2)) ^
324             Te2(byte(s3, 1)) ^
325             Te3(byte(s0, 0)) ^
326             rk[1];
327         t2 =
328             Te0(byte(s2, 3)) ^
329             Te1(byte(s3, 2)) ^
330             Te2(byte(s0, 1)) ^
331             Te3(byte(s1, 0)) ^
332             rk[2];
333         t3 =
334             Te0(byte(s3, 3)) ^
335             Te1(byte(s0, 2)) ^
336             Te2(byte(s1, 1)) ^
337             Te3(byte(s2, 0)) ^
338             rk[3];
339         if (r == Nr-2) {
340             break;
341         }
342         s0 = t0; s1 = t1; s2 = t2; s3 = t3;
343     }
344     rk += 4;
345
346 #else
347     /*
348      * Nr - 1 full rounds:
349      */
350

```

```
351     r = Nr >> 1;
352     for (;;) {
353         t0 =
354             Te0(byte(s0, 3)) ^
355             Te1(byte(s1, 2)) ^
356             Te2(byte(s2, 1)) ^
357             Te3(byte(s3, 0)) ^
358             rk[4];
359         t1 =
360             Te0(byte(s1, 3)) ^
361             Te1(byte(s2, 2)) ^
362             Te2(byte(s3, 1)) ^
363             Te3(byte(s0, 0)) ^
364             rk[5];
365         t2 =
366             Te0(byte(s2, 3)) ^
367             Te1(byte(s3, 2)) ^
368             Te2(byte(s0, 1)) ^
369             Te3(byte(s1, 0)) ^
370             rk[6];
371         t3 =
372             Te0(byte(s3, 3)) ^
373             Te1(byte(s0, 2)) ^
374             Te2(byte(s1, 1)) ^
375             Te3(byte(s2, 0)) ^
376             rk[7];
377
378         rk += 8;
379         if (--r == 0) {
380             break;
381         }
382
383         s0 =
384             Te0(byte(t0, 3)) ^
385             Te1(byte(t1, 2)) ^
386             Te2(byte(t2, 1)) ^
387             Te3(byte(t3, 0)) ^
388             rk[0];
389         s1 =
390             Te0(byte(t1, 3)) ^
391             Te1(byte(t2, 2)) ^
392             Te2(byte(t3, 1)) ^
393             Te3(byte(t0, 0)) ^
394             rk[1];
395         s2 =
396             Te0(byte(t2, 3)) ^
397             Te1(byte(t3, 2)) ^
398             Te2(byte(t0, 1)) ^
399             Te3(byte(t1, 0)) ^
400             rk[2];
401         s3 =
402             Te0(byte(t3, 3)) ^
403             Te1(byte(t0, 2)) ^
404             Te2(byte(t1, 1)) ^
405             Te3(byte(t2, 0)) ^
406             rk[3];
407     }
408
409 #endif
410
411 /*
412  * apply last round and
413  * map cipher state to byte array block:
414  */
415     s0 =
416         (Te4_3(byte(t0, 3))) ^
417         (Te4_2(byte(t1, 2))) ^
```

```

418         (Te4_1[byte(t2, 1)]) ^
419         (Te4_0[byte(t3, 0)]) ^
420         rk[0];
421     STORE32H(s0, ct);
422     s1 =
423         (Te4_3[byte(t1, 3)]) ^
424         (Te4_2[byte(t2, 2)]) ^
425         (Te4_1[byte(t3, 1)]) ^
426         (Te4_0[byte(t0, 0)]) ^
427         rk[1];
428     STORE32H(s1, ct+4);
429     s2 =
430         (Te4_3[byte(t2, 3)]) ^
431         (Te4_2[byte(t3, 2)]) ^
432         (Te4_1[byte(t0, 1)]) ^
433         (Te4_0[byte(t1, 0)]) ^
434         rk[2];
435     STORE32H(s2, ct+8);
436     s3 =
437         (Te4_3[byte(t3, 3)]) ^
438         (Te4_2[byte(t0, 2)]) ^
439         (Te4_1[byte(t1, 1)]) ^
440         (Te4_0[byte(t2, 0)]) ^
441         rk[3];
442     STORE32H(s3, ct+12);
443
444     return CRYPT_OK;
445 }

```

#### 5.1.3.4 int ECB\_KS (int \* *keysize*)

Gets suitable key size.

##### Parameters:

***keysize*** [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

##### Returns:

CRYPT\_OK if the input key size is acceptable.

Definition at line 737 of file aes.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

738 {
739     LTC_ARGCHK(keysize != NULL);
740
741     if (*keysize < 16)
742         return CRYPT_INVALID_KEYSIZE;
743     if (*keysize < 24) {
744         *keysize = 16;
745         return CRYPT_OK;
746     } else if (*keysize < 32) {
747         *keysize = 24;
748         return CRYPT_OK;
749     } else {
750         *keysize = 32;
751         return CRYPT_OK;
752     }
753 }

```

### 5.1.3.5 int ECB\_TEST (void)

Performs a self-test of the AES block cipher.

**Returns:**

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 640 of file aes.c.

References CRYPT\_NOP, CRYPT\_OK, and zeromem().

```

641 {
642     #ifndef LTC_TEST
643         return CRYPT_NOP;
644     #else
645     int err;
646     static const struct {
647         int keylen;
648         unsigned char key[32], pt[16], ct[16];
649     } tests[] = {
650         { 16,
651           { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
652             0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
653           { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
654             0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff },
655           { 0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30,
656             0xd8, 0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5, 0x5a }
657         }, {
658           24,
659           { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
660             0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
661             0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 },
662           { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
663             0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff },
664           { 0xdd, 0xa9, 0x7c, 0xa4, 0x86, 0x4c, 0xdf, 0xe0,
665             0x6e, 0xaf, 0x70, 0xa0, 0xec, 0x0d, 0x71, 0x91 }
666         }, {
667           32,
668           { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
669             0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
670             0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
671             0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f },
672           { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
673             0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff },
674           { 0x8e, 0xa2, 0xb7, 0xca, 0x51, 0x67, 0x45, 0xbf,
675             0xea, 0xfc, 0x49, 0x90, 0x4b, 0x49, 0x60, 0x89 }
676         }
677     };
678
679     symmetric_key key;
680     unsigned char tmp[2][16];
681     int i, y;
682
683     for (i = 0; i < (int)(sizeof(tests)/sizeof(tests[0])); i++) {
684         zeromem(&key, sizeof(key));
685         if ((err = rijndael_setup(tests[i].key, tests[i].keylen, 0, &key)) != CRYPT_OK) {
686             return err;
687         }
688
689         rijndael_ecb_encrypt(tests[i].pt, tmp[0], &key);
690         rijndael_ecb_decrypt(tmp[0], tmp[1], &key);
691         if (XMEMCMP(tmp[0], tests[i].ct, 16) || XMEMCMP(tmp[1], tests[i].pt, 16)) {
692             #if 0
693                 printf("\n\nTest %d failed\n", i);
694                 if (XMEMCMP(tmp[0], tests[i].ct, 16)) {

```

```

695         printf("CT: ");
696         for (i = 0; i < 16; i++) {
697             printf("%02x ", tmp[0][i]);
698         }
699         printf("\n");
700     } else {
701         printf("PT: ");
702         for (i = 0; i < 16; i++) {
703             printf("%02x ", tmp[1][i]);
704         }
705         printf("\n");
706     }
707 #endif
708     return CRYPT_FAIL_TESTVECTOR;
709 }
710
711 /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
712 for (y = 0; y < 16; y++) tmp[0][y] = 0;
713 for (y = 0; y < 1000; y++) rijndael_ecb_encrypt(tmp[0], tmp[0], &key);
714 for (y = 0; y < 1000; y++) rijndael_ecb_decrypt(tmp[0], tmp[0], &key);
715 for (y = 0; y < 16; y++) if (tmp[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
716 }
717 return CRYPT_OK;
718 #endif
719 }

```

Here is the call graph for this function:

#### 5.1.3.6 int SETUP (const unsigned char \* *key*, int *keylen*, int *num\_rounds*, [symmetric\\_key](#) \* *skey*)

Initialize the AES (Rijndael) block cipher.

##### Parameters:

- key*** The symmetric key you wish to pass
- keylen*** The key length in bytes
- num\_rounds*** The number of rounds desired (0 for default)
- skey*** The key in as scheduled by this function.

##### Returns:

CRYPT\_OK if successful

Definition at line 121 of file aes.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, LTC\_ARGCHK, rcon, and setup\_mix().

```

122 {
123     int i, j;
124     ulong32 temp, *rk;
125 #ifndef ENCRYPT_ONLY
126     ulong32 *rrk;
127 #endif
128     LTC_ARGCHK(key != NULL);
129     LTC_ARGCHK(skey != NULL);
130
131     if (keylen != 16 && keylen != 24 && keylen != 32) {
132         return CRYPT_INVALID_KEYSIZE;
133     }
134

```



```
135     if (num_rounds != 0 && num_rounds != (10 + ((keylen/8)-2)*2)) {
136         return CRYPT_INVALID_ROUNDS;
137     }
138
139     skey->rijndael.Nr = 10 + ((keylen/8)-2)*2;
140
141     /* setup the forward key */
142     i = 0;
143     rk = skey->rijndael.eK;
144     LOAD32H(rk[0], key);
145     LOAD32H(rk[1], key + 4);
146     LOAD32H(rk[2], key + 8);
147     LOAD32H(rk[3], key + 12);
148     if (keylen == 16) {
149         j = 44;
150         for (;;) {
151             temp = rk[3];
152             rk[4] = rk[0] ^ setup_mix(temp) ^ rcon[i];
153             rk[5] = rk[1] ^ rk[4];
154             rk[6] = rk[2] ^ rk[5];
155             rk[7] = rk[3] ^ rk[6];
156             if (++i == 10) {
157                 break;
158             }
159             rk += 4;
160         }
161     } else if (keylen == 24) {
162         j = 52;
163         LOAD32H(rk[4], key + 16);
164         LOAD32H(rk[5], key + 20);
165         for (;;) {
166             #ifdef _MSC_VER
167                 temp = skey->rijndael.eK[rk - skey->rijndael.eK + 5];
168             #else
169                 temp = rk[5];
170             #endif
171             rk[ 6] = rk[ 0] ^ setup_mix(temp) ^ rcon[i];
172             rk[ 7] = rk[ 1] ^ rk[ 6];
173             rk[ 8] = rk[ 2] ^ rk[ 7];
174             rk[ 9] = rk[ 3] ^ rk[ 8];
175             if (++i == 8) {
176                 break;
177             }
178             rk[10] = rk[ 4] ^ rk[ 9];
179             rk[11] = rk[ 5] ^ rk[10];
180             rk += 6;
181         }
182     } else if (keylen == 32) {
183         j = 60;
184         LOAD32H(rk[4], key + 16);
185         LOAD32H(rk[5], key + 20);
186         LOAD32H(rk[6], key + 24);
187         LOAD32H(rk[7], key + 28);
188         for (;;) {
189             #ifdef _MSC_VER
190                 temp = skey->rijndael.eK[rk - skey->rijndael.eK + 7];
191             #else
192                 temp = rk[7];
193             #endif
194             rk[ 8] = rk[ 0] ^ setup_mix(temp) ^ rcon[i];
195             rk[ 9] = rk[ 1] ^ rk[ 8];
196             rk[10] = rk[ 2] ^ rk[ 9];
197             rk[11] = rk[ 3] ^ rk[10];
198             if (++i == 7) {
199                 break;
200             }
201             temp = rk[11];
```

```

202         rk[12] = rk[ 4] ^ setup_mix(RORc(temp, 8));
203         rk[13] = rk[ 5] ^ rk[12];
204         rk[14] = rk[ 6] ^ rk[13];
205         rk[15] = rk[ 7] ^ rk[14];
206         rk += 8;
207     }
208 } else {
209     /* this can't happen */
210     return CRYPT_ERROR;
211 }
212
213 #ifndef ENCRYPT_ONLY
214     /* setup the inverse key now */
215     rk = skey->rijndael.dK;
216     rrk = skey->rijndael.eK + j - 4;
217
218     /* apply the inverse MixColumn transform to all round keys but the first and the last: */
219     /* copy first */
220     *rk++ = *rrk++;
221     *rk++ = *rrk++;
222     *rk++ = *rrk++;
223     *rk = *rrk;
224     rk -= 3; rrk -= 3;
225
226     for (i = 1; i < skey->rijndael.Nr; i++) {
227         rrk -= 4;
228         rk += 4;
229         #ifdef LTC_SMALL_CODE
230             temp = rrk[0];
231             rk[0] = setup_mix2(temp);
232             temp = rrk[1];
233             rk[1] = setup_mix2(temp);
234             temp = rrk[2];
235             rk[2] = setup_mix2(temp);
236             temp = rrk[3];
237             rk[3] = setup_mix2(temp);
238         #else
239             temp = rrk[0];
240             rk[0] =
241                 Tks0[byte(temp, 3)] ^
242                 Tks1[byte(temp, 2)] ^
243                 Tks2[byte(temp, 1)] ^
244                 Tks3[byte(temp, 0)];
245             temp = rrk[1];
246             rk[1] =
247                 Tks0[byte(temp, 3)] ^
248                 Tks1[byte(temp, 2)] ^
249                 Tks2[byte(temp, 1)] ^
250                 Tks3[byte(temp, 0)];
251             temp = rrk[2];
252             rk[2] =
253                 Tks0[byte(temp, 3)] ^
254                 Tks1[byte(temp, 2)] ^
255                 Tks2[byte(temp, 1)] ^
256                 Tks3[byte(temp, 0)];
257             temp = rrk[3];
258             rk[3] =
259                 Tks0[byte(temp, 3)] ^
260                 Tks1[byte(temp, 2)] ^
261                 Tks2[byte(temp, 1)] ^
262                 Tks3[byte(temp, 0)];
263         #endif
264     }
265
266     /* copy last */
267     rrk -= 4;

```

```

269     rk  += 4;
270     *rk++ = *rrk++;
271     *rk++ = *rrk++;
272     *rk++ = *rrk++;
273     *rk  = *rrk;
274 #endif /* ENCRYPT_ONLY */
275
276     return CRYPT_OK;
277 }

```

Here is the call graph for this function:

#### 5.1.3.7 static [ulong32](#) setup\_mix ([ulong32](#) temp) [static]

Definition at line 93 of file aes.c.

References byte, Te4\_0, Te4\_1, Te4\_2, and Te4\_3.

Referenced by SETUP().

```

94 {
95     return (Te4_3[byte(temp, 2)]) ^
96           (Te4_2[byte(temp, 1)]) ^
97           (Te4_1[byte(temp, 0)]) ^
98           (Te4_0[byte(temp, 3)]);
99 }

```

### 5.1.4 Variable Documentation

#### 5.1.4.1 const struct [ltc\\_cipher\\_descriptor](#) aes\_desc

**Initial value:**

```

{
    "aes",
    6,
    16, 32, 16, 10,
    SETUP, ECB_ENC, ECB_DEC, ECB_TEST, ECB_DONE, ECB_KS,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}

```

Definition at line 55 of file aes.c.

Referenced by yarrow\_start().

#### 5.1.4.2 const struct [ltc\\_cipher\\_descriptor](#) rijndael\_desc

**Initial value:**

```

{
    "rijndael",
    6,
    16, 32, 16, 10,
    SETUP, ECB_ENC, ECB_DEC, ECB_TEST, ECB_DONE, ECB_KS,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}

```

Definition at line 46 of file aes.c.

Referenced by yarrow\_start().

## 5.2 ciphers/aes/aes\_tab.c File Reference

### 5.2.1 Detailed Description

AES tables.

Definition in file [aes\\_tab.c](#).

This graph shows which files directly or indirectly include this file:

#### Defines

- #define [Te0\(x\)](#) [TE0\[x\]](#)
- #define [Te1\(x\)](#) [TE1\[x\]](#)
- #define [Te2\(x\)](#) [TE2\[x\]](#)
- #define [Te3\(x\)](#) [TE3\[x\]](#)
- #define [Td0\(x\)](#) [TD0\[x\]](#)
- #define [Td1\(x\)](#) [TD1\[x\]](#)
- #define [Td2\(x\)](#) [TD2\[x\]](#)
- #define [Td3\(x\)](#) [TD3\[x\]](#)

#### Variables

- static const [ulong32](#) [TE0](#) [256]
- static const [ulong32](#) [Te4](#) [256]
- static const [ulong32](#) [TD0](#) [256]
- static const [ulong32](#) [Td4](#) [256]
- static const [ulong32](#) [TE1](#) [256]
- static const [ulong32](#) [TE2](#) [256]
- static const [ulong32](#) [TE3](#) [256]
- static const [ulong32](#) [Te4\\_0](#) []
- static const [ulong32](#) [Te4\\_1](#) []
- static const [ulong32](#) [Te4\\_2](#) []
- static const [ulong32](#) [Te4\\_3](#) []
- static const [ulong32](#) [TD1](#) [256]
- static const [ulong32](#) [TD2](#) [256]
- static const [ulong32](#) [TD3](#) [256]
- static const [ulong32](#) [Tks0](#) []
- static const [ulong32](#) [Tks1](#) []
- static const [ulong32](#) [Tks2](#) []
- static const [ulong32](#) [Tks3](#) []
- static const [ulong32](#) [rcon](#) []

### 5.2.2 Define Documentation

#### 5.2.2.1 #define [Td0\(x\)](#) [TD0\[x\]](#)

Definition at line 328 of file [aes\\_tab.c](#).

Referenced by [ECB\\_DEC\(\)](#).

#### 5.2.2.2 #define Td1(x) TD1[x]

Definition at line 329 of file aes\_tab.c.

Referenced by ECB\_DEC().

#### 5.2.2.3 #define Td2(x) TD2[x]

Definition at line 330 of file aes\_tab.c.

Referenced by ECB\_DEC().

#### 5.2.2.4 #define Td3(x) TD3[x]

Definition at line 331 of file aes\_tab.c.

Referenced by ECB\_DEC().

#### 5.2.2.5 #define Te0(x) TE0[x]

Definition at line 323 of file aes\_tab.c.

Referenced by ECB\_ENC(), and four\_rounds().

#### 5.2.2.6 #define Te1(x) TE1[x]

Definition at line 324 of file aes\_tab.c.

Referenced by ECB\_ENC(), and four\_rounds().

#### 5.2.2.7 #define Te2(x) TE2[x]

Definition at line 325 of file aes\_tab.c.

Referenced by ECB\_ENC(), and four\_rounds().

#### 5.2.2.8 #define Te3(x) TE3[x]

Definition at line 326 of file aes\_tab.c.

Referenced by ECB\_ENC(), and four\_rounds().

### 5.2.3 Variable Documentation

#### 5.2.3.1 const **ulong32** rcon[] [static]

**Initial value:**

```
{
    0x01000000UL, 0x02000000UL, 0x04000000UL, 0x08000000UL,
    0x10000000UL, 0x20000000UL, 0x40000000UL, 0x80000000UL,
    0x1B000000UL, 0x36000000UL,
}
```

Definition at line 1020 of file aes\_tab.c.

Referenced by SETUP().

#### **5.2.3.2**   **const** **ulong32** **TD0**[256]   [static]

Definition at line 168 of file aes\_tab.c.

#### **5.2.3.3**   **const** **ulong32** **TD1**[256]   [static]

Definition at line 677 of file aes\_tab.c.

#### **5.2.3.4**   **const** **ulong32** **TD2**[256]   [static]

Definition at line 743 of file aes\_tab.c.

#### **5.2.3.5**   **const** **ulong32** **TD3**[256]   [static]

Definition at line 809 of file aes\_tab.c.

#### **5.2.3.6**   **const** **ulong32** **Td4**[256]   [static]

Definition at line 235 of file aes\_tab.c.

#### **5.2.3.7**   **const** **ulong32** **TE0**[256]   [static]

Definition at line 30 of file aes\_tab.c.

#### **5.2.3.8**   **const** **ulong32** **TE1**[256]   [static]

Definition at line 333 of file aes\_tab.c.

#### **5.2.3.9**   **const** **ulong32** **TE2**[256]   [static]

Definition at line 399 of file aes\_tab.c.

#### **5.2.3.10**   **const** **ulong32** **TE3**[256]   [static]

Definition at line 465 of file aes\_tab.c.

#### **5.2.3.11**   **const** **ulong32** **Te4**[256]   [static]

Definition at line 98 of file aes\_tab.c.

**5.2.3.12** `const ulong32 Te4_0[]` `[static]`

Definition at line 534 of file aes\_tab.c.

Referenced by setup\_mix().

**5.2.3.13** `const ulong32 Te4_1[]` `[static]`

Definition at line 569 of file aes\_tab.c.

Referenced by setup\_mix().

**5.2.3.14** `const ulong32 Te4_2[]` `[static]`

Definition at line 604 of file aes\_tab.c.

Referenced by setup\_mix().

**5.2.3.15** `const ulong32 Te4_3[]` `[static]`

Definition at line 639 of file aes\_tab.c.

Referenced by setup\_mix().

**5.2.3.16** `const ulong32 Tks0[]` `[static]`

Definition at line 876 of file aes\_tab.c.

**5.2.3.17** `const ulong32 Tks1[]` `[static]`

Definition at line 911 of file aes\_tab.c.

**5.2.3.18** `const ulong32 Tks2[]` `[static]`

Definition at line 946 of file aes\_tab.c.

**5.2.3.19** `const ulong32 Tks3[]` `[static]`

Definition at line 981 of file aes\_tab.c.

## 5.3 ciphers/anubis.c File Reference

### 5.3.1 Detailed Description

Anubis implementation derived from public domain source Authors: Paulo S.L.M.

Barreto and Vincent Rijmen.

Definition in file [anubis.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for anubis.c:

### Defines

- `#define MIN_N 4`
- `#define MAX_N 10`
- `#define MIN_ROUNDS (8 + MIN_N)`
- `#define MAX_ROUNDS (8 + MAX_N)`
- `#define MIN_KEYSIZEB (4*MIN_N)`
- `#define MAX_KEYSIZEB (4*MAX_N)`
- `#define BLOCKSIZE 128`
- `#define BLOCKSIZEB (BLOCKSIZE/8)`

### Functions

- `int anubis_setup (const unsigned char *key, int keylen, int num_rounds, symmetric\_key *skey)`  
*Initialize the Anubis block cipher.*
- `static void anubis_crypt (const unsigned char *plaintext, unsigned char *ciphertext, ulong32 round-Key[18+1][4], int R)`
- `int anubis_ecb_encrypt (const unsigned char *pt, unsigned char *ct, symmetric\_key *skey)`  
*Encrypts a block of text with Anubis.*
- `int anubis_ecb_decrypt (const unsigned char *ct, unsigned char *pt, symmetric\_key *skey)`  
*Decrypts a block of text with Anubis.*
- `int anubis_test (void)`  
*Performs a self-test of the Anubis block cipher.*
- `void anubis_done (symmetric\_key *skey)`  
*Terminate the context.*
- `int anubis_keysize (int *keysize)`  
*Gets suitable key size.*



## Variables

- const struct `ltc_cipher_descriptor` `anubis_desc`
- static const `ulong32` `T0` [256]
- static const `ulong32` `T1` [256]
- static const `ulong32` `T2` [256]
- static const `ulong32` `T3` [256]
- static const `ulong32` `T4` [256]
- static const `ulong32` `T5` [256]
- static const `ulong32` `rc` [ ]

*The round constants.*

### 5.3.2 Define Documentation

#### 5.3.2.1 `#define BLOCKSIZE 128`

Definition at line 41 of file `anubis.c`.

#### 5.3.2.2 `#define BLOCKSIZEB (BLOCKSIZE/8)`

Definition at line 42 of file `anubis.c`.

#### 5.3.2.3 `#define MAX_KEYSIZEB (4*MAX_N)`

Definition at line 40 of file `anubis.c`.

#### 5.3.2.4 `#define MAX_N 10`

Definition at line 36 of file `anubis.c`.

Referenced by `anubis_setup()`.

#### 5.3.2.5 `#define MAX_ROUNDS (8 + MAX_N)`

Definition at line 38 of file `anubis.c`.

#### 5.3.2.6 `#define MIN_KEYSIZEB (4*MIN_N)`

Definition at line 39 of file `anubis.c`.

#### 5.3.2.7 `#define MIN_N 4`

Definition at line 35 of file `anubis.c`.

#### 5.3.2.8 `#define MIN_ROUNDS (8 + MIN_N)`

Definition at line 37 of file `anubis.c`.

### 5.3.3 Function Documentation

#### 5.3.3.1 static void anubis\_crypt (const unsigned char \* *plaintext*, unsigned char \* *ciphertext*, ulong32 roundKey[18+1][4], int R) [static]

Definition at line 1039 of file anubis.c.

Referenced by anubis\_ecb\_decrypt(), and anubis\_ecb\_encrypt().

```

1040
1041     int i, pos, r;
1042     ulong32 state[4];
1043     ulong32 inter[4];
1044
1045     /*
1046     * map plaintext block to cipher state (mu)
1047     * and add initial round key (sigma[K^0]):
1048     */
1049     for (i = 0, pos = 0; i < 4; i++, pos += 4) {
1050         state[i] =
1051             (plaintext[pos] << 24) ^
1052             (plaintext[pos + 1] << 16) ^
1053             (plaintext[pos + 2] << 8) ^
1054             (plaintext[pos + 3] ) ^
1055             roundKey[0][i];
1056     }
1057
1058     /*
1059     * R - 1 full rounds:
1060     */
1061     for (r = 1; r < R; r++) {
1062         inter[0] =
1063             T0[(state[0] >> 24) & 0xff] ^
1064             T1[(state[1] >> 24) & 0xff] ^
1065             T2[(state[2] >> 24) & 0xff] ^
1066             T3[(state[3] >> 24) & 0xff] ^
1067             roundKey[r][0];
1068         inter[1] =
1069             T0[(state[0] >> 16) & 0xff] ^
1070             T1[(state[1] >> 16) & 0xff] ^
1071             T2[(state[2] >> 16) & 0xff] ^
1072             T3[(state[3] >> 16) & 0xff] ^
1073             roundKey[r][1];
1074         inter[2] =
1075             T0[(state[0] >> 8) & 0xff] ^
1076             T1[(state[1] >> 8) & 0xff] ^
1077             T2[(state[2] >> 8) & 0xff] ^
1078             T3[(state[3] >> 8) & 0xff] ^
1079             roundKey[r][2];
1080         inter[3] =
1081             T0[(state[0] ) & 0xff] ^
1082             T1[(state[1] ) & 0xff] ^
1083             T2[(state[2] ) & 0xff] ^
1084             T3[(state[3] ) & 0xff] ^
1085             roundKey[r][3];
1086         state[0] = inter[0];
1087         state[1] = inter[1];
1088         state[2] = inter[2];
1089         state[3] = inter[3];
1090     }
1091
1092     /*
1093     * last round:
1094     */
1095     inter[0] =
1096         (T0[(state[0] >> 24) & 0xff] & 0xff000000U) ^

```

```

1097     (T1[(state[1] >> 24) & 0xff] & 0x00ff0000U) ^
1098     (T2[(state[2] >> 24) & 0xff] & 0x0000ff00U) ^
1099     (T3[(state[3] >> 24) & 0xff] & 0x000000ffU) ^
1100     roundKey[R][0];
1101     inter[1] =
1102     (T0[(state[0] >> 16) & 0xff] & 0xff000000U) ^
1103     (T1[(state[1] >> 16) & 0xff] & 0x00ff0000U) ^
1104     (T2[(state[2] >> 16) & 0xff] & 0x0000ff00U) ^
1105     (T3[(state[3] >> 16) & 0xff] & 0x000000ffU) ^
1106     roundKey[R][1];
1107     inter[2] =
1108     (T0[(state[0] >> 8) & 0xff] & 0xff000000U) ^
1109     (T1[(state[1] >> 8) & 0xff] & 0x00ff0000U) ^
1110     (T2[(state[2] >> 8) & 0xff] & 0x0000ff00U) ^
1111     (T3[(state[3] >> 8) & 0xff] & 0x000000ffU) ^
1112     roundKey[R][2];
1113     inter[3] =
1114     (T0[(state[0] >> 0) & 0xff] & 0xff000000U) ^
1115     (T1[(state[1] >> 0) & 0xff] & 0x00ff0000U) ^
1116     (T2[(state[2] >> 0) & 0xff] & 0x0000ff00U) ^
1117     (T3[(state[3] >> 0) & 0xff] & 0x000000ffU) ^
1118     roundKey[R][3];
1119
1120     /*
1121     * map cipher state to ciphertext block (mu^{-1}):
1122     */
1123     for (i = 0, pos = 0; i < 4; i++, pos += 4) {
1124         ulong32 w = inter[i];
1125         ciphertext[pos] = (unsigned char)(w >> 24);
1126         ciphertext[pos + 1] = (unsigned char)(w >> 16);
1127         ciphertext[pos + 2] = (unsigned char)(w >> 8);
1128         ciphertext[pos + 3] = (unsigned char)(w >> 0);
1129     }
1130 }

```

### 5.3.3.2 void anubis\_done (symmetric\_key \* skey)

Terminate the context.

#### Parameters:

*skey* The scheduled key

Definition at line 1521 of file anubis.c.

```

1522 {
1523 }

```

### 5.3.3.3 int anubis\_ecb\_decrypt (const unsigned char \* ct, unsigned char \* pt, symmetric\_key \* skey)

Decrypts a block of text with Anubis.

#### Parameters:

*ct* The input ciphertext (16 bytes)

*pt* The output plaintext (16 bytes)

*skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 1155 of file anubis.c.

References anubis\_crypt(), CRYPT\_OK, and LTC\_ARGCHK.

Referenced by anubis\_test().

```

1156 {
1157     LTC_ARGCHK(pt != NULL);
1158     LTC_ARGCHK(ct != NULL);
1159     LTC_ARGCHK(skey != NULL);
1160     anubis_crypt(ct, pt, skey->anubis.roundKeyDec, skey->anubis.R);
1161     return CRYPT_OK;
1162 }
```

Here is the call graph for this function:

#### 5.3.3.4 int anubis\_ecb\_encrypt (const unsigned char \* *pt*, unsigned char \* *ct*, [symmetric\\_key](#) \* *skey*)

Encrypts a block of text with Anubis.

**Parameters:**

- pt* The input plaintext (16 bytes)
- ct* The output ciphertext (16 bytes)
- skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 1139 of file anubis.c.

References anubis\_crypt(), CRYPT\_OK, and LTC\_ARGCHK.

Referenced by anubis\_test().

```

1140 {
1141     LTC_ARGCHK(pt != NULL);
1142     LTC_ARGCHK(ct != NULL);
1143     LTC_ARGCHK(skey != NULL);
1144     anubis_crypt(pt, ct, skey->anubis.roundKeyEnc, skey->anubis.R);
1145     return CRYPT_OK;
1146 }
```

Here is the call graph for this function:

#### 5.3.3.5 int anubis\_keysize (int \* *keysize*)

Gets suitable key size.

**Parameters:**

- keysize* [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

**Returns:**

CRYPT\_OK if the input key size is acceptable.

Definition at line 1530 of file anubis.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

1531 {
1532     LTC_ARGCHK(keysize != NULL);
1533     if (*keysize >= 40) {
1534         *keysize = 40;
1535     } else if (*keysize >= 36) {
1536         *keysize = 36;
1537     } else if (*keysize >= 32) {
1538         *keysize = 32;
1539     } else if (*keysize >= 28) {
1540         *keysize = 28;
1541     } else if (*keysize >= 24) {
1542         *keysize = 24;
1543     } else if (*keysize >= 20) {
1544         *keysize = 20;
1545     } else if (*keysize >= 16) {
1546         *keysize = 16;
1547     } else {
1548         return CRYPT_INVALID_KEYSIZE;
1549     }
1550     return CRYPT_OK;
1551 }
```

### 5.3.3.6 int anubis\_setup (const unsigned char \* *key*, int *keylen*, int *num\_rounds*, symmetric\_key \* *skey*)

Initialize the Anubis block cipher.

**Parameters:**

*key* The symmetric key you wish to pass

*keylen* The key length in bytes

*num\_rounds* The number of rounds desired (0 for default)

*skey* The key in as scheduled by this function.

**Returns:**

CRYPT\_OK if successful

Definition at line 897 of file anubis.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, LTC\_ARGCHK, MAX\_N, N, and R.

Referenced by anubis\_test().

```

899 {
900     int N, R, i, pos, r;
901     ulong32 kappa[MAX_N];
902     ulong32 inter[MAX_N];
903     ulong32 v, K0, K1, K2, K3;
904
905     LTC_ARGCHK(key != NULL);
906     LTC_ARGCHK(skey != NULL);
```

```

907
908 /* Valid sizes (in bytes) are 16, 20, 24, 28, 32, 36, and 40. */
909 if ((keylen & 3) || (keylen < 16) || (keylen > 40)) {
910     return CRYPT_INVALID_KEYSIZE;
911 }
912 skey->anubis.keyBits = keylen*8;
913
914 /*
915  * determine the N length parameter:
916  * (N.B. it is assumed that the key length is valid!)
917  */
918 N = skey->anubis.keyBits >> 5;
919
920 /*
921  * determine number of rounds from key size:
922  */
923 skey->anubis.R = R = 8 + N;
924
925 if (num_rounds != 0 && num_rounds != skey->anubis.R) {
926     return CRYPT_INVALID_ROUNDS;
927 }
928
929 /*
930  * map cipher key to initial key state (mu):
931  */
932 for (i = 0, pos = 0; i < N; i++, pos += 4) {
933     kappa[i] =
934         (key[pos] << 24) ^
935         (key[pos + 1] << 16) ^
936         (key[pos + 2] << 8) ^
937         (key[pos + 3]);
938 }
939
940 /*
941  * generate R + 1 round keys:
942  */
943 for (r = 0; r <= R; r++) {
944     /*
945      * generate r-th round key K^r:
946      */
947     K0 = T4[(kappa[N - 1] >> 24) & 0xff];
948     K1 = T4[(kappa[N - 1] >> 16) & 0xff];
949     K2 = T4[(kappa[N - 1] >> 8) & 0xff];
950     K3 = T4[(kappa[N - 1]) & 0xff];
951     for (i = N - 2; i >= 0; i--) {
952         K0 = T4[(kappa[i] >> 24) & 0xff] ^
953             (T5[(K0 >> 24) & 0xff] & 0xff000000U) ^
954             (T5[(K0 >> 16) & 0xff] & 0x00ff0000U) ^
955             (T5[(K0 >> 8) & 0xff] & 0x0000ff00U) ^
956             (T5[(K0) & 0xff] & 0x000000ffU);
957         K1 = T4[(kappa[i] >> 16) & 0xff] ^
958             (T5[(K1 >> 24) & 0xff] & 0xff000000U) ^
959             (T5[(K1 >> 16) & 0xff] & 0x00ff0000U) ^
960             (T5[(K1 >> 8) & 0xff] & 0x0000ff00U) ^
961             (T5[(K1) & 0xff] & 0x000000ffU);
962         K2 = T4[(kappa[i] >> 8) & 0xff] ^
963             (T5[(K2 >> 24) & 0xff] & 0xff000000U) ^
964             (T5[(K2 >> 16) & 0xff] & 0x00ff0000U) ^
965             (T5[(K2 >> 8) & 0xff] & 0x0000ff00U) ^
966             (T5[(K2) & 0xff] & 0x000000ffU);
967         K3 = T4[(kappa[i]) & 0xff] ^
968             (T5[(K3 >> 24) & 0xff] & 0xff000000U) ^
969             (T5[(K3 >> 16) & 0xff] & 0x00ff0000U) ^
970             (T5[(K3 >> 8) & 0xff] & 0x0000ff00U) ^
971             (T5[(K3) & 0xff] & 0x000000ffU);
972     }
973     /*

```

```

974     -- this is the code to use with the large U tables:
975     K0 = K1 = K2 = K3 = 0;
976     for (i = 0; i < N; i++) {
977         K0 ^= U[i][(kappa[i] >> 24) & 0xff];
978         K1 ^= U[i][(kappa[i] >> 16) & 0xff];
979         K2 ^= U[i][(kappa[i] >> 8) & 0xff];
980         K3 ^= U[i][(kappa[i]      ) & 0xff];
981     }
982     */
983     skey->anubis.roundKeyEnc[r][0] = K0;
984     skey->anubis.roundKeyEnc[r][1] = K1;
985     skey->anubis.roundKeyEnc[r][2] = K2;
986     skey->anubis.roundKeyEnc[r][3] = K3;
987
988     /*
989     * compute kappa^{r+1} from kappa^r:
990     */
991     if (r == R) {
992         break;
993     }
994     for (i = 0; i < N; i++) {
995         int j = i;
996         inter[i] = T0[(kappa[j--] >> 24) & 0xff]; if (j < 0) j = N - 1;
997         inter[i] ^= T1[(kappa[j--] >> 16) & 0xff]; if (j < 0) j = N - 1;
998         inter[i] ^= T2[(kappa[j--] >> 8) & 0xff]; if (j < 0) j = N - 1;
999         inter[i] ^= T3[(kappa[j]      ) & 0xff];
1000     }
1001     kappa[0] = inter[0] ^ rc[r];
1002     for (i = 1; i < N; i++) {
1003         kappa[i] = inter[i];
1004     }
1005 }
1006
1007 /*
1008 * generate inverse key schedule: K'^0 = K^R, K'^R = K^0, K'^r = theta(K^{R-r}):
1009 */
1010 for (i = 0; i < 4; i++) {
1011     skey->anubis.roundKeyDec[0][i] = skey->anubis.roundKeyEnc[R][i];
1012     skey->anubis.roundKeyDec[R][i] = skey->anubis.roundKeyEnc[0][i];
1013 }
1014 for (r = 1; r < R; r++) {
1015     for (i = 0; i < 4; i++) {
1016         v = skey->anubis.roundKeyEnc[R - r][i];
1017         skey->anubis.roundKeyDec[r][i] =
1018             T0[T4[(v >> 24) & 0xff] & 0xff] ^
1019             T1[T4[(v >> 16) & 0xff] & 0xff] ^
1020             T2[T4[(v >> 8) & 0xff] & 0xff] ^
1021             T3[T4[(v      ) & 0xff] & 0xff];
1022     }
1023 }
1024
1025 return CRYPT_OK;
1026 }

```

### 5.3.3.7 int anubis\_test (void)

Performs a self-test of the Anubis block cipher.

#### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 1168 of file anubis.c.

References `anubis_ecb_decrypt()`, `anubis_ecb_encrypt()`, `anubis_setup()`, `CRYPT_FAIL_TESTVECTOR`, `CRYPT_NOP`, and `XMEMCMP`.

```

1169 {
1170 #if !defined(LTC_TEST)
1171     return CRYPT_NOP;
1172 #else
1173     static const struct test {
1174         int keylen;
1175         unsigned char pt[16], ct[16], key[40];
1176     } tests[] = {
1177 #ifndef ANUBIS_TWEAK
1178         /***** ORIGINAL ANUBIS *****/
1179         /* 128 bit keys */
1180         {
1181             16,
1182             { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1183               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1184             { 0xF0, 0x68, 0x60, 0xFC, 0x67, 0x30, 0xE8, 0x18,
1185               0xF1, 0x32, 0xC7, 0x8A, 0xF4, 0x13, 0x2A, 0xFE },
1186             { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1187               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
1188         }, {
1189             16,
1190             { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1191               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1192             { 0xA8, 0x66, 0x84, 0x80, 0x07, 0x74, 0x5C, 0x89,
1193               0xFC, 0x5E, 0xB5, 0xBA, 0xD4, 0xFE, 0x32, 0x6D },
1194             { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1195               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
1196         },
1197         /* 160-bit keys */
1198         {
1199             20,
1200             { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1201               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1202             { 0xBD, 0x5E, 0x32, 0xBE, 0x51, 0x67, 0xA8, 0xE2,
1203               0x72, 0xD7, 0x95, 0x0F, 0x83, 0xC6, 0x8C, 0x31 },
1204             { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1205               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1206             { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1207               0x00, 0x00, 0x00, 0x00 }
1208         }, {
1209             20,
1210             { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1211               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1212             { 0x4C, 0x1F, 0x86, 0x2E, 0x11, 0xEB, 0xCE, 0xEB,
1213               0xFE, 0xB9, 0x73, 0xC9, 0xDF, 0xEF, 0x7A, 0xDB },
1214             { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1215               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1216             { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1217               0x00, 0x00, 0x00, 0x00 }
1218         },
1219         /* 192-bit keys */
1220         {
1221             24,
1222             { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1223               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1224             { 0x17, 0xAC, 0x57, 0x44, 0x9D, 0x59, 0x61, 0x66,
1225               0xD0, 0xC7, 0x9E, 0x04, 0x7C, 0xC7, 0x58, 0xF0 },
1226             { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1227               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1228             { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1229               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
1230         }, {
1231             24,
1232             { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1233               0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },

```



```

1233     { 0x71, 0x52, 0xB4, 0xEB, 0x1D, 0xAA, 0x36, 0xFD,
1234       0x57, 0x14, 0x5F, 0x57, 0x04, 0x9F, 0x70, 0x74 },
1235     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1236       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1237       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 }
1238 },
1239
1240 /* 224-bit keys */
1241 {
1242     28,
1243     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1244       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1245     { 0xA2, 0xF0, 0xA6, 0xB9, 0x17, 0x93, 0x2A, 0x3B,
1246       0xEF, 0x08, 0xE8, 0x7A, 0x58, 0xD6, 0xF8, 0x53 },
1247     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1248       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1249       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1250       0x00, 0x00, 0x00, 0x00 }
1251 }, {
1252     28,
1253     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1254       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1255     { 0xF0, 0xCA, 0xFC, 0x78, 0x8B, 0x4B, 0x4E, 0x53,
1256       0x8B, 0xC4, 0x32, 0x6A, 0xF5, 0xB9, 0x1B, 0x5F },
1257     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1258       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1259       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1260       0x00, 0x00, 0x00, 0x01 }
1261 },
1262
1263 /* 256-bit keys */
1264 {
1265     32,
1266     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1267       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1268     { 0xE0, 0x86, 0xAC, 0x45, 0x6B, 0x3C, 0xE5, 0x13,
1269       0xED, 0xF5, 0xDF, 0xDD, 0xD6, 0x3B, 0x71, 0x93 },
1270     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1271       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1272       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1273       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
1274 }, {
1275     32,
1276     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1277       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1278     { 0x50, 0x01, 0xB9, 0xF5, 0x21, 0xC1, 0xC1, 0x29,
1279       0x00, 0xD5, 0xEC, 0x98, 0x2B, 0x9E, 0xE8, 0x21 },
1280     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1281       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1282       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1283       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 }
1284 },
1285
1286 /* 288-bit keys */
1287 {
1288     36,
1289     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1290       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1291     { 0xE8, 0xF4, 0xAF, 0x2B, 0x21, 0xA0, 0x87, 0x9B,
1292       0x41, 0x95, 0xB9, 0x71, 0x75, 0x79, 0x04, 0x7C },
1293     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1294       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1295       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1296       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1297       0x00, 0x00, 0x00, 0x00 }
1298 }, {
1299     36,

```

```

1300     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1301       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1302     { 0xE6, 0xA6, 0xA5, 0xBC, 0x8B, 0x63, 0x6F, 0xE2,
1303       0xBD, 0xA7, 0xA7, 0x53, 0xAB, 0x40, 0x22, 0xE0 },
1304     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1305       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1306       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1307       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1308       0x00, 0x00, 0x00, 0x01 }
1309 },
1310
1311 /* 320-bit keys */
1312 {
1313     40,
1314     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1315       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1316     { 0x17, 0x04, 0xD7, 0x2C, 0xC6, 0x85, 0x76, 0x02,
1317       0x4B, 0xCC, 0x39, 0x80, 0xD8, 0x22, 0xEA, 0xA4 },
1318     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1319       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1320       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1321       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1322       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
1323 }, {
1324     40,
1325     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1326       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1327     { 0x7A, 0x41, 0xE6, 0x7D, 0x4F, 0xD8, 0x64, 0xF0,
1328       0x44, 0xA8, 0x3C, 0x73, 0x81, 0x7E, 0x53, 0xD8 },
1329     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1330       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1331       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1332       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1333       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 }
1334 }
1335 #else
1336 /**** Tweaked ANUBIS *****/
1337 /* 128 bit keys */
1338 {
1339     16,
1340     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1341       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1342     { 0xB8, 0x35, 0xBD, 0xC3, 0x34, 0x82, 0x9D, 0x83,
1343       0x71, 0xBF, 0xA3, 0x71, 0xE4, 0xB3, 0xC4, 0xFD },
1344     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1345       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
1346 }, {
1347     16,
1348     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1349       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1350     { 0xE6, 0x14, 0x1E, 0xAF, 0xEB, 0xE0, 0x59, 0x3C,
1351       0x48, 0xE1, 0xCD, 0xF2, 0x1B, 0xBA, 0xA1, 0x89 },
1352     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1353       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 }
1354 },
1355
1356 /* 160-bit keys */
1357 {
1358     20,
1359     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1360       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1361     { 0x97, 0x59, 0x79, 0x4B, 0x5C, 0xA0, 0x70, 0x73,
1362       0x24, 0xEF, 0xB3, 0x58, 0x67, 0xCA, 0xD4, 0xB3 },
1363     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1364       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1365       0x00, 0x00, 0x00, 0x00 }
1366 }, {

```

```

1367     20,
1368     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1369       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1370     { 0xB8, 0x0D, 0xFB, 0x9B, 0xE4, 0xA1, 0x58, 0x87,
1371       0xB3, 0x76, 0xD5, 0x02, 0x18, 0x95, 0xC1, 0x2E },
1372     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1373       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1374       0x00, 0x00, 0x00, 0x01 }
1375 },
1376
1377 /* 192-bit keys */
1378 {
1379     24,
1380     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1381       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1382     { 0x7D, 0x62, 0x3B, 0x52, 0xC7, 0x4C, 0x64, 0xD8,
1383       0xEB, 0xC7, 0x2D, 0x57, 0x97, 0x85, 0x43, 0x8F },
1384     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1385       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1386       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
1387 }, {
1388     24,
1389     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1390       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1391     { 0xB1, 0x0A, 0x59, 0xDD, 0x5D, 0x5D, 0x8D, 0x67,
1392       0xEC, 0xEE, 0x4A, 0xC4, 0xBE, 0x4F, 0xA8, 0x4F },
1393     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1394       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1395       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 }
1396 },
1397
1398 /* 224-bit keys */
1399 {
1400     28,
1401     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1402       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1403     { 0x68, 0x9E, 0x05, 0x94, 0x6A, 0x94, 0x43, 0x8F,
1404       0xE7, 0x8E, 0x37, 0x3D, 0x24, 0x97, 0x92, 0xF5 },
1405     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1406       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1407       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1408       0x00, 0x00, 0x00, 0x00 }
1409 }, {
1410     28,
1411     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1412       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1413     { 0xDD, 0xB7, 0xB0, 0xB4, 0xE9, 0xB4, 0x9B, 0x9C,
1414       0x38, 0x20, 0x25, 0x0B, 0x47, 0xC2, 0x1F, 0x89 },
1415     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1416       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1417       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1418       0x00, 0x00, 0x00, 0x01 }
1419 },
1420
1421 /* 256-bit keys */
1422 {
1423     32,
1424     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1425       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1426     { 0x96, 0x00, 0xF0, 0x76, 0x91, 0x69, 0x29, 0x87,
1427       0xF5, 0xE5, 0x97, 0xDB, 0xDB, 0xAF, 0x1B, 0x0A },
1428     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1429       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1430       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1431       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
1432 }, {
1433     32,

```

```

1434     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1435       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1436     { 0x69, 0x9C, 0xAF, 0xDD, 0x94, 0xC7, 0xBC, 0x60,
1437       0x44, 0xFE, 0x02, 0x05, 0x8A, 0x6E, 0xEF, 0xBD },
1438     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1439       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1440       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1441       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 }
1442 },
1443
1444 /* 288-bit keys */
1445 {
1446     36,
1447     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1448       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1449     { 0x0F, 0xC7, 0xA2, 0xC0, 0x11, 0x17, 0xAC, 0x43,
1450       0x52, 0x5E, 0xDF, 0x6C, 0xF3, 0x96, 0x33, 0x6C },
1451     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1452       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1453       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1454       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1455       0x00, 0x00, 0x00, 0x00 }
1456 }, {
1457     36,
1458     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1459       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1460     { 0xAD, 0x08, 0x4F, 0xED, 0x55, 0xA6, 0x94, 0x3E,
1461       0x7E, 0x5E, 0xED, 0x05, 0xA1, 0x9D, 0x41, 0xB4 },
1462     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1463       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1464       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1465       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1466       0x00, 0x00, 0x00, 0x01 }
1467 },
1468
1469 /* 320-bit keys */
1470 {
1471     40,
1472     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1473       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1474     { 0xFE, 0xE2, 0x0E, 0x2A, 0x9D, 0xC5, 0x83, 0xBA,
1475       0xA3, 0xA6, 0xD6, 0xA6, 0xF2, 0xE8, 0x06, 0xA5 },
1476     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1477       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1478       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1479       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1480       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
1481 }, {
1482     40,
1483     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1484       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1485     { 0x86, 0x3D, 0xCC, 0x4A, 0x60, 0x34, 0x9C, 0x28,
1486       0xA7, 0xDA, 0xA4, 0x3B, 0x0A, 0xD7, 0xFD, 0xC7 },
1487     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1488       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1489       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1490       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
1491       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 }
1492 }
1493 #endif
1494 };
1495
1496 int x, y;
1497 unsigned char buf[2][16];
1498 symmetric_key skey;
1499
1500 for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
1501     anubis_setup(tests[x].key, tests[x].keylen, 0, &skey);

```

```

1501     anubis_ecb_encrypt(tests[x].pt, buf[0], &skey);
1502     anubis_ecb_decrypt(buf[0], buf[1], &skey);
1503     if (XMEMCMP(buf[0], tests[x].ct, 16) || XMEMCMP(buf[1], tests[x].pt, 16)) {
1504         return CRYPT_FAIL_TESTVECTOR;
1505     }
1506
1507     for (y = 0; y < 1000; y++) anubis_ecb_encrypt(buf[0], buf[0], &skey);
1508     for (y = 0; y < 1000; y++) anubis_ecb_decrypt(buf[0], buf[0], &skey);
1509     if (XMEMCMP(buf[0], tests[x].ct, 16)) {
1510         return CRYPT_FAIL_TESTVECTOR;
1511     }
1512
1513     }
1514     return CRYPT_OK;
1515 #endif
1516 }

```

Here is the call graph for this function:

## 5.3.4 Variable Documentation

### 5.3.4.1 const struct [ltc\\_cipher\\_descriptor](#) [anubis\\_desc](#)

**Initial value:**

```

{
    "anubis",
    19,
    16, 40, 16, 12,
    &anubis_setup,
    &anubis_ecb_encrypt,
    &anubis_ecb_decrypt,
    &anubis_test,
    &anubis_done,
    &anubis_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}

```

Definition at line 22 of file anubis.c.

Referenced by [yarrow\\_start\(\)](#).

### 5.3.4.2 const [ulong32](#) [rc\[\]](#) [static]

**Initial value:**

```

{
    0xba542f74U, 0x53d3d24dU, 0x50ac8dbfU, 0x70529a4cU,
    0xad597d1U, 0x33515ba6U, 0xde48a899U, 0xdb32b7fcU,
    0xe39e919bU, 0xe2bb416eU, 0xa5cb6b95U, 0xa1f3b102U,
    0xcc41d14U, 0xc363da5dU, 0x5fdc7dcdU, 0x7f5a6c5cU,
    0xf726ffedU, 0xe89d6f8eU, 0x19a0f089U,
}

```

The round constants.

Definition at line 458 of file anubis.c.

**5.3.4.3** `const ulong32 T0[256]` `[static]`

Definition at line 53 of file anubis.c.

Referenced by `khazad_crypt()`, and `khazad_setup()`.

**5.3.4.4** `const ulong32 T1[256]` `[static]`

Definition at line 120 of file anubis.c.

Referenced by `khazad_crypt()`, and `khazad_setup()`.

**5.3.4.5** `const ulong32 T2[256]` `[static]`

Definition at line 187 of file anubis.c.

Referenced by `khazad_crypt()`, `khazad_setup()`, and `rounds()`.

**5.3.4.6** `const ulong32 T3[256]` `[static]`

Definition at line 254 of file anubis.c.

Referenced by `khazad_crypt()`, and `khazad_setup()`.

**5.3.4.7** `const ulong32 T4[256]` `[static]`

Definition at line 321 of file anubis.c.

Referenced by `khazad_crypt()`, and `khazad_setup()`.

**5.3.4.8** `const ulong32 T5[256]` `[static]`

Definition at line 388 of file anubis.c.

Referenced by `khazad_crypt()`, and `khazad_setup()`.

## 5.4 ciphers/blowfish.c File Reference

### 5.4.1 Detailed Description

Implementation of the Blowfish block cipher, Tom St Denis.

Definition in file [blowfish.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for blowfish.c:

### Defines

- `#define F(x) ((S1[byte(x,3)] + S2[byte(x,2)]) ^ S3[byte(x,1)]) + S4[byte(x,0)]`

### Functions

- `int blowfish_setup (const unsigned char *key, int keylen, int num_rounds, symmetric_key *skey)`  
*Initialize the Blowfish block cipher.*
- `int blowfish_ecb_encrypt (const unsigned char *pt, unsigned char *ct, symmetric_key *skey)`  
*Encrypts a block of text with Blowfish.*
- `int blowfish_ecb_decrypt (const unsigned char *ct, unsigned char *pt, symmetric_key *skey)`  
*Decrypts a block of text with Blowfish.*
- `int blowfish_test (void)`  
*Performs a self-test of the Blowfish block cipher.*
- `void blowfish_done (symmetric_key *skey)`  
*Terminate the context.*
- `int blowfish_keysize (int *keysize)`  
*Gets suitable key size.*

### Variables

- `const struct ltc_cipher_descriptor blowfish_desc`
- `static const ulong32 ORIG_P [16+2]`
- `static const ulong32 ORIG_S [4][256]`

### 5.4.2 Define Documentation

#### 5.4.2.1 `#define F(x) ((S1[byte(x,3)] + S2[byte(x,2)]) ^ S3[byte(x,1)]) + S4[byte(x,0)]`

Definition at line 378 of file blowfish.c.

Referenced by `blowfish_ecb_decrypt()`, `blowfish_ecb_encrypt()`, and `rounds()`.

### 5.4.3 Function Documentation

#### 5.4.3.1 void blowfish\_done ([symmetric\\_key](#) \* *skey*)

Terminate the context.

**Parameters:**

*skey* The scheduled key

Definition at line 568 of file blowfish.c.

```
569 {
570 }
```

#### 5.4.3.2 int blowfish\_ecb\_decrypt (const unsigned char \* *ct*, unsigned char \* *pt*, [symmetric\\_key](#) \* *skey*)

Decrypts a block of text with Blowfish.

**Parameters:**

*ct* The input ciphertext (8 bytes)

*pt* The output plaintext (8 bytes)

*skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 455 of file blowfish.c.

References F, LTC\_ARGCHK, R, S1, S2, S3, and S4.

```
457 {
458     ulong32 L, R;
459     int r;
460 #ifndef __GNUC__
461     ulong32 *S1, *S2, *S3, *S4;
462 #endif
463
464     LTC_ARGCHK(pt != NULL);
465     LTC_ARGCHK(ct != NULL);
466     LTC_ARGCHK(skey != NULL);
467
468 #ifndef __GNUC__
469     S1 = skey->blowfish.S[0];
470     S2 = skey->blowfish.S[1];
471     S3 = skey->blowfish.S[2];
472     S4 = skey->blowfish.S[3];
473 #endif
474
475     /* load it */
476     LOAD32H(R, &ct[0]);
477     LOAD32H(L, &ct[4]);
478
479     /* undo last keying */
480     R ^= skey->blowfish.K[17];
481     L ^= skey->blowfish.K[16];
482 }
```



```

483     /* do 16 rounds */
484     for (r = 15; r > 0; ) {
485         L ^= F(R); R ^= skey->blowfish.K[r--];
486         R ^= F(L); L ^= skey->blowfish.K[r--];
487         L ^= F(R); R ^= skey->blowfish.K[r--];
488         R ^= F(L); L ^= skey->blowfish.K[r--];
489     }
490
491     /* store */
492     STORE32H(L, &pt[0]);
493     STORE32H(R, &pt[4]);
494     return CRYPT_OK;
495 }

```

#### 5.4.3.3 int blowfish\_ecb\_encrypt (const unsigned char \* *pt*, unsigned char \* *ct*, [symmetric\\_key](#) \* *skey*)

Encrypts a block of text with Blowfish.

##### Parameters:

- pt* The input plaintext (8 bytes)
- ct* The output ciphertext (8 bytes)
- skey* The key as scheduled

##### Returns:

CRYPT\_OK if successful

Definition at line 393 of file blowfish.c.

References F, LTC\_ARGCHK, R, S1, S2, S3, and S4.

```

395 {
396     ulong32 L, R;
397     int r;
398     #ifndef __GNUC__
399     ulong32 *S1, *S2, *S3, *S4;
400 #endif
401
402     LTC_ARGCHK(pt != NULL);
403     LTC_ARGCHK(ct != NULL);
404     LTC_ARGCHK(skey != NULL);
405
406     #ifndef __GNUC__
407     S1 = skey->blowfish.S[0];
408     S2 = skey->blowfish.S[1];
409     S3 = skey->blowfish.S[2];
410     S4 = skey->blowfish.S[3];
411 #endif
412
413     /* load it */
414     LOAD32H(L, &pt[0]);
415     LOAD32H(R, &pt[4]);
416
417     /* do 16 rounds */
418     for (r = 0; r < 16; ) {
419         L ^= skey->blowfish.K[r++]; R ^= F(L);
420         R ^= skey->blowfish.K[r++]; L ^= F(R);
421         L ^= skey->blowfish.K[r++]; R ^= F(L);
422         R ^= skey->blowfish.K[r++]; L ^= F(R);
423     }

```

```

424
425     /* last keying */
426     R ^= skey->blowfish.K[17];
427     L ^= skey->blowfish.K[16];
428
429     /* store */
430     STORE32H(R, &ct[0]);
431     STORE32H(L, &ct[4]);
432
433     return CRYPT_OK;
434 }

```

#### 5.4.3.4 int blowfish\_keysize (int \*keysize)

Gets suitable key size.

##### Parameters:

**keysize** [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

##### Returns:

CRYPT\_OK if the input key size is acceptable.

Definition at line 577 of file blowfish.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

578 {
579     LTC_ARGCHK(keysize != NULL);
580
581     if (*keysize < 8) {
582         return CRYPT_INVALID_KEYSIZE;
583     } else if (*keysize > 56) {
584         *keysize = 56;
585     }
586     return CRYPT_OK;
587 }

```

#### 5.4.3.5 int blowfish\_setup (const unsigned char \*key, int keylen, int num\_rounds, symmetric\_key \*skey)

Initialize the Blowfish block cipher.

##### Parameters:

**key** The symmetric key you wish to pass

**keylen** The key length in bytes

**num\_rounds** The number of rounds desired (0 for default)

**skey** The key in as scheduled by this function.

##### Returns:

CRYPT\_OK if successful

Definition at line 308 of file blowfish.c.

References B, CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, and LTC\_ARGCHK.

Referenced by blowfish\_test().

```
310 {
311     ulong32 x, y, z, A;
312     unsigned char B[8];
313
314     LTC_ARGCHK(key != NULL);
315     LTC_ARGCHK(skey != NULL);
316
317     /* check key length */
318     if (keylen < 8 || keylen > 56) {
319         return CRYPT_INVALID_KEYSIZE;
320     }
321
322     /* check rounds */
323     if (num_rounds != 0 && num_rounds != 16) {
324         return CRYPT_INVALID_ROUNDS;
325     }
326
327     /* load in key bytes (Supplied by David Hopwood) */
328     for (x = y = 0; x < 18; x++) {
329         A = 0;
330         for (z = 0; z < 4; z++) {
331             A = (A << 8) | ((ulong32)key[y++] & 255);
332             if (y == (ulong32)keylen) {
333                 y = 0;
334             }
335         }
336         skey->blowfish.K[x] = ORIG_P[x] ^ A;
337     }
338
339     /* copy sboxes */
340     for (x = 0; x < 4; x++) {
341         for (y = 0; y < 256; y++) {
342             skey->blowfish.S[x][y] = ORIG_S[x][y];
343         }
344     }
345
346     /* encrypt K array */
347     for (x = 0; x < 8; x++) {
348         B[x] = 0;
349     }
350
351     for (x = 0; x < 18; x += 2) {
352         /* encrypt it */
353         blowfish_ecb_encrypt(B, B, skey);
354         /* copy it */
355         LOAD32H(skey->blowfish.K[x], &B[0]);
356         LOAD32H(skey->blowfish.K[x+1], &B[4]);
357     }
358
359     /* encrypt S array */
360     for (x = 0; x < 4; x++) {
361         for (y = 0; y < 256; y += 2) {
362             /* encrypt it */
363             blowfish_ecb_encrypt(B, B, skey);
364             /* copy it */
365             LOAD32H(skey->blowfish.S[x][y], &B[0]);
366             LOAD32H(skey->blowfish.S[x][y+1], &B[4]);
367         }
368     }
369
370 #ifdef LTC_CLEAN_STACK
371     zeromem(B, sizeof(B));
372 #endif
373
374     return CRYPT_OK;
375 }
```

### 5.4.3.6 int blowfish\_test (void)

Performs a self-test of the Blowfish block cipher.

#### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 511 of file blowfish.c.

References blowfish\_setup(), CRYPT\_NOP, and CRYPT\_OK.

```

512 {
513     #ifndef LTC_TEST
514         return CRYPT_NOP;
515     #else
516         int err;
517         symmetric_key key;
518         static const struct {
519             unsigned char key[8], pt[8], ct[8];
520         } tests[] = {
521             {
522                 { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
523                 { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
524                 { 0x4E, 0xF9, 0x97, 0x45, 0x61, 0x98, 0xDD, 0x78}
525             },
526             {
527                 { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF},
528                 { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF},
529                 { 0x51, 0x86, 0x6F, 0xD5, 0xB8, 0x5E, 0xCB, 0x8A}
530             },
531             {
532                 { 0x30, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
533                 { 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01},
534                 { 0x7D, 0x85, 0x6F, 0x9A, 0x61, 0x30, 0x63, 0xF2}
535             }
536         };
537         unsigned char tmp[2][8];
538         int x, y;
539
540         for (x = 0; x < (int)(sizeof(tests) / sizeof(tests[0])); x++) {
541             /* setup key */
542             if ((err = blowfish_setup(tests[x].key, 8, 16, &key)) != CRYPT_OK) {
543                 return err;
544             }
545
546             /* encrypt and decrypt */
547             blowfish_ecb_encrypt(tests[x].pt, tmp[0], &key);
548             blowfish_ecb_decrypt(tmp[0], tmp[1], &key);
549
550             /* compare */
551             if ((XMEMCMP(tmp[0], tests[x].ct, 8) != 0) || (XMEMCMP(tmp[1], tests[x].pt, 8) != 0)) {
552                 return CRYPT_FAIL_TESTVECTOR;
553             }
554
555             /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
556             for (y = 0; y < 8; y++) tmp[0][y] = 0;
557             for (y = 0; y < 1000; y++) blowfish_ecb_encrypt(tmp[0], tmp[0], &key);
558             for (y = 0; y < 1000; y++) blowfish_ecb_decrypt(tmp[0], tmp[0], &key);
559             for (y = 0; y < 8; y++) if (tmp[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
560         }
561         return CRYPT_OK;
562     #endif
563 }

```

Here is the call graph for this function:

## 5.4.4 Variable Documentation

### 5.4.4.1 `const struct ltc_cipher_descriptor blowfish_desc`

**Initial value:**

```
{
    "blowfish",
    0,
    8, 56, 8, 16,
    &blowfish_setup,
    &blowfish_ecb_encrypt,
    &blowfish_ecb_decrypt,
    &blowfish_test,
    &blowfish_done,
    &blowfish_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 19 of file blowfish.c.

Referenced by `yarrow_start()`.

### 5.4.4.2 `const ulong32 ORIG_P[16+2]` [static]

**Initial value:**

```
{
    0x243F6A88UL, 0x85A308D3UL, 0x13198A2EUL, 0x03707344UL,
    0xA4093822UL, 0x299F31D0UL, 0x082EFA98UL, 0xEC4E6C89UL,
    0x452821E6UL, 0x38D01377UL, 0xBE5466CFUL, 0x34E90C6CUL,
    0xC0AC29B7UL, 0xC97C50DDUL, 0x3F84D5B5UL, 0xB5470917UL,
    0x9216D5D9UL, 0x8979FB1BUL
}
```

Definition at line 33 of file blowfish.c.

### 5.4.4.3 `const ulong32 ORIG_S[4][256]` [static]

Definition at line 41 of file blowfish.c.

## 5.5 ciphers/cast5.c File Reference

### 5.5.1 Detailed Description

Implementation of CAST5 (RFC 2144) by Tom St Denis.

Definition in file [cast5.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cast5.c:

### Defines

- `#define GB(x, i) (((x[(15-i)>>2])>>(unsigned)(8*((15-i)&3)))&255)`
- `#define INLINE`

### Functions

- `int cast5_setup` (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the CAST5 block cipher.*
- static `INLINE ulong32 FI` (ulong32 R, [ulong32](#) Km, [ulong32](#) Kr)
- static `INLINE ulong32 FII` (ulong32 R, [ulong32](#) Km, [ulong32](#) Kr)
- static `INLINE ulong32 FIII` (ulong32 R, [ulong32](#) Km, [ulong32](#) Kr)
- `int cast5_ecb_encrypt` (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with CAST5.*
- `int cast5_ecb_decrypt` (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with CAST5.*
- `int cast5_test` (void)  
*Performs a self-test of the CAST5 block cipher.*
- `void cast5_done` ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- `int cast5_keysize` (int \*keysize)  
*Gets suitable key size.*

### Variables

- const struct [ltc\\_cipher\\_descriptor](#) [cast5\\_desc](#)
- static const [ulong32](#) S1 [256]
- static const [ulong32](#) S2 [256]
- static const [ulong32](#) S3 [256]
- static const [ulong32](#) S4 [256]
- static const [ulong32](#) S5 [256]
- static const [ulong32](#) S6 [256]
- static const [ulong32](#) S7 [256]
- static const [ulong32](#) S8 [256]

## 5.5.2 Define Documentation

### 5.5.2.1 `#define GB(x, i) (((x[(15-i)>>2])>>(unsigned)(8*((15-i)&3)))&255)`

Definition at line 397 of file cast5.c.

### 5.5.2.2 `#define INLINE`

Definition at line 505 of file cast5.c.

## 5.5.3 Function Documentation

### 5.5.3.1 `void cast5_done (symmetric_key * skey)`

Terminate the context.

#### Parameters:

*skey* The scheduled key

Definition at line 696 of file cast5.c.

```
697 {
698 }
```

### 5.5.3.2 `int cast5_ecb_decrypt (const unsigned char * ct, unsigned char * pt, symmetric_key * skey)`

Decrypts a block of text with CAST5.

#### Parameters:

*ct* The input ciphertext (8 bytes)

*pt* The output plaintext (8 bytes)

*skey* The key as scheduled

Definition at line 594 of file cast5.c.

References CRYPT\_OK, FI(), FII(), FIII(), LTC\_ARGCHK, and R.

```
596 {
597     ulong32 R, L;
598
599     LTC_ARGCHK(pt != NULL);
600     LTC_ARGCHK(ct != NULL);
601     LTC_ARGCHK(skey != NULL);
602
603     LOAD32H(R, &ct[0]);
604     LOAD32H(L, &ct[4]);
605     if (skey->cast5.keylen > 10) {
606         R ^= FI(L, skey->cast5.K[15], skey->cast5.K[31]);
607         L ^= FIII(R, skey->cast5.K[14], skey->cast5.K[30]);
608         R ^= FII(L, skey->cast5.K[13], skey->cast5.K[29]);
609         L ^= FI(R, skey->cast5.K[12], skey->cast5.K[28]);
610     }
611     R ^= FIII(L, skey->cast5.K[11], skey->cast5.K[27]);
```

```

612     L ^= FII(R, skey->cast5.K[10], skey->cast5.K[26]);
613     R ^= FI(L, skey->cast5.K[9], skey->cast5.K[25]);
614     L ^= FIII(R, skey->cast5.K[8], skey->cast5.K[24]);
615     R ^= FII(L, skey->cast5.K[7], skey->cast5.K[23]);
616     L ^= FI(R, skey->cast5.K[6], skey->cast5.K[22]);
617     R ^= FIII(L, skey->cast5.K[5], skey->cast5.K[21]);
618     L ^= FII(R, skey->cast5.K[4], skey->cast5.K[20]);
619     R ^= FI(L, skey->cast5.K[3], skey->cast5.K[19]);
620     L ^= FIII(R, skey->cast5.K[2], skey->cast5.K[18]);
621     R ^= FII(L, skey->cast5.K[1], skey->cast5.K[17]);
622     L ^= FI(R, skey->cast5.K[0], skey->cast5.K[16]);
623     STORE32H(L, &pt[0]);
624     STORE32H(R, &pt[4]);
625
626     return CRYPT_OK;
627 }

```

Here is the call graph for this function:

### 5.5.3.3 int cast5\_ecb\_encrypt (const unsigned char \* *pt*, unsigned char \* *ct*, symmetric\_key \* *skey*)

Encrypts a block of text with CAST5.

#### Parameters:

- pt* The input plaintext (8 bytes)
- ct* The output ciphertext (8 bytes)
- skey* The key as scheduled

Definition at line 541 of file cast5.c.

References CRYPT\_OK, FI(), FII(), FIII(), LTC\_ARGCHK, and R.

```

543 {
544     ulong32 R, L;
545
546     LTC_ARGCHK(pt != NULL);
547     LTC_ARGCHK(ct != NULL);
548     LTC_ARGCHK(skey != NULL);
549
550     LOAD32H(L, &pt[0]);
551     LOAD32H(R, &pt[4]);
552     L ^= FI(R, skey->cast5.K[0], skey->cast5.K[16]);
553     R ^= FII(L, skey->cast5.K[1], skey->cast5.K[17]);
554     L ^= FIII(R, skey->cast5.K[2], skey->cast5.K[18]);
555     R ^= FI(L, skey->cast5.K[3], skey->cast5.K[19]);
556     L ^= FII(R, skey->cast5.K[4], skey->cast5.K[20]);
557     R ^= FIII(L, skey->cast5.K[5], skey->cast5.K[21]);
558     L ^= FI(R, skey->cast5.K[6], skey->cast5.K[22]);
559     R ^= FII(L, skey->cast5.K[7], skey->cast5.K[23]);
560     L ^= FIII(R, skey->cast5.K[8], skey->cast5.K[24]);
561     R ^= FI(L, skey->cast5.K[9], skey->cast5.K[25]);
562     L ^= FII(R, skey->cast5.K[10], skey->cast5.K[26]);
563     R ^= FIII(L, skey->cast5.K[11], skey->cast5.K[27]);
564     if (skey->cast5.keylen > 10) {
565         L ^= FI(R, skey->cast5.K[12], skey->cast5.K[28]);
566         R ^= FII(L, skey->cast5.K[13], skey->cast5.K[29]);
567         L ^= FIII(R, skey->cast5.K[14], skey->cast5.K[30]);
568         R ^= FI(L, skey->cast5.K[15], skey->cast5.K[31]);
569     }
570     STORE32H(R, &ct[0]);
571     STORE32H(L, &ct[4]);
572     return CRYPT_OK;
573 }

```



Here is the call graph for this function:

#### 5.5.3.4 int cast5\_keysize (int \* *keysize*)

Gets suitable key size.

##### Parameters:

*keysize* [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

##### Returns:

CRYPT\_OK if the input key size is acceptable.

Definition at line 705 of file cast5.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

706 {
707     LTC_ARGCHK(keysize != NULL);
708     if (*keysize < 5) {
709         return CRYPT_INVALID_KEYSIZE;
710     } else if (*keysize > 16) {
711         *keysize = 16;
712     }
713     return CRYPT_OK;
714 }
```

#### 5.5.3.5 int cast5\_setup (const unsigned char \* *key*, int *keylen*, int *num\_rounds*, [symmetric\\_key](#) \* *skey*)

Initialize the CAST5 block cipher.

##### Parameters:

*key* The symmetric key you wish to pass

*keylen* The key length in bytes

*num\_rounds* The number of rounds desired (0 for default)

*skey* The key in as scheduled by this function.

##### Returns:

CRYPT\_OK if successful

Definition at line 411 of file cast5.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, LTC\_ARGCHK, XMEMCPY, and zeromem().

Referenced by cast5\_test().

```

413 {
414     ulong32 x[4], z[4];
415     unsigned char buf[16];
416     int y, i;
417
418     LTC_ARGCHK(key != NULL);
```



```

486     zeromem(z, sizeof(z));
487 #endif
488
489     return CRYPT_OK;
490 }

```

Here is the call graph for this function:

### 5.5.3.6 int cast5\_test (void)

Performs a self-test of the CAST5 block cipher.

#### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 642 of file cast5.c.

References cast5\_setup(), CRYPT\_NOP, and CRYPT\_OK.

```

643 {
644 #ifndef LTC_TEST
645     return CRYPT_NOP;
646 #else
647     static const struct {
648         int keylen;
649         unsigned char key[16];
650         unsigned char pt[8];
651         unsigned char ct[8];
652     } tests[] = {
653         { 16,
654           {0x01, 0x23, 0x45, 0x67, 0x12, 0x34, 0x56, 0x78, 0x23, 0x45, 0x67, 0x89, 0x34, 0x56, 0x78, 0x9A},
655           {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF},
656           {0x23, 0x8B, 0x4F, 0xE5, 0x84, 0x7E, 0x44, 0xB2}
657         },
658         { 10,
659           {0x01, 0x23, 0x45, 0x67, 0x12, 0x34, 0x56, 0x78, 0x23, 0x45, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
660           {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF},
661           {0xEB, 0x6A, 0x71, 0x1A, 0x2C, 0x02, 0x27, 0x1B},
662         },
663         { 5,
664           {0x01, 0x23, 0x45, 0x67, 0x12, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
665           {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF},
666           {0x7A, 0xC8, 0x16, 0xD1, 0x6E, 0x9B, 0x30, 0x2E}
667         }
668     };
669     int i, y, err;
670     symmetric_key key;
671     unsigned char tmp[2][8];
672
673     for (i = 0; i < (int)(sizeof(tests) / sizeof(tests[0])); i++) {
674         if ((err = cast5_setup(tests[i].key, tests[i].keylen, 0, &key)) != CRYPT_OK) {
675             return err;
676         }
677         cast5_ecb_encrypt(tests[i].pt, tmp[0], &key);
678         cast5_ecb_decrypt(tmp[0], tmp[1], &key);
679         if ((XMEMCMP(tmp[0], tests[i].ct, 8) != 0) || (XMEMCMP(tmp[1], tests[i].pt, 8) != 0)) {
680             return CRYPT_FAIL_TESTVECTOR;
681         }
682         /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
683         for (y = 0; y < 8; y++) tmp[0][y] = 0;
684         for (y = 0; y < 1000; y++) cast5_ecb_encrypt(tmp[0], tmp[0], &key);
685         for (y = 0; y < 1000; y++) cast5_ecb_decrypt(tmp[0], tmp[0], &key);
686         for (y = 0; y < 8; y++) if (tmp[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;

```

```

687
688     }
689     return CRYPT_OK;
690 #endif
691 }

```

Here is the call graph for this function:

#### 5.5.3.7 static **INLINE** **ulong32** FI (**ulong32** R, **ulong32** Km, **ulong32** Kr) [static]

Definition at line 508 of file cast5.c.

References byte, I, ROL, S1, S2, S3, and S4.

Referenced by cast5\_ecb\_decrypt(), cast5\_ecb\_encrypt(), and FO().

```

509 {
510     ulong32 I;
511     I = (Km + R);
512     I = ROL(I, Kr);
513     return ((S1[byte(I, 3)] ^ S2[byte(I,2)]) - S3[byte(I,1)]) + S4[byte(I,0)];
514 }

```

#### 5.5.3.8 static **INLINE** **ulong32** FII (**ulong32** R, **ulong32** Km, **ulong32** Kr) [static]

Definition at line 516 of file cast5.c.

References byte, I, ROL, S1, S2, S3, and S4.

Referenced by cast5\_ecb\_decrypt(), and cast5\_ecb\_encrypt().

```

517 {
518     ulong32 I;
519     I = (Km ^ R);
520     I = ROL(I, Kr);
521     return ((S1[byte(I, 3)] - S2[byte(I,2)]) + S3[byte(I,1)]) ^ S4[byte(I,0)];
522 }

```

#### 5.5.3.9 static **INLINE** **ulong32** FIII (**ulong32** R, **ulong32** Km, **ulong32** Kr) [static]

Definition at line 524 of file cast5.c.

References byte, I, ROL, S1, S2, S3, and S4.

Referenced by cast5\_ecb\_decrypt(), and cast5\_ecb\_encrypt().

```

525 {
526     ulong32 I;
527     I = (Km - R);
528     I = ROL(I, Kr);
529     return ((S1[byte(I, 3)] + S2[byte(I,2)]) ^ S3[byte(I,1)]) - S4[byte(I,0)];
530 }

```

### 5.5.4 Variable Documentation

#### 5.5.4.1 const struct **ltc\_cipher\_descriptor** cast5\_desc

Initial value:

```
{
    "cast5",
    15,
    5, 16, 8, 16,
    &cast5_setup,
    &cast5_ecb_encrypt,
    &cast5_ecb_decrypt,
    &cast5_test,
    &cast5_done,
    &cast5_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 20 of file cast5.c.

Referenced by yarrow\_start().

#### 5.5.4.2 const [ulong32 S1](#)[256] [static]

Definition at line 33 of file cast5.c.

Referenced by blowfish\_ecb\_decrypt(), blowfish\_ecb\_encrypt(), FI(), FII(), FIII(), twofish\_ecb\_decrypt(), and twofish\_ecb\_encrypt().

#### 5.5.4.3 const [ulong32 S2](#)[256] [static]

Definition at line 78 of file cast5.c.

Referenced by blowfish\_ecb\_decrypt(), blowfish\_ecb\_encrypt(), FI(), FII(), FIII(), twofish\_ecb\_decrypt(), and twofish\_ecb\_encrypt().

#### 5.5.4.4 const [ulong32 S3](#)[256] [static]

Definition at line 123 of file cast5.c.

Referenced by blowfish\_ecb\_decrypt(), blowfish\_ecb\_encrypt(), FI(), FII(), FIII(), twofish\_ecb\_decrypt(), and twofish\_ecb\_encrypt().

#### 5.5.4.5 const [ulong32 S4](#)[256] [static]

Definition at line 168 of file cast5.c.

Referenced by blowfish\_ecb\_decrypt(), blowfish\_ecb\_encrypt(), FI(), FII(), FIII(), twofish\_ecb\_decrypt(), and twofish\_ecb\_encrypt().

#### 5.5.4.6 const [ulong32 S5](#)[256] [static]

Definition at line 213 of file cast5.c.

#### 5.5.4.7 const [ulong32 S6](#)[256] [static]

Definition at line 258 of file cast5.c.

**5.5.4.8** `const ulong32 S7[256]` `[static]`

Definition at line 303 of file cast5.c.

Referenced by FI().

**5.5.4.9** `const ulong32 S8[256]` `[static]`

Definition at line 348 of file cast5.c.

## 5.6 ciphers/des.c File Reference

### 5.6.1 Detailed Description

DES code submitted by Dobes Vandermeer.

Definition in file [des.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for des.c:

### Defines

- `#define` [EN0](#) 0
- `#define` [DE1](#) 1

### Functions

- static void [cookey](#) (const [ulong32](#) \*raw1, [ulong32](#) \*keyout)
- static void [deskey](#) (const unsigned char \*key, short edf, [ulong32](#) \*keyout)
- static void [desfunc](#) ([ulong32](#) \*block, const [ulong32](#) \*keys)
- int [des\\_setup](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the DES block cipher.*
- int [des3\\_setup](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the 3DES-EDE block cipher.*
- int [des\\_ecb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with DES.*
- int [des\\_ecb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with DES.*
- int [des3\\_ecb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with 3DES-EDE.*
- int [des3\\_ecb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with 3DES-EDE.*
- int [des\\_test](#) (void)  
*Performs a self-test of the DES block cipher.*
- int [des3\\_test](#) (void)
- void [des\\_done](#) ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- void [des3\\_done](#) ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- int [des\\_keysize](#) (int \*keysize)

*Gets suitable key size.*

- int [des3\\_keysize](#) (int \*keysize)

*Gets suitable key size.*

## Variables

- const struct [ltc\\_cipher\\_descriptor](#) [des\\_desc](#)
- const struct [ltc\\_cipher\\_descriptor](#) [des3\\_desc](#)
- static const [ulong32](#) [bytebit](#) [8]
- static const [ulong32](#) [bigbyte](#) [24]
- static const unsigned char [pc1](#) [56]
- static const unsigned char [totrot](#) [16]
- static const unsigned char [pc2](#) [48]
- static const [ulong32](#) [SP1](#) [64]
- static const [ulong32](#) [SP2](#) [64]
- static const [ulong32](#) [SP3](#) [64]
- static const [ulong32](#) [SP4](#) [64]
- static const [ulong32](#) [SP5](#) [64]
- static const [ulong32](#) [SP6](#) [64]
- static const [ulong32](#) [SP7](#) [64]
- static const [ulong32](#) [SP8](#) [64]
- static const [ulong64](#) [des\\_ip](#) [8][256]
- static const [ulong64](#) [des\\_fp](#) [8][256]

## 5.6.2 Define Documentation

### 5.6.2.1 #define DE1 1

Definition at line 21 of file [des.c](#).

Referenced by [des3\\_setup\(\)](#), and [des\\_setup\(\)](#).

### 5.6.2.2 #define EN0 0

Definition at line 20 of file [des.c](#).

Referenced by [des3\\_setup\(\)](#), and [des\\_setup\(\)](#).

## 5.6.3 Function Documentation

### 5.6.3.1 static void [cookey](#) (const [ulong32](#) \* [raw1](#), [ulong32](#) \* [keyout](#)) [static]

Definition at line 1366 of file [des.c](#).

```
1368 {
1369     ulong32 *cook;
1370     const ulong32 *raw0;
1371     ulong32 dough[32];
1372     int i;
```



```

1373
1374     cook = dough;
1375     for(i=0; i < 16; i++, rawl++)
1376     {
1377         raw0 = rawl++;
1378         *cook    = (*raw0 & 0x00fc0000L) << 6;
1379         *cook    |= (*raw0 & 0x00000fc0L) << 10;
1380         *cook    |= (*raw1 & 0x00fc0000L) >> 10;
1381         *cook++  |= (*raw1 & 0x00000fc0L) >> 6;
1382         *cook    = (*raw0 & 0x0003f000L) << 12;
1383         *cook    |= (*raw0 & 0x0000003fL) << 16;
1384         *cook    |= (*raw1 & 0x0003f000L) >> 4;
1385         *cook++  |= (*raw1 & 0x0000003fL);
1386     }
1387
1388     XMEMCPY(keyout, dough, sizeof dough);
1389 }

```

### 5.6.3.2 void des3\_done (symmetric\_key \*skey)

Terminate the context.

#### Parameters:

*skey* The scheduled key

Definition at line 1862 of file des.c.

```

1863 {
1864 }

```

### 5.6.3.3 int des3\_ecb\_decrypt (const unsigned char \*ct, unsigned char \*pt, symmetric\_key \*skey)

Decrypts a block of text with 3DES-EDE.

#### Parameters:

*ct* The input ciphertext (8 bytes)

*pt* The output plaintext (8 bytes)

*skey* The key as scheduled

#### Returns:

CRYPT\_OK if successful

Definition at line 1653 of file des.c.

References CRYPT\_OK, desfunc(), and LTC\_ARGCHK.

```

1654 {
1655     ulong32 work[2];
1656     LTC_ARGCHK(pt != NULL);
1657     LTC_ARGCHK(ct != NULL);
1658     LTC_ARGCHK(skey != NULL);
1659     LOAD32H(work[0], ct+0);
1660     LOAD32H(work[1], ct+4);
1661     desfunc(work, skey->des3.dk[0]);
1662     desfunc(work, skey->des3.dk[1]);

```

```

1663     desfunc(work, skey->des3.dk[2]);
1664     STORE32H(work[0],pt+0);
1665     STORE32H(work[1],pt+4);
1666     return CRYPT_OK;
1667 }

```

Here is the call graph for this function:

#### 5.6.3.4 int des3\_ecb\_encrypt (const unsigned char \* *pt*, unsigned char \* *ct*, symmetric\_key \* *skey*)

Encrypts a block of text with 3DES-EDE.

##### Parameters:

- pt* The input plaintext (8 bytes)
- ct* The output ciphertext (8 bytes)
- skey* The key as scheduled

##### Returns:

CRYPT\_OK if successful

Definition at line 1629 of file des.c.

References CRYPT\_OK, desfunc(), and LTC\_ARGCHK.

```

1630 {
1631     ulong32 work[2];
1632
1633     LTC_ARGCHK(pt != NULL);
1634     LTC_ARGCHK(ct != NULL);
1635     LTC_ARGCHK(skey != NULL);
1636     LOAD32H(work[0], pt+0);
1637     LOAD32H(work[1], pt+4);
1638     desfunc(work, skey->des3.ek[0]);
1639     desfunc(work, skey->des3.ek[1]);
1640     desfunc(work, skey->des3.ek[2]);
1641     STORE32H(work[0], ct+0);
1642     STORE32H(work[1], ct+4);
1643     return CRYPT_OK;
1644 }

```

Here is the call graph for this function:

#### 5.6.3.5 int des3\_keysize (int \* *keysize*)

Gets suitable key size.

##### Parameters:

- keysize* [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

##### Returns:

CRYPT\_OK if the input key size is acceptable.

Definition at line 1887 of file des.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```
1888 {
1889     LTC_ARGCHK(keysize != NULL);
1890     if(*keysize < 24) {
1891         return CRYPT_INVALID_KEYSIZE;
1892     }
1893     *keysize = 24;
1894     return CRYPT_OK;
1895 }
```

### 5.6.3.6 int des3\_setup (const unsigned char \* key, int keylen, int num\_rounds, symmetric\_key \* skey)

Initialize the 3DES-EDE block cipher.

#### Parameters:

- key* The symmetric key you wish to pass
- keylen* The key length in bytes
- num\_rounds* The number of rounds desired (0 for default)
- skey* The key in as scheduled by this function.

#### Returns:

CRYPT\_OK if successful

Definition at line 1556 of file des.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, CRYPT\_OK, DE1, deskey(), EN0, and LTC\_ARGCHK.

```
1557 {
1558     LTC_ARGCHK(key != NULL);
1559     LTC_ARGCHK(skey != NULL);
1560
1561     if(num_rounds != 0 && num_rounds != 16) {
1562         return CRYPT_INVALID_ROUNDS;
1563     }
1564
1565     if (keylen != 24) {
1566         return CRYPT_INVALID_KEYSIZE;
1567     }
1568
1569     deskey(key, EN0, skey->des3.ek[0]);
1570     deskey(key+8, DE1, skey->des3.ek[1]);
1571     deskey(key+16, EN0, skey->des3.ek[2]);
1572
1573     deskey(key, DE1, skey->des3.dk[2]);
1574     deskey(key+8, EN0, skey->des3.dk[1]);
1575     deskey(key+16, DE1, skey->des3.dk[0]);
1576
1577     return CRYPT_OK;
1578 }
```

Here is the call graph for this function:

### 5.6.3.7 int des3\_test (void)

Definition at line 1816 of file des.c.

References CRYPT\_NOP, CRYPT\_OK, and des\_test().

```

1817 {
1818     #ifndef LTC_TEST
1819         return CRYPT_NOP;
1820     #else
1821         unsigned char key[24], pt[8], ct[8], tmp[8];
1822         symmetric_key skey;
1823         int x, err;
1824
1825         if ((err = des_test()) != CRYPT_OK) {
1826             return err;
1827         }
1828
1829         for (x = 0; x < 8; x++) {
1830             pt[x] = x;
1831         }
1832
1833         for (x = 0; x < 24; x++) {
1834             key[x] = x;
1835         }
1836
1837         if ((err = des3_setup(key, 24, 0, &skey)) != CRYPT_OK) {
1838             return err;
1839         }
1840
1841         des3_ecb_encrypt(pt, ct, &skey);
1842         des3_ecb_decrypt(ct, tmp, &skey);
1843
1844         if (XMEMCMP(pt, tmp, 8) != 0) {
1845             return CRYPT_FAIL_TESTVECTOR;
1846         }
1847
1848         return CRYPT_OK;
1849     #endif
1850 }

```

Here is the call graph for this function:

#### 5.6.3.8 void des\_done ([symmetric\\_key](#) \* *skey*)

Terminate the context.

##### Parameters:

*skey* The scheduled key

Definition at line 1855 of file des.c.

```

1856 {
1857 }

```

#### 5.6.3.9 int des\_ecb\_decrypt (const unsigned char \* *ct*, unsigned char \* *pt*, [symmetric\\_key](#) \* *skey*)

Decrypts a block of text with DES.

##### Parameters:

*ct* The input ciphertext (8 bytes)

*pt* The output plaintext (8 bytes)

*skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 1608 of file des.c.

References CRYPT\_OK, desfunc(), and LTC\_ARGCHK.

```

1609 {
1610     ulong32 work[2];
1611     LTC_ARGCHK(pt != NULL);
1612     LTC_ARGCHK(ct != NULL);
1613     LTC_ARGCHK(skey != NULL);
1614     LOAD32H(work[0], ct+0);
1615     LOAD32H(work[1], ct+4);
1616     desfunc(work, skey->des.dk);
1617     STORE32H(work[0], pt+0);
1618     STORE32H(work[1], pt+4);
1619     return CRYPT_OK;
1620 }
```

Here is the call graph for this function:

### 5.6.3.10 int des\_ecb\_encrypt (const unsigned char \* *pt*, unsigned char \* *ct*, [symmetric\\_key](#) \* *skey*)

Encrypts a block of text with DES.

**Parameters:**

- pt* The input plaintext (8 bytes)
- ct* The output ciphertext (8 bytes)
- skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 1587 of file des.c.

References CRYPT\_OK, desfunc(), and LTC\_ARGCHK.

```

1588 {
1589     ulong32 work[2];
1590     LTC_ARGCHK(pt != NULL);
1591     LTC_ARGCHK(ct != NULL);
1592     LTC_ARGCHK(skey != NULL);
1593     LOAD32H(work[0], pt+0);
1594     LOAD32H(work[1], pt+4);
1595     desfunc(work, skey->des.ek);
1596     STORE32H(work[0], ct+0);
1597     STORE32H(work[1], ct+4);
1598     return CRYPT_OK;
1599 }
```

Here is the call graph for this function:

### 5.6.3.11 int des\_keysize (int \* *keysize*)

Gets suitable key size.

**Parameters:**

*keysize* [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

**Returns:**

CRYPT\_OK if the input key size is acceptable.

Definition at line 1872 of file des.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

1873 {
1874     LTC_ARGCHK(keysize != NULL);
1875     if(*keysize < 8) {
1876         return CRYPT_INVALID_KEYSIZE;
1877     }
1878     *keysize = 8;
1879     return CRYPT_OK;
1880 }
```

### 5.6.3.12 int des\_setup (const unsigned char \* key, int keylen, int num\_rounds, symmetric\_key \* skey)

Initialize the DES block cipher.

**Parameters:**

*key* The symmetric key you wish to pass  
*keylen* The key length in bytes  
*num\_rounds* The number of rounds desired (0 for default)  
*skey* The key in as scheduled by this function.

**Returns:**

CRYPT\_OK if successful

Definition at line 1529 of file des.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, CRYPT\_OK, DE1, deskey(), EN0, and LTC\_ARGCHK.

Referenced by des\_test().

```

1530 {
1531     LTC_ARGCHK(key != NULL);
1532     LTC_ARGCHK(skey != NULL);
1533
1534     if (num_rounds != 0 && num_rounds != 16) {
1535         return CRYPT_INVALID_ROUNDS;
1536     }
1537
1538     if (keylen != 8) {
1539         return CRYPT_INVALID_KEYSIZE;
1540     }
1541
1542     deskey(key, EN0, skey->des.ek);
1543     deskey(key, DE1, skey->des.dk);
1544
1545     return CRYPT_OK;
1546 }
```

Here is the call graph for this function:

### 5.6.3.13 int des\_test (void)

Performs a self-test of the DES block cipher.

#### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 1673 of file des.c.

References CRYPT\_NOP, CRYPT\_OK, and des\_setup().

Referenced by des3\_test().

```

1674 {
1675     #ifndef LTC_TEST
1676         return CRYPT_NOP;
1677     #else
1678         int err;
1679         static const struct des_test_case {
1680             int num, mode; /* mode 1 = encrypt */
1681             unsigned char key[8], txt[8], out[8];
1682         } cases[] = {
1683             { 1, 1,          { 0x10, 0x31, 0x6E, 0x02, 0x8C, 0x8F, 0x3B, 0x4A },
1684                           { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1685                           { 0x82, 0xDC, 0xBA, 0xFB, 0xDE, 0xAB, 0x66, 0x02 } },
1686             { 2, 1,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1687                           { 0x95, 0xF8, 0xA5, 0xE5, 0xDD, 0x31, 0xD9, 0x00 },
1688                           { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } },
1689             { 3, 1,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1690                           { 0xDD, 0x7F, 0x12, 0x1C, 0xA5, 0x01, 0x56, 0x19 },
1691                           { 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } },
1692             { 4, 1,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1693                           { 0x2E, 0x86, 0x53, 0x10, 0x4F, 0x38, 0x34, 0xEA },
1694                           { 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } },
1695             { 5, 1,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1696                           { 0x4B, 0xD3, 0x88, 0xFF, 0x6C, 0xD8, 0x1D, 0x4F },
1697                           { 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } },
1698             { 6, 1,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1699                           { 0x20, 0xB9, 0xE7, 0x67, 0xB2, 0xFB, 0x14, 0x56 },
1700                           { 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } },
1701             { 7, 1,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1702                           { 0x55, 0x57, 0x93, 0x80, 0xD7, 0x71, 0x38, 0xEF },
1703                           { 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } },
1704             { 8, 1,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1705                           { 0x6C, 0xC5, 0xDE, 0xFA, 0xAF, 0x04, 0x51, 0x2F },
1706                           { 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } },
1707             { 9, 1,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1708                           { 0x0D, 0x9F, 0x27, 0x9B, 0xA5, 0xD8, 0x72, 0x60 },
1709                           { 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } },
1710             {10, 1,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1711                           { 0xD9, 0x03, 0x1B, 0x02, 0x71, 0xBD, 0x5A, 0x0A },
1712                           { 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } },
1713             { 1, 0,          { 0x10, 0x31, 0x6E, 0x02, 0x8C, 0x8F, 0x3B, 0x4A },
1714                           { 0x82, 0xDC, 0xBA, 0xFB, 0xDE, 0xAB, 0x66, 0x02 },
1715                           { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } },
1716             { 2, 0,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1717                           { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1718                           { 0x95, 0xF8, 0xA5, 0xE5, 0xDD, 0x31, 0xD9, 0x00 } },
1719             { 3, 0,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1720                           { 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1721                           { 0xDD, 0x7F, 0x12, 0x1C, 0xA5, 0x01, 0x56, 0x19 } },
1722             { 4, 0,          { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1723                           { 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1724                           { 0x2E, 0x86, 0x53, 0x10, 0x4F, 0x38, 0x34, 0xEA } },

```

```

1726     { 5, 0,      { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1727                  { 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1728                  { 0x4B, 0xD3, 0x88, 0xFF, 0x6C, 0xD8, 0x1D, 0x4F } },
1729     { 6, 0,      { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1730                  { 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1731                  { 0x20, 0xB9, 0xE7, 0x67, 0xB2, 0xFB, 0x14, 0x56 } },
1732     { 7, 0,      { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1733                  { 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1734                  { 0x55, 0x57, 0x93, 0x80, 0xD7, 0x71, 0x38, 0xEF } },
1735     { 8, 0,      { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1736                  { 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1737                  { 0x6C, 0xC5, 0xDE, 0xFA, 0xAF, 0x04, 0x51, 0x2F } },
1738     { 9, 0,      { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1739                  { 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1740                  { 0x0D, 0x9F, 0x27, 0x9B, 0xA5, 0xD8, 0x72, 0x60 } },
1741     {10, 0,      { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 },
1742                  { 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
1743                  { 0xD9, 0x03, 0x1B, 0x02, 0x71, 0xBD, 0x5A, 0x0A } }
1744
1745     /** more test cases you could add if you are not convinced (the above test cases aren't real
1746
1747     key          plaintext          ciphertext
1748     0000000000000000 0000000000000000 8CA64DE9C1B123A7
1749     FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF 7359B2163E4EDC58
1750     3000000000000000 10000000000000001 958E6E627A05557B
1751     1111111111111111 1111111111111111 F40379AB9E0EC533
1752     0123456789ABCDEF 1111111111111111 17668DFC7292532D
1753     1111111111111111 0123456789ABCDEF 8A5AE1F81AB8F2DD
1754     0000000000000000 0000000000000000 8CA64DE9C1B123A7
1755     FEDCBA9876543210 0123456789ABCDEF ED39D950FA74BCC4
1756     7CA110454A1A6E57 01A1D6D039776742 690F5B0D9A26939B
1757     0131D9619DC1376E 5CD54CA83DEF57DA 7A389D10354BD271
1758     07A1133E4A0B2686 0248D43806F67172 868EBB51CAB4599A
1759     3849674C2602319E 51454B582DDF440A 7178876E01F19B2A
1760     04B915BA43FEB5B6 42FD443059577FA2 AF37FB421F8C4095
1761     0113B970FD34F2CE 059B5E0851CF143A 86A560F10EC6D85B
1762     0170F175468FB5E6 0756D8E0774761D2 0CD3DA020021DC09
1763     43297FAD38E373FE 762514B829BF486A EA676B2CB7DB2B7A
1764     07A7137045DA2A16 3BDD119049372802 DFD64A815CAF1A0F
1765     04689104C2FD3B2F 26955F6835AF609A 5C513C9C4886C088
1766     37D06BB516CB7546 164D5E404F275232 0A2AEEAE3FF4AB77
1767     1F08260D1AC2465E 6B056E18759F5CCA EF1BF03E5DFA575A
1768     584023641ABA6176 004BD6EF09176062 88BF0DB6D70DEE56
1769     025816164629B007 480D39006EE762F2 A1F9915541020B56
1770     49793EBC79B3258F 437540C8698F3CFA 6FBF1CAFCCFD0556
1771     4FB05E1515AB73A7 072D43A077075292 2F22E49BAB7CA1AC
1772     49E95D6D4CA229BF 02FE55778117F12A 5A6B612CC26CCE4A
1773     018310DC409B26D6 1D9D5C5018F728C2 5F4C038ED12B2E41
1774     1C587F1C13924FEF 305532286D6F295A 63FAC0D034D9F793
1775     0101010101010101 0123456789ABCDEF 617B3A0CE8F07100
1776     1F1F1F1F0E0E0E0E 0123456789ABCDEF DB958605F8C8C606
1777     E0FEE0FEF1FEF1FE 0123456789ABCDEF EDBFD1C66C29CCC7
1778     0000000000000000 FFFFFFFFFFFFFFFF 355550B2150E2451
1779     FFFFFFFFFFFFFFFF 0000000000000000 CAAAF4DEAF1DBAE
1780     0123456789ABCDEF 0000000000000000 D5D44FF720683D0D
1781     FEDCBA9876543210 FFFFFFFFFFFFFFFF 2A2BB008DF97C2F2
1782
1783     http://www.ecs.soton.ac.uk/~prw99r/ez438/vectors.txt
1784     ***/
1785 };
1786 int i, y;
1787 unsigned char tmp[8];
1788 symmetric_key des;
1789
1790 for(i=0; i < (int)(sizeof(cases)/sizeof(cases[0])); i++)
1791 {
1792     if ((err = des_setup(cases[i].key, 8, 0, &des)) != CRYPT_OK) {

```



```

1793         return err;
1794     }
1795     if (cases[i].mode != 0) {
1796         des_ecb_encrypt(cases[i].txt, tmp, &des);
1797     } else {
1798         des_ecb_decrypt(cases[i].txt, tmp, &des);
1799     }
1800
1801     if (XMEMCMP(cases[i].out, tmp, sizeof(tmp)) != 0) {
1802         return CRYPT_FAIL_TESTVECTOR;
1803     }
1804
1805     /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started
1806     for (y = 0; y < 8; y++) tmp[y] = 0;
1807     for (y = 0; y < 1000; y++) des_ecb_encrypt(tmp, tmp, &des);
1808     for (y = 0; y < 1000; y++) des_ecb_decrypt(tmp, tmp, &des);
1809     for (y = 0; y < 8; y++) if (tmp[y] != 0) return CRYPT_FAIL_TESTVECTOR;
1810 }
1811
1812     return CRYPT_OK;
1813 #endif
1814 }

```

Here is the call graph for this function:

#### 5.6.3.14 static void desfunc (ulong32 \* block, const ulong32 \* keys) [static]

Definition at line 1400 of file des.c.

References byte, des\_ip, ROLc, RORc, SP1, SP2, SP3, SP4, SP5, SP6, SP7, and SP8.

Referenced by des3\_ecb\_decrypt(), des3\_ecb\_encrypt(), des\_ecb\_decrypt(), and des\_ecb\_encrypt().

```

1404 {
1405     ulong32 work, right, leftt;
1406     int cur_round;
1407
1408     leftt = block[0];
1409     right = block[1];
1410
1411     #ifdef LTC_SMALL_CODE
1412     work = ((leftt >> 4) ^ right) & 0x0f0f0f0fL;
1413     right ^= work;
1414     leftt ^= (work << 4);
1415
1416     work = ((leftt >> 16) ^ right) & 0x0000ffffL;
1417     right ^= work;
1418     leftt ^= (work << 16);
1419
1420     work = ((right >> 2) ^ leftt) & 0x33333333L;
1421     leftt ^= work;
1422     right ^= (work << 2);
1423
1424     work = ((right >> 8) ^ leftt) & 0x00ff00ffL;
1425     leftt ^= work;
1426     right ^= (work << 8);
1427
1428     right = ROLc(right, 1);
1429     work = (leftt ^ right) & 0xaaaaaaaaL;
1430
1431     leftt ^= work;
1432     right ^= work;
1433     leftt = ROLc(leftt, 1);
1434 #else
1435     {

```

```

1436     ulong64 tmp;
1437     tmp = des_ip[0][byte(leftt, 0)] ^
1438         des_ip[1][byte(leftt, 1)] ^
1439         des_ip[2][byte(leftt, 2)] ^
1440         des_ip[3][byte(leftt, 3)] ^
1441         des_ip[4][byte(right, 0)] ^
1442         des_ip[5][byte(right, 1)] ^
1443         des_ip[6][byte(right, 2)] ^
1444         des_ip[7][byte(right, 3)];
1445     leftt = (ulong32)(tmp >> 32);
1446     right = (ulong32)(tmp & 0xFFFFFFFFUL);
1447 }
1448 #endif
1449
1450     for (cur_round = 0; cur_round < 8; cur_round++) {
1451         work = RORc(right, 4) ^ *keys++;
1452         leftt ^= SP7[work & 0x3fL]
1453             ^ SP5[(work >> 8) & 0x3fL]
1454             ^ SP3[(work >> 16) & 0x3fL]
1455             ^ SP1[(work >> 24) & 0x3fL];
1456         work = right ^ *keys++;
1457         leftt ^= SP8[work & 0x3fL]
1458             ^ SP6[(work >> 8) & 0x3fL]
1459             ^ SP4[(work >> 16) & 0x3fL]
1460             ^ SP2[(work >> 24) & 0x3fL];
1461
1462         work = RORc(leftt, 4) ^ *keys++;
1463         right ^= SP7[work & 0x3fL]
1464             ^ SP5[(work >> 8) & 0x3fL]
1465             ^ SP3[(work >> 16) & 0x3fL]
1466             ^ SP1[(work >> 24) & 0x3fL];
1467         work = leftt ^ *keys++;
1468         right ^= SP8[work & 0x3fL]
1469             ^ SP6[(work >> 8) & 0x3fL]
1470             ^ SP4[(work >> 16) & 0x3fL]
1471             ^ SP2[(work >> 24) & 0x3fL];
1472     }
1473
1474 #ifdef LTC_SMALL_CODE
1475     right = RORc(right, 1);
1476     work = (leftt ^ right) & 0xaaaaaaaaL;
1477     leftt ^= work;
1478     right ^= work;
1479     leftt = RORc(leftt, 1);
1480     work = ((leftt >> 8) ^ right) & 0x00ff00ffL;
1481     right ^= work;
1482     leftt ^= (work << 8);
1483     /* -- */
1484     work = ((leftt >> 2) ^ right) & 0x33333333L;
1485     right ^= work;
1486     leftt ^= (work << 2);
1487     work = ((right >> 16) ^ leftt) & 0x0000ffffL;
1488     leftt ^= work;
1489     right ^= (work << 16);
1490     work = ((right >> 4) ^ leftt) & 0x0f0f0f0fL;
1491     leftt ^= work;
1492     right ^= (work << 4);
1493 #else
1494     {
1495         ulong64 tmp;
1496         tmp = des_fp[0][byte(leftt, 0)] ^
1497             des_fp[1][byte(leftt, 1)] ^
1498             des_fp[2][byte(leftt, 2)] ^
1499             des_fp[3][byte(leftt, 3)] ^
1500             des_fp[4][byte(right, 0)] ^
1501             des_fp[5][byte(right, 1)] ^
1502             des_fp[6][byte(right, 2)] ^

```

```

1503         des_fp[7][byte(right, 3)];
1504         leftt = (ulong32)(tmp >> 32);
1505         right = (ulong32)(tmp & 0xFFFFFFFFUL);
1506     }
1507 #endif
1508
1509     block[0] = right;
1510     block[1] = leftt;
1511 }

```

### 5.6.3.15 static void deskey (const unsigned char \*key, short edf, [ulong32](#) \*keyout) [static]

Definition at line 1306 of file des.c.

References [bytebit](#), and [pc1](#).

Referenced by [des3\\_setup\(\)](#), and [des\\_setup\(\)](#).

```

1308 {
1309     ulong32 i, j, l, m, n, kn[32];
1310     unsigned char pclm[56], pcr[56];
1311
1312     for (j=0; j < 56; j++) {
1313         l = (ulong32)pc1[j];
1314         m = l & 7;
1315         pclm[j] = (unsigned char)((key[l >> 3U] & bytebit[m]) == bytebit[m] ? 1 : 0);
1316     }
1317
1318     for (i=0; i < 16; i++) {
1319         if (edf == DE1) {
1320             m = (15 - i) << 1;
1321         } else {
1322             m = i << 1;
1323         }
1324         n = m + 1;
1325         kn[m] = kn[n] = 0L;
1326         for (j=0; j < 28; j++) {
1327             l = j + (ulong32)totrot[i];
1328             if (l < 28) {
1329                 pcr[j] = pclm[l];
1330             } else {
1331                 pcr[j] = pclm[l - 28];
1332             }
1333         }
1334         for (/*j = 28*/; j < 56; j++) {
1335             l = j + (ulong32)totrot[i];
1336             if (l < 56) {
1337                 pcr[j] = pclm[l];
1338             } else {
1339                 pcr[j] = pclm[l - 28];
1340             }
1341         }
1342         for (j=0; j < 24; j++) {
1343             if ((int)pcr[(int)pc2[j]] != 0) {
1344                 kn[m] |= bigbyte[j];
1345             }
1346             if ((int)pcr[(int)pc2[j+24]] != 0) {
1347                 kn[n] |= bigbyte[j];
1348             }
1349         }
1350     }
1351
1352     cookey(kn, keyout);
1353 }

```

## 5.6.4 Variable Documentation

### 5.6.4.1 `const ulong32 bigbyte[24]` `[static]`

**Initial value:**

```
{
    0x800000UL, 0x400000UL, 0x200000UL, 0x100000UL,
    0x800000UL, 0x400000UL, 0x200000UL, 0x100000UL,
    0x80000UL, 0x40000UL, 0x20000UL, 0x10000UL,
    0x800UL, 0x400UL, 0x200UL, 0x100UL,
    0x8UL, 0x4UL, 0x2UL, 0x1L
}
```

Definition at line 56 of file des.c.

### 5.6.4.2 `const ulong32 bytebit[8]` `[static]`

**Initial value:**

```
{
    0200, 0100, 040, 020, 010, 04, 02, 01
}
```

Definition at line 51 of file des.c.

Referenced by deskey().

### 5.6.4.3 `const struct ltc_cipher_descriptor des3_desc`

**Initial value:**

```
{
    "3des",
    14,
    24, 24, 8, 16,
    &des3_setup,
    &des3_ecb_encrypt,
    &des3_ecb_decrypt,
    &des3_test,
    &des3_done,
    &des3_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 37 of file des.c.

Referenced by yarrow\_start().

### 5.6.4.4 `const struct ltc_cipher_descriptor des_desc`

**Initial value:**

```
{
    "des",
```

```

    13,
    8, 8, 8, 16,
    &des_setup,
    &des_ecb_encrypt,
    &des_ecb_decrypt,
    &des_test,
    &des_done,
    &des_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}

```

Definition at line 23 of file des.c.

#### 5.6.4.5 const [ulong64 des\\_fp](#)[8][256] [static]

Definition at line 775 of file des.c.

#### 5.6.4.6 const [ulong64 des\\_ip](#)[8][256] [static]

Definition at line 252 of file des.c.

Referenced by desfunc().

#### 5.6.4.7 const unsigned char [pc1](#)[56] [static]

**Initial value:**

```

{
    56, 48, 40, 32, 24, 16, 8, 0, 57, 49, 41, 33, 25, 17,
    9, 1, 58, 50, 42, 34, 26, 18, 10, 2, 59, 51, 43, 35,
    62, 54, 46, 38, 30, 22, 14, 6, 61, 53, 45, 37, 29, 21,
    13, 5, 60, 52, 44, 36, 28, 20, 12, 4, 27, 19, 11, 3
}

```

Definition at line 68 of file des.c.

Referenced by deskey().

#### 5.6.4.8 const unsigned char [pc2](#)[48] [static]

**Initial value:**

```

{
    13, 16, 10, 23, 0, 4, 2, 27, 14, 5, 20, 9,
    22, 18, 11, 3, 25, 7, 15, 6, 26, 19, 12, 1,
    40, 51, 30, 36, 46, 54, 29, 39, 50, 44, 32, 47,
    43, 48, 38, 55, 33, 52, 45, 41, 49, 35, 28, 31
}

```

Definition at line 82 of file des.c.

#### 5.6.4.9 const [ulong32 SP1](#)[64] [static]

**Initial value:**

```
{
    0x01010400UL, 0x00000000UL, 0x00010000UL, 0x01010404UL,
    0x01010004UL, 0x00010404UL, 0x00000004UL, 0x00010000UL,
    0x00000400UL, 0x01010400UL, 0x01010404UL, 0x00000400UL,
    0x01000404UL, 0x01010004UL, 0x01000000UL, 0x00000004UL,
    0x00000404UL, 0x01000400UL, 0x01000400UL, 0x00010400UL,
    0x00010400UL, 0x01010000UL, 0x01010000UL, 0x01000404UL,
    0x00010004UL, 0x01000004UL, 0x01000004UL, 0x00010004UL,
    0x00000000UL, 0x00000404UL, 0x00010404UL, 0x01000000UL,
    0x00010000UL, 0x01010404UL, 0x00000004UL, 0x01010000UL,
    0x01010400UL, 0x01000000UL, 0x01000000UL, 0x00000400UL,
    0x01010004UL, 0x00010000UL, 0x00010400UL, 0x01000004UL,
    0x00000400UL, 0x00000004UL, 0x01000404UL, 0x00010404UL,
    0x01010404UL, 0x00010004UL, 0x01010000UL, 0x01000404UL,
    0x01000004UL, 0x00000404UL, 0x00010404UL, 0x01010400UL,
    0x00000404UL, 0x01000400UL, 0x01000400UL, 0x00000000UL,
    0x00010004UL, 0x00010400UL, 0x00000000UL, 0x01010004UL
}
```

Definition at line 90 of file des.c.

Referenced by desfunc().

#### 5.6.4.10 const [ulong32 SP2](#)[64] [static]

Initial value:

```
{
    0x80108020UL, 0x80008000UL, 0x00008000UL, 0x00108020UL,
    0x00100000UL, 0x00000020UL, 0x80100020UL, 0x80008020UL,
    0x80000020UL, 0x80108020UL, 0x80108000UL, 0x80000000UL,
    0x80008000UL, 0x00100000UL, 0x00000020UL, 0x80100020UL,
    0x00108000UL, 0x00100020UL, 0x80008020UL, 0x00000000UL,
    0x80000000UL, 0x00008000UL, 0x00108020UL, 0x80100000UL,
    0x00100020UL, 0x80000020UL, 0x00000000UL, 0x00108000UL,
    0x00008020UL, 0x80108000UL, 0x80100000UL, 0x00008020UL,
    0x00000000UL, 0x00108020UL, 0x80100020UL, 0x00100000UL,
    0x80008020UL, 0x80100000UL, 0x80108000UL, 0x00008000UL,
    0x80100000UL, 0x80008000UL, 0x00000020UL, 0x80108020UL,
    0x00108020UL, 0x00000020UL, 0x00008000UL, 0x80000000UL,
    0x00008020UL, 0x80108000UL, 0x00100000UL, 0x80000020UL,
    0x00100020UL, 0x80008020UL, 0x80000020UL, 0x00100020UL,
    0x00108000UL, 0x00000000UL, 0x80008000UL, 0x00008020UL,
    0x80000000UL, 0x80100020UL, 0x80108020UL, 0x00108000UL
}
```

Definition at line 110 of file des.c.

Referenced by desfunc().

#### 5.6.4.11 const [ulong32 SP3](#)[64] [static]

Initial value:

```
{
    0x00000208UL, 0x08020200UL, 0x00000000UL, 0x08020008UL,
    0x08000200UL, 0x00000000UL, 0x00020208UL, 0x08000200UL,
    0x00020008UL, 0x08000008UL, 0x08000008UL, 0x00020000UL,
    0x08020208UL, 0x00020008UL, 0x08020000UL, 0x00000208UL,
    0x08000000UL, 0x00000008UL, 0x08020200UL, 0x00000200UL,
    0x00020200UL, 0x08020000UL, 0x08020008UL, 0x00020208UL,
}
```

```

    0x08000208UL, 0x00020200UL, 0x00020000UL, 0x08000208UL,
    0x00000008UL, 0x08020208UL, 0x00000200UL, 0x08000000UL,
    0x08020200UL, 0x08000000UL, 0x00020008UL, 0x00000208UL,
    0x00020000UL, 0x08020200UL, 0x08000200UL, 0x00000000UL,
    0x00000200UL, 0x00020008UL, 0x08020208UL, 0x08000200UL,
    0x08000008UL, 0x00000200UL, 0x00000000UL, 0x08020008UL,
    0x08000208UL, 0x00020000UL, 0x08000000UL, 0x08020208UL,
    0x00000008UL, 0x00020208UL, 0x00020200UL, 0x08000008UL,
    0x08020000UL, 0x08000208UL, 0x00000208UL, 0x08020000UL,
    0x00020208UL, 0x00000008UL, 0x08020008UL, 0x00020200UL
}

```

Definition at line 130 of file des.c.

Referenced by desfunc().

#### 5.6.4.12 const [ulong32](#) SP4[64] [static]

**Initial value:**

```

{
    0x00802001UL, 0x00002081UL, 0x00002081UL, 0x00000080UL,
    0x00802080UL, 0x00800081UL, 0x00800001UL, 0x00002001UL,
    0x00000000UL, 0x00802000UL, 0x00802000UL, 0x00802081UL,
    0x00000081UL, 0x00000000UL, 0x00800080UL, 0x00800001UL,
    0x00000001UL, 0x00002000UL, 0x00800000UL, 0x00802001UL,
    0x00000080UL, 0x00800000UL, 0x00002001UL, 0x00002080UL,
    0x00800081UL, 0x00000001UL, 0x00002080UL, 0x00800080UL,
    0x00002000UL, 0x00802080UL, 0x00802081UL, 0x00000081UL,
    0x00800080UL, 0x00800001UL, 0x00802000UL, 0x00802081UL,
    0x00000081UL, 0x00000000UL, 0x00000000UL, 0x00802000UL,
    0x00002080UL, 0x00800080UL, 0x00800081UL, 0x00000001UL,
    0x00802001UL, 0x00002081UL, 0x00002081UL, 0x00000080UL,
    0x00802081UL, 0x00000081UL, 0x00000001UL, 0x00002000UL,
    0x00800001UL, 0x00002001UL, 0x00802080UL, 0x00800081UL,
    0x00002001UL, 0x00002080UL, 0x00800000UL, 0x00802001UL,
    0x00000080UL, 0x00800000UL, 0x00002000UL, 0x00802080UL
}

```

Definition at line 150 of file des.c.

Referenced by desfunc().

#### 5.6.4.13 const [ulong32](#) SP5[64] [static]

**Initial value:**

```

{
    0x00000100UL, 0x02080100UL, 0x02080000UL, 0x42000100UL,
    0x00080000UL, 0x00000100UL, 0x40000000UL, 0x02080000UL,
    0x40080100UL, 0x00080000UL, 0x02000100UL, 0x40080100UL,
    0x42000100UL, 0x42080000UL, 0x00080100UL, 0x40000000UL,
    0x02000000UL, 0x40080000UL, 0x40080000UL, 0x00000000UL,
    0x40000100UL, 0x42080100UL, 0x42080100UL, 0x02000100UL,
    0x42080000UL, 0x40000100UL, 0x00000000UL, 0x42000000UL,
    0x02080100UL, 0x02000000UL, 0x42000000UL, 0x00080100UL,
    0x00080000UL, 0x42000100UL, 0x00000100UL, 0x02000000UL,
    0x40000000UL, 0x02080000UL, 0x42000100UL, 0x40080100UL,
    0x02000100UL, 0x40000000UL, 0x42080000UL, 0x02080100UL,
    0x40080100UL, 0x00000100UL, 0x02000000UL, 0x42080000UL,
    0x42080100UL, 0x00080100UL, 0x42000000UL, 0x42080100UL,
}

```

```

    0x02080000UL, 0x00000000UL, 0x40080000UL, 0x42000000UL,
    0x00080100UL, 0x02000100UL, 0x40000100UL, 0x00080000UL,
    0x00000000UL, 0x40080000UL, 0x02080100UL, 0x40000100UL
}

```

Definition at line 170 of file des.c.

Referenced by desfunc().

#### 5.6.4.14 const [ulong32](#) SP6[64] [static]

Initial value:

```

{
    0x20000010UL, 0x20400000UL, 0x00004000UL, 0x20404010UL,
    0x20400000UL, 0x00000010UL, 0x20404010UL, 0x00400000UL,
    0x20004000UL, 0x00404010UL, 0x00400000UL, 0x20000010UL,
    0x00400010UL, 0x20004000UL, 0x20000000UL, 0x00004010UL,
    0x00000000UL, 0x00400010UL, 0x20004010UL, 0x00004000UL,
    0x00404000UL, 0x20004010UL, 0x00000010UL, 0x20400010UL,
    0x20400010UL, 0x00000000UL, 0x00404010UL, 0x20404000UL,
    0x00004010UL, 0x00404000UL, 0x20400010UL, 0x00404000UL,
    0x20004000UL, 0x00000010UL, 0x20400010UL, 0x00404000UL,
    0x20404010UL, 0x00400000UL, 0x00004010UL, 0x20000010UL,
    0x00400000UL, 0x20004000UL, 0x20000000UL, 0x00004010UL,
    0x20000010UL, 0x20404010UL, 0x00404000UL, 0x20400000UL,
    0x00404010UL, 0x20404000UL, 0x00000000UL, 0x20400010UL,
    0x00000010UL, 0x00004000UL, 0x20400000UL, 0x00404010UL,
    0x00004000UL, 0x00400010UL, 0x20004010UL, 0x00000000UL,
    0x20404000UL, 0x20000000UL, 0x00400010UL, 0x20004010UL
}

```

Definition at line 190 of file des.c.

Referenced by desfunc().

#### 5.6.4.15 const [ulong32](#) SP7[64] [static]

Initial value:

```

{
    0x00200000UL, 0x04200002UL, 0x04000802UL, 0x00000000UL,
    0x00000800UL, 0x04000802UL, 0x00200802UL, 0x04200800UL,
    0x04200802UL, 0x00200000UL, 0x00000000UL, 0x04000002UL,
    0x00000002UL, 0x04000000UL, 0x04200002UL, 0x00000802UL,
    0x04000800UL, 0x00200802UL, 0x00200002UL, 0x04000800UL,
    0x04200000UL, 0x00000800UL, 0x00000802UL, 0x04200802UL,
    0x00200800UL, 0x00000002UL, 0x04000000UL, 0x00200800UL,
    0x04000802UL, 0x04200002UL, 0x04200002UL, 0x00000002UL,
    0x00200002UL, 0x04000000UL, 0x04000800UL, 0x00200000UL,
    0x04200800UL, 0x00000802UL, 0x00200802UL, 0x04200800UL,
    0x00000802UL, 0x04000002UL, 0x04200802UL, 0x04200000UL,
    0x00200800UL, 0x00000000UL, 0x00000002UL, 0x04200802UL,
    0x00000000UL, 0x00200802UL, 0x04200000UL, 0x00000800UL,
    0x04000002UL, 0x04000800UL, 0x00000800UL, 0x00200002UL
}

```

Definition at line 210 of file des.c.

Referenced by desfunc().



**5.6.4.16** `const ulong32 SP8[64]` `[static]`**Initial value:**

```
{
    0x10001040UL, 0x00001000UL, 0x00040000UL, 0x10041040UL,
    0x10000000UL, 0x10001040UL, 0x00000040UL, 0x10000000UL,
    0x00040040UL, 0x10040000UL, 0x10041040UL, 0x00041000UL,
    0x10041000UL, 0x00041040UL, 0x00001000UL, 0x00000040UL,
    0x10040000UL, 0x10000040UL, 0x10001000UL, 0x00001040UL,
    0x00041000UL, 0x00040040UL, 0x10040040UL, 0x10041000UL,
    0x00001040UL, 0x00000000UL, 0x00000000UL, 0x10040040UL,
    0x10000040UL, 0x10001000UL, 0x00041040UL, 0x00040000UL,
    0x00041040UL, 0x00040000UL, 0x10041000UL, 0x00001000UL,
    0x00000040UL, 0x10040040UL, 0x00001000UL, 0x00041040UL,
    0x10001000UL, 0x00000040UL, 0x10000040UL, 0x10040000UL,
    0x10040040UL, 0x10000000UL, 0x00040000UL, 0x10001040UL,
    0x00000000UL, 0x10041040UL, 0x00040040UL, 0x10000040UL,
    0x10040000UL, 0x10001000UL, 0x10001040UL, 0x00000000UL,
    0x10041040UL, 0x00041000UL, 0x00041000UL, 0x00001040UL,
    0x00001040UL, 0x00040040UL, 0x10000000UL, 0x10041000UL
}
```

Definition at line 230 of file des.c.

Referenced by desfunc().

**5.6.4.17** `const unsigned char totrot[16]` `[static]`**Initial value:**

```
{
    1,  2,  4,  6,
    8, 10, 12, 14,
    15, 17, 19, 21,
    23, 25, 27, 28
}
```

Definition at line 75 of file des.c.

## 5.7 ciphers/kasumi.c File Reference

### 5.7.1 Detailed Description

Implementation of the 3GPP Kasumi block cipher Derived from the 3GPP standard source code.

Definition in file [kasumi.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for kasumi.c:

#### Defines

- #define [ROL16](#)(x, y) (((x)<<(y)) | ((x)>>(16-(y)))) & 0xFFFF

#### Typedefs

- typedef unsigned [u16](#)

#### Functions

- static [u16](#) [FI](#) ([u16](#) in, [u16](#) subkey)
- static [ulong32](#) [FO](#) ([ulong32](#) in, int round\_no, [symmetric\\_key](#) \*key)
- static [ulong32](#) [FL](#) ([ulong32](#) in, int round\_no, [symmetric\\_key](#) \*key)
- int [kasumi\\_ecb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)
- int [kasumi\\_ecb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)
- int [kasumi\\_setup](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)
- void [kasumi\\_done](#) ([symmetric\\_key](#) \*skey)
- int [kasumi\\_keysize](#) (int \*keysize)
- int [kasumi\\_test](#) (void)

#### Variables

- const struct [ltc\\_cipher\\_descriptor](#) [kasumi\\_desc](#)

### 5.7.2 Define Documentation

#### 5.7.2.1 #define [ROL16](#)(x, y) (((x)<<(y)) | ((x)>>(16-(y)))) & 0xFFFF

Definition at line 24 of file kasumi.c.

Referenced by [FL\(\)](#).

### 5.7.3 Typedef Documentation

#### 5.7.3.1 typedef unsigned [u16](#)

Definition at line 22 of file kasumi.c.

## 5.7.4 Function Documentation

### 5.7.4.1 static u16 FI (u16 in, u16 subkey) [static]

Definition at line 39 of file kasumi.c.

References S7.

```

40 {
41     u16 nine, seven;
42     static const u16 S7[128] = {
43         54, 50, 62, 56, 22, 34, 94, 96, 38, 6, 63, 93, 2, 18, 123, 33,
44         55, 113, 39, 114, 21, 67, 65, 12, 47, 73, 46, 27, 25, 111, 124, 81,
45         53, 9, 121, 79, 52, 60, 58, 48, 101, 127, 40, 120, 104, 70, 71, 43,
46         20, 122, 72, 61, 23, 109, 13, 100, 77, 1, 16, 7, 82, 10, 105, 98,
47         117, 116, 76, 11, 89, 106, 0, 125, 118, 99, 86, 69, 30, 57, 126, 87,
48         112, 51, 17, 5, 95, 14, 90, 84, 91, 8, 35, 103, 32, 97, 28, 66,
49         102, 31, 26, 45, 75, 4, 85, 92, 37, 74, 80, 49, 68, 29, 115, 44,
50         64, 107, 108, 24, 110, 83, 36, 78, 42, 19, 15, 41, 88, 119, 59, 3 };
51     static const u16 S9[512] = {
52         167, 239, 161, 379, 391, 334, 9, 338, 38, 226, 48, 358, 452, 385, 90, 397,
53         183, 253, 147, 331, 415, 340, 51, 362, 306, 500, 262, 82, 216, 159, 356, 177,
54         175, 241, 489, 37, 206, 17, 0, 333, 44, 254, 378, 58, 143, 220, 81, 400,
55         95, 3, 315, 245, 54, 235, 218, 405, 472, 264, 172, 494, 371, 290, 399, 76,
56         165, 197, 395, 121, 257, 480, 423, 212, 240, 28, 462, 176, 406, 507, 288, 223,
57         501, 407, 249, 265, 89, 186, 221, 428, 164, 74, 440, 196, 458, 421, 350, 163,
58         232, 158, 134, 354, 13, 250, 491, 142, 191, 69, 193, 425, 152, 227, 366, 135,
59         344, 300, 276, 242, 437, 320, 113, 278, 11, 243, 87, 317, 36, 93, 496, 27,
60         487, 446, 482, 41, 68, 156, 457, 131, 326, 403, 339, 20, 39, 115, 442, 124,
61         475, 384, 508, 53, 112, 170, 479, 151, 126, 169, 73, 268, 279, 321, 168, 364,
62         363, 292, 46, 499, 393, 327, 324, 24, 456, 267, 157, 460, 488, 426, 309, 229,
63         439, 506, 208, 271, 349, 401, 434, 236, 16, 209, 359, 52, 56, 120, 199, 277,
64         465, 416, 252, 287, 246, 6, 83, 305, 420, 345, 153, 502, 65, 61, 244, 282,
65         173, 222, 418, 67, 386, 368, 261, 101, 476, 291, 195, 430, 49, 79, 166, 330,
66         280, 383, 373, 128, 382, 408, 155, 495, 367, 388, 274, 107, 459, 417, 62, 454,
67         132, 225, 203, 316, 234, 14, 301, 91, 503, 286, 424, 211, 347, 307, 140, 374,
68         35, 103, 125, 427, 19, 214, 453, 146, 498, 314, 444, 230, 256, 329, 198, 285,
69         50, 116, 78, 410, 10, 205, 510, 171, 231, 45, 139, 467, 29, 86, 505, 32,
70         72, 26, 342, 150, 313, 490, 431, 238, 411, 325, 149, 473, 40, 119, 174, 355,
71         185, 233, 389, 71, 448, 273, 372, 55, 110, 178, 322, 12, 469, 392, 369, 190,
72         1, 109, 375, 137, 181, 88, 75, 308, 260, 484, 98, 272, 370, 275, 412, 111,
73         336, 318, 4, 504, 492, 259, 304, 77, 337, 435, 21, 357, 303, 332, 483, 18,
74         47, 85, 25, 497, 474, 289, 100, 269, 296, 478, 270, 106, 31, 104, 433, 84,
75         414, 486, 394, 96, 99, 154, 511, 148, 413, 361, 409, 255, 162, 215, 302, 201,
76         266, 351, 343, 144, 441, 365, 108, 298, 251, 34, 182, 509, 138, 210, 335, 133,
77         311, 352, 328, 141, 396, 346, 123, 319, 450, 281, 429, 228, 443, 481, 92, 404,
78         485, 422, 248, 297, 23, 213, 130, 466, 22, 217, 283, 70, 294, 360, 419, 127,
79         312, 377, 7, 468, 194, 2, 117, 295, 463, 258, 224, 447, 247, 187, 80, 398,
80         284, 353, 105, 390, 299, 471, 470, 184, 57, 200, 348, 63, 204, 188, 33, 451,
81         97, 30, 310, 219, 94, 160, 129, 493, 64, 179, 263, 102, 189, 207, 114, 402,
82         438, 477, 387, 122, 192, 42, 381, 5, 145, 118, 180, 449, 293, 323, 136, 380,
83         43, 66, 60, 455, 341, 445, 202, 432, 8, 237, 15, 376, 436, 464, 59, 461 };
84
85     /* The sixteen bit input is split into two unequal halves, *
86     * nine bits and seven bits - as is the subkey */
87
88     nine = (u16) (in >> 7) & 0x1FF;
89     seven = (u16) (in & 0x7F);
90
91     /* Now run the various operations */
92     nine = (u16) (S9[nine] ^ seven);
93     seven = (u16) (S7[seven] ^ (nine & 0x7F));
94     seven ^= (subkey >> 9);
95     nine ^= (subkey & 0x1FF);
96     nine = (u16) (S9[nine] ^ seven);
97     seven = (u16) (S7[seven] ^ (nine & 0x7F));

```

```

98  return (u16)(seven<<9) + nine;
99 }

```

#### 5.7.4.2 static **ulong32** FL (**ulong32** in, int round\_no, **symmetric\_key** \* key) [static]

Definition at line 125 of file kasumi.c.

References ROL16.

Referenced by kasumi\_ecb\_decrypt(), and kasumi\_ecb\_encrypt().

```

126 {
127     u16 l, r, a, b;
128     /* split out the left and right halves */
129     l = (u16)(in>>16);
130     r = (u16)(in)&0xFFFF;
131     /* do the FL() operations */
132     a = (u16)(l & key->kasumi.KLi1[round_no]);
133     r ^= ROL16(a,1);
134     b = (u16)(r | key->kasumi.KLi2[round_no]);
135     l ^= ROL16(b,1);
136     /* put the two halves back together */
137
138     return (((ulong32)l)<<16) + r;
139 }

```

#### 5.7.4.3 static **ulong32** FO (**ulong32** in, int round\_no, **symmetric\_key** \* key) [static]

Definition at line 101 of file kasumi.c.

References FI().

Referenced by kasumi\_ecb\_decrypt(), and kasumi\_ecb\_encrypt().

```

102 {
103     u16 left, right;
104
105     /* Split the input into two 16-bit words */
106     left = (u16)(in>>16);
107     right = (u16) in&0xFFFF;
108
109     /* Now apply the same basic transformation three times */
110     left ^= key->kasumi.KOi1[round_no];
111     left = FI( left, key->kasumi.KIi1[round_no] );
112     left ^= right;
113
114     right ^= key->kasumi.KOi2[round_no];
115     right = FI( right, key->kasumi.KIi2[round_no] );
116     right ^= left;
117
118     left ^= key->kasumi.KOi3[round_no];
119     left = FI( left, key->kasumi.KIi3[round_no] );
120     left ^= right;
121
122     return (((ulong32)right)<<16)+left;
123 }

```

Here is the call graph for this function:

#### 5.7.4.4 void kasumi\_done (symmetric\_key \* skey)

Definition at line 237 of file kasumi.c.

```
238 {  
239 }
```

#### 5.7.4.5 int kasumi\_ecb\_decrypt (const unsigned char \* ct, unsigned char \* pt, symmetric\_key \* skey)

Definition at line 168 of file kasumi.c.

References FL(), FO(), and LTC\_ARGCHK.

```
169 {  
170     ulong32 left, right, temp;  
171     int n;  
172  
173     LTC_ARGCHK(pt != NULL);  
174     LTC_ARGCHK(ct != NULL);  
175     LTC_ARGCHK(skey != NULL);  
176  
177     LOAD32H(left, ct);  
178     LOAD32H(right, ct+4);  
179  
180     for (n = 7; n >= 0; ) {  
181         temp = FO(right, n, skey);  
182         temp = FL(temp, n--, skey);  
183         left ^= temp;  
184         temp = FL(left, n, skey);  
185         temp = FO(temp, n--, skey);  
186         right ^= temp;  
187     }  
188  
189     STORE32H(left, pt);  
190     STORE32H(right, pt+4);  
191  
192     return CRYPT_OK;  
193 }
```

Here is the call graph for this function:

#### 5.7.4.6 int kasumi\_ecb\_encrypt (const unsigned char \* pt, unsigned char \* ct, symmetric\_key \* skey)

Definition at line 141 of file kasumi.c.

References FL(), FO(), and LTC\_ARGCHK.

```
142 {  
143     ulong32 left, right, temp;  
144     int n;  
145  
146     LTC_ARGCHK(pt != NULL);  
147     LTC_ARGCHK(ct != NULL);  
148     LTC_ARGCHK(skey != NULL);  
149  
150     LOAD32H(left, pt);  
151     LOAD32H(right, pt+4);
```

```

152
153     for (n = 0; n <= 7; ) {
154         temp = FL(left, n, skey);
155         temp = FO(temp, n++, skey);
156         right ^= temp;
157         temp = FO(right, n, skey);
158         temp = FL(temp, n++, skey);
159         left ^= temp;
160     }
161
162     STORE32H(left, ct);
163     STORE32H(right, ct+4);
164
165     return CRYPT_OK;
166 }

```

Here is the call graph for this function:

#### 5.7.4.7 int kasumi\_keysize (int \* *keysize*)

Definition at line 241 of file kasumi.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

242 {
243     LTC_ARGCHK(keysize != NULL);
244     if (*keysize >= 16) {
245         *keysize = 16;
246         return CRYPT_OK;
247     } else {
248         return CRYPT_INVALID_KEYSIZE;
249     }
250 }

```

#### 5.7.4.8 int kasumi\_setup (const unsigned char \* *key*, int *keylen*, int *num\_rounds*, [symmetric\\_key](#) \* *skey*)

Definition at line 195 of file kasumi.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, and LTC\_ARGCHK.

Referenced by kasumi\_test().

```

196 {
197     static const u16 C[8] = { 0x0123, 0x4567, 0x89AB, 0xCDEF, 0xFEDC, 0xBA98, 0x7654, 0x3210 };
198     u16 ukey[8], Kprime[8];
199     int n;
200
201     LTC_ARGCHK(key != NULL);
202     LTC_ARGCHK(skey != NULL);
203
204     if (keylen != 16) {
205         return CRYPT_INVALID_KEYSIZE;
206     }
207
208     if (num_rounds != 0 && num_rounds != 8) {
209         return CRYPT_INVALID_ROUNDS;
210     }
211
212     /* Start by ensuring the subkeys are endian correct on a 16-bit basis */
213     for (n = 0; n < 8; n++) {

```

```

214     ukey[n] = (((u16)key[2*n]) << 8) | key[2*n+1];
215 }
216
217 /* Now build the K'[] keys */
218 for (n = 0; n < 8; n++) {
219     Kprime[n] = ukey[n] ^ C[n];
220 }
221
222 /* Finally construct the various sub keys */
223 for(n = 0; n < 8; n++) {
224     skey->kasumi.KLi1[n] = ROL16(ukey[n],1);
225     skey->kasumi.KLi2[n] = Kprime[(n+2)&0x7];
226     skey->kasumi.KOi1[n] = ROL16(ukey[(n+1)&0x7],5);
227     skey->kasumi.KOi2[n] = ROL16(ukey[(n+5)&0x7],8);
228     skey->kasumi.KOi3[n] = ROL16(ukey[(n+6)&0x7],13);
229     skey->kasumi.KIi1[n] = Kprime[(n+4)&0x7];
230     skey->kasumi.KIi2[n] = Kprime[(n+3)&0x7];
231     skey->kasumi.KIi3[n] = Kprime[(n+7)&0x7];
232 }
233
234 return CRYPT_OK;
235 }

```

#### 5.7.4.9 int kasumi\_test (void)

Definition at line 252 of file kasumi.c.

References CRYPT\_NOP, CRYPT\_OK, and kasumi\_setup().

```

253 {
254 #ifndef LTC_TEST
255     return CRYPT_NOP;
256 #else
257     static const struct {
258         unsigned char key[16], pt[8], ct[8];
259     } tests[] = {
260
261 {
262     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
263     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
264     { 0x4B, 0x58, 0xA7, 0x71, 0xAF, 0xC7, 0xE5, 0xE8 }
265 },
266
267 {
268     { 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
269     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
270     { 0x7E, 0xEF, 0x11, 0x3C, 0x95, 0xBB, 0x5A, 0x77 }
271 },
272
273 {
274     { 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
275     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
276     { 0x5F, 0x14, 0x06, 0x86, 0xD7, 0xAD, 0x5A, 0x39 }
277 },
278
279 {
280     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 },
281     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
282     { 0x2E, 0x14, 0x91, 0xCF, 0x70, 0xAA, 0x46, 0x5D }
283 },
284
285 {
286     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00 },
287     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },

```

```

288     { 0xB5, 0x45, 0x86, 0xF4, 0xAB, 0x9A, 0xE5, 0x46 }
289 },
290
291 };
292 unsigned char buf[2][8];
293 symmetric_key key;
294 int err, x;
295
296 for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
297     if ((err = kasumi_setup(tests[x].key, 16, 0, &key)) != CRYPT_OK) {
298         return err;
299     }
300     if ((err = kasumi_ecb_encrypt(tests[x].pt, buf[0], &key)) != CRYPT_OK) {
301         return err;
302     }
303     if ((err = kasumi_ecb_decrypt(tests[x].ct, buf[1], &key)) != CRYPT_OK) {
304         return err;
305     }
306     if (XMEMCMP(tests[x].pt, buf[1], 8) || XMEMCMP(tests[x].ct, buf[0], 8)) {
307         return CRYPT_FAIL_TESTVECTOR;
308     }
309 }
310 return CRYPT_OK;
311 #endif
312 }

```

Here is the call graph for this function:

## 5.7.5 Variable Documentation

### 5.7.5.1 const struct [ltc\\_cipher\\_descriptor](#) kasumi\_desc

**Initial value:**

```

{
    "kasumi",
    21,
    16, 16, 8, 8,
    &kasumi_setup,
    &kasumi_ecb_encrypt,
    &kasumi_ecb_decrypt,
    &kasumi_test,
    &kasumi_done,
    &kasumi_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}

```

Definition at line 26 of file kasumi.c.



## 5.8 ciphers/khazad.c File Reference

### 5.8.1 Detailed Description

Khazad implementation derived from public domain source Authors: Paulo S.L.M.

Barreto and Vincent Rijmen.

Definition in file [khazad.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for khazad.c:

### Defines

- `#define R 8`
- `#define KEYSIZE 128`
- `#define KEYSIZEB (KEYSIZE/8)`
- `#define BLOCKSIZE 64`
- `#define BLOCKSIZEB (BLOCKSIZE/8)`

### Functions

- `int khazad_setup` (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the Khazad block cipher.*
- `static void khazad_crypt` (const unsigned char \*plaintext, unsigned char \*ciphertext, const [ulong64](#) \*roundKey)
- `int khazad_ecb_encrypt` (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with Khazad.*
- `int khazad_ecb_decrypt` (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with Khazad.*
- `int khazad_test` (void)  
*Performs a self-test of the Khazad block cipher.*
- `void khazad_done` ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- `int khazad_keysize` (int \*keysize)  
*Gets suitable key size.*

### Variables

- `const struct ltc\_cipher\_descriptor khazad_desc`
- `static const ulong64 T0 [256]`
- `static const ulong64 T1 [256]`
- `static const ulong64 T2 [256]`

- static const [ulong64](#) T3 [256]
- static const [ulong64](#) T4 [256]
- static const [ulong64](#) T5 [256]
- static const [ulong64](#) T6 [256]
- static const [ulong64](#) T7 [256]
- static const [ulong64](#) c [R+1]

## 5.8.2 Define Documentation

### 5.8.2.1 #define BLOCKSIZE 64

Definition at line 37 of file khazad.c.

### 5.8.2.2 #define BLOCKSIZEB (BLOCKSIZE/8)

Definition at line 38 of file khazad.c.

### 5.8.2.3 #define KEYSIZE 128

Definition at line 35 of file khazad.c.

### 5.8.2.4 #define KEYSIZEB (KEYSIZE/8)

Definition at line 36 of file khazad.c.

### 5.8.2.5 #define R 8

Definition at line 34 of file khazad.c.

Referenced by `anubis_setup()`, `blowfish_ecb_decrypt()`, `blowfish_ecb_encrypt()`, `cast5_ecb_decrypt()`, `cast5_ecb_encrypt()`, and `khazad_setup()`.

## 5.8.3 Function Documentation

### 5.8.3.1 static void khazad\_crypt (const unsigned char \* *plaintext*, unsigned char \* *ciphertext*, const [ulong64](#) \* *roundKey*) [static]

Definition at line 677 of file khazad.c.

References T0, T1, T2, T3, T4, T5, T6, and T7.

Referenced by `khazad_ecb_decrypt()`, and `khazad_ecb_encrypt()`.

```

678                                     {
679     int      r;
680     ulong64 state;
681     /*
682     * map plaintext block to cipher state (mu)
683     * and add initial round key (sigma[K^0]):
684     */
685     state =
686         ((ulong64)plaintext[0] << 56) ^

```

```

687     ((ulong64)plaintext[1] << 48) ^
688     ((ulong64)plaintext[2] << 40) ^
689     ((ulong64)plaintext[3] << 32) ^
690     ((ulong64)plaintext[4] << 24) ^
691     ((ulong64)plaintext[5] << 16) ^
692     ((ulong64)plaintext[6] << 8) ^
693     ((ulong64)plaintext[7]      ) ^
694     roundKey[0];
695
696     /*
697     * R - 1 full rounds:
698     */
699     for (r = 1; r < R; r++) {
700         state =
701             T0[(int)(state >> 56)      ] ^
702             T1[(int)(state >> 48) & 0xff] ^
703             T2[(int)(state >> 40) & 0xff] ^
704             T3[(int)(state >> 32) & 0xff] ^
705             T4[(int)(state >> 24) & 0xff] ^
706             T5[(int)(state >> 16) & 0xff] ^
707             T6[(int)(state >> 8) & 0xff] ^
708             T7[(int)(state      ) & 0xff] ^
709             roundKey[r];
710     }
711
712     /*
713     * last round:
714     */
715     state =
716         (T0[(int)(state >> 56)      ] & CONST64(0xff00000000000000)) ^
717         (T1[(int)(state >> 48) & 0xff] & CONST64(0x00ff000000000000)) ^
718         (T2[(int)(state >> 40) & 0xff] & CONST64(0x0000ff0000000000)) ^
719         (T3[(int)(state >> 32) & 0xff] & CONST64(0x000000ff00000000)) ^
720         (T4[(int)(state >> 24) & 0xff] & CONST64(0x00000000ff000000)) ^
721         (T5[(int)(state >> 16) & 0xff] & CONST64(0x0000000000ff0000)) ^
722         (T6[(int)(state >> 8) & 0xff] & CONST64(0x000000000000ff00)) ^
723         (T7[(int)(state      ) & 0xff] & CONST64(0x00000000000000ff)) ^
724         roundKey[R];
725
726     /*
727     * map cipher state to ciphertext block (mu^{-1}):
728     */
729     ciphertext[0] = (unsigned char)(state >> 56);
730     ciphertext[1] = (unsigned char)(state >> 48);
731     ciphertext[2] = (unsigned char)(state >> 40);
732     ciphertext[3] = (unsigned char)(state >> 32);
733     ciphertext[4] = (unsigned char)(state >> 24);
734     ciphertext[5] = (unsigned char)(state >> 16);
735     ciphertext[6] = (unsigned char)(state >> 8);
736     ciphertext[7] = (unsigned char)(state      );
737 }

```

### 5.8.3.2 void khazad\_done (symmetric\_key \* skey)

Terminate the context.

#### Parameters:

*skey* The scheduled key

Definition at line 831 of file khazad.c.

```

832 {
833 }

```

### 5.8.3.3 `int khazad_ecb_decrypt (const unsigned char * ct, unsigned char * pt, symmetric\_key * skey)`

Decrypts a block of text with Khazad.

#### Parameters:

*ct* The input ciphertext (8 bytes)

*pt* The output plaintext (8 bytes)

*skey* The key as scheduled

#### Returns:

CRYPT\_OK if successful

Definition at line 762 of file khazad.c.

References CRYPT\_OK, khazad\_crypt(), and LTC\_ARGCHK.

Referenced by khazad\_test().

```
763 {  
764     LTC_ARGCHK (pt != NULL);  
765     LTC_ARGCHK (ct != NULL);  
766     LTC_ARGCHK (skey != NULL);  
767     khazad_crypt (ct, pt, skey->khazad.roundKeyDec);  
768     return CRYPT_OK;  
769 }
```

Here is the call graph for this function:

### 5.8.3.4 `int khazad_ecb_encrypt (const unsigned char * pt, unsigned char * ct, symmetric\_key * skey)`

Encrypts a block of text with Khazad.

#### Parameters:

*pt* The input plaintext (8 bytes)

*ct* The output ciphertext (8 bytes)

*skey* The key as scheduled

#### Returns:

CRYPT\_OK if successful

Definition at line 746 of file khazad.c.

References CRYPT\_OK, khazad\_crypt(), and LTC\_ARGCHK.

Referenced by khazad\_test().

```
747 {  
748     LTC_ARGCHK (pt != NULL);  
749     LTC_ARGCHK (ct != NULL);  
750     LTC_ARGCHK (skey != NULL);  
751     khazad_crypt (pt, ct, skey->khazad.roundKeyEnc);  
752     return CRYPT_OK;  
753 }
```

Here is the call graph for this function:

### 5.8.3.5 int khazad\_keysize (int \* *keysize*)

Gets suitable key size.

**Parameters:**

*keysize* [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

**Returns:**

CRYPT\_OK if the input key size is acceptable.

Definition at line 840 of file khazad.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```
841 {
842     LTC_ARGCHK(keysize != NULL);
843     if (*keysize >= 16) {
844         *keysize = 16;
845         return CRYPT_OK;
846     } else {
847         return CRYPT_INVALID_KEYSIZE;
848     }
849 }
```

### 5.8.3.6 int khazad\_setup (const unsigned char \* *key*, int *keylen*, int *num\_rounds*, [symmetric\\_key](#) \* *skey*)

Initialize the Khazad block cipher.

**Parameters:**

*key* The symmetric key you wish to pass

*keylen* The key length in bytes

*num\_rounds* The number of rounds desired (0 for default)

*skey* The key in as scheduled by this function.

**Returns:**

CRYPT\_OK if successful

Definition at line 596 of file khazad.c.

References c, CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, LTC\_ARGCHK, R, S, T0, T1, T2, T3, T4, T5, T6, and T7.

Referenced by khazad\_test().

```
597 {
598     int                r;
599     const_ulong64     *S;
600     ulong64           K2, K1;
601
602     LTC_ARGCHK(key != NULL);
603     LTC_ARGCHK(skey != NULL);
604     if (keylen != 16) {
605         return CRYPT_INVALID_KEYSIZE;
606     }
```

```

607     if (num_rounds != 8 && num_rounds != 0) {
608         return CRYPT_INVALID_ROUNDS;
609     }
610
611     /* use 7th table */
612     S = T7;
613
614     /*
615      * map unsigned char array cipher key to initial key state (mu):
616      */
617     K2 =
618         ((ulong64)key[ 0] << 56) ^
619         ((ulong64)key[ 1] << 48) ^
620         ((ulong64)key[ 2] << 40) ^
621         ((ulong64)key[ 3] << 32) ^
622         ((ulong64)key[ 4] << 24) ^
623         ((ulong64)key[ 5] << 16) ^
624         ((ulong64)key[ 6] <<  8) ^
625         ((ulong64)key[ 7]      );
626     K1 =
627         ((ulong64)key[ 8] << 56) ^
628         ((ulong64)key[ 9] << 48) ^
629         ((ulong64)key[10] << 40) ^
630         ((ulong64)key[11] << 32) ^
631         ((ulong64)key[12] << 24) ^
632         ((ulong64)key[13] << 16) ^
633         ((ulong64)key[14] <<  8) ^
634         ((ulong64)key[15]      );
635
636     /*
637      * compute the round keys:
638      */
639     for (r = 0; r <= R; r++) {
640         /*
641          * K[r] = rho(c[r], K1) ^ K2;
642          */
643         skey->khazad.roundKeyEnc[r] =
644             T0[(int)(K1 >> 56)      ] ^
645             T1[(int)(K1 >> 48) & 0xff] ^
646             T2[(int)(K1 >> 40) & 0xff] ^
647             T3[(int)(K1 >> 32) & 0xff] ^
648             T4[(int)(K1 >> 24) & 0xff] ^
649             T5[(int)(K1 >> 16) & 0xff] ^
650             T6[(int)(K1 >>  8) & 0xff] ^
651             T7[(int)(K1      ) & 0xff] ^
652             c[r] ^ K2;
653         K2 = K1; K1 = skey->khazad.roundKeyEnc[r];
654     }
655     /*
656      * compute the inverse key schedule:
657      *  $K'^0 = K^R$ ,  $K'^R = K^0$ ,  $K'^r = \text{theta}(K^{R-r})$ 
658      */
659     skey->khazad.roundKeyDec[0] = skey->khazad.roundKeyEnc[R];
660     for (r = 1; r < R; r++) {
661         K1 = skey->khazad.roundKeyEnc[R - r];
662         skey->khazad.roundKeyDec[r] =
663             T0[(int)S[(int)(K1 >> 56)      ] & 0xff] ^
664             T1[(int)S[(int)(K1 >> 48) & 0xff] & 0xff] ^
665             T2[(int)S[(int)(K1 >> 40) & 0xff] & 0xff] ^
666             T3[(int)S[(int)(K1 >> 32) & 0xff] & 0xff] ^
667             T4[(int)S[(int)(K1 >> 24) & 0xff] & 0xff] ^
668             T5[(int)S[(int)(K1 >> 16) & 0xff] & 0xff] ^
669             T6[(int)S[(int)(K1 >>  8) & 0xff] & 0xff] ^
670             T7[(int)S[(int)(K1      ) & 0xff] & 0xff];
671     }
672     skey->khazad.roundKeyDec[R] = skey->khazad.roundKeyEnc[0];
673

```

```

674     return CRYPT_OK;
675 }

```

### 5.8.3.7 int khazad\_test (void)

Performs a self-test of the Khazad block cipher.

#### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 775 of file khazad.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, khazad\_ecb\_decrypt(), khazad\_ecb\_encrypt(), khazad\_setup(), and XMEMCMP.

```

776 {
777 #ifndef LTC_TEST
778     return CRYPT_NOP;
779 #else
780     static const struct test {
781         unsigned char pt[8], ct[8], key[16];
782     } tests[] = {
783     {
784         { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
785         { 0x49, 0xA4, 0xCE, 0x32, 0xAC, 0x19, 0x0E, 0x3F },
786         { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
787           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
788     }, {
789         { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
790         { 0x64, 0x5D, 0x77, 0x3E, 0x40, 0xAB, 0xDD, 0x53 },
791         { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
792           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 },
793     }, {
794         { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
795         { 0x9E, 0x39, 0x98, 0x64, 0xF7, 0x8E, 0xCA, 0x02 },
796         { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
797           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
798     }, {
799         { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 },
800         { 0xA9, 0xDF, 0x3D, 0x2C, 0x64, 0xD3, 0xEA, 0x28 },
801         { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
802           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
803     }
804 };
805     int x, y;
806     unsigned char buf[2][8];
807     symmetric_key skey;
808
809     for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
810         khazad_setup(tests[x].key, 16, 0, &skey);
811         khazad_ecb_encrypt(tests[x].pt, buf[0], &skey);
812         khazad_ecb_decrypt(buf[0], buf[1], &skey);
813         if (XMEMCMP(buf[0], tests[x].ct, 8) || XMEMCMP(buf[1], tests[x].pt, 8)) {
814             return CRYPT_FAIL_TESTVECTOR;
815         }
816
817         for (y = 0; y < 1000; y++) khazad_ecb_encrypt(buf[0], buf[0], &skey);
818         for (y = 0; y < 1000; y++) khazad_ecb_decrypt(buf[0], buf[0], &skey);
819         if (XMEMCMP(buf[0], tests[x].ct, 8)) {
820             return CRYPT_FAIL_TESTVECTOR;
821         }
822     }

```

```
823     }
824     return CRYPT_OK;
825 #endif
826 }
```

Here is the call graph for this function:

### 5.8.4 Variable Documentation

#### 5.8.4.1 const **ulong64** c[R+1] [static]

**Initial value:**

```
{
    CONST64(0xba542f7453d3d24d),
    CONST64(0x50ac8dbf70529a4c),
    CONST64(0xead597d133515ba6),
    CONST64(0xde48a899db32b7fc),
    CONST64(0xe39e919be2bb416e),
    CONST64(0xa5cb6b95a1f3b102),
    CONST64(0xccc41d14c363da5d),
    CONST64(0x5fdc7dcd7f5a6c5c),
    CONST64(0xf726ffede89d6f8e),
}
```

Definition at line 576 of file khazad.c.

Referenced by `base64_decode()`, `der_object_identifier_bits()`, `khazad_setup()`, `md4_compress()`, `md5_compress()`, `noekeon_ecb_decrypt()`, `noekeon_ecb_encrypt()`, `ocb_ntz()`, `pmac_ntz()`, `rc6_ecb_decrypt()`, `rc6_ecb_encrypt()`, `safer_ecb_decrypt()`, `safer_ecb_encrypt()`, `sha1_compress()`, `sober128_add_entropy()`, `sober128_read()`, `sober128_start()`, `tiger_compress()`, `twofish_ecb_decrypt()`, and `twofish_ecb_encrypt()`.

#### 5.8.4.2 const struct ltc\_cipher\_descriptor khazad\_desc

**Initial value:**

```
{
    "khazad",
    18,
    16, 16, 8, 8,
    &khazad_setup,
    &khazad_ecb_encrypt,
    &khazad_ecb_decrypt,
    &khazad_test,
    &khazad_done,
    &khazad_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 21 of file khazad.c.

Referenced by `yarrow_start()`.

#### 5.8.4.3 const **ulong64** T0[256] [static]

Definition at line 40 of file khazad.c.



**5.8.4.4** `const ulong64 T1[256]` `[static]`

Definition at line 107 of file khazad.c.

**5.8.4.5** `const ulong64 T2[256]` `[static]`

Definition at line 174 of file khazad.c.

**5.8.4.6** `const ulong64 T3[256]` `[static]`

Definition at line 241 of file khazad.c.

**5.8.4.7** `const ulong64 T4[256]` `[static]`

Definition at line 308 of file khazad.c.

**5.8.4.8** `const ulong64 T5[256]` `[static]`

Definition at line 375 of file khazad.c.

**5.8.4.9** `const ulong64 T6[256]` `[static]`

Definition at line 442 of file khazad.c.

Referenced by `khazad_crypt()`, and `khazad_setup()`.

**5.8.4.10** `const ulong64 T7[256]` `[static]`

Definition at line 509 of file khazad.c.

Referenced by `khazad_crypt()`, and `khazad_setup()`.

## 5.9 ciphers/kseed.c File Reference

### 5.9.1 Detailed Description

seed implementation of SEED derived from RFC4269 Tom St Denis

Definition in file [kseed.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for kseed.c:

### Defines

- #define [G\(x\)](#) ([SS3](#)[((x)>>24)&255] ^ [SS2](#)[((x)>>16)&255] ^ [SS1](#)[((x)>>8)&255] ^ [SS0](#)[(x)&255])
- #define [F](#)(L1, L2, R1, R2, K1, K2)

### Functions

- int [kseed\\_setup](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the SEED block cipher.*
- static void [rounds](#) ([ulong32](#) \*P, [ulong32](#) \*K)
- int [kseed\\_ecb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with SEED.*
- int [kseed\\_ecb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with SEED.*
- void [kseed\\_done](#) ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- int [kseed\\_test](#) (void)  
*Performs a self-test of the SEED block cipher.*
- int [kseed\\_keysize](#) (int \*keysize)  
*Gets suitable key size.*

### Variables

- const struct [ltc\\_cipher\\_descriptor](#) [kseed\\_desc](#)
- static const [ulong32](#) [SS0](#) [256]
- static const [ulong32](#) [SS1](#) [256]
- static const [ulong32](#) [SS2](#) [256]
- static const [ulong32](#) [SS3](#) [256]
- static const [ulong32](#) [KCi](#) [16]

## 5.9.2 Define Documentation

### 5.9.2.1 #define F(L1, L2, R1, R2, K1, K2)

**Value:**

```
T2 = G((R1 ^ K1) ^ (R2 ^ K2)); \
T = G(G(T2 + (R1 ^ K1)) + T2); \
L2 ^= T; \
L1 ^= (T + G(T2 + (R1 ^ K1))); \
```

Definition at line 188 of file kseed.c.

### 5.9.2.2 #define G(x) (SS3[(x)>>24]&255] ^ SS2[(x)>>16]&255] ^ SS1[(x)>>8]&255] ^ SS0[(x)&255])

Definition at line 186 of file kseed.c.

Referenced by ecc\_test(), and kseed\_setup().

## 5.9.3 Function Documentation

### 5.9.3.1 void kseed\_done (symmetric\_key \* skey)

Terminate the context.

**Parameters:**

*skey* The scheduled key

Definition at line 299 of file kseed.c.

```
300 {
301 }
```

### 5.9.3.2 int kseed\_ecb\_decrypt (const unsigned char \* ct, unsigned char \* pt, symmetric\_key \* skey)

Decrypts a block of text with SEED.

**Parameters:**

*ct* The input ciphertext (16 bytes)

*pt* The output plaintext (16 bytes)

*skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 281 of file kseed.c.

References CRYPT\_OK, and rounds().

Referenced by kseed\_test().

```

282 {
283     ulong32 P[4];
284     LOAD32H(P[0], ct);
285     LOAD32H(P[1], ct+4);
286     LOAD32H(P[2], ct+8);
287     LOAD32H(P[3], ct+12);
288     rounds(P, skey->kseed.dK);
289     STORE32H(P[2], pt);
290     STORE32H(P[3], pt+4);
291     STORE32H(P[0], pt+8);
292     STORE32H(P[1], pt+12);
293     return CRYPT_OK;
294 }

```

Here is the call graph for this function:

### 5.9.3.3 int kseed\_ecb\_encrypt (const unsigned char \* *pt*, unsigned char \* *ct*, [symmetric\\_key](#) \* *skey*)

Encrypts a block of text with SEED.

#### Parameters:

- pt* The input plaintext (16 bytes)
- ct* The output ciphertext (16 bytes)
- skey* The key as scheduled

#### Returns:

CRYPT\_OK if successful

Definition at line 259 of file kseed.c.

References CRYPT\_OK, and rounds().

Referenced by kseed\_test().

```

260 {
261     ulong32 P[4];
262     LOAD32H(P[0], pt);
263     LOAD32H(P[1], pt+4);
264     LOAD32H(P[2], pt+8);
265     LOAD32H(P[3], pt+12);
266     rounds(P, skey->kseed.K);
267     STORE32H(P[2], ct);
268     STORE32H(P[3], ct+4);
269     STORE32H(P[0], ct+8);
270     STORE32H(P[1], ct+12);
271     return CRYPT_OK;
272 }

```

Here is the call graph for this function:

### 5.9.3.4 int kseed\_keysize (int \* *keysize*)

Gets suitable key size.

#### Parameters:

- keysize* [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

**Returns:**

CRYPT\_OK if the input key size is acceptable.

Definition at line 361 of file kseed.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

362 {
363     LTC_ARGCHK(keysize != NULL);
364     if (*keysize >= 16) {
365         *keysize = 16;
366     } else {
367         return CRYPT_INVALID_KEYSIZE;
368     }
369     return CRYPT_OK;
370 }
```

### 5.9.3.5 int kseed\_setup (const unsigned char \* *key*, int *keylen*, int *num\_rounds*, symmetric\_key \* *skey*)

Initialize the SEED block cipher.

**Parameters:**

*key* The symmetric key you wish to pass

*keylen* The key length in bytes

*num\_rounds* The number of rounds desired (0 for default)

*skey* The key in as scheduled by this function.

**Returns:**

CRYPT\_OK if successful

Definition at line 202 of file kseed.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, G, and KCi.

Referenced by kseed\_test().

```

203 {
204     int i;
205     ulong32 tmp, k1, k2, k3, k4;
206
207     if (keylen != 16) {
208         return CRYPT_INVALID_KEYSIZE;
209     }
210
211     if (num_rounds != 16 && num_rounds != 0) {
212         return CRYPT_INVALID_ROUNDS;
213     }
214
215     /* load key */
216     LOAD32H(k1, key);
217     LOAD32H(k2, key+4);
218     LOAD32H(k3, key+8);
219     LOAD32H(k4, key+12);
220
221     for (i = 0; i < 16; i++) {
222         skey->kseed.K[2*i+0] = G(k1 + k3 - KCi[i]);
223         skey->kseed.K[2*i+1] = G(k2 - k4 + KCi[i]);

```

```

224     if (i&1) {
225         tmp = k3;
226         k3 = ((k3 << 8) | (k4 >> 24)) & 0xFFFFFFFF;
227         k4 = ((k4 << 8) | (tmp >> 24)) & 0xFFFFFFFF;
228     } else {
229         tmp = k1;
230         k1 = ((k1 >> 8) | (k2 << 24)) & 0xFFFFFFFF;
231         k2 = ((k2 >> 8) | (tmp << 24)) & 0xFFFFFFFF;
232     }
233     /* reverse keys for decrypt */
234     skey->kseed.dK[2*(15-i)+0] = skey->kseed.K[2*i+0];
235     skey->kseed.dK[2*(15-i)+1] = skey->kseed.K[2*i+1];
236 }
237
238 return CRYPT_OK;
239 }

```

### 5.9.3.6 int kseed\_test (void)

Performs a self-test of the SEED block cipher.

#### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 307 of file kseed.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, kseed\_ecb\_decrypt(), kseed\_ecb\_encrypt(), kseed\_setup(), and XMEMCMP.

```

308 {
309 #if !defined(LTC_TEST)
310     return CRYPT_NOP;
311 #else
312     static const struct test {
313         unsigned char pt[16], ct[16], key[16];
314     } tests[] = {
315
316 {
317     { 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F },
318     { 0x5E,0xBA,0xC6,0xE0,0x05,0x4E,0x16,0x68,0x19,0xAF,0xF1,0xCC,0x6D,0x34,0x6C,0xDB },
319     { 0 },
320 },
321
322 {
323     { 0 },
324     { 0xC1,0x1F,0x22,0xF2,0x01,0x40,0x50,0x50,0x84,0x48,0x35,0x97,0xE4,0x37,0x0F,0x43 },
325     { 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F },
326 },
327
328 {
329     { 0x83,0xA2,0xF8,0xA2,0x88,0x64,0x1F,0xB9,0xA4,0xE9,0xA5,0xCC,0x2F,0x13,0x1C,0x7D },
330     { 0xEE,0x54,0xD1,0x3E,0xBC,0xAE,0x70,0x6D,0x22,0x6B,0xC3,0x14,0x2C,0xD4,0x0D,0x4A },
331     { 0x47,0x06,0x48,0x08,0x51,0xE6,0x1B,0xE8,0x5D,0x74,0xBF,0xB3,0xFD,0x95,0x61,0x85 },
332 },
333
334 {
335     { 0xB4,0x1E,0x6B,0xE2,0xEB,0xA8,0x4A,0x14,0x8E,0x2E,0xED,0x84,0x59,0x3C,0x5E,0xC7 },
336     { 0x9B,0x9B,0x7B,0xFC,0xD1,0x81,0x3C,0xB9,0x5D,0x0B,0x36,0x18,0xF4,0x0F,0x51,0x22 },
337     { 0x28,0xDB,0xC3,0xBC,0x49,0xFF,0xD8,0x7D,0xCF,0xA5,0x09,0xB1,0x1D,0x42,0x2B,0xE7 },
338 },
339 };
340     int x;

```

```

341     unsigned char buf[2][16];
342     symmetric_key skey;
343
344     for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
345         kseed_setup(tests[x].key, 16, 0, &skey);
346         kseed_ecb_encrypt(tests[x].pt, buf[0], &skey);
347         kseed_ecb_decrypt(buf[0], buf[1], &skey);
348         if (XMEMCMP(buf[0], tests[x].ct, 16) || XMEMCMP(buf[1], tests[x].pt, 16)) {
349             return CRYPT_FAIL_TESTVECTOR;
350         }
351     }
352     return CRYPT_OK;
353 #endif
354 }

```

Here is the call graph for this function:

### 5.9.3.7 static void rounds (ulong32 \* P, ulong32 \* K) [static]

Definition at line 241 of file kseed.c.

References F, and T2.

Referenced by kseed\_ecb\_decrypt(), kseed\_ecb\_encrypt(), and saferp\_setup().

```

242 {
243     ulong32 T, T2;
244     int i;
245     for (i = 0; i < 16; i += 2) {
246         F(P[0], P[1], P[2], P[3], K[0], K[1]);
247         F(P[2], P[3], P[0], P[1], K[2], K[3]);
248         K += 4;
249     }
250 }

```

## 5.9.4 Variable Documentation

### 5.9.4.1 const ulong32 KCi[16] [static]

**Initial value:**

```

{
0x9E3779B9, 0x3C6EF373,
0x78DDE6E6, 0xF1BBCDCC,
0xE3779B99, 0xC6EF3733,
0x8DDE6E67, 0x1BBCDCCF,
0x3779B99E, 0x6EF3733C,
0xDDE6E678, 0xBBCCDCCF1,
0x779B99E3, 0xEF3733C6,
0xDE6E678D, 0xBCDCCF1B
}

```

Definition at line 175 of file kseed.c.

Referenced by kseed\_setup().

### 5.9.4.2 const struct ltc\_cipher\_descriptor kseed\_desc

**Initial value:**

```
{
    "seed",
    20,
    16, 16, 16, 16,
    &kseed_setup,
    &kseed_ecb_encrypt,
    &kseed_ecb_decrypt,
    &kseed_test,
    &kseed_done,
    &kseed_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 22 of file kseed.c.

Referenced by yarrow\_start().

#### **5.9.4.3**   **const** **ulong32** **SS0**[256]   [static]

Definition at line 35 of file kseed.c.

#### **5.9.4.4**   **const** **ulong32** **SS1**[256]   [static]

Definition at line 70 of file kseed.c.

#### **5.9.4.5**   **const** **ulong32** **SS2**[256]   [static]

Definition at line 105 of file kseed.c.

#### **5.9.4.6**   **const** **ulong32** **SS3**[256]   [static]

Definition at line 140 of file kseed.c.



## 5.10 ciphers/noekeon.c File Reference

### 5.10.1 Detailed Description

Implementation of the Noekeon block cipher by Tom St Denis.

Definition in file [noekeon.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for noekeon.c:

#### Defines

- #define [kTHETA](#)(a, b, [c](#), d)
- #define [THETA](#)(k, a, b, [c](#), d)
- #define [GAMMA](#)(a, b, [c](#), d)
- #define [PI1](#)(a, b, [c](#), d) a = ROLc(a, 1); [c](#) = ROLc([c](#), 5); d = ROLc(d, 2);
- #define [PI2](#)(a, b, [c](#), d) a = RORc(a, 1); [c](#) = RORc([c](#), 5); d = RORc(d, 2);
- #define [ROUND](#)(i)
- #define [ROUND](#)(i)

#### Functions

- int [noekeon\\_setup](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the Noekeon block cipher.*
- int [noekeon\\_ecb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with Noekeon.*
- int [noekeon\\_ecb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with Noekeon.*
- int [noekeon\\_test](#) (void)  
*Performs a self-test of the Noekeon block cipher.*
- void [noekeon\\_done](#) ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- int [noekeon\\_keysize](#) (int \*keysize)  
*Gets suitable key size.*

#### Variables

- const struct [ltc\\_cipher\\_descriptor](#) [noekeon\\_desc](#)
- static const [ulong32](#) [RC](#) []

## 5.10.2 Define Documentation

### 5.10.2.1 #define GAMMA(a, b, c, d)

**Value:**

```
b ^= ~(d|c);          \
a ^= c&b;             \
temp = d; d = a; a = temp; \
c ^= a ^ b ^ d;       \
b ^= ~(d|c);          \
a ^= c&b;
```

Definition at line 53 of file noekeon.c.

### 5.10.2.2 #define kTHETA(a, b, c, d)

**Value:**

```
temp = a^c; temp = temp ^ ROLc(temp, 8) ^ RORc(temp, 8); \
b ^= temp; d ^= temp; \
temp = b^d; temp = temp ^ ROLc(temp, 8) ^ RORc(temp, 8); \
a ^= temp; c ^= temp;
```

Definition at line 41 of file noekeon.c.

Referenced by noekeon\_setup().

### 5.10.2.3 #define PI1(a, b, c, d) a = ROLc(a, 1); c = ROLc(c, 5); d = ROLc(d, 2);

Definition at line 61 of file noekeon.c.

### 5.10.2.4 #define PI2(a, b, c, d) a = RORc(a, 1); c = RORc(c, 5); d = RORc(d, 2);

Definition at line 64 of file noekeon.c.

### 5.10.2.5 #define ROUND(i)

**Value:**

```
THETA(skey->noekeon.dK, a, b, c, d); \
a ^= RC[i]; \
PI1(a, b, c, d); \
GAMMA(a, b, c, d); \
PI2(a, b, c, d);
```

### 5.10.2.6 #define ROUND(i)

**Value:**

```

a ^= RC[i]; \
    THETA(skey->noekeon.K, a,b,c,d); \
    PI1(a,b,c,d); \
    GAMMA(a,b,c,d); \
    PI2(a,b,c,d);

```

Referenced by `noekeon_ecb_decrypt()`, and `noekeon_ecb_encrypt()`.

#### 5.10.2.7 #define THETA(k, a, b, c, d)

##### Value:

```

temp = a^c; temp = temp ^ ROLc(temp, 8) ^ RORc(temp, 8); \
    b ^= temp ^ k[1]; d ^= temp ^ k[3]; \
    temp = b^d; temp = temp ^ ROLc(temp, 8) ^ RORc(temp, 8); \
    a ^= temp ^ k[0]; c ^= temp ^ k[2];

```

Definition at line 47 of file `noekeon.c`.

### 5.10.3 Function Documentation

#### 5.10.3.1 void noekeon\_done (symmetric\_key \* skey)

Terminate the context.

##### Parameters:

*skey* The scheduled key

Definition at line 278 of file `noekeon.c`.

```

279 {
280 }

```

#### 5.10.3.2 int noekeon\_ecb\_decrypt (const unsigned char \* ct, unsigned char \* pt, symmetric\_key \* skey)

Decrypts a block of text with Noekeon.

##### Parameters:

*ct* The input ciphertext (16 bytes)

*pt* The output plaintext (16 bytes)

*skey* The key as scheduled

##### Returns:

CRYPT\_OK if successful

Definition at line 169 of file `noekeon.c`.

References `c`, `LTC_ARGCHK`, and `ROUND`.

```

171 {
172     ulong32 a,b,c,d, temp;
173     int r;
174
175     LTC_ARGCHK(skey != NULL);
176     LTC_ARGCHK(pt != NULL);
177     LTC_ARGCHK(ct != NULL);
178
179     LOAD32H(a,&ct[0]); LOAD32H(b,&ct[4]);
180     LOAD32H(c,&ct[8]); LOAD32H(d,&ct[12]);
181
182
183 #define ROUND(i) \
184     THETA(skey->noekeon.dK, a,b,c,d); \
185     a ^= RC[i]; \
186     P11(a,b,c,d); \
187     GAMMA(a,b,c,d); \
188     P12(a,b,c,d);
189
190     for (r = 16; r > 0; --r) {
191         ROUND(r);
192     }
193
194 #undef ROUND
195
196     THETA(skey->noekeon.dK, a,b,c,d);
197     a ^= RC[0];
198     STORE32H(a,&pt[0]); STORE32H(b, &pt[4]);
199     STORE32H(c,&pt[8]); STORE32H(d, &pt[12]);
200     return CRYPT_OK;
201 }

```

### 5.10.3.3 int noekeon\_ecb\_encrypt (const unsigned char \**pt*, unsigned char \**ct*, [symmetric\\_key](#) \**skey*)

Encrypts a block of text with Noekeon.

#### Parameters:

- pt* The input plaintext (16 bytes)
- ct* The output ciphertext (16 bytes)
- skey* The key as scheduled

#### Returns:

CRYPT\_OK if successful

Definition at line 115 of file noekeon.c.

References `c`, `LTC_ARGCHK`, and `ROUND`.

```

117 {
118     ulong32 a,b,c,d,temp;
119     int r;
120
121     LTC_ARGCHK(skey != NULL);
122     LTC_ARGCHK(pt != NULL);
123     LTC_ARGCHK(ct != NULL);
124
125     LOAD32H(a,&pt[0]); LOAD32H(b,&pt[4]);
126     LOAD32H(c,&pt[8]); LOAD32H(d,&pt[12]);
127

```

```

128 #define ROUND(i) \
129     a ^= RC[i]; \
130     THETA(skey->noekeon.K, a, b, c, d); \
131     PI1(a, b, c, d); \
132     GAMMA(a, b, c, d); \
133     PI2(a, b, c, d);
134
135     for (r = 0; r < 16; ++r) {
136         ROUND(r);
137     }
138
139 #undef ROUND
140
141     a ^= RC[16];
142     THETA(skey->noekeon.K, a, b, c, d);
143
144     STORE32H(a, &ct[0]); STORE32H(b, &ct[4]);
145     STORE32H(c, &ct[8]); STORE32H(d, &ct[12]);
146
147     return CRYPT_OK;
148 }

```

#### 5.10.3.4 int noekeon\_keysize (int \* *keysize*)

Gets suitable key size.

##### Parameters:

***keysize*** [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

##### Returns:

CRYPT\_OK if the input key size is acceptable.

Definition at line 287 of file noekeon.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

288 {
289     LTC_ARGCHK(keysize != NULL);
290     if (*keysize < 16) {
291         return CRYPT_INVALID_KEYSIZE;
292     } else {
293         *keysize = 16;
294         return CRYPT_OK;
295     }
296 }

```

#### 5.10.3.5 int noekeon\_setup (const unsigned char \* *key*, int *keylen*, int *num\_rounds*, [symmetric\\_key](#) \* *skey*)

Initialize the Noekeon block cipher.

##### Parameters:

***key*** The symmetric key you wish to pass

***keylen*** The key length in bytes

***num\_rounds*** The number of rounds desired (0 for default)

*skey* The key in as scheduled by this function.

**Returns:**

CRYPT\_OK if successful

Definition at line 75 of file noekeon.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, CRYPT\_OK, kTHETA, and LTC\_ARGCHK.

Referenced by noekeon\_test().

```

76 {
77     ulong32 temp;
78
79     LTC_ARGCHK(key != NULL);
80     LTC_ARGCHK(skey != NULL);
81
82     if (keylen != 16) {
83         return CRYPT_INVALID_KEYSIZE;
84     }
85
86     if (num_rounds != 16 && num_rounds != 0) {
87         return CRYPT_INVALID_ROUNDS;
88     }
89
90     LOAD32H(skey->noekeon.K[0], &key[0]);
91     LOAD32H(skey->noekeon.K[1], &key[4]);
92     LOAD32H(skey->noekeon.K[2], &key[8]);
93     LOAD32H(skey->noekeon.K[3], &key[12]);
94
95     LOAD32H(skey->noekeon.dK[0], &key[0]);
96     LOAD32H(skey->noekeon.dK[1], &key[4]);
97     LOAD32H(skey->noekeon.dK[2], &key[8]);
98     LOAD32H(skey->noekeon.dK[3], &key[12]);
99
100     kTHETA(skey->noekeon.dK[0], skey->noekeon.dK[1], skey->noekeon.dK[2], skey->noekeon.dK[3]);
101
102     return CRYPT_OK;
103 }
```

### 5.10.3.6 int noekeon\_test (void)

Performs a self-test of the Noekeon block cipher.

**Returns:**

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 216 of file noekeon.c.

References CRYPT\_NOP, CRYPT\_OK, noekeon\_setup(), and zeromem().

```

217 {
218     #ifndef LTC_TEST
219         return CRYPT_NOP;
220     #else
221     static const struct {
222         int keylen;
223         unsigned char key[16], pt[16], ct[16];
224     } tests[] = {
```

```

225     {
226         16,
227         { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
228         { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
229         { 0x18, 0xa6, 0xec, 0xe5, 0x28, 0xaa, 0x79, 0x73,
230           0x28, 0xb2, 0xc0, 0x91, 0xa0, 0x2f, 0x54, 0xc5}
231     }
232 };
233 symmetric_key key;
234 unsigned char tmp[2][16];
235 int err, i, y;
236
237 for (i = 0; i < (int)(sizeof(tests)/sizeof(tests[0])); i++) {
238     zeromem(&key, sizeof(key));
239     if ((err = noekeon_setup(tests[i].key, tests[i].keylen, 0, &key)) != CRYPT_OK) {
240         return err;
241     }
242
243     noekeon_ecb_encrypt(tests[i].pt, tmp[0], &key);
244     noekeon_ecb_decrypt(tmp[0], tmp[1], &key);
245     if (XMEMCMP(tmp[0], tests[i].ct, 16) || XMEMCMP(tmp[1], tests[i].pt, 16)) {
246 #if 0
247         printf("\n\nTest %d failed\n", i);
248         if (XMEMCMP(tmp[0], tests[i].ct, 16)) {
249             printf("CT: ");
250             for (i = 0; i < 16; i++) {
251                 printf("%02x ", tmp[0][i]);
252             }
253             printf("\n");
254         } else {
255             printf("PT: ");
256             for (i = 0; i < 16; i++) {
257                 printf("%02x ", tmp[1][i]);
258             }
259             printf("\n");
260         }
261 #endif
262         return CRYPT_FAIL_TESTVECTOR;
263     }
264
265     /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
266     for (y = 0; y < 16; y++) tmp[0][y] = 0;
267     for (y = 0; y < 1000; y++) noekeon_ecb_encrypt(tmp[0], tmp[0], &key);
268     for (y = 0; y < 1000; y++) noekeon_ecb_decrypt(tmp[0], tmp[0], &key);
269     for (y = 0; y < 16; y++) if (tmp[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
270 }
271 return CRYPT_OK;
272 #endif
273 }

```

Here is the call graph for this function:

## 5.10.4 Variable Documentation

### 5.10.4.1 const struct `ltc_cipher_descriptor` `noekeon_desc`

**Initial value:**

```

{
    "noekeon",
    16,
    16, 16, 16, 16,
    &noekeon_setup,
    &noekeon_ecb_encrypt,

```

```
&noekeon_ecb_decrypt,  
&noekeon_test,  
&noekeon_done,  
&noekeon_keysize,  
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,  
}
```

Definition at line 19 of file noekeon.c.

Referenced by `yarrow_start()`.

#### 5.10.4.2 `const ulong32 RC[]` `[static]`

**Initial value:**

```
{  
    0x00000080UL, 0x0000001bUL, 0x00000036UL, 0x0000006cUL,  
    0x000000d8UL, 0x000000abUL, 0x0000004dUL, 0x0000009aUL,  
    0x0000002fUL, 0x0000005eUL, 0x000000bcUL, 0x00000063UL,  
    0x000000c6UL, 0x00000097UL, 0x00000035UL, 0x0000006aUL,  
    0x000000d4UL  
}
```

Definition at line 33 of file noekeon.c.



## 5.11 ciphers/rc2.c File Reference

### 5.11.1 Detailed Description

Implementation of RC2.

Definition in file [rc2.c](#).

```
#include <tomcrypt.h>
```

Include dependency graph for rc2.c:

### Functions

- int [rc2\\_setup](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the RC2 block cipher.*
- int [rc2\\_ecb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with RC2.*
- int [rc2\\_ecb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with RC2.*
- int [rc2\\_test](#) (void)  
*Performs a self-test of the RC2 block cipher.*
- void [rc2\\_done](#) ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- int [rc2\\_keysize](#) (int \*keysize)  
*Gets suitable key size.*

### Variables

- const struct [ltc\\_cipher\\_descriptor](#) [rc2\\_desc](#)
- static const unsigned char [permute](#) [256]

### 5.11.2 Function Documentation

#### 5.11.2.1 void [rc2\\_done](#) ([symmetric\\_key](#) \* *skey*)

Terminate the context.

#### Parameters:

*skey* The scheduled key

Definition at line 335 of file rc2.c.

```
336 {  
337 }
```

### 5.11.2.2 int rc2\_ecb\_decrypt (const unsigned char \* *ct*, unsigned char \* *pt*, symmetric\_key \* *skey*)

Decrypts a block of text with RC2.

#### Parameters:

*ct* The input ciphertext (8 bytes)

*pt* The output plaintext (8 bytes)

*skey* The key as scheduled

#### Returns:

CRYPT\_OK if successful

Definition at line 213 of file rc2.c.

References LTC\_ARGCHK.

```

217 {
218     unsigned x76, x54, x32, x10;
219     unsigned *xkey;
220     int i;
221
222     LTC_ARGCHK(pt != NULL);
223     LTC_ARGCHK(ct != NULL);
224     LTC_ARGCHK(skey != NULL);
225
226     xkey = skey->rc2.xkey;
227
228     x76 = ((unsigned)ct[7] << 8) + (unsigned)ct[6];
229     x54 = ((unsigned)ct[5] << 8) + (unsigned)ct[4];
230     x32 = ((unsigned)ct[3] << 8) + (unsigned)ct[2];
231     x10 = ((unsigned)ct[1] << 8) + (unsigned)ct[0];
232
233     for (i = 15; i >= 0; i--) {
234         if (i == 4 || i == 10) {
235             x76 = (x76 - xkey[x54 & 63]) & 0xFFFF;
236             x54 = (x54 - xkey[x32 & 63]) & 0xFFFF;
237             x32 = (x32 - xkey[x10 & 63]) & 0xFFFF;
238             x10 = (x10 - xkey[x76 & 63]) & 0xFFFF;
239         }
240
241         x76 = ((x76 << 11) | (x76 >> 5));
242         x76 = (x76 - ((x10 & ~x54) + (x32 & x54) + xkey[4*i+3])) & 0xFFFF;
243
244         x54 = ((x54 << 13) | (x54 >> 3));
245         x54 = (x54 - ((x76 & ~x32) + (x10 & x32) + xkey[4*i+2])) & 0xFFFF;
246
247         x32 = ((x32 << 14) | (x32 >> 2));
248         x32 = (x32 - ((x54 & ~x10) + (x76 & x10) + xkey[4*i+1])) & 0xFFFF;
249
250         x10 = ((x10 << 15) | (x10 >> 1));
251         x10 = (x10 - ((x32 & ~x76) + (x54 & x76) + xkey[4*i+0])) & 0xFFFF;
252     }
253
254     pt[0] = (unsigned char)x10;
255     pt[1] = (unsigned char)(x10 >> 8);
256     pt[2] = (unsigned char)x32;
257     pt[3] = (unsigned char)(x32 >> 8);
258     pt[4] = (unsigned char)x54;
259     pt[5] = (unsigned char)(x54 >> 8);
260     pt[6] = (unsigned char)x76;
261     pt[7] = (unsigned char)(x76 >> 8);
262
263     return CRYPT_OK;
264 }

```

**5.11.2.3 int rc2\_ecb\_encrypt (const unsigned char \* *pt*, unsigned char \* *ct*, symmetric\_key \* *skey*)**

Encrypts a block of text with RC2.

**Parameters:**

- pt* The input plaintext (8 bytes)
- ct* The output ciphertext (8 bytes)
- skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 135 of file rc2.c.

References LTC\_ARGCHK.

```

139 {
140     unsigned *xkey;
141     unsigned x76, x54, x32, x10, i;
142
143     LTC_ARGCHK(pt != NULL);
144     LTC_ARGCHK(ct != NULL);
145     LTC_ARGCHK(skey != NULL);
146
147     xkey = skey->rc2.xkey;
148
149     x76 = ((unsigned)pt[7] << 8) + (unsigned)pt[6];
150     x54 = ((unsigned)pt[5] << 8) + (unsigned)pt[4];
151     x32 = ((unsigned)pt[3] << 8) + (unsigned)pt[2];
152     x10 = ((unsigned)pt[1] << 8) + (unsigned)pt[0];
153
154     for (i = 0; i < 16; i++) {
155         x10 = (x10 + (x32 & ~x76) + (x54 & x76) + xkey[4*i+0]) & 0xFFFF;
156         x10 = ((x10 << 1) | (x10 >> 15));
157
158         x32 = (x32 + (x54 & ~x10) + (x76 & x10) + xkey[4*i+1]) & 0xFFFF;
159         x32 = ((x32 << 2) | (x32 >> 14));
160
161         x54 = (x54 + (x76 & ~x32) + (x10 & x32) + xkey[4*i+2]) & 0xFFFF;
162         x54 = ((x54 << 3) | (x54 >> 13));
163
164         x76 = (x76 + (x10 & ~x54) + (x32 & x54) + xkey[4*i+3]) & 0xFFFF;
165         x76 = ((x76 << 5) | (x76 >> 11));
166
167         if (i == 4 || i == 10) {
168             x10 = (x10 + xkey[x76 & 63]) & 0xFFFF;
169             x32 = (x32 + xkey[x10 & 63]) & 0xFFFF;
170             x54 = (x54 + xkey[x32 & 63]) & 0xFFFF;
171             x76 = (x76 + xkey[x54 & 63]) & 0xFFFF;
172         }
173     }
174
175     ct[0] = (unsigned char)x10;
176     ct[1] = (unsigned char)(x10 >> 8);
177     ct[2] = (unsigned char)x32;
178     ct[3] = (unsigned char)(x32 >> 8);
179     ct[4] = (unsigned char)x54;
180     ct[5] = (unsigned char)(x54 >> 8);
181     ct[6] = (unsigned char)x76;
182     ct[7] = (unsigned char)(x76 >> 8);
183
184     return CRYPT_OK;
185 }
```

#### 5.11.2.4 int rc2\_keysize (int \* *keysize*)

Gets suitable key size.

##### Parameters:

*keysize* [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

##### Returns:

CRYPT\_OK if the input key size is acceptable.

Definition at line 344 of file rc2.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

345 {
346     LTC_ARGCHK(keysize != NULL);
347     if (*keysize < 8) {
348         return CRYPT_INVALID_KEYSIZE;
349     } else if (*keysize > 128) {
350         *keysize = 128;
351     }
352     return CRYPT_OK;
353 }
```

#### 5.11.2.5 int rc2\_setup (const unsigned char \* *key*, int *keylen*, int *num\_rounds*, [symmetric\\_key](#) \* *skey*)

Initialize the RC2 block cipher.

##### Parameters:

*key* The symmetric key you wish to pass

*keylen* The key length in bytes

*num\_rounds* The number of rounds desired (0 for default)

*skey* The key in as scheduled by this function.

##### Returns:

CRYPT\_OK if successful

Definition at line 70 of file rc2.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, and LTC\_ARGCHK.

Referenced by rc2\_test().

```

71 {
72     unsigned *xkey = skey->rc2.xkey;
73     unsigned char tmp[128];
74     unsigned T8, TM;
75     int i, bits;
76
77     LTC_ARGCHK(key != NULL);
78     LTC_ARGCHK(skey != NULL);
79
80     if (keylen < 8 || keylen > 128) {
81         return CRYPT_INVALID_KEYSIZE;
82     }
```

```

82     }
83
84     if (num_rounds != 0 && num_rounds != 16) {
85         return CRYPT_INVALID_ROUNDS;
86     }
87
88     for (i = 0; i < keylen; i++) {
89         tmp[i] = key[i] & 255;
90     }
91
92     /* Phase 1: Expand input key to 128 bytes */
93     if (keylen < 128) {
94         for (i = keylen; i < 128; i++) {
95             tmp[i] = permute[(tmp[i - 1] + tmp[i - keylen]) & 255];
96         }
97     }
98
99     /* Phase 2 - reduce effective key size to "bits" */
100    bits = keylen<<3;
101    T8   = (unsigned)(bits+7)>>3;
102    TM   = (255 >> (unsigned)(7 & -bits));
103    tmp[128 - T8] = permute[tmp[128 - T8] & TM];
104    for (i = 127 - T8; i >= 0; i--) {
105        tmp[i] = permute[tmp[i + 1] ^ tmp[i + T8]];
106    }
107
108    /* Phase 3 - copy to xkey in little-endian order */
109    for (i = 0; i < 64; i++) {
110        xkey[i] = (unsigned)tmp[2*i] + ((unsigned)tmp[2*i+1] << 8);
111    }
112
113    #ifdef LTC_CLEAN_STACK
114        zeromem(tmp, sizeof(tmp));
115    #endif
116
117    return CRYPT_OK;
118 }

```

### 5.11.2.6 int rc2\_test (void)

Performs a self-test of the RC2 block cipher.

#### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 281 of file rc2.c.

References CRYPT\_NOP, CRYPT\_OK, rc2\_setup(), and zeromem().

```

282 {
283     #ifndef LTC_TEST
284         return CRYPT_NOP;
285     #else
286         static const struct {
287             int keylen;
288             unsigned char key[16], pt[8], ct[8];
289         } tests[] = {
290
291             { 8,
292               { 0x30, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
293                 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
294               { 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 },
295               { 0x30, 0x64, 0x9e, 0xdf, 0x9b, 0xe7, 0xd2, 0xc2 }

```

```

296
297 },
298 { 16,
299   { 0x88, 0xbc, 0xa9, 0x0e, 0x90, 0x87, 0x5a, 0x7f,
300     0x0f, 0x79, 0xc3, 0x84, 0x62, 0x7b, 0xaf, 0xb2 },
301   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
302   { 0x22, 0x69, 0x55, 0x2a, 0xb0, 0xf8, 0x5c, 0xa6 }
303 }
304 };
305 int x, y, err;
306 symmetric_key skey;
307 unsigned char tmp[2][8];
308
309 for (x = 0; x < (int)(sizeof(tests) / sizeof(tests[0])); x++) {
310     zeromem(tmp, sizeof(tmp));
311     if ((err = rc2_setup(tests[x].key, tests[x].keylen, 0, &skey)) != CRYPT_OK) {
312         return err;
313     }
314
315     rc2_ecb_encrypt(tests[x].pt, tmp[0], &skey);
316     rc2_ecb_decrypt(tmp[0], tmp[1], &skey);
317
318     if (XMEMCMP(tmp[0], tests[x].ct, 8) != 0 || XMEMCMP(tmp[1], tests[x].pt, 8) != 0) {
319         return CRYPT_FAIL_TESTVECTOR;
320     }
321
322     /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
323     for (y = 0; y < 8; y++) tmp[0][y] = 0;
324     for (y = 0; y < 1000; y++) rc2_ecb_encrypt(tmp[0], tmp[0], &skey);
325     for (y = 0; y < 1000; y++) rc2_ecb_decrypt(tmp[0], tmp[0], &skey);
326     for (y = 0; y < 8; y++) if (tmp[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
327 }
328 return CRYPT_OK;
329 #endif
330 }

```

Here is the call graph for this function:

### 5.11.3 Variable Documentation

#### 5.11.3.1 const unsigned char [permute](#)[256] [static]

Initial value:

```

{
    217,120,249,196, 25,221,181,237, 40,233,253,121, 74,160,216,157,
    198,126, 55,131, 43,118, 83,142, 98, 76,100,136, 68,139,251,162,
    23,154, 89,245,135,179, 79, 19, 97, 69,109,141,  9,129,125, 50,
    189,143, 64,235,134,183,123, 11,240,149, 33, 34, 92,107, 78,130,
    84,214,101,147,206, 96,178, 28,115, 86,192, 20,167,140,241,220,
    18,117,202, 31, 59,190,228,209, 66, 61,212, 48,163, 60,182, 38,
    111,191, 14,218, 70,105,  7, 87, 39,242, 29,155,188,148, 67,  3,
    248, 17,199,246,144,239, 62,231,  6,195,213, 47,200,102, 30,215,
    8,232,234,222,128, 82,238,247,132,170,114,172, 53, 77,106, 42,
    150, 26,210,113, 90, 21, 73,116, 75,159,208, 94,  4, 24,164,236,
    194,224, 65,110, 15, 81,203,204, 36,145,175, 80,161,244,112, 57,
    153,124, 58,133, 35,184,180,122,252,  2, 54, 91, 37, 85,151, 49,
    45, 93,250,152,227,138,146,174,  5,223, 41, 16,103,108,186,201,
    211,  0,230,207,225,158,168, 44, 99, 22,  1, 63, 88,226,137,169,
    13, 56, 52, 27,171, 51,255,176,187, 72, 12, 95,185,177,205, 46,
    197,243,219, 71,229,165,156,119, 10,166, 32,104,254,127,193,173
}

```

Definition at line 43 of file rc2.c.

### 5.11.3.2 const struct [ltc\\_cipher\\_descriptor](#) rc2\_desc

**Initial value:**

```
{
    "rc2",
    12, 8, 128, 8, 16,
    &rc2_setup,
    &rc2_ecb_encrypt,
    &rc2_ecb_decrypt,
    &rc2_test,
    &rc2_done,
    &rc2_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 30 of file rc2.c.

Referenced by [yarrow\\_start\(\)](#).

## 5.12 ciphers/rc5.c File Reference

### 5.12.1 Detailed Description

RC5 code by Tom St Denis.

Definition in file [rc5.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for rc5.c:

### Functions

- int [rc5\\_setup](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the RC5 block cipher.*
- int [rc5\\_ecb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with RC5.*
- int [rc5\\_ecb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with RC5.*
- int [rc5\\_test](#) (void)  
*Performs a self-test of the RC5 block cipher.*
- void [rc5\\_done](#) ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- int [rc5\\_keysize](#) (int \*keysize)  
*Gets suitable key size.*

### Variables

- const struct [ltc\\_cipher\\_descriptor](#) [rc5\\_desc](#)
- static const [ulong32](#) [stab](#) [50]

### 5.12.2 Function Documentation

#### 5.12.2.1 void [rc5\\_done](#) ([symmetric\\_key](#) \* *skey*)

Terminate the context.

#### Parameters:

*skey* The scheduled key

Definition at line 295 of file rc5.c.

```
296 {  
297 }
```



**5.12.2.2 int rc5\_ecb\_decrypt (const unsigned char \* *ct*, unsigned char \* *pt*, symmetric\_key \* *skey*)**

Decrypts a block of text with RC5.

**Parameters:**

- ct* The input ciphertext (8 bytes)
- pt* The output plaintext (8 bytes)
- skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 186 of file rc5.c.

References B, CRYPT\_OK, K, LTC\_ARGCHK, and ROR.

```

188 {
189     ulong32 A, B, *K;
190     int r;
191     LTC_ARGCHK(skey != NULL);
192     LTC_ARGCHK(pt != NULL);
193     LTC_ARGCHK(ct != NULL);
194
195     LOAD32L(A, &ct[0]);
196     LOAD32L(B, &ct[4]);
197     K = skey->rc5.K + (skey->rc5.rounds << 1);
198
199     if ((skey->rc5.rounds & 1) == 0) {
200         K -= 2;
201         for (r = skey->rc5.rounds - 1; r >= 0; r -= 2) {
202             B = ROR(B - K[3], A) ^ A;
203             A = ROR(A - K[2], B) ^ B;
204             B = ROR(B - K[1], A) ^ A;
205             A = ROR(A - K[0], B) ^ B;
206             K -= 4;
207         }
208     } else {
209         for (r = skey->rc5.rounds - 1; r >= 0; r--) {
210             B = ROR(B - K[1], A) ^ A;
211             A = ROR(A - K[0], B) ^ B;
212             K -= 2;
213         }
214     }
215     A -= skey->rc5.K[0];
216     B -= skey->rc5.K[1];
217     STORE32L(A, &pt[0]);
218     STORE32L(B, &pt[4]);
219
220     return CRYPT_OK;
221 }
```

**5.12.2.3 int rc5\_ecb\_encrypt (const unsigned char \* *pt*, unsigned char \* *ct*, symmetric\_key \* *skey*)**

Encrypts a block of text with RC5.

**Parameters:**

- pt* The input plaintext (8 bytes)
- ct* The output ciphertext (8 bytes)

*skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 131 of file rc5.c.

References B, K, LTC\_ARGCHK, and ROL.

```

133 {
134     ulong32 A, B, *K;
135     int r;
136     LTC_ARGCHK(skey != NULL);
137     LTC_ARGCHK(pt != NULL);
138     LTC_ARGCHK(ct != NULL);
139
140     LOAD32L(A, &pt[0]);
141     LOAD32L(B, &pt[4]);
142     A += skey->rc5.K[0];
143     B += skey->rc5.K[1];
144     K = skey->rc5.K + 2;
145
146     if ((skey->rc5.rounds & 1) == 0) {
147         for (r = 0; r < skey->rc5.rounds; r += 2) {
148             A = ROL(A ^ B, B) + K[0];
149             B = ROL(B ^ A, A) + K[1];
150             A = ROL(A ^ B, B) + K[2];
151             B = ROL(B ^ A, A) + K[3];
152             K += 4;
153         }
154     } else {
155         for (r = 0; r < skey->rc5.rounds; r++) {
156             A = ROL(A ^ B, B) + K[0];
157             B = ROL(B ^ A, A) + K[1];
158             K += 2;
159         }
160     }
161     STORE32L(A, &ct[0]);
162     STORE32L(B, &ct[4]);
163
164     return CRYPT_OK;
165 }
```

#### 5.12.2.4 int rc5\_keysize (int \* *keysize*)

Gets suitable key size.

**Parameters:**

*keysize* [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

**Returns:**

CRYPT\_OK if the input key size is acceptable.

Definition at line 304 of file rc5.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

305 {
```

```

306     LTC_ARGCHK(keysize != NULL);
307     if (*keysize < 8) {
308         return CRYPT_INVALID_KEYSIZE;
309     } else if (*keysize > 128) {
310         *keysize = 128;
311     }
312     return CRYPT_OK;
313 }

```

### 5.12.2.5 int rc5\_setup (const unsigned char \*key, int keylen, int num\_rounds, symmetric\_key \*skey)

Initialize the RC5 block cipher.

#### Parameters:

- key* The symmetric key you wish to pass
- keylen* The key length in bytes
- num\_rounds* The number of rounds desired (0 for default)
- skey* The key in as scheduled by this function.

#### Returns:

CRYPT\_OK if successful

Definition at line 56 of file rc5.c.

References B, BSWAP, CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, ltc\_cipher\_descriptor::default\_rounds, LTC\_ARGCHK, rc5\_desc, and S.

Referenced by rc5\_test().

```

58 {
59     ulong32 L[64], *S, A, B, i, j, v, s, t, l;
60
61     LTC_ARGCHK(skey != NULL);
62     LTC_ARGCHK(key != NULL);
63
64     /* test parameters */
65     if (num_rounds == 0) {
66         num_rounds = rc5_desc.default_rounds;
67     }
68
69     if (num_rounds < 12 || num_rounds > 24) {
70         return CRYPT_INVALID_ROUNDS;
71     }
72
73     /* key must be between 64 and 1024 bits */
74     if (keylen < 8 || keylen > 128) {
75         return CRYPT_INVALID_KEYSIZE;
76     }
77
78     skey->rc5.rounds = num_rounds;
79     S = skey->rc5.K;
80
81     /* copy the key into the L array */
82     for (A = i = j = 0; i < (ulong32)keylen; ) {
83         A = (A << 8) | ((ulong32)(key[i++] & 255));
84         if ((i & 3) == 0) {
85             L[j++] = BSWAP(A);
86             A = 0;

```

```

87     }
88 }
89
90 if ((keylen & 3) != 0) {
91     A <<= (ulong32)((8 * (4 - (keylen&3))));
92     L[j++] = BSWAP(A);
93 }
94
95 /* setup the S array */
96 t = (ulong32)(2 * (num_rounds + 1));
97 XMEMCPY(S, stab, t * sizeof(*S));
98
99 /* mix buffer */
100 s = 3 * MAX(t, j);
101 l = j;
102 for (A = B = i = j = v = 0; v < s; v++) {
103     A = S[i] = ROLc(S[i] + A + B, 3);
104     B = L[j] = ROL(L[j] + A + B, (A+B));
105     if (++i == t) { i = 0; }
106     if (++j == l) { j = 0; }
107 }
108 return CRYPT_OK;
109 }

```

#### 5.12.2.6 int rc5\_test(void)

Performs a self-test of the RC5 block cipher.

##### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 236 of file rc5.c.

References CRYPT\_NOP, CRYPT\_OK, and rc5\_setup().

```

237 {
238 #ifndef LTC_TEST
239     return CRYPT_NOP;
240 #else
241     static const struct {
242         unsigned char key[16], pt[8], ct[8];
243     } tests[] = {
244     {
245         { 0x91, 0x5f, 0x46, 0x19, 0xbe, 0x41, 0xb2, 0x51,
246           0x63, 0x55, 0xa5, 0x01, 0x10, 0xa9, 0xce, 0x91 },
247         { 0x21, 0xa5, 0xdb, 0xee, 0x15, 0x4b, 0x8f, 0x6d },
248         { 0xf7, 0xc0, 0x13, 0xac, 0x5b, 0x2b, 0x89, 0x52 }
249     },
250     {
251         { 0x78, 0x33, 0x48, 0xe7, 0x5a, 0xeb, 0x0f, 0x2f,
252           0xd7, 0xb1, 0x69, 0xbb, 0x8d, 0xc1, 0x67, 0x87 },
253         { 0xf7, 0xc0, 0x13, 0xac, 0x5b, 0x2b, 0x89, 0x52 },
254         { 0x2f, 0x42, 0xb3, 0xb7, 0x03, 0x69, 0xfc, 0x92 }
255     },
256     {
257         { 0xdc, 0x49, 0xdb, 0x13, 0x75, 0xa5, 0x58, 0x4f,
258           0x64, 0x85, 0xb4, 0x13, 0xb5, 0xf1, 0x2b, 0xaf },
259         { 0x2f, 0x42, 0xb3, 0xb7, 0x03, 0x69, 0xfc, 0x92 },
260         { 0x65, 0xc1, 0x78, 0xb2, 0x84, 0xd1, 0x97, 0xcc }
261     }
262     };
263     unsigned char tmp[2][8];
264     int x, y, err;

```

```

265     symmetric_key key;
266
267     for (x = 0; x < (int)(sizeof(tests) / sizeof(tests[0])); x++) {
268         /* setup key */
269         if ((err = rc5_setup(tests[x].key, 16, 12, &key)) != CRYPT_OK) {
270             return err;
271         }
272
273         /* encrypt and decrypt */
274         rc5_ecb_encrypt(tests[x].pt, tmp[0], &key);
275         rc5_ecb_decrypt(tmp[0], tmp[1], &key);
276
277         /* compare */
278         if (XMEMCMP(tmp[0], tests[x].ct, 8) != 0 || XMEMCMP(tmp[1], tests[x].pt, 8) != 0) {
279             return CRYPT_FAIL_TESTVECTOR;
280         }
281
282         /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
283         for (y = 0; y < 8; y++) tmp[0][y] = 0;
284         for (y = 0; y < 1000; y++) rc5_ecb_encrypt(tmp[0], tmp[0], &key);
285         for (y = 0; y < 1000; y++) rc5_ecb_decrypt(tmp[0], tmp[0], &key);
286         for (y = 0; y < 8; y++) if (tmp[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
287     }
288     return CRYPT_OK;
289 #endif
290 }

```

Here is the call graph for this function:

### 5.12.3 Variable Documentation

#### 5.12.3.1 `const struct ltc_cipher_descriptor rc5_desc`

**Initial value:**

```

{
    "rc5",
    2,
    8, 128, 8, 12,
    &rc5_setup,
    &rc5_ecb_encrypt,
    &rc5_ecb_decrypt,
    &rc5_test,
    &rc5_done,
    &rc5_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}

```

Definition at line 21 of file rc5.c.

Referenced by rc5\_setup(), and yarrow\_start().

#### 5.12.3.2 `const ulong32 stab[50]` [static]

**Initial value:**

```

{
    0xb7e15163UL, 0x5618cb1cUL, 0xf45044d5UL, 0x9287be8eUL, 0x30bf3847UL, 0xcef6b200UL, 0x6d2e2bb9UL, 0xb65a5
    0xa99d1f2bUL, 0x47d498e4UL, 0xe60c129dUL, 0x84438c56UL, 0x227b060fUL, 0xc0b27fc8UL, 0xee9f981UL, 0xfd2173
    0x9b58ecf3UL, 0x399066acUL, 0xd7c7e065UL, 0x75ff5a1eUL, 0x1436d3d7UL, 0xb26e4d90UL, 0x50a5c749UL, 0xeedd41

```

```
0x8d14babbUL, 0x2b4c3474UL, 0xc983ae2dUL, 0x67bb27e6UL, 0x05f2a19fUL, 0xa42a1b58UL, 0x42619511UL, 0xe0990e
0x7ed08883UL, 0x1d08023cUL, 0xbb3f7bf5UL, 0x5976f5aeUL, 0xf7ae6f67UL, 0x95e5e920UL, 0x341d62d9UL, 0xd254dc
0x708c564bUL, 0x0ec3d004UL, 0xacfb49bdUL, 0x4b32c376UL, 0xe96a3d2fUL, 0x87a1b6e8UL, 0x25d930a1UL, 0xc410aa
0x62482413UL, 0x007f9dccUL
}
```

Definition at line 35 of file rc5.c.

## 5.13 ciphers/rc6.c File Reference

### 5.13.1 Detailed Description

RC6 code by Tom St Denis.

Definition in file [rc6.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for rc6.c:

#### Defines

- `#define RND(a, b, c, d)`
- `#define RND(a, b, c, d)`

#### Functions

- `int rc6_setup` (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the RC6 block cipher.*
- `int rc6_ecb_encrypt` (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with RC6.*
- `int rc6_ecb_decrypt` (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with RC6.*
- `int rc6_test` (void)  
*Performs a self-test of the RC6 block cipher.*
- `void rc6_done` ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- `int rc6_keysize` (int \*keysize)  
*Gets suitable key size.*

#### Variables

- const struct [ltc\\_cipher\\_descriptor](#) [rc6\\_desc](#)
- static const [ulong32](#) [stab](#) [44]

### 5.13.2 Define Documentation

#### 5.13.2.1 `#define RND(a, b, c, d)`

Value:

```

t = (b * (b + b + 1)); t = ROLc(t, 5); \
u = (d * (d + d + 1)); u = ROLc(u, 5); \
c = ROR(c - K[1], t) ^ u; \
a = ROR(a - K[0], u) ^ t; K -= 2;

```

### 5.13.2.2 #define RND(a, b, c, d)

**Value:**

```

t = (b * (b + b + 1)); t = ROLc(t, 5); \
u = (d * (d + d + 1)); u = ROLc(u, 5); \
a = ROL(a^t, u) + K[0]; \
c = ROL(c^u, t) + K[1]; K += 2;

```

Referenced by rc6\_ecb\_decrypt(), and rc6\_ecb\_encrypt().

## 5.13.3 Function Documentation

### 5.13.3.1 void rc6\_done (symmetric\_key \* skey)

Terminate the context.

**Parameters:**

*skey* The scheduled key

Definition at line 322 of file rc6.c.

```

323 {
324 }

```

### 5.13.3.2 int rc6\_ecb\_decrypt (const unsigned char \* ct, unsigned char \* pt, symmetric\_key \* skey)

Decrypts a block of text with RC6.

**Parameters:**

*ct* The input ciphertext (16 bytes)

*pt* The output plaintext (16 bytes)

*skey* The key as scheduled

Definition at line 179 of file rc6.c.

References c, K, LTC\_ARGCHK, and RND.

```

181 {
182     ulong32 a, b, c, d, t, u, *K;
183     int r;
184
185     LTC_ARGCHK(skey != NULL);
186     LTC_ARGCHK(pt != NULL);
187     LTC_ARGCHK(ct != NULL);
188
189     LOAD32L(a, &ct[0]); LOAD32L(b, &ct[4]); LOAD32L(c, &ct[8]); LOAD32L(d, &ct[12]);

```



```

190     a -= skey->rc6.K[42];
191     c -= skey->rc6.K[43];
192
193 #define RND(a,b,c,d) \
194     t = (b * (b + b + 1)); t = ROLc(t, 5); \
195     u = (d * (d + d + 1)); u = ROLc(u, 5); \
196     c = ROR(c - K[1], t) ^ u; \
197     a = ROR(a - K[0], u) ^ t; K -= 2;
198
199     K = skey->rc6.K + 40;
200
201     for (r = 0; r < 20; r += 4) {
202         RND(d,a,b,c);
203         RND(c,d,a,b);
204         RND(b,c,d,a);
205         RND(a,b,c,d);
206     }
207
208 #undef RND
209
210     b -= skey->rc6.K[0];
211     d -= skey->rc6.K[1];
212     STORE32L(a, &pt[0]); STORE32L(b, &pt[4]); STORE32L(c, &pt[8]); STORE32L(d, &pt[12]);
213
214     return CRYPT_OK;
215 }

```

### 5.13.3.3 int rc6\_ecb\_encrypt (const unsigned char \*pt, unsigned char \*ct, symmetric\_key \*skey)

Encrypts a block of text with RC6.

#### Parameters:

- pt** The input plaintext (16 bytes)
- ct** The output ciphertext (16 bytes)
- skey** The key as scheduled

Definition at line 125 of file rc6.c.

References c, K, LTC\_ARGCHK, and RND.

```

127 {
128     ulong32 a,b,c,d,t,u, *K;
129     int r;
130
131     LTC_ARGCHK(skey != NULL);
132     LTC_ARGCHK(pt != NULL);
133     LTC_ARGCHK(ct != NULL);
134     LOAD32L(a, &pt[0]); LOAD32L(b, &pt[4]); LOAD32L(c, &pt[8]); LOAD32L(d, &pt[12]);
135
136     b += skey->rc6.K[0];
137     d += skey->rc6.K[1];
138
139 #define RND(a,b,c,d) \
140     t = (b * (b + b + 1)); t = ROLc(t, 5); \
141     u = (d * (d + d + 1)); u = ROLc(u, 5); \
142     a = ROL(a^t,u) + K[0]; \
143     c = ROL(c^u,t) + K[1]; K += 2;
144
145     K = skey->rc6.K + 2;
146     for (r = 0; r < 20; r += 4) {
147         RND(a,b,c,d);

```

```

148         RND (b, c, d, a);
149         RND (c, d, a, b);
150         RND (d, a, b, c);
151     }
152
153 #undef RND
154
155     a += skey->rc6.K[42];
156     c += skey->rc6.K[43];
157     STORE32L(a, &ct[0]); STORE32L(b, &ct[4]); STORE32L(c, &ct[8]); STORE32L(d, &ct[12]);
158     return CRYPT_OK;
159 }

```

#### 5.13.3.4 int rc6\_keysize (int \* *keysize*)

Gets suitable key size.

##### Parameters:

*keysize* [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

##### Returns:

CRYPT\_OK if the input key size is acceptable.

Definition at line 331 of file rc6.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

332 {
333     LTC_ARGCHK(keysize != NULL);
334     if (*keysize < 8) {
335         return CRYPT_INVALID_KEYSIZE;
336     } else if (*keysize > 128) {
337         *keysize = 128;
338     }
339     return CRYPT_OK;
340 }

```

#### 5.13.3.5 int rc6\_setup (const unsigned char \* *key*, int *keylen*, int *num\_rounds*, symmetric\_key \* *skey*)

Initialize the RC6 block cipher.

##### Parameters:

*key* The symmetric key you wish to pass

*keylen* The key length in bytes

*num\_rounds* The number of rounds desired (0 for default)

*skey* The key in as scheduled by this function.

##### Returns:

CRYPT\_OK if successful

Definition at line 53 of file rc6.c.

References B, BSWAP, CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, LTC\_ARGCHK, and S.

Referenced by rc6\_test().

```

55 {
56     ulong32 L[64], S[50], A, B, i, j, v, s, l;
57
58     LTC_ARGCHK(key != NULL);
59     LTC_ARGCHK(skey != NULL);
60
61     /* test parameters */
62     if (num_rounds != 0 && num_rounds != 20) {
63         return CRYPT_INVALID_ROUNDS;
64     }
65
66     /* key must be between 64 and 1024 bits */
67     if (keylen < 8 || keylen > 128) {
68         return CRYPT_INVALID_KEYSIZE;
69     }
70
71     /* copy the key into the L array */
72     for (A = i = j = 0; i < (ulong32)keylen; ) {
73         A = (A << 8) | ((ulong32)(key[i++] & 255));
74         if (!(i & 3)) {
75             L[j++] = BSWAP(A);
76             A = 0;
77         }
78     }
79
80     /* handle odd sized keys */
81     if (keylen & 3) {
82         A <<= (8 * (4 - (keylen&3)));
83         L[j++] = BSWAP(A);
84     }
85
86     /* setup the S array */
87     XMEMCPY(S, stab, 44 * sizeof(stab[0]));
88
89     /* mix buffer */
90     s = 3 * MAX(44, j);
91     l = j;
92     for (A = B = i = j = v = 0; v < s; v++) {
93         A = S[i] = ROLc(S[i] + A + B, 3);
94         B = L[j] = ROL(L[j] + A + B, (A+B));
95         if (++i == 44) { i = 0; }
96         if (++j == l) { j = 0; }
97     }
98
99     /* copy to key */
100     for (i = 0; i < 44; i++) {
101         skey->rc6.K[i] = S[i];
102     }
103     return CRYPT_OK;
104 }

```

### 5.13.3.6 int rc6\_test(void)

Performs a self-test of the RC6 block cipher.

#### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 230 of file rc6.c.

References CRYPT\_NOP, CRYPT\_OK, and rc6\_setup().

```

231 {
232     #ifndef LTC_TEST
233         return CRYPT_NOP;
234     #else
235         static const struct {
236             int keylen;
237             unsigned char key[32], pt[16], ct[16];
238         } tests[] = {
239             {
240                 16,
241                 { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef,
242                   0x01, 0x12, 0x23, 0x34, 0x45, 0x56, 0x67, 0x78,
243                   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
244                   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
245                 { 0x02, 0x13, 0x24, 0x35, 0x46, 0x57, 0x68, 0x79,
246                   0x8a, 0x9b, 0xac, 0xbd, 0xce, 0xdf, 0xe0, 0xf1 },
247                 { 0x52, 0x4e, 0x19, 0x2f, 0x47, 0x15, 0xc6, 0x23,
248                   0x1f, 0x51, 0xf6, 0x36, 0x7e, 0xa4, 0x3f, 0x18 }
249             },
250             {
251                 24,
252                 { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef,
253                   0x01, 0x12, 0x23, 0x34, 0x45, 0x56, 0x67, 0x78,
254                   0x89, 0x9a, 0xab, 0xbc, 0xcd, 0xde, 0xef, 0xf0,
255                   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
256                 { 0x02, 0x13, 0x24, 0x35, 0x46, 0x57, 0x68, 0x79,
257                   0x8a, 0x9b, 0xac, 0xbd, 0xce, 0xdf, 0xe0, 0xf1 },
258                 { 0x68, 0x83, 0x29, 0xd0, 0x19, 0xe5, 0x05, 0x04,
259                   0x1e, 0x52, 0xe9, 0x2a, 0xf9, 0x52, 0x91, 0xd4 }
260             },
261             {
262                 32,
263                 { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef,
264                   0x01, 0x12, 0x23, 0x34, 0x45, 0x56, 0x67, 0x78,
265                   0x89, 0x9a, 0xab, 0xbc, 0xcd, 0xde, 0xef, 0xf0,
266                   0x10, 0x32, 0x54, 0x76, 0x98, 0xba, 0xdc, 0xfe },
267                 { 0x02, 0x13, 0x24, 0x35, 0x46, 0x57, 0x68, 0x79,
268                   0x8a, 0x9b, 0xac, 0xbd, 0xce, 0xdf, 0xe0, 0xf1 },
269                 { 0xc8, 0x24, 0x18, 0x16, 0xf0, 0xd7, 0xe4, 0x89,
270                   0x20, 0xad, 0x16, 0xa1, 0x67, 0x4e, 0x5d, 0x48 }
271             }
272         };
273         unsigned char tmp[2][16];
274         int x, y, err;
275         symmetric_key key;
276
277         for (x = 0; x < (int)(sizeof(tests) / sizeof(tests[0])); x++) {
278             /* setup key */
279             if ((err = rc6_setup(tests[x].key, tests[x].keylen, 0, &key)) != CRYPT_OK) {
280                 return err;
281             }
282
283             /* encrypt and decrypt */
284             rc6_ecb_encrypt(tests[x].pt, tmp[0], &key);
285             rc6_ecb_decrypt(tmp[0], tmp[1], &key);
286
287             /* compare */
288             if (XMEMCMP(tmp[0], tests[x].ct, 16) || XMEMCMP(tmp[1], tests[x].pt, 16)) {
289 #if 0
290                 printf("\n\nFailed test %d\n", x);
291                 if (XMEMCMP(tmp[0], tests[x].ct, 16)) {
292                     printf("Ciphertext:  ");
293                     for (y = 0; y < 16; y++) printf("%02x ", tmp[0][y]);

```

```

294         printf("\nExpected : ");
295         for (y = 0; y < 16; y++) printf("%02x ", tests[x].ct[y]);
296         printf("\n");
297     }
298     if (XMEMCMP(tmp[1], tests[x].pt, 16)) {
299         printf("Plaintext: ");
300         for (y = 0; y < 16; y++) printf("%02x ", tmp[0][y]);
301         printf("\nExpected : ");
302         for (y = 0; y < 16; y++) printf("%02x ", tests[x].pt[y]);
303         printf("\n");
304     }
305 #endif
306     return CRYPT_FAIL_TESTVECTOR;
307 }
308
309 /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
310 for (y = 0; y < 16; y++) tmp[0][y] = 0;
311 for (y = 0; y < 1000; y++) rc6_ecb_encrypt(tmp[0], tmp[0], &key);
312 for (y = 0; y < 1000; y++) rc6_ecb_decrypt(tmp[0], tmp[0], &key);
313 for (y = 0; y < 16; y++) if (tmp[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
314 }
315 return CRYPT_OK;
316 #endif
317 }

```

Here is the call graph for this function:

## 5.13.4 Variable Documentation

### 5.13.4.1 const struct `ltc_cipher_descriptor` `rc6_desc`

**Initial value:**

```

{
    "rc6",
    3,
    8, 128, 16, 20,
    &rc6_setup,
    &rc6_ecb_encrypt,
    &rc6_ecb_decrypt,
    &rc6_test,
    &rc6_done,
    &rc6_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}

```

Definition at line 20 of file rc6.c.

Referenced by `yarrow_start()`.

### 5.13.4.2 const `ulong32` `stab`[44] [static]

**Initial value:**

```

{
    0xb7e15163UL, 0x5618cb1cUL, 0xf45044d5UL, 0x9287be8eUL, 0x30bf3847UL, 0xcef6b200UL, 0x6d2e2bb9UL, 0x0b65a5
    0xa99d1f2bUL, 0x47d498e4UL, 0xe60c129dUL, 0x84438c56UL, 0x227b060fUL, 0xc0b27fc8UL, 0x5ee9f981UL, 0xfd2173
    0x9b58ecf3UL, 0x399066acUL, 0xd7c7e065UL, 0x75ff5a1eUL, 0x1436d3d7UL, 0xb26e4d90UL, 0x50a5c749UL, 0xeedd41
    0x8d14babbUL, 0x2b4c3474UL, 0xc983ae2dUL, 0x67bb27e6UL, 0x05f2a19fUL, 0xa42a1b58UL, 0x42619511UL, 0xe0990e
    0x7ed08883UL, 0x1d08023cUL, 0xbb3f7bf5UL, 0x5976f5aeUL, 0xf7ae6f67UL, 0x95e5e920UL, 0x341d62d9UL, 0xd254dc
    0x708c564bUL, 0x0ec3d004UL, 0xacfb49bdUL, 0x4b32c376UL }

```

Definition at line 34 of file rc6.c.

## 5.14 ciphers/safer/safer.c File Reference

```
#include <tomcrypt.h>
```

Include dependency graph for safer.c:

### Defines

- #define [ROL8](#)(x, n)
- #define [EXP](#)(x) [safer\\_ebox](#)[(x) & 0xFF]
- #define [LOG](#)(x) [safer\\_lbox](#)[(x) & 0xFF]
- #define [PHT](#)(x, y) { y += x; x += y; }
- #define [IPHT](#)(x, y) { x -= y; y -= x; }

### Functions

- static void [Safer\\_Expand\\_Userkey](#) (const unsigned char \*userkey\_1, const unsigned char \*userkey\_2, unsigned int nof\_rounds, int strengthened, safer\_key\_t key)
- int [safer\\_k64\\_setup](#) (const unsigned char \*key, int keylen, int numrounds, [symmetric\\_key](#) \*skey)
- int [safer\\_sk64\\_setup](#) (const unsigned char \*key, int keylen, int numrounds, [symmetric\\_key](#) \*skey)
- int [safer\\_k128\\_setup](#) (const unsigned char \*key, int keylen, int numrounds, [symmetric\\_key](#) \*skey)
- int [safer\\_sk128\\_setup](#) (const unsigned char \*key, int keylen, int numrounds, [symmetric\\_key](#) \*skey)
- int [safer\\_ecb\\_encrypt](#) (const unsigned char \*block\_in, unsigned char \*block\_out, [symmetric\\_key](#) \*skey)
- int [safer\\_ecb\\_decrypt](#) (const unsigned char \*block\_in, unsigned char \*block\_out, [symmetric\\_key](#) \*skey)
- int [safer\\_64\\_keysize](#) (int \*keysize)
- int [safer\\_128\\_keysize](#) (int \*keysize)
- int [safer\\_k64\\_test](#) (void)
- int [safer\\_sk64\\_test](#) (void)
- void [safer\\_done](#) ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- int [safer\\_sk128\\_test](#) (void)

### Variables

- const struct [ltc\\_cipher\\_descriptor](#) [safer\\_k64\\_desc](#)
- const struct [ltc\\_cipher\\_descriptor](#) [safer\\_sk64\\_desc](#)
- const struct [ltc\\_cipher\\_descriptor](#) [safer\\_k128\\_desc](#)
- const struct [ltc\\_cipher\\_descriptor](#) [safer\\_sk128\\_desc](#)
- const unsigned char [safer\\_ebox](#) []
- const unsigned char [safer\\_lbox](#) []

#### 5.14.1 Define Documentation

##### 5.14.1.1 #define [EXP](#)(x) [safer\\_ebox](#)[(x) & 0xFF]

Definition at line 92 of file safer.c.

Referenced by [safer\\_ecb\\_decrypt](#)(), and [safer\\_ecb\\_encrypt](#)().

**5.14.1.2 #define IPHT(x, y) { x -= y; y -= x; }**

Definition at line 95 of file safer.c.

Referenced by safer\_ecb\_decrypt().

**5.14.1.3 #define LOG(x) safer\_lbox[(x) & 0xFF]**

Definition at line 93 of file safer.c.

Referenced by safer\_ecb\_decrypt(), and safer\_ecb\_encrypt().

**5.14.1.4 #define PHT(x, y) { y += x; x += y; }**

Definition at line 94 of file safer.c.

Referenced by safer\_ecb\_encrypt().

**5.14.1.5 #define ROL8(x, n)**

**Value:**

```
((unsigned char)((unsigned int)(x) << (n)\
| (unsigned int)((x) & 0xFF) >> (8 - (n))))
```

Definition at line 90 of file safer.c.

Referenced by Safer\_Expand\_Userkey().

**5.14.2 Function Documentation****5.14.2.1 int safer\_128\_keysize (int \* *keysize*)**

Definition at line 368 of file safer.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```
369 {
370     LTC_ARGCHK(keysize != NULL);
371     if (*keysize < 16) {
372         return CRYPT_INVALID_KEYSIZE;
373     } else {
374         *keysize = 16;
375         return CRYPT_OK;
376     }
377 }
```

**5.14.2.2 int safer\_64\_keysize (int \* *keysize*)**

Definition at line 357 of file safer.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.



```

358 {
359     LTC_ARGCHK(keysize != NULL);
360     if (*keysize < 8) {
361         return CRYPT_INVALID_KEYSIZE;
362     } else {
363         *keysize = 8;
364         return CRYPT_OK;
365     }
366 }

```

#### 5.14.2.3 void safer\_done (symmetric\_key \*skey)

Terminate the context.

##### Parameters:

*skey* The scheduled key

Definition at line 446 of file safer.c.

```

447 {
448 }

```

#### 5.14.2.4 int safer\_ecb\_decrypt (const unsigned char \*block\_in, unsigned char \*block\_out, symmetric\_key \*skey)

Definition at line 307 of file safer.c.

References c, CRYPT\_OK, EXP, IPHT, LOG, and LTC\_ARGCHK.

Referenced by safer\_k64\_test(), safer\_sk128\_test(), and safer\_sk64\_test().

```

311 {    unsigned char a, b, c, d, e, f, g, h, t;
312        unsigned int round;
313        unsigned char *key;
314
315        LTC_ARGCHK(block_in != NULL);
316        LTC_ARGCHK(block_out != NULL);
317        LTC_ARGCHK(skey != NULL);
318
319        key = skey->safer.key;
320        a = block_in[0]; b = block_in[1]; c = block_in[2]; d = block_in[3];
321        e = block_in[4]; f = block_in[5]; g = block_in[6]; h = block_in[7];
322        if (SAFER_MAX_NOF_ROUNDS < (round = *key)) round = SAFER_MAX_NOF_ROUNDS;
323        key += SAFER_BLOCK_LEN * (1 + 2 * round);
324        h ^= *key; g -= *--key; f -= *--key; e ^= *--key;
325        d ^= *--key; c -= *--key; b -= *--key; a ^= *--key;
326        while (round--)
327        {
328            t = e; e = b; b = c; c = t; t = f; f = d; d = g; g = t;
329            IPHT(a, e); IPHT(b, f); IPHT(c, g); IPHT(d, h);
330            IPHT(a, c); IPHT(e, g); IPHT(b, d); IPHT(f, h);
331            IPHT(a, b); IPHT(c, d); IPHT(e, f); IPHT(g, h);
332            h -= *--key; g ^= *--key; f ^= *--key; e -= *--key;
333            d -= *--key; c ^= *--key; b ^= *--key; a -= *--key;
334            h = LOG(h) ^ *--key; g = EXP(g) - *--key;
335            f = EXP(f) - *--key; e = LOG(e) ^ *--key;
336            d = LOG(d) ^ *--key; c = EXP(c) - *--key;
337            b = EXP(b) - *--key; a = LOG(a) ^ *--key;
338        }

```

```

339     block_out[0] = a & 0xFF; block_out[1] = b & 0xFF;
340     block_out[2] = c & 0xFF; block_out[3] = d & 0xFF;
341     block_out[4] = e & 0xFF; block_out[5] = f & 0xFF;
342     block_out[6] = g & 0xFF; block_out[7] = h & 0xFF;
343     return CRYPT_OK;
344 }

```

#### 5.14.2.5 `int safer_ecb_encrypt (const unsigned char * block_in, unsigned char * block_out, symmetric_key * skey)`

Definition at line 253 of file safer.c.

References `c`, `CRYPT_OK`, `EXP`, `LOG`, `LTC_ARGCHK`, and `PHT`.

Referenced by `safer_k64_test()`, `safer_sk128_test()`, and `safer_sk64_test()`.

```

257 {     unsigned char a, b, c, d, e, f, g, h, t;
258     unsigned int round;
259     unsigned char *key;
260
261     LTC_ARGCHK(block_in != NULL);
262     LTC_ARGCHK(block_out != NULL);
263     LTC_ARGCHK(skey != NULL);
264
265     key = skey->safer.key;
266     a = block_in[0]; b = block_in[1]; c = block_in[2]; d = block_in[3];
267     e = block_in[4]; f = block_in[5]; g = block_in[6]; h = block_in[7];
268     if (SAFER_MAX_NOF_ROUNDS < (round = *key)) round = SAFER_MAX_NOF_ROUNDS;
269     while(round-- > 0)
270     {
271         a ^= ++key; b += ++key; c += ++key; d ^= ++key;
272         e ^= ++key; f += ++key; g += ++key; h ^= ++key;
273         a = EXP(a) + ++key; b = LOG(b) ^ ++key;
274         c = LOG(c) ^ ++key; d = EXP(d) + ++key;
275         e = EXP(e) + ++key; f = LOG(f) ^ ++key;
276         g = LOG(g) ^ ++key; h = EXP(h) + ++key;
277         PHT(a, b); PHT(c, d); PHT(e, f); PHT(g, h);
278         PHT(a, c); PHT(e, g); PHT(b, d); PHT(f, h);
279         PHT(a, e); PHT(b, f); PHT(c, g); PHT(d, h);
280         t = b; b = e; e = c; c = t; t = d; d = f; f = g; g = t;
281     }
282     a ^= ++key; b += ++key; c += ++key; d ^= ++key;
283     e ^= ++key; f += ++key; g += ++key; h ^= ++key;
284     block_out[0] = a & 0xFF; block_out[1] = b & 0xFF;
285     block_out[2] = c & 0xFF; block_out[3] = d & 0xFF;
286     block_out[4] = e & 0xFF; block_out[5] = f & 0xFF;
287     block_out[6] = g & 0xFF; block_out[7] = h & 0xFF;
288     return CRYPT_OK;
289 }

```

#### 5.14.2.6 `static void Safer_Expand_Userkey (const unsigned char * userkey_1, const unsigned char * userkey_2, unsigned int nof_rounds, int strengthened, safer_key_t key)` `[static]`

Definition at line 107 of file safer.c.

References `ROL8`.

Referenced by `safer_k128_setup()`, `safer_k64_setup()`, `safer_sk128_setup()`, and `safer_sk64_setup()`.

```

113 {     unsigned int i, j, k;
114     unsigned char ka[SAFER_BLOCK_LEN + 1];

```

```

115     unsigned char kb[SAFER_BLOCK_LEN + 1];
116
117     if (SAFER_MAX_NOF_ROUNDS < nof_rounds)
118         nof_rounds = SAFER_MAX_NOF_ROUNDS;
119     *key++ = (unsigned char)nof_rounds;
120     ka[SAFER_BLOCK_LEN] = (unsigned char)0;
121     kb[SAFER_BLOCK_LEN] = (unsigned char)0;
122     k = 0;
123     for (j = 0; j < SAFER_BLOCK_LEN; j++) {
124         ka[j] = ROL8(userkey_1[j], 5);
125         ka[SAFER_BLOCK_LEN] ^= ka[j];
126         kb[j] = *key++ = userkey_2[j];
127         kb[SAFER_BLOCK_LEN] ^= kb[j];
128     }
129     for (i = 1; i <= nof_rounds; i++) {
130         for (j = 0; j < SAFER_BLOCK_LEN + 1; j++) {
131             ka[j] = ROL8(ka[j], 6);
132             kb[j] = ROL8(kb[j], 6);
133         }
134         if (strengthened) {
135             k = 2 * i - 1;
136             while (k >= (SAFER_BLOCK_LEN + 1)) { k -= SAFER_BLOCK_LEN + 1; }
137         }
138         for (j = 0; j < SAFER_BLOCK_LEN; j++) {
139             if (strengthened) {
140                 *key++ = (ka[k]
141                     + safer_ebox[(int)safer_ebox[(int)((18 * i + j + 1)&0xFF])]) & 0xFF;
142                 if (++k == (SAFER_BLOCK_LEN + 1)) { k = 0; }
143             } else {
144                 *key++ = (ka[j] + safer_ebox[(int)safer_ebox[(int)((18 * i + j + 1)&0xFF])]) & 0xFF;
145             }
146         }
147         if (strengthened) {
148             k = 2 * i;
149             while (k >= (SAFER_BLOCK_LEN + 1)) { k -= SAFER_BLOCK_LEN + 1; }
150         }
151         for (j = 0; j < SAFER_BLOCK_LEN; j++) {
152             if (strengthened) {
153                 *key++ = (kb[k]
154                     + safer_ebox[(int)safer_ebox[(int)((18 * i + j + 10)&0xFF])]) & 0xFF;
155                 if (++k == (SAFER_BLOCK_LEN + 1)) { k = 0; }
156             } else {
157                 *key++ = (kb[j] + safer_ebox[(int)safer_ebox[(int)((18 * i + j + 10)&0xFF])]) & 0xFF;
158             }
159         }
160     }
161
162 #ifdef LTC_CLEAN_STACK
163     zeromem(ka, sizeof(ka));
164     zeromem(kb, sizeof(kb));
165 #endif
166 }

```

#### 5.14.2.7 int safer\_k128\_setup (const unsigned char \* key, int keylen, int numrounds, symmetric\_key \* skey)

Definition at line 214 of file safer.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, CRYPT\_OK, LTC\_ARGCHK, and Safer\_Expand\_Userkey().

```

215 {
216     LTC_ARGCHK(key != NULL);
217     LTC_ARGCHK(skey != NULL);

```

```

218
219     if (numrounds != 0 && (numrounds < 6 || numrounds > SAFER_MAX_NOF_ROUNDS)) {
220         return CRYPT_INVALID_ROUNDS;
221     }
222
223     if (keylen != 16) {
224         return CRYPT_INVALID_KEYSIZE;
225     }
226
227     Safer_Expand_Userkey(key, key+8, (unsigned int)(numrounds != 0 ? numrounds : SAFER_K128_DEFAULT_NOF_ROUNDS));
228     return CRYPT_OK;
229 }

```

Here is the call graph for this function:

#### 5.14.2.8 int safer\_k64\_setup (const unsigned char \* key, int keylen, int numrounds, [symmetric\\_key](#) \* skey)

Definition at line 180 of file safer.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, CRYPT\_OK, LTC\_ARGCHK, and Safer\_Expand\_Userkey().

Referenced by safer\_k64\_test().

```

181 {
182     LTC_ARGCHK(key != NULL);
183     LTC_ARGCHK(skey != NULL);
184
185     if (numrounds != 0 && (numrounds < 6 || numrounds > SAFER_MAX_NOF_ROUNDS)) {
186         return CRYPT_INVALID_ROUNDS;
187     }
188
189     if (keylen != 8) {
190         return CRYPT_INVALID_KEYSIZE;
191     }
192
193     Safer_Expand_Userkey(key, key, (unsigned int)(numrounds != 0 ? numrounds : SAFER_K64_DEFAULT_NOF_ROUNDS));
194     return CRYPT_OK;
195 }

```

Here is the call graph for this function:

#### 5.14.2.9 int safer\_k64\_test (void)

Definition at line 379 of file safer.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, CRYPT\_OK, safer\_ecb\_decrypt(), safer\_ecb\_encrypt(), safer\_k64\_setup(), and XMEMCMP.

```

380 {
381     #ifndef LTC_TEST
382         return CRYPT_NOP;
383     #else
384         static const unsigned char k64_pt[] = { 1, 2, 3, 4, 5, 6, 7, 8 },
385                                         k64_key[] = { 8, 7, 6, 5, 4, 3, 2, 1 },
386                                         k64_ct[] = { 200, 242, 156, 221, 135, 120, 62, 217 };
387
388         symmetric_key skey;
389         unsigned char buf[2][8];
390         int err;

```

```

391
392     /* test K64 */
393     if ((err = safer_k64_setup(k64_key, 8, 6, &skey)) != CRYPT_OK) {
394         return err;
395     }
396     safer_ecb_encrypt(k64_pt, buf[0], &skey);
397     safer_ecb_decrypt(buf[0], buf[1], &skey);
398
399     if (XMEMCMP(buf[0], k64_ct, 8) != 0 || XMEMCMP(buf[1], k64_pt, 8) != 0) {
400         return CRYPT_FAIL_TESTVECTOR;
401     }
402
403     return CRYPT_OK;
404 #endif
405 }

```

Here is the call graph for this function:

#### 5.14.2.10 int safer\_sk128\_setup (const unsigned char \* *key*, int *keylen*, int *numrounds*, symmetric\_key \* *skey*)

Definition at line 231 of file safer.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, CRYPT\_OK, LTC\_ARGCHK, and Safer\_Expand\_Userkey().

Referenced by safer\_sk128\_test().

```

232 {
233     LTC_ARGCHK(key != NULL);
234     LTC_ARGCHK(skey != NULL);
235
236     if (numrounds != 0 && (numrounds < 6 || numrounds > SAFER_MAX_NOF_ROUNDS)) {
237         return CRYPT_INVALID_ROUNDS;
238     }
239
240     if (keylen != 16) {
241         return CRYPT_INVALID_KEYSIZE;
242     }
243
244     Safer_Expand_Userkey(key, key+8, (unsigned int)(numrounds != 0?numrounds:SAFER_SK128_DEFAULT_NOF_ROUNDS));
245     return CRYPT_OK;
246 }

```

Here is the call graph for this function:

#### 5.14.2.11 int safer\_sk128\_test (void)

Definition at line 450 of file safer.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, CRYPT\_OK, safer\_ecb\_decrypt(), safer\_ecb\_encrypt(), safer\_sk128\_setup(), and XMEMCMP.

```

451 {
452     #ifndef LTC_TEST
453         return CRYPT_NOP;
454     #else
455         static const unsigned char sk128_pt[] = { 1, 2, 3, 4, 5, 6, 7, 8 },
456                                         sk128_key[] = { 1, 2, 3, 4, 5, 6, 7, 8,
457                                                         0, 0, 0, 0, 0, 0, 0, 0 },
458                                         sk128_ct[] = { 255, 120, 17, 228, 179, 167, 46, 113 };

```

```

459
460     symmetric_key skey;
461     unsigned char buf[2][8];
462     int err, y;
463
464     /* test SK128 */
465     if ((err = safer_sk128_setup(sk128_key, 16, 0, &skey)) != CRYPT_OK) {
466         return err;
467     }
468     safer_ecb_encrypt(sk128_pt, buf[0], &skey);
469     safer_ecb_decrypt(buf[0], buf[1], &skey);
470
471     if (XMEMCMP(buf[0], sk128_ct, 8) != 0 || XMEMCMP(buf[1], sk128_pt, 8) != 0) {
472         return CRYPT_FAIL_TESTVECTOR;
473     }
474
475     /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
476     for (y = 0; y < 8; y++) buf[0][y] = 0;
477     for (y = 0; y < 1000; y++) safer_ecb_encrypt(buf[0], buf[0], &skey);
478     for (y = 0; y < 1000; y++) safer_ecb_decrypt(buf[0], buf[0], &skey);
479     for (y = 0; y < 8; y++) if (buf[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
480     return CRYPT_OK;
481 #endif
482 }

```

Here is the call graph for this function:

#### 5.14.2.12 `int safer_sk64_setup (const unsigned char * key, int keylen, int numrounds, symmetric_key * skey)`

Definition at line 197 of file safer.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, CRYPT\_OK, LTC\_ARGCHK, and Safer\_Expand\_Userkey().

Referenced by safer\_sk64\_test().

```

198 {
199     LTC_ARGCHK(key != NULL);
200     LTC_ARGCHK(skey != NULL);
201
202     if (numrounds != 0 && (numrounds < 6 || numrounds > SAFER_MAX_NOF_ROUNDS)) {
203         return CRYPT_INVALID_ROUNDS;
204     }
205
206     if (keylen != 8) {
207         return CRYPT_INVALID_KEYSIZE;
208     }
209
210     Safer_Expand_Userkey(key, key, (unsigned int)(numrounds != 0 ? numrounds : SAFER_SK64_DEFAULT_NOF_ROUNDS));
211     return CRYPT_OK;
212 }

```

Here is the call graph for this function:

#### 5.14.2.13 `int safer_sk64_test (void)`

Definition at line 408 of file safer.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, CRYPT\_OK, safer\_ecb\_decrypt(), safer\_ecb\_encrypt(), safer\_sk64\_setup(), and XMEMCMP.

```

409 {
410     #ifndef LTC_TEST
411         return CRYPT_NOP;
412     #else
413         static const unsigned char sk64_pt[] = { 1, 2, 3, 4, 5, 6, 7, 8 },
414                                           sk64_key[] = { 1, 2, 3, 4, 5, 6, 7, 8 },
415                                           sk64_ct[] = { 95, 206, 155, 162, 5, 132, 56, 199 };
416
417         symmetric_key skey;
418         unsigned char buf[2][8];
419         int err, y;
420
421         /* test SK64 */
422         if ((err = safer_sk64_setup(sk64_key, 8, 6, &skey)) != CRYPT_OK) {
423             return err;
424         }
425
426         safer_ecb_encrypt(sk64_pt, buf[0], &skey);
427         safer_ecb_decrypt(buf[0], buf[1], &skey);
428
429         if (XMEMCMP(buf[0], sk64_ct, 8) != 0 || XMEMCMP(buf[1], sk64_pt, 8) != 0) {
430             return CRYPT_FAIL_TESTVECTOR;
431         }
432
433         /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
434         for (y = 0; y < 8; y++) buf[0][y] = 0;
435         for (y = 0; y < 1000; y++) safer_ecb_encrypt(buf[0], buf[0], &skey);
436         for (y = 0; y < 1000; y++) safer_ecb_decrypt(buf[0], buf[0], &skey);
437         for (y = 0; y < 8; y++) if (buf[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
438
439         return CRYPT_OK;
440     #endif
441 }

```

Here is the call graph for this function:

### 5.14.3 Variable Documentation

#### 5.14.3.1 const unsigned char [safer\\_ebox\[\]](#)

Definition at line 23 of file safer\_tab.c.

#### 5.14.3.2 const struct [ltc\\_cipher\\_descriptor](#) [safer\\_k128\\_desc](#)

**Initial value:**

```

{
    "safer-k128",
    10, 16, 16, 8, SAFER_K128_DEFAULT_NOF_ROUNDS,
    &safer_k128_setup,
    &safer_ecb_encrypt,
    &safer_ecb_decrypt,
    &safer_sk128_test,
    &safer_done,
    &safer_128_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}

```

Definition at line 60 of file safer.c.

### 5.14.3.3 `const struct ltc_cipher_descriptor safer_k64_desc`

**Initial value:**

```
{
    "safer-k64",
    8, 8, 8, 8, SAFER_K64_DEFAULT_NOF_ROUNDS,
    &safer_k64_setup,
    &safer_ecb_encrypt,
    &safer_ecb_decrypt,
    &safer_k64_test,
    &safer_done,
    &safer_64_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 36 of file safer.c.

### 5.14.3.4 `const unsigned char safer_lbox[]`

Definition at line 43 of file safer\_tab.c.

### 5.14.3.5 `const struct ltc_cipher_descriptor safer_sk128_desc`

**Initial value:**

```
{
    "safer-sk128",
    11, 16, 16, 8, SAFER_SK128_DEFAULT_NOF_ROUNDS,
    &safer_sk128_setup,
    &safer_ecb_encrypt,
    &safer_ecb_decrypt,
    &safer_sk128_test,
    &safer_done,
    &safer_128_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 72 of file safer.c.

Referenced by `yarrow_start()`.

### 5.14.3.6 `const struct ltc_cipher_descriptor safer_sk64_desc`

**Initial value:**

```
{
    "safer-sk64",
    9, 8, 8, 8, SAFER_SK64_DEFAULT_NOF_ROUNDS,
    &safer_sk64_setup,
    &safer_ecb_encrypt,
    &safer_ecb_decrypt,
    &safer_sk64_test,
    &safer_done,
    &safer_64_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 48 of file safer.c.



## 5.15 ciphers/safer/safer\_tab.c File Reference

### 5.15.1 Detailed Description

Tables for SAFER block ciphers.

Definition in file [safer\\_tab.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for safer\_tab.c:

### Variables

- const unsigned char [safer\\_ebox](#) [256]
- const unsigned char [safer\\_lbox](#) [256]

### 5.15.2 Variable Documentation

#### 5.15.2.1 const unsigned char [safer\\_ebox](#)[256]

**Initial value:**

```
{
  1, 45, 226, 147, 190, 69, 21, 174, 120, 3, 135, 164, 184, 56, 207, 63,
  8, 103, 9, 148, 235, 38, 168, 107, 189, 24, 52, 27, 187, 191, 114, 247,
  64, 53, 72, 156, 81, 47, 59, 85, 227, 192, 159, 216, 211, 243, 141, 177,
  255, 167, 62, 220, 134, 119, 215, 166, 17, 251, 244, 186, 146, 145, 100, 131,
  241, 51, 239, 218, 44, 181, 178, 43, 136, 209, 153, 203, 140, 132, 29, 20,
  129, 151, 113, 202, 95, 163, 139, 87, 60, 130, 196, 82, 92, 28, 232, 160,
  4, 180, 133, 74, 246, 19, 84, 182, 223, 12, 26, 142, 222, 224, 57, 252,
  32, 155, 36, 78, 169, 152, 158, 171, 242, 96, 208, 108, 234, 250, 199, 217,
  0, 212, 31, 110, 67, 188, 236, 83, 137, 254, 122, 93, 73, 201, 50, 194,
  249, 154, 248, 109, 22, 219, 89, 150, 68, 233, 205, 230, 70, 66, 143, 10,
  193, 204, 185, 101, 176, 210, 198, 172, 30, 65, 98, 41, 46, 14, 116, 80,
  2, 90, 195, 37, 123, 138, 42, 91, 240, 6, 13, 71, 111, 112, 157, 126,
  16, 206, 18, 39, 213, 76, 79, 214, 121, 48, 104, 54, 117, 125, 228, 237,
  128, 106, 144, 55, 162, 94, 118, 170, 197, 127, 61, 175, 165, 229, 25, 97,
  253, 77, 124, 183, 11, 238, 173, 75, 34, 245, 231, 115, 35, 33, 200, 5,
  225, 102, 221, 179, 88, 105, 99, 86, 15, 161, 49, 149, 23, 7, 58, 40
}
```

Definition at line 23 of file safer\_tab.c.

#### 5.15.2.2 const unsigned char [safer\\_lbox](#)[256]

**Initial value:**

```
{
  128, 0, 176, 9, 96, 239, 185, 253, 16, 18, 159, 228, 105, 186, 173, 248,
  192, 56, 194, 101, 79, 6, 148, 252, 25, 222, 106, 27, 93, 78, 168, 130,
  112, 237, 232, 236, 114, 179, 21, 195, 255, 171, 182, 71, 68, 1, 172, 37,
  201, 250, 142, 65, 26, 33, 203, 211, 13, 110, 254, 38, 88, 218, 50, 15,
  32, 169, 157, 132, 152, 5, 156, 187, 34, 140, 99, 231, 197, 225, 115, 198,
  175, 36, 91, 135, 102, 39, 247, 87, 244, 150, 177, 183, 92, 139, 213, 84,
  121, 223, 170, 246, 62, 163, 241, 17, 202, 245, 209, 23, 123, 147, 131, 188,
  189, 82, 30, 235, 174, 204, 214, 53, 8, 200, 138, 180, 226, 205, 191, 217,
}
```

```
208, 80, 89, 63, 77, 98, 52, 10, 72, 136, 181, 86, 76, 46, 107, 158,
210, 61, 60, 3, 19, 251, 151, 81, 117, 74, 145, 113, 35, 190, 118, 42,
95, 249, 212, 85, 11, 220, 55, 49, 22, 116, 215, 119, 167, 230, 7, 219,
164, 47, 70, 243, 97, 69, 103, 227, 12, 162, 59, 28, 133, 24, 4, 29,
41, 160, 143, 178, 90, 216, 166, 126, 238, 141, 83, 75, 161, 154, 193, 14,
122, 73, 165, 44, 129, 196, 199, 54, 43, 127, 67, 149, 51, 242, 108, 104,
109, 240, 2, 40, 206, 221, 155, 234, 94, 153, 124, 20, 134, 207, 229, 66,
184, 64, 120, 45, 58, 233, 100, 31, 146, 144, 125, 57, 111, 224, 137, 48
}
```

Definition at line 43 of file safer\_tab.c.

## 5.16 ciphers/safer/saferp.c File Reference

### 5.16.1 Detailed Description

SAFER+ Implementation by Tom St Denis.

Definition in file [saferp.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for saferp.c:

#### Defines

- #define [ROUND](#)(b, i)
- #define [iROUND](#)(b, i)
- #define [PHT](#)(b)
- #define [iPHT](#)(b)
- #define [SHUF](#)(b, b2)
- #define [iSHUF](#)(b, b2)
- #define [LT](#)(b, b2)
- #define [iLT](#)(b, b2)

#### Functions

- int [saferp\\_setup](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the SAFER+ block cipher.*
- int [saferp\\_ecb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with SAFER+.*
- int [saferp\\_ecb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with SAFER+.*
- int [saferp\\_test](#) (void)  
*Performs a self-test of the SAFER+ block cipher.*
- void [saferp\\_done](#) ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- int [saferp\\_keysize](#) (int \*keysize)  
*Gets suitable key size.*

#### Variables

- const struct [ltc\\_cipher\\_descriptor](#) [saferp\\_desc](#)
- const unsigned char [safer\\_ebox](#) []
- const unsigned char [safer\\_lbox](#) []
- static const unsigned char [safer\\_bias](#) [33][16]

## 5.16.2 Define Documentation

### 5.16.2.1 #define iLT(b, b2)

**Value:**

```
iPHT(b);
    iSHUF(b, b2); iPHT(b2); \
    iSHUF(b2, b); iPHT(b); \
    iSHUF(b, b2); iPHT(b2);
```

Definition at line 131 of file saferp.c.

Referenced by saferp\_ecb\_decrypt().

### 5.16.2.2 #define iPHT(b)

**Value:**

```
b[15] = (b[15] - (b[14] = (b[14] - b[15]) & 255)) & 255; \
b[13] = (b[13] - (b[12] = (b[12] - b[13]) & 255)) & 255; \
b[11] = (b[11] - (b[10] = (b[10] - b[11]) & 255)) & 255; \
b[9]  = (b[9] - (b[8] = (b[8] - b[9]) & 255)) & 255; \
b[7]  = (b[7] - (b[6] = (b[6] - b[7]) & 255)) & 255; \
b[5]  = (b[5] - (b[4] = (b[4] - b[5]) & 255)) & 255; \
b[3]  = (b[3] - (b[2] = (b[2] - b[3]) & 255)) & 255; \
b[1]  = (b[1] - (b[0] = (b[0] - b[1]) & 255)) & 255; \
```

Definition at line 96 of file saferp.c.

### 5.16.2.3 #define iROUND(b, i)

**Value:**

```
b[0] = safer_lbox[(b[0] - skey->saferp.K[i+1][0]) & 255] ^ skey->saferp.K[i][0]; \
b[1] = (safer_ebox[(b[1] ^ skey->saferp.K[i+1][1]) & 255] - skey->saferp.K[i][1]) & 255; \
b[2] = (safer_ebox[(b[2] ^ skey->saferp.K[i+1][2]) & 255] - skey->saferp.K[i][2]) & 255; \
b[3] = safer_lbox[(b[3] - skey->saferp.K[i+1][3]) & 255] ^ skey->saferp.K[i][3]; \
b[4] = safer_lbox[(b[4] - skey->saferp.K[i+1][4]) & 255] ^ skey->saferp.K[i][4]; \
b[5] = (safer_ebox[(b[5] ^ skey->saferp.K[i+1][5]) & 255] - skey->saferp.K[i][5]) & 255; \
b[6] = (safer_ebox[(b[6] ^ skey->saferp.K[i+1][6]) & 255] - skey->saferp.K[i][6]) & 255; \
b[7] = safer_lbox[(b[7] - skey->saferp.K[i+1][7]) & 255] ^ skey->saferp.K[i][7]; \
b[8] = safer_lbox[(b[8] - skey->saferp.K[i+1][8]) & 255] ^ skey->saferp.K[i][8]; \
b[9] = (safer_ebox[(b[9] ^ skey->saferp.K[i+1][9]) & 255] - skey->saferp.K[i][9]) & 255; \
b[10] = (safer_ebox[(b[10] ^ skey->saferp.K[i+1][10]) & 255] - skey->saferp.K[i][10]) & 255; \
b[11] = safer_lbox[(b[11] - skey->saferp.K[i+1][11]) & 255] ^ skey->saferp.K[i][11]; \
b[12] = safer_lbox[(b[12] - skey->saferp.K[i+1][12]) & 255] ^ skey->saferp.K[i][12]; \
b[13] = (safer_ebox[(b[13] ^ skey->saferp.K[i+1][13]) & 255] - skey->saferp.K[i][13]) & 255; \
b[14] = (safer_ebox[(b[14] ^ skey->saferp.K[i+1][14]) & 255] - skey->saferp.K[i][14]) & 255; \
b[15] = safer_lbox[(b[15] - skey->saferp.K[i+1][15]) & 255] ^ skey->saferp.K[i][15];
```

Definition at line 66 of file saferp.c.

Referenced by saferp\_ecb\_decrypt().

### 5.16.2.4 #define iSHUF(b, b2)

**Value:**

```

b2[0] = b[12]; b2[1] = b[5]; b2[2] = b[4]; b2[3] = b[15];      \
    b2[4] = b[14]; b2[5] = b[7]; b2[6] = b[6]; b2[7] = b[13];    \
    b2[8] = b[0]; b2[9] = b[9]; b2[10] = b[8]; b2[11] = b[1];    \
    b2[12] = b[2]; b2[13] = b[11]; b2[14] = b[10]; b2[15] = b[3];

```

Definition at line 114 of file saferp.c.

### 5.16.2.5 #define LT(b, b2)

**Value:**

```

PHT(b);  SHUF(b, b2);      \
    PHT(b2); SHUF(b2, b);    \
    PHT(b);  SHUF(b, b2);    \
    PHT(b2);

```

Definition at line 124 of file saferp.c.

### 5.16.2.6 #define PHT(b)

**Value:**

```

b[0] = (b[0] + (b[1] = (b[0] + b[1]) & 255)) & 255;      \
    b[2] = (b[2] + (b[3] = (b[3] + b[2]) & 255)) & 255;    \
    b[4] = (b[4] + (b[5] = (b[5] + b[4]) & 255)) & 255;    \
    b[6] = (b[6] + (b[7] = (b[7] + b[6]) & 255)) & 255;    \
    b[8] = (b[8] + (b[9] = (b[9] + b[8]) & 255)) & 255;    \
    b[10] = (b[10] + (b[11] = (b[11] + b[10]) & 255)) & 255; \
    b[12] = (b[12] + (b[13] = (b[13] + b[12]) & 255)) & 255; \
    b[14] = (b[14] + (b[15] = (b[15] + b[14]) & 255)) & 255;

```

Definition at line 85 of file saferp.c.

### 5.16.2.7 #define ROUND(b, i)

**Value:**

```

b[0] = (safer_ebox[(b[0] ^ skey->saferp.K[i][0]) & 255] + skey->saferp.K[i+1][0]) & 255;      \
    b[1] = safer_lbox[(b[1] + skey->saferp.K[i][1]) & 255] ^ skey->saferp.K[i+1][1];          \
    b[2] = safer_lbox[(b[2] + skey->saferp.K[i][2]) & 255] ^ skey->saferp.K[i+1][2];          \
    b[3] = (safer_ebox[(b[3] ^ skey->saferp.K[i][3]) & 255] + skey->saferp.K[i+1][3]) & 255;    \
    b[4] = (safer_ebox[(b[4] ^ skey->saferp.K[i][4]) & 255] + skey->saferp.K[i+1][4]) & 255;    \
    b[5] = safer_lbox[(b[5] + skey->saferp.K[i][5]) & 255] ^ skey->saferp.K[i+1][5];          \
    b[6] = safer_lbox[(b[6] + skey->saferp.K[i][6]) & 255] ^ skey->saferp.K[i+1][6];          \
    b[7] = (safer_ebox[(b[7] ^ skey->saferp.K[i][7]) & 255] + skey->saferp.K[i+1][7]) & 255;    \
    b[8] = (safer_ebox[(b[8] ^ skey->saferp.K[i][8]) & 255] + skey->saferp.K[i+1][8]) & 255;    \
    b[9] = safer_lbox[(b[9] + skey->saferp.K[i][9]) & 255] ^ skey->saferp.K[i+1][9];          \
    b[10] = safer_lbox[(b[10] + skey->saferp.K[i][10]) & 255] ^ skey->saferp.K[i+1][10];        \
    b[11] = (safer_ebox[(b[11] ^ skey->saferp.K[i][11]) & 255] + skey->saferp.K[i+1][11]) & 255; \
    b[12] = (safer_ebox[(b[12] ^ skey->saferp.K[i][12]) & 255] + skey->saferp.K[i+1][12]) & 255; \
    b[13] = safer_lbox[(b[13] + skey->saferp.K[i][13]) & 255] ^ skey->saferp.K[i+1][13];      \
    b[14] = safer_lbox[(b[14] + skey->saferp.K[i][14]) & 255] ^ skey->saferp.K[i+1][14];      \
    b[15] = (safer_ebox[(b[15] ^ skey->saferp.K[i][15]) & 255] + skey->saferp.K[i+1][15]) & 255;

```

Definition at line 47 of file saferp.c.

### 5.16.2.8 #define SHUF(b, b2)

**Value:**

```
b2[0] = b[8]; b2[1] = b[11]; b2[2] = b[12]; b2[3] = b[15]; \
    b2[4] = b[2]; b2[5] = b[1]; b2[6] = b[6]; b2[7] = b[5]; \
    b2[8] = b[10]; b2[9] = b[9]; b2[10] = b[14]; b2[11] = b[13]; \
    b2[12] = b[0]; b2[13] = b[7]; b2[14] = b[4]; b2[15] = b[3];
```

Definition at line 107 of file saferp.c.

## 5.16.3 Function Documentation

### 5.16.3.1 void saferp\_done (symmetric\_key \* skey)

Terminate the context.

**Parameters:**

*skey* The scheduled key

Definition at line 528 of file saferp.c.

```
529 {
530 }
```

### 5.16.3.2 int saferp\_ecb\_decrypt (const unsigned char \* ct, unsigned char \* pt, symmetric\_key \* skey)

Decrypts a block of text with SAFER+.

**Parameters:**

*ct* The input ciphertext (16 bytes)

*pt* The output plaintext (16 bytes)

*skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 398 of file saferp.c.

References iLT, iROUND, and LTC\_ARGCHK.

```
399 {
400     unsigned char b[16];
401     int x;
402
403     LTC_ARGCHK(pt != NULL);
404     LTC_ARGCHK(ct != NULL);
405     LTC_ARGCHK(skey != NULL);
406
407     /* do eight rounds */
408     b[0] = ct[0] ^ skey->saferp.K[skey->saferp.rounds*2][0];
409     b[1] = (ct[1] - skey->saferp.K[skey->saferp.rounds*2][1]) & 255;
```

```

410     b[2] = (ct[2] - skey->saferp.K[skey->saferp.rounds*2][2]) & 255;
411     b[3] = ct[3] ^ skey->saferp.K[skey->saferp.rounds*2][3];
412     b[4] = ct[4] ^ skey->saferp.K[skey->saferp.rounds*2][4];
413     b[5] = (ct[5] - skey->saferp.K[skey->saferp.rounds*2][5]) & 255;
414     b[6] = (ct[6] - skey->saferp.K[skey->saferp.rounds*2][6]) & 255;
415     b[7] = ct[7] ^ skey->saferp.K[skey->saferp.rounds*2][7];
416     b[8] = ct[8] ^ skey->saferp.K[skey->saferp.rounds*2][8];
417     b[9] = (ct[9] - skey->saferp.K[skey->saferp.rounds*2][9]) & 255;
418     b[10] = (ct[10] - skey->saferp.K[skey->saferp.rounds*2][10]) & 255;
419     b[11] = ct[11] ^ skey->saferp.K[skey->saferp.rounds*2][11];
420     b[12] = ct[12] ^ skey->saferp.K[skey->saferp.rounds*2][12];
421     b[13] = (ct[13] - skey->saferp.K[skey->saferp.rounds*2][13]) & 255;
422     b[14] = (ct[14] - skey->saferp.K[skey->saferp.rounds*2][14]) & 255;
423     b[15] = ct[15] ^ skey->saferp.K[skey->saferp.rounds*2][15];
424     /* 256-bit key? */
425     if (skey->saferp.rounds > 12) {
426         iLT(b, pt); iROUND(pt, 30);
427         iLT(pt, b); iROUND(b, 28);
428         iLT(b, pt); iROUND(pt, 26);
429         iLT(pt, b); iROUND(b, 24);
430     }
431     /* 192-bit key? */
432     if (skey->saferp.rounds > 8) {
433         iLT(b, pt); iROUND(pt, 22);
434         iLT(pt, b); iROUND(b, 20);
435         iLT(b, pt); iROUND(pt, 18);
436         iLT(pt, b); iROUND(b, 16);
437     }
438     iLT(b, pt); iROUND(pt, 14);
439     iLT(pt, b); iROUND(b, 12);
440     iLT(b, pt); iROUND(pt, 10);
441     iLT(pt, b); iROUND(b, 8);
442     iLT(b, pt); iROUND(pt, 6);
443     iLT(pt, b); iROUND(b, 4);
444     iLT(b, pt); iROUND(pt, 2);
445     iLT(pt, b); iROUND(b, 0);
446     for (x = 0; x < 16; x++) {
447         pt[x] = b[x];
448     }
449 #ifdef LTC_CLEAN_STACK
450     zeromem(b, sizeof(b));
451 #endif
452     return CRYPT_OK;
453 }

```

### 5.16.3.3 `int saferp_ecb_encrypt (const unsigned char * pt, unsigned char * ct, symmetric\_key * skey)`

Encrypts a block of text with SAFER+.

#### Parameters:

- pt* The input plaintext (16 bytes)
- ct* The output ciphertext (16 bytes)
- skey* The key as scheduled

#### Returns:

CRYPT\_OK if successful

Definition at line 334 of file saferp.c.

References LTC\_ARGCHK.

```

335 {
336     unsigned char b[16];
337     int x;
338
339     LTC_ARGCHK(pt != NULL);
340     LTC_ARGCHK(ct != NULL);
341     LTC_ARGCHK(skey != NULL);
342
343     /* do eight rounds */
344     for (x = 0; x < 16; x++) {
345         b[x] = pt[x];
346     }
347     ROUND(b, 0); LT(b, ct);
348     ROUND(ct, 2); LT(ct, b);
349     ROUND(b, 4); LT(b, ct);
350     ROUND(ct, 6); LT(ct, b);
351     ROUND(b, 8); LT(b, ct);
352     ROUND(ct, 10); LT(ct, b);
353     ROUND(b, 12); LT(b, ct);
354     ROUND(ct, 14); LT(ct, b);
355     /* 192-bit key? */
356     if (skey->saferp.rounds > 8) {
357         ROUND(b, 16); LT(b, ct);
358         ROUND(ct, 18); LT(ct, b);
359         ROUND(b, 20); LT(b, ct);
360         ROUND(ct, 22); LT(ct, b);
361     }
362     /* 256-bit key? */
363     if (skey->saferp.rounds > 12) {
364         ROUND(b, 24); LT(b, ct);
365         ROUND(ct, 26); LT(ct, b);
366         ROUND(b, 28); LT(b, ct);
367         ROUND(ct, 30); LT(ct, b);
368     }
369     ct[0] = b[0] ^ skey->saferp.K[skey->saferp.rounds*2][0];
370     ct[1] = (b[1] + skey->saferp.K[skey->saferp.rounds*2][1]) & 255;
371     ct[2] = (b[2] + skey->saferp.K[skey->saferp.rounds*2][2]) & 255;
372     ct[3] = b[3] ^ skey->saferp.K[skey->saferp.rounds*2][3];
373     ct[4] = b[4] ^ skey->saferp.K[skey->saferp.rounds*2][4];
374     ct[5] = (b[5] + skey->saferp.K[skey->saferp.rounds*2][5]) & 255;
375     ct[6] = (b[6] + skey->saferp.K[skey->saferp.rounds*2][6]) & 255;
376     ct[7] = b[7] ^ skey->saferp.K[skey->saferp.rounds*2][7];
377     ct[8] = b[8] ^ skey->saferp.K[skey->saferp.rounds*2][8];
378     ct[9] = (b[9] + skey->saferp.K[skey->saferp.rounds*2][9]) & 255;
379     ct[10] = (b[10] + skey->saferp.K[skey->saferp.rounds*2][10]) & 255;
380     ct[11] = b[11] ^ skey->saferp.K[skey->saferp.rounds*2][11];
381     ct[12] = b[12] ^ skey->saferp.K[skey->saferp.rounds*2][12];
382     ct[13] = (b[13] + skey->saferp.K[skey->saferp.rounds*2][13]) & 255;
383     ct[14] = (b[14] + skey->saferp.K[skey->saferp.rounds*2][14]) & 255;
384     ct[15] = b[15] ^ skey->saferp.K[skey->saferp.rounds*2][15];
385 #ifdef LTC_CLEAN_STACK
386     zeromem(b, sizeof(b));
387 #endif
388     return CRYPT_OK;
389 }

```

#### 5.16.3.4 int saferp\_keysize (int \* keysize)

Gets suitable key size.

##### Parameters:

*keysize* [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.



**Returns:**

CRYPT\_OK if the input key size is acceptable.

Definition at line 537 of file saferp.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

538 {
539     LTC_ARGCHK(keysize != NULL);
540
541     if (*keysize < 16)
542         return CRYPT_INVALID_KEYSIZE;
543     if (*keysize < 24) {
544         *keysize = 16;
545     } else if (*keysize < 32) {
546         *keysize = 24;
547     } else {
548         *keysize = 32;
549     }
550     return CRYPT_OK;
551 }
```

### 5.16.3.5 int saferp\_setup (const unsigned char \*key, int keylen, int num\_rounds, symmetric\_key \*skey)

Initialize the SAFER+ block cipher.

**Parameters:**

**key** The symmetric key you wish to pass

**keylen** The key length in bytes

**num\_rounds** The number of rounds desired (0 for default)

**skey** The key in as scheduled by this function.

**Returns:**

CRYPT\_OK if successful

Definition at line 216 of file saferp.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, LTC\_ARGCHK, and rounds().

Referenced by saferp\_test().

```

217 {
218     unsigned x, y, z;
219     unsigned char t[33];
220     static const int rounds[3] = { 8, 12, 16 };
221
222     LTC_ARGCHK(key != NULL);
223     LTC_ARGCHK(skey != NULL);
224
225     /* check arguments */
226     if (keylen != 16 && keylen != 24 && keylen != 32) {
227         return CRYPT_INVALID_KEYSIZE;
228     }
229
230     /* Is the number of rounds valid? Either use zero for default or
231      * 8,12,16 rounds for 16,24,32 byte keys
232     */
```

```

233     if (num_rounds != 0 && num_rounds != rounds[(keylen/8)-2]) {
234         return CRYPT_INVALID_ROUNDS;
235     }
236
237     /* 128 bit key version */
238     if (keylen == 16) {
239         /* copy key into t */
240         for (x = y = 0; x < 16; x++) {
241             t[x] = key[x];
242             y ^= key[x];
243         }
244         t[16] = y;
245
246         /* make round keys */
247         for (x = 0; x < 16; x++) {
248             skey->saferp.K[0][x] = t[x];
249         }
250
251         /* make the 16 other keys as a transformation of the first key */
252         for (x = 1; x < 17; x++) {
253             /* rotate 3 bits each */
254             for (y = 0; y < 17; y++) {
255                 t[y] = ((t[y]<<3)|(t[y]>>5)) & 255;
256             }
257
258             /* select and add */
259             z = x;
260             for (y = 0; y < 16; y++) {
261                 skey->saferp.K[x][y] = (t[z] + safer_bias[x-1][y]) & 255;
262                 if (++z == 17) { z = 0; }
263             }
264         }
265         skey->saferp.rounds = 8;
266     } else if (keylen == 24) {
267         /* copy key into t */
268         for (x = y = 0; x < 24; x++) {
269             t[x] = key[x];
270             y ^= key[x];
271         }
272         t[24] = y;
273
274         /* make round keys */
275         for (x = 0; x < 16; x++) {
276             skey->saferp.K[0][x] = t[x];
277         }
278
279         for (x = 1; x < 25; x++) {
280             /* rotate 3 bits each */
281             for (y = 0; y < 25; y++) {
282                 t[y] = ((t[y]<<3)|(t[y]>>5)) & 255;
283             }
284
285             /* select and add */
286             z = x;
287             for (y = 0; y < 16; y++) {
288                 skey->saferp.K[x][y] = (t[z] + safer_bias[x-1][y]) & 255;
289                 if (++z == 25) { z = 0; }
290             }
291         }
292         skey->saferp.rounds = 12;
293     } else {
294         /* copy key into t */
295         for (x = y = 0; x < 32; x++) {
296             t[x] = key[x];
297             y ^= key[x];
298         }
299         t[32] = y;

```

```

300
301     /* make round keys */
302     for (x = 0; x < 16; x++) {
303         skey->saferp.K[0][x] = t[x];
304     }
305
306     for (x = 1; x < 33; x++) {
307         /* rotate 3 bits each */
308         for (y = 0; y < 33; y++) {
309             t[y] = ((t[y]<<3)|(t[y]>>5)) & 255;
310         }
311
312         /* select and add */
313         z = x;
314         for (y = 0; y < 16; y++) {
315             skey->saferp.K[x][y] = (t[z] + safer_bias[x-1][y]) & 255;
316             if (++z == 33) { z = 0; }
317         }
318     }
319     skey->saferp.rounds = 16;
320 }
321 #ifdef LTC_CLEAN_STACK
322     zeromem(t, sizeof(t));
323 #endif
324     return CRYPT_OK;
325 }

```

Here is the call graph for this function:

#### 5.16.3.6 int saferp\_test (void)

Performs a self-test of the SAFER+ block cipher.

##### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 459 of file saferp.c.

References CRYPT\_NOP, CRYPT\_OK, and saferp\_setup().

```

460 {
461     #ifndef LTC_TEST
462         return CRYPT_NOP;
463     #else
464         static const struct {
465             int keylen;
466             unsigned char key[32], pt[16], ct[16];
467         } tests[] = {
468             {
469                 16,
470                 { 41, 35, 190, 132, 225, 108, 214, 174,
471                   82, 144, 73, 241, 241, 187, 233, 235 },
472                 { 179, 166, 219, 60, 135, 12, 62, 153,
473                   36, 94, 13, 28, 6, 183, 71, 222 },
474                 { 224, 31, 182, 10, 12, 255, 84, 70,
475                   127, 13, 89, 249, 9, 57, 165, 220 }
476             }, {
477                 24,
478                 { 72, 211, 143, 117, 230, 217, 29, 42,
479                   229, 192, 247, 43, 120, 129, 135, 68,
480                   14, 95, 80, 0, 212, 97, 141, 190 },
481                 { 123, 5, 21, 7, 59, 51, 130, 31,
482                   24, 112, 146, 218, 100, 84, 206, 177 },

```

```

483         { 92, 136, 4, 63, 57, 95, 100, 0,
484           150, 130, 130, 16, 193, 111, 219, 133 }
485     }, {
486         32,
487         { 243, 168, 141, 254, 190, 242, 235, 113,
488           255, 160, 208, 59, 117, 6, 140, 126,
489           135, 120, 115, 77, 208, 190, 130, 190,
490           219, 194, 70, 65, 43, 140, 250, 48 },
491         { 127, 112, 240, 167, 84, 134, 50, 149,
492           170, 91, 104, 19, 11, 230, 252, 245 },
493         { 88, 11, 25, 36, 172, 229, 202, 213,
494           170, 65, 105, 153, 220, 104, 153, 138 }
495     }
496 };
497
498 unsigned char tmp[2][16];
499 symmetric_key skey;
500 int err, i, y;
501
502 for (i = 0; i < (int)(sizeof(tests) / sizeof(tests[0])); i++) {
503     if ((err = saferp_setup(tests[i].key, tests[i].keylen, 0, &skey)) != CRYPT_OK) {
504         return err;
505     }
506     saferp_ecb_encrypt(tests[i].pt, tmp[0], &skey);
507     saferp_ecb_decrypt(tmp[0], tmp[1], &skey);
508
509     /* compare */
510     if (XMEMCMP(tmp[0], tests[i].ct, 16) || XMEMCMP(tmp[1], tests[i].pt, 16)) {
511         return CRYPT_FAIL_TESTVECTOR;
512     }
513
514     /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
515     for (y = 0; y < 16; y++) tmp[0][y] = 0;
516     for (y = 0; y < 1000; y++) saferp_ecb_encrypt(tmp[0], tmp[0], &skey);
517     for (y = 0; y < 1000; y++) saferp_ecb_decrypt(tmp[0], tmp[0], &skey);
518     for (y = 0; y < 16; y++) if (tmp[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
519 }
520
521 return CRYPT_OK;
522 #endif
523 }

```

Here is the call graph for this function:

## 5.16.4 Variable Documentation

### 5.16.4.1 const unsigned char [safer\\_bias](#)[33][16] [static]

Definition at line 174 of file saferp.c.

### 5.16.4.2 const unsigned char [safer\\_ebox](#)[]

Definition at line 23 of file safer\_tab.c.

### 5.16.4.3 const unsigned char [safer\\_lbox](#)[]

Definition at line 43 of file safer\_tab.c.

#### 5.16.4.4 const struct [ltc\\_cipher\\_descriptor](#) saferp\_desc

**Initial value:**

```
{
    "safer+",
    4,
    16, 32, 16, 8,
    &saferp_setup,
    &saferp_ecb_encrypt,
    &saferp_ecb_decrypt,
    &saferp_test,
    &saferp_done,
    &saferp_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 20 of file saferp.c.

Referenced by `yarrow_start()`.

## 5.17 ciphers/skipjack.c File Reference

### 5.17.1 Detailed Description

Skipjack Implementation by Tom St Denis.

Definition in file [skipjack.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for skipjack.c:

#### Defines

- #define [RULE\\_A](#)
- #define [RULE\\_B](#)
- #define [RULE\\_A1](#)
- #define [RULE\\_B1](#)

#### Functions

- int [skipjack\\_setup](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)  
*Initialize the Skipjack block cipher.*
- static unsigned [g\\_func](#) (unsigned w, int \*kp, unsigned char \*key)
- static unsigned [ig\\_func](#) (unsigned w, int \*kp, unsigned char \*key)
- int [skipjack\\_ecb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with Skipjack.*
- int [skipjack\\_ecb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with Skipjack.*
- int [skipjack\\_test](#) (void)  
*Performs a self-test of the Skipjack block cipher.*
- void [skipjack\\_done](#) ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- int [skipjack\\_keysize](#) (int \*keysize)  
*Gets suitable key size.*

#### Variables

- const struct [lte\\_cipher\\_descriptor](#) [skipjack\\_desc](#)
- static const unsigned char [sbox](#) [256]
- static const int [keystep](#) [] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 }
- static const int [ikeystep](#) [] = { 9, 0, 1, 2, 3, 4, 5, 6, 7, 8 }

## 5.17.2 Define Documentation

### 5.17.2.1 #define RULE\_A

**Value:**

```
tmp = g_func(w1, &kp, skey->skipjack.key); \
w1  = tmp ^ w4 ^ x;                      \
w4  = w3; w3 = w2;                      \
w2  = tmp;
```

Definition at line 90 of file skipjack.c.

Referenced by skipjack\_ecb\_encrypt().

### 5.17.2.2 #define RULE\_A1

**Value:**

```
tmp = w1 ^ w2 ^ x; \
w1  = ig_func(w2, &kp, skey->skipjack.key); \
w2  = w3; w3 = w4; w4 = tmp;
```

Definition at line 102 of file skipjack.c.

### 5.17.2.3 #define RULE\_B

**Value:**

```
tmp  = g_func(w1, &kp, skey->skipjack.key); \
tmp1 = w4; w4  = w3;                      \
w3   = w1 ^ w2 ^ x;                      \
w1   = tmp1; w2 = tmp;
```

Definition at line 96 of file skipjack.c.

### 5.17.2.4 #define RULE\_B1

**Value:**

```
tmp = ig_func(w2, &kp, skey->skipjack.key); \
w2  = tmp ^ w3 ^ x;                      \
w3  = w4; w4 = w1; w1 = tmp;
```

Definition at line 107 of file skipjack.c.

Referenced by skipjack\_ecb\_decrypt().

## 5.17.3 Function Documentation

### 5.17.3.1 static unsigned g\_func (unsigned w, int \*kp, unsigned char \*key) [static]

Definition at line 112 of file skipjack.c.

References keystep, and sbbox.

```

113 {
114     unsigned char g1,g2;
115
116     g1 = (w >> 8) & 255; g2 = w & 255;
117     g1 ^= sbbox[g2^key[*kp]]; *kp = keystep[*kp];
118     g2 ^= sbbox[g1^key[*kp]]; *kp = keystep[*kp];
119     g1 ^= sbbox[g2^key[*kp]]; *kp = keystep[*kp];
120     g2 ^= sbbox[g1^key[*kp]]; *kp = keystep[*kp];
121     return ((unsigned)g1<<8) | (unsigned)g2;
122 }

```

### 5.17.3.2 static unsigned ig\_func (unsigned w, int \*kp, unsigned char \*key) [static]

Definition at line 124 of file skipjack.c.

References `ikeystep`, and `sbox`.

```

125 {
126     unsigned char g1,g2;
127
128     g1 = (w >> 8) & 255; g2 = w & 255;
129     *kp = ikeystep[*kp]; g2 ^= sbbox[g1^key[*kp]];
130     *kp = ikeystep[*kp]; g1 ^= sbbox[g2^key[*kp]];
131     *kp = ikeystep[*kp]; g2 ^= sbbox[g1^key[*kp]];
132     *kp = ikeystep[*kp]; g1 ^= sbbox[g2^key[*kp]];
133     return ((unsigned)g1<<8) | (unsigned)g2;
134 }

```

### 5.17.3.3 void skipjack\_done (symmetric\_key \*skey)

Terminate the context.

#### Parameters:

*skey* The scheduled key

Definition at line 319 of file skipjack.c.

```

320 {
321 }

```

### 5.17.3.4 int skipjack\_ecb\_decrypt (const unsigned char \*ct, unsigned char \*pt, symmetric\_key \*skey)

Decrypts a block of text with Skipjack.

#### Parameters:

*ct* The input ciphertext (8 bytes)

*pt* The output plaintext (8 bytes)

*skey* The key as scheduled

#### Returns:

CRYPT\_OK if successful



Definition at line 210 of file skipjack.c.

References LTC\_ARGCHK, and RULE\_B1.

```

212 {
213     unsigned w1,w2,w3,w4,tmp;
214     int x, kp;
215
216     LTC_ARGCHK(pt != NULL);
217     LTC_ARGCHK(ct != NULL);
218     LTC_ARGCHK(skey != NULL);
219
220     /* load block */
221     w1 = ((unsigned)ct[0]<<8)|ct[1];
222     w2 = ((unsigned)ct[2]<<8)|ct[3];
223     w3 = ((unsigned)ct[4]<<8)|ct[5];
224     w4 = ((unsigned)ct[6]<<8)|ct[7];
225
226     /* 8 rounds of RULE B^-1
227
228     Note the value "kp = 8" comes from "kp = (32 * 4) mod 10" where 32*4 is 128 which mod 10 is 8
229     */
230     for (x = 32, kp = 8; x > 24; x--) {
231         RULE_B1;
232     }
233
234     /* 8 rounds of RULE A^-1 */
235     for (; x > 16; x--) {
236         RULE_A1;
237     }
238
239
240     /* 8 rounds of RULE B^-1 */
241     for (; x > 8; x--) {
242         RULE_B1;
243     }
244
245     /* 8 rounds of RULE A^-1 */
246     for (; x > 0; x--) {
247         RULE_A1;
248     }
249
250     /* store block */
251     pt[0] = (w1>>8)&255; pt[1] = w1&255;
252     pt[2] = (w2>>8)&255; pt[3] = w2&255;
253     pt[4] = (w3>>8)&255; pt[5] = w3&255;
254     pt[6] = (w4>>8)&255; pt[7] = w4&255;
255
256     return CRYPT_OK;
257 }

```

### 5.17.3.5 int skipjack\_ecb\_encrypt (const unsigned char \* *pt*, unsigned char \* *ct*, [symmetric\\_key](#) \* *skey*)

Encrypts a block of text with Skipjack.

#### Parameters:

- pt* The input plaintext (8 bytes)
- ct* The output ciphertext (8 bytes)
- skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 146 of file skipjack.c.

References LTC\_ARGCHK, and RULE\_A.

```

148 {
149     unsigned w1,w2,w3,w4,tmp,tmp1;
150     int x, kp;
151
152     LTC_ARGCHK(pt != NULL);
153     LTC_ARGCHK(ct != NULL);
154     LTC_ARGCHK(skey != NULL);
155
156     /* load block */
157     w1 = ((unsigned)pt[0]<<8)|pt[1];
158     w2 = ((unsigned)pt[2]<<8)|pt[3];
159     w3 = ((unsigned)pt[4]<<8)|pt[5];
160     w4 = ((unsigned)pt[6]<<8)|pt[7];
161
162     /* 8 rounds of RULE A */
163     for (x = 1, kp = 0; x < 9; x++) {
164         RULE_A;
165     }
166
167     /* 8 rounds of RULE B */
168     for (; x < 17; x++) {
169         RULE_B;
170     }
171
172     /* 8 rounds of RULE A */
173     for (; x < 25; x++) {
174         RULE_A;
175     }
176
177     /* 8 rounds of RULE B */
178     for (; x < 33; x++) {
179         RULE_B;
180     }
181
182     /* store block */
183     ct[0] = (w1>>8)&255; ct[1] = w1&255;
184     ct[2] = (w2>>8)&255; ct[3] = w2&255;
185     ct[4] = (w3>>8)&255; ct[5] = w3&255;
186     ct[6] = (w4>>8)&255; ct[7] = w4&255;
187
188     return CRYPT_OK;
189 }
```

**5.17.3.6 int skipjack\_keysize (int \* keysize)**

Gets suitable key size.

**Parameters:**

*keysize* [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

**Returns:**

CRYPT\_OK if the input key size is acceptable.

Definition at line 328 of file skipjack.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```
329 {
330     LTC_ARGCHK(keysize != NULL);
331     if (*keysize < 10) {
332         return CRYPT_INVALID_KEYSIZE;
333     } else if (*keysize > 10) {
334         *keysize = 10;
335     }
336     return CRYPT_OK;
337 }
```

### 5.17.3.7 int skipjack\_setup (const unsigned char \* *key*, int *keylen*, int *num\_rounds*, [symmetric\\_key](#) \* *skey*)

Initialize the Skipjack block cipher.

#### Parameters:

- key*** The symmetric key you wish to pass
- keylen*** The key length in bytes
- num\_rounds*** The number of rounds desired (0 for default)
- skey*** The key in as scheduled by this function.

#### Returns:

CRYPT\_OK if successful

Definition at line 67 of file skipjack.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, and LTC\_ARGCHK.

Referenced by skipjack\_test().

```
68 {
69     int x;
70
71     LTC_ARGCHK(key != NULL);
72     LTC_ARGCHK(skey != NULL);
73
74     if (keylen != 10) {
75         return CRYPT_INVALID_KEYSIZE;
76     }
77
78     if (num_rounds != 32 && num_rounds != 0) {
79         return CRYPT_INVALID_ROUNDS;
80     }
81
82     /* make sure the key is in range for platforms where CHAR_BIT != 8 */
83     for (x = 0; x < 10; x++) {
84         skey->skipjack.key[x] = key[x] & 255;
85     }
86
87     return CRYPT_OK;
88 }
```

### 5.17.3.8 int skipjack\_test (void)

Performs a self-test of the Skipjack block cipher.

#### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 272 of file skipjack.c.

References CRYPT\_NOP, CRYPT\_OK, and skipjack\_setup().

```

273 {
274     #ifndef LTC_TEST
275         return CRYPT_NOP;
276     #else
277         static const struct {
278             unsigned char key[10], pt[8], ct[8];
279         } tests[] = {
280             {
281                 { 0x00, 0x99, 0x88, 0x77, 0x66, 0x55, 0x44, 0x33, 0x22, 0x11 },
282                 { 0x33, 0x22, 0x11, 0x00, 0xdd, 0xcc, 0xbb, 0xaa },
283                 { 0x25, 0x87, 0xca, 0xe2, 0x7a, 0x12, 0xd3, 0x00 }
284             }
285         };
286         unsigned char buf[2][8];
287         int x, y, err;
288         symmetric_key key;
289
290         for (x = 0; x < (int)(sizeof(tests) / sizeof(tests[0])); x++) {
291             /* setup key */
292             if ((err = skipjack_setup(tests[x].key, 10, 0, &key)) != CRYPT_OK) {
293                 return err;
294             }
295
296             /* encrypt and decrypt */
297             skipjack_ecb_encrypt(tests[x].pt, buf[0], &key);
298             skipjack_ecb_decrypt(buf[0], buf[1], &key);
299
300             /* compare */
301             if (XMEMCMP(buf[0], tests[x].ct, 8) != 0 || XMEMCMP(buf[1], tests[x].pt, 8) != 0) {
302                 return CRYPT_FAIL_TESTVECTOR;
303             }
304
305             /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
306             for (y = 0; y < 8; y++) buf[0][y] = 0;
307             for (y = 0; y < 1000; y++) skipjack_ecb_encrypt(buf[0], buf[0], &key);
308             for (y = 0; y < 1000; y++) skipjack_ecb_decrypt(buf[0], buf[0], &key);
309             for (y = 0; y < 8; y++) if (buf[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
310         }
311         return CRYPT_OK;
312     #endif
313 }
314 
```

Here is the call graph for this function:

## 5.17.4 Variable Documentation

### 5.17.4.1 const int [ikeystep](#)[] = { 9, 0, 1, 2, 3, 4, 5, 6, 7, 8 } [static]

Definition at line 57 of file skipjack.c.

Referenced by [ig\\_func\(\)](#).

**5.17.4.2** `const int keystep[ ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 } [static]`

Definition at line 54 of file skipjack.c.

Referenced by `g_func()`.

**5.17.4.3** `const unsigned char sbox[256] [static]`

**Initial value:**

```
{
    0xa3,0xd7,0x09,0x83,0xf8,0x48,0xf6,0xf4,0xb3,0x21,0x15,0x78,0x99,0xb1,0xaf,0xf9,
    0xe7,0x2d,0x4d,0x8a,0xce,0x4c,0xca,0x2e,0x52,0x95,0xd9,0x1e,0x4e,0x38,0x44,0x28,
    0x0a,0xdf,0x02,0xa0,0x17,0xf1,0x60,0x68,0x12,0xb7,0x7a,0xc3,0xe9,0xfa,0x3d,0x53,
    0x96,0x84,0x6b,0xba,0xf2,0x63,0x9a,0x19,0x7c,0xae,0xe5,0xf5,0xf7,0x16,0x6a,0xa2,
    0x39,0xb6,0x7b,0x0f,0xc1,0x93,0x81,0x1b,0xee,0xb4,0x1a,0xea,0xd0,0x91,0x2f,0xb8,
    0x55,0xb9,0xda,0x85,0x3f,0x41,0xbf,0xe0,0x5a,0x58,0x80,0x5f,0x66,0x0b,0xd8,0x90,
    0x35,0xd5,0xc0,0xa7,0x33,0x06,0x65,0x69,0x45,0x00,0x94,0x56,0x6d,0x98,0x9b,0x76,
    0x97,0xfc,0xb2,0xc2,0xb0,0xfe,0xdb,0x20,0xe1,0xeb,0xd6,0xe4,0xdd,0x47,0x4a,0x1d,
    0x42,0xed,0x9e,0x6e,0x49,0x3c,0xcd,0x43,0x27,0xd2,0x07,0xd4,0xde,0xc7,0x67,0x18,
    0x89,0xcb,0x30,0x1f,0x8d,0xc6,0x8f,0xaa,0xc8,0x74,0xdc,0xc9,0x5d,0x5c,0x31,0xa4,
    0x70,0x88,0x61,0x2c,0x9f,0x0d,0x2b,0x87,0x50,0x82,0x54,0x64,0x26,0x7d,0x03,0x40,
    0x34,0x4b,0x1c,0x73,0xd1,0xc4,0xfd,0x3b,0xcc,0xfb,0x7f,0xab,0xe6,0x3e,0x5b,0xa5,
    0xad,0x04,0x23,0x9c,0x14,0x51,0x22,0xf0,0x29,0x79,0x71,0x7e,0xff,0x8c,0x0e,0xe2,
    0x0c,0xef,0xbc,0x72,0x75,0x6f,0x37,0xa1,0xec,0xd3,0x8e,0x62,0x8b,0x86,0x10,0xe8,
    0x08,0x77,0x11,0xbe,0x92,0x4f,0x24,0xc5,0x32,0x36,0x9d,0xcf,0xf3,0xa6,0xbb,0xac,
    0x5e,0x6c,0xa9,0x13,0x57,0x25,0xb5,0xe3,0xbd,0xa8,0x3a,0x01,0x05,0x59,0x2a,0x46
}
```

Definition at line 34 of file skipjack.c.

**5.17.4.4** `const struct ltc\_cipher\_descriptor skipjack\_desc`

**Initial value:**

```
{
    "skipjack",
    17,
    10, 10, 8, 32,
    &skipjack_setup,
    &skipjack_ecb_encrypt,
    &skipjack_ecb_decrypt,
    &skipjack_test,
    &skipjack_done,
    &skipjack_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 20 of file skipjack.c.

## 5.18 ciphers/twofish/twofish.c File Reference

### 5.18.1 Detailed Description

Implementation of Twofish by Tom St Denis.

Definition in file [twofish.c](#).

```
#include "tomcrypt.h"
#include "twofish_tab.c"
```

Include dependency graph for twofish.c:

### Defines

- #define [MDS\\_POLY](#) 0x169
- #define [RS\\_POLY](#) 0x14D
- #define [sbox](#)(i, x) (([ulong32](#))SBOX[i][(x)&255])
- #define [mds\\_column\\_mult](#)(x, i) mds\_tab[i][x]
- #define [g\\_func](#)(x, dum) ([S1](#)[byte(x,0)] ^ [S2](#)[byte(x,1)] ^ [S3](#)[byte(x,2)] ^ [S4](#)[byte(x,3)])
- #define [g1\\_func](#)(x, dum) ([S2](#)[byte(x,0)] ^ [S3](#)[byte(x,1)] ^ [S4](#)[byte(x,2)] ^ [S1](#)[byte(x,3)])

### Functions

- static [ulong32](#) [gf\\_mult](#) ([ulong32](#) a, [ulong32](#) b, [ulong32](#) p)
- static void [mds\\_mult](#) (const unsigned char \*in, unsigned char \*out)
- static void [rs\\_mult](#) (const unsigned char \*in, unsigned char \*out)
- static void [h\\_func](#) (const unsigned char \*in, unsigned char \*out, unsigned char \*M, int k, int offset)
- int [twofish\\_setup](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)

*Initialize the Twofish block cipher.*

- int [twofish\\_ecb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)

*Encrypts a block of text with Twofish.*

- int [twofish\\_ecb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)

*Decrypts a block of text with Twofish.*

- int [twofish\\_test](#) (void)

*Performs a self-test of the Twofish block cipher.*

- void [twofish\\_done](#) ([symmetric\\_key](#) \*skey)

*Terminate the context.*

- int [twofish\\_keysize](#) (int \*keysize)

*Gets suitable key size.*

## Variables

- const struct `ltc_cipher_descriptor` `twofish_desc`
- static const unsigned char `MDS` [4][4]
- static const unsigned char `RS` [4][8]
- static const unsigned char `qord` [4][5]

## 5.18.2 Define Documentation

### 5.18.2.1 `#define g1_func(x, dum) (S2[byte(x,0)] ^ S3[byte(x,1)] ^ S4[byte(x,2)] ^ S1[byte(x,3)])`

Definition at line 286 of file `twofish.c`.

Referenced by `twofish_ecb_decrypt()`, and `twofish_ecb_encrypt()`.

### 5.18.2.2 `#define g_func(x, dum) (S1[byte(x,0)] ^ S2[byte(x,1)] ^ S3[byte(x,2)] ^ S4[byte(x,3)])`

Definition at line 285 of file `twofish.c`.

Referenced by `twofish_ecb_decrypt()`, and `twofish_ecb_encrypt()`.

### 5.18.2.3 `#define mds_column_mult(x, i) mds_tab[i][x]`

Definition at line 205 of file `twofish.c`.

Referenced by `mds_mult()`.

### 5.18.2.4 `#define MDS_POLY 0x169`

Definition at line 42 of file `twofish.c`.

### 5.18.2.5 `#define RS_POLY 0x14D`

Definition at line 43 of file `twofish.c`.

Referenced by `rs_mult()`.

### 5.18.2.6 `#define sbbox(i, x) ((ulong32)SBOX[i][(x)&255])`

Definition at line 73 of file `twofish.c`.

Referenced by `g_func()`, and `ig_func()`.

## 5.18.3 Function Documentation

### 5.18.3.1 `static ulong32 gf_mult (ulong32 a, ulong32 b, ulong32 p) [static]`

Definition at line 146 of file `twofish.c`.

References B.

Referenced by `rs_mult()`.

```

147 {
148     ulong32 result, B[2], P[2];
149
150     P[1] = p;
151     B[1] = b;
152     result = P[0] = B[0] = 0;
153
154     /* unrolled branchless GF multiplier */
155     result ^= B[a&1]; a >>= 1; B[1] = P[B[1]>>7] ^ (B[1] << 1);
156     result ^= B[a&1]; a >>= 1; B[1] = P[B[1]>>7] ^ (B[1] << 1);
157     result ^= B[a&1]; a >>= 1; B[1] = P[B[1]>>7] ^ (B[1] << 1);
158     result ^= B[a&1]; a >>= 1; B[1] = P[B[1]>>7] ^ (B[1] << 1);
159     result ^= B[a&1]; a >>= 1; B[1] = P[B[1]>>7] ^ (B[1] << 1);
160     result ^= B[a&1]; a >>= 1; B[1] = P[B[1]>>7] ^ (B[1] << 1);
161     result ^= B[a&1]; a >>= 1; B[1] = P[B[1]>>7] ^ (B[1] << 1);
162     result ^= B[a&1];
163
164     return result;
165 }

```

### 5.18.3.2 static void h\_func (const unsigned char \* *in*, unsigned char \* *out*, unsigned char \* *M*, int *k*, int *offset*) [static]

Definition at line 247 of file twofish.c.

```

248 {
249     int x;
250     unsigned char y[4];
251     for (x = 0; x < 4; x++) {
252         y[x] = in[x];
253     }
254     switch (k) {
255         case 4:
256             y[0] = (unsigned char) (sbox(1, (ulong32)y[0]) ^ M[4 * (6 + offset) + 0]);
257             y[1] = (unsigned char) (sbox(0, (ulong32)y[1]) ^ M[4 * (6 + offset) + 1]);
258             y[2] = (unsigned char) (sbox(0, (ulong32)y[2]) ^ M[4 * (6 + offset) + 2]);
259             y[3] = (unsigned char) (sbox(1, (ulong32)y[3]) ^ M[4 * (6 + offset) + 3]);
260         case 3:
261             y[0] = (unsigned char) (sbox(1, (ulong32)y[0]) ^ M[4 * (4 + offset) + 0]);
262             y[1] = (unsigned char) (sbox(1, (ulong32)y[1]) ^ M[4 * (4 + offset) + 1]);
263             y[2] = (unsigned char) (sbox(0, (ulong32)y[2]) ^ M[4 * (4 + offset) + 2]);
264             y[3] = (unsigned char) (sbox(0, (ulong32)y[3]) ^ M[4 * (4 + offset) + 3]);
265         case 2:
266             y[0] = (unsigned char) (sbox(1, sbox(0, sbox(0, (ulong32)y[0]) ^ M[4 * (2 + offset) + 0]) ^
267             y[1] = (unsigned char) (sbox(0, sbox(0, sbox(1, (ulong32)y[1]) ^ M[4 * (2 + offset) + 1]) ^
268             y[2] = (unsigned char) (sbox(1, sbox(1, sbox(0, (ulong32)y[2]) ^ M[4 * (2 + offset) + 2]) ^
269             y[3] = (unsigned char) (sbox(0, sbox(1, sbox(1, (ulong32)y[3]) ^ M[4 * (2 + offset) + 3]) ^
270     }
271     mds_mult(y, out);
272 }

```

### 5.18.3.3 static void mds\_mult (const unsigned char \* *in*, unsigned char \* *out*) [static]

Definition at line 210 of file twofish.c.

References mds\_column\_mult.

```

211 {
212     int x;
213     ulong32 tmp;
214     for (tmp = x = 0; x < 4; x++) {

```



```

215         tmp ^= mds_column_mult(in[x], x);
216     }
217     STORE32L(tmp, out);
218 }

```

#### 5.18.3.4 static void rs\_mult (const unsigned char \* *in*, unsigned char \* *out*) [static]

Definition at line 233 of file twofish.c.

References gf\_mult(), RS, and RS\_POLY.

```

234 {
235     int x, y;
236     for (x = 0; x < 4; x++) {
237         out[x] = 0;
238         for (y = 0; y < 8; y++) {
239             out[x] ^= gf_mult(in[y], RS[x][y], RS_POLY);
240         }
241     }
242 }

```

Here is the call graph for this function:

#### 5.18.3.5 void twofish\_done (symmetric\_key \* *skey*)

Terminate the context.

##### Parameters:

*skey* The scheduled key

Definition at line 683 of file twofish.c.

```

684 {
685 }

```

#### 5.18.3.6 int twofish\_ecb\_decrypt (const unsigned char \* *ct*, unsigned char \* *pt*, symmetric\_key \* *skey*)

Decrypts a block of text with Twofish.

##### Parameters:

*ct* The input ciphertext (16 bytes)

*pt* The output plaintext (16 bytes)

*skey* The key as scheduled

##### Returns:

CRYPT\_OK if successful

Definition at line 546 of file twofish.c.

References c, g1\_func, g\_func, LTC\_ARGCHK, ROLc, RORc, S1, S2, S3, S4, t1, and t2.

```

548 {
549     ulong32 a,b,c,d,ta,tb,tc,td,t1,t2, *k;
550     int r;
551     #if !defined(TWOFISH_SMALL) && !defined(__GNUC__)
552         ulong32 *S1, *S2, *S3, *S4;
553     #endif
554
555     LTC_ARGCHK(pt != NULL);
556     LTC_ARGCHK(ct != NULL);
557     LTC_ARGCHK(skey != NULL);
558
559     #if !defined(TWOFISH_SMALL) && !defined(__GNUC__)
560         S1 = skey->twofish.S[0];
561         S2 = skey->twofish.S[1];
562         S3 = skey->twofish.S[2];
563         S4 = skey->twofish.S[3];
564     #endif
565
566     /* load input */
567     LOAD32L(ta,&ct[0]); LOAD32L(tb,&ct[4]);
568     LOAD32L(tc,&ct[8]); LOAD32L(td,&ct[12]);
569
570     /* undo undo final swap */
571     a = tc ^ skey->twofish.K[6];
572     b = td ^ skey->twofish.K[7];
573     c = ta ^ skey->twofish.K[4];
574     d = tb ^ skey->twofish.K[5];
575
576     k = skey->twofish.K + 36;
577     for (r = 8; r != 0; --r) {
578         t2 = g1_func(d, skey);
579         t1 = g_func(c, skey) + t2;
580         a = ROLc(a, 1) ^ (t1 + k[2]);
581         b = RORc(b ^ (t2 + t1 + k[3]), 1);
582
583         t2 = g1_func(b, skey);
584         t1 = g_func(a, skey) + t2;
585         c = ROLc(c, 1) ^ (t1 + k[0]);
586         d = RORc(d ^ (t2 + t1 + k[1]), 1);
587         k -= 4;
588     }
589
590     /* pre-white */
591     a ^= skey->twofish.K[0];
592     b ^= skey->twofish.K[1];
593     c ^= skey->twofish.K[2];
594     d ^= skey->twofish.K[3];
595
596     /* store */
597     STORE32L(a, &pt[0]); STORE32L(b, &pt[4]);
598     STORE32L(c, &pt[8]); STORE32L(d, &pt[12]);
599     return CRYPT_OK;
600 }

```

### 5.18.3.7 `int twofish_ecb_encrypt (const unsigned char * pt, unsigned char * ct, symmetric\_key * skey)`

Encrypts a block of text with Twofish.

#### Parameters:

- pt* The input plaintext (16 bytes)
- ct* The output ciphertext (16 bytes)
- skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 473 of file twofish.c.

References c, g1\_func, g\_func, LTC\_ARGCHK, ROLc, RORc, S1, S2, S3, S4, t1, and t2.

```

475 {
476     ulong32 a,b,c,d,ta,tb,tc,td,t1,t2, *k;
477     int r;
478     #if !defined(TWOFISH_SMALL) && !defined(__GNUC__)
479         ulong32 *S1, *S2, *S3, *S4;
480     #endif
481
482     LTC_ARGCHK(pt != NULL);
483     LTC_ARGCHK(ct != NULL);
484     LTC_ARGCHK(skey != NULL);
485
486     #if !defined(TWOFISH_SMALL) && !defined(__GNUC__)
487         S1 = skey->twofish.S[0];
488         S2 = skey->twofish.S[1];
489         S3 = skey->twofish.S[2];
490         S4 = skey->twofish.S[3];
491     #endif
492
493     LOAD32L(a,&pt[0]); LOAD32L(b,&pt[4]);
494     LOAD32L(c,&pt[8]); LOAD32L(d,&pt[12]);
495     a ^= skey->twofish.K[0];
496     b ^= skey->twofish.K[1];
497     c ^= skey->twofish.K[2];
498     d ^= skey->twofish.K[3];
499
500     k = skey->twofish.K + 8;
501     for (r = 8; r != 0; --r) {
502         t2 = g1_func(b, skey);
503         t1 = g_func(a, skey) + t2;
504         c = RORc(c ^ (t1 + k[0]), 1);
505         d = ROLc(d, 1) ^ (t2 + t1 + k[1]);
506
507         t2 = g1_func(d, skey);
508         t1 = g_func(c, skey) + t2;
509         a = RORc(a ^ (t1 + k[2]), 1);
510         b = ROLc(b, 1) ^ (t2 + t1 + k[3]);
511         k += 4;
512     }
513
514     /* output with "undo last swap" */
515     ta = c ^ skey->twofish.K[4];
516     tb = d ^ skey->twofish.K[5];
517     tc = a ^ skey->twofish.K[6];
518     td = b ^ skey->twofish.K[7];
519
520     /* store output */
521     STORE32L(ta,&ct[0]); STORE32L(tb,&ct[4]);
522     STORE32L(tc,&ct[8]); STORE32L(td,&ct[12]);
523
524     return CRYPT_OK;
525 }

```

**5.18.3.8 int twofish\_keysize (int \*keysize)**

Gets suitable key size.

**Parameters:**

**keysize** [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

**Returns:**

CRYPT\_OK if the input key size is acceptable.

Definition at line 692 of file twofish.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

693 {
694     LTC_ARGCHK(keysize);
695     if (*keysize < 16)
696         return CRYPT_INVALID_KEYSIZE;
697     if (*keysize < 24) {
698         *keysize = 16;
699         return CRYPT_OK;
700     } else if (*keysize < 32) {
701         *keysize = 24;
702         return CRYPT_OK;
703     } else {
704         *keysize = 32;
705         return CRYPT_OK;
706     }
707 }
```

### 5.18.3.9 int twofish\_setup (const unsigned char \*key, int keylen, int num\_rounds, symmetric\_key \*skey)

Initialize the Twofish block cipher.

**Parameters:**

**key** The symmetric key you wish to pass

**keylen** The key length in bytes

**num\_rounds** The number of rounds desired (0 for default)

**skey** The key in as scheduled by this function.

**Returns:**

CRYPT\_OK if successful

Definition at line 346 of file twofish.c.

References B, CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, LTC\_ARGCHK, and S.

Referenced by twofish\_test().

```

348 {
349 #ifndef TWOFISH_SMALL
350     unsigned char S[4*4], tmpx0, tmpx1;
351 #endif
352     int k, x, y;
353     unsigned char tmp[4], tmp2[4], M[8*4];
354     ulong32 A, B;
355
356     LTC_ARGCHK(key != NULL);
357     LTC_ARGCHK(skey != NULL);
```

```

358
359     /* invalid arguments? */
360     if (num_rounds != 16 && num_rounds != 0) {
361         return CRYPT_INVALID_ROUNDS;
362     }
363
364     if (keylen != 16 && keylen != 24 && keylen != 32) {
365         return CRYPT_INVALID_KEYSIZE;
366     }
367
368     /* k = keysize/64 [but since our keysize is in bytes...] */
369     k = keylen / 8;
370
371     /* copy the key into M */
372     for (x = 0; x < keylen; x++) {
373         M[x] = key[x] & 255;
374     }
375
376     /* create the S[...] words */
377 #ifndef TWOFISH_SMALL
378     for (x = 0; x < k; x++) {
379         rs_mult(M+(x*8), S+(x*4));
380     }
381 #else
382     for (x = 0; x < k; x++) {
383         rs_mult(M+(x*8), skey->twofish.S+(x*4));
384     }
385 #endif
386
387     /* make subkeys */
388     for (x = 0; x < 20; x++) {
389         /* A = h(p * 2x, Me) */
390         for (y = 0; y < 4; y++) {
391             tmp[y] = x+x;
392         }
393         h_func(tmp, tmp2, M, k, 0);
394         LOAD32L(A, tmp2);
395
396         /* B = ROL(h(p * (2x + 1), Mo), 8) */
397         for (y = 0; y < 4; y++) {
398             tmp[y] = (unsigned char)(x+x+1);
399         }
400         h_func(tmp, tmp2, M, k, 1);
401         LOAD32L(B, tmp2);
402         B = ROLc(B, 8);
403
404         /* K[2i] = A + B */
405         skey->twofish.K[x+x] = (A + B) & 0xFFFFFFFFFUL;
406
407         /* K[2i+1] = (A + 2B) <<< 9 */
408         skey->twofish.K[x+x+1] = ROLc(B + B + A, 9);
409     }
410
411 #ifndef TWOFISH_SMALL
412     /* make the sboxes (large ram variant) */
413     if (k == 2) {
414         for (x = 0; x < 256; x++) {
415             tmpx0 = sbbox(0, x);
416             tmpx1 = sbbox(1, x);
417             skey->twofish.S[0][x] = mds_column_mult(sbox(1, (sbox(0, tmpx0) ^ S[0]) ^ S[4])),0);
418             skey->twofish.S[1][x] = mds_column_mult(sbox(0, (sbox(0, tmpx1) ^ S[1]) ^ S[5])),1);
419             skey->twofish.S[2][x] = mds_column_mult(sbox(1, (sbox(1, tmpx0) ^ S[2]) ^ S[6])),2);
420             skey->twofish.S[3][x] = mds_column_mult(sbox(0, (sbox(1, tmpx1) ^ S[3]) ^ S[7])),3);
421         }
422     } else if (k == 3) {
423         for (x = 0; x < 256; x++) {
424             tmpx0 = sbbox(0, x);

```

```

425         tmpx1 = sbbox(1, x);
426         skey->twofish.S[0][x] = mds_column_mult(sbbox(1, (sbbox(0, sbbox(0, tmpx1 ^ S[0]) ^ S[4]) ^ S[8]) ^ S[12]),
427         skey->twofish.S[1][x] = mds_column_mult(sbbox(0, (sbbox(0, sbbox(1, tmpx1 ^ S[1]) ^ S[5]) ^ S[9]) ^ S[13]),
428         skey->twofish.S[2][x] = mds_column_mult(sbbox(1, (sbbox(1, sbbox(0, tmpx0 ^ S[2]) ^ S[6]) ^ S[10]) ^ S[14]),
429         skey->twofish.S[3][x] = mds_column_mult(sbbox(0, (sbbox(1, sbbox(1, tmpx0 ^ S[3]) ^ S[7]) ^ S[11]) ^ S[15]);
430     }
431 } else {
432     for (x = 0; x < 256; x++) {
433         tmpx0 = sbbox(0, x);
434         tmpx1 = sbbox(1, x);
435         skey->twofish.S[0][x] = mds_column_mult(sbbox(1, (sbbox(0, sbbox(0, sbbox(1, tmpx1 ^ S[0]) ^ S[4]) ^ S[8]) ^ S[12]),
436         skey->twofish.S[1][x] = mds_column_mult(sbbox(0, (sbbox(0, sbbox(1, sbbox(1, tmpx0 ^ S[1]) ^ S[5]) ^ S[9]) ^ S[13]),
437         skey->twofish.S[2][x] = mds_column_mult(sbbox(1, (sbbox(1, sbbox(0, sbbox(0, tmpx0 ^ S[2]) ^ S[6]) ^ S[10]) ^ S[14]),
438         skey->twofish.S[3][x] = mds_column_mult(sbbox(0, (sbbox(1, sbbox(1, sbbox(0, tmpx1 ^ S[3]) ^ S[7]) ^ S[11]) ^ S[15]);
439     }
440 }
441 #else
442 /* where to start in the sbbox layers */
443 /* small ram variant */
444 switch (k) {
445     case 4 : skey->twofish.start = 0; break;
446     case 3 : skey->twofish.start = 1; break;
447     default: skey->twofish.start = 2; break;
448 }
449 #endif
450 return CRYPT_OK;
451 }

```

### 5.18.3.10 int twofish\_test (void)

Performs a self-test of the Twofish block cipher.

#### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 615 of file twofish.c.

References CRYPT\_NOP, CRYPT\_OK, and twofish\_setup().

```

616 {
617     #ifndef LTC_TEST
618         return CRYPT_NOP;
619     #else
620     static const struct {
621         int keylen;
622         unsigned char key[32], pt[16], ct[16];
623     } tests[] = {
624         { 16,
625         { 0x9F, 0x58, 0x9F, 0x5C, 0xF6, 0x12, 0x2C, 0x32,
626         { 0xB6, 0xBF, 0xEC, 0x2F, 0x2A, 0xE8, 0xC3, 0x5A },
627         { 0xD4, 0x91, 0xDB, 0x16, 0xE7, 0xB1, 0xC3, 0x9E },
628         { 0x86, 0xCB, 0x08, 0x6B, 0x78, 0x9F, 0x54, 0x19 },
629         { 0x01, 0x9F, 0x98, 0x09, 0xDE, 0x17, 0x11, 0x85,
630         { 0x8F, 0xAA, 0xC3, 0xA3, 0xBA, 0x20, 0xFB, 0xC3 }
631         }, {
632         24,
633         { 0x88, 0xB2, 0xB2, 0x70, 0x6B, 0x10, 0x5E, 0x36,
634         { 0xB4, 0x46, 0xBB, 0x6D, 0x73, 0x1A, 0x1E, 0x88,
635         { 0xEF, 0xA7, 0x1F, 0x78, 0x89, 0x65, 0xBD, 0x44 },
636         { 0x39, 0xDA, 0x69, 0xD6, 0xBA, 0x49, 0x97, 0xD5,
637         { 0x85, 0xB6, 0xDC, 0x07, 0x3C, 0xA3, 0x41, 0xB2 },
638         { 0x18, 0x2B, 0x02, 0xD8, 0x14, 0x97, 0xEA, 0x45,
639         { 0xF9, 0xDA, 0xAC, 0xDC, 0x29, 0x19, 0x3A, 0x65 }

```

```

640     }, {
641         32,
642         { 0xD4, 0x3B, 0xB7, 0x55, 0x6E, 0xA3, 0x2E, 0x46,
643           0xF2, 0xA2, 0x82, 0xB7, 0xD4, 0x5B, 0x4E, 0x0D,
644           0x57, 0xFF, 0x73, 0x9D, 0x4D, 0xC9, 0x2C, 0x1B,
645           0xD7, 0xFC, 0x01, 0x70, 0x0C, 0xC8, 0x21, 0x6F },
646         { 0x90, 0xAF, 0xE9, 0x1B, 0xB2, 0x88, 0x54, 0x4F,
647           0x2C, 0x32, 0xDC, 0x23, 0x9B, 0x26, 0x35, 0xE6 },
648         { 0x6C, 0xB4, 0x56, 0x1C, 0x40, 0xBF, 0x0A, 0x97,
649           0x05, 0x93, 0x1C, 0xB6, 0xD4, 0x08, 0xE7, 0xFA }
650     }
651 };
652
653
654 symmetric_key key;
655 unsigned char tmp[2][16];
656 int err, i, y;
657
658 for (i = 0; i < (int)(sizeof(tests)/sizeof(tests[0])); i++) {
659     if ((err = twofish_setup(tests[i].key, tests[i].keylen, 0, &key)) != CRYPT_OK) {
660         return err;
661     }
662     twofish_ecb_encrypt(tests[i].pt, tmp[0], &key);
663     twofish_ecb_decrypt(tmp[0], tmp[1], &key);
664     if (XMEMCMP(tmp[0], tests[i].ct, 16) != 0 || XMEMCMP(tmp[1], tests[i].pt, 16) != 0) {
665 #if 0
666         printf("Twofish failed test %d, %d, %d\n", i, XMEMCMP(tmp[0], tests[i].ct, 16), XMEMCMP(tmp[1],
667 #endif
668         return CRYPT_FAIL_TESTVECTOR;
669     }
670     /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
671     for (y = 0; y < 16; y++) tmp[0][y] = 0;
672     for (y = 0; y < 1000; y++) twofish_ecb_encrypt(tmp[0], tmp[0], &key);
673     for (y = 0; y < 1000; y++) twofish_ecb_decrypt(tmp[0], tmp[0], &key);
674     for (y = 0; y < 16; y++) if (tmp[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
675 }
676 return CRYPT_OK;
677 #endif
678 }

```

Here is the call graph for this function:

## 5.18.4 Variable Documentation

### 5.18.4.1 const unsigned char [MDS](#)[4][4] [static]

**Initial value:**

```

{
    { 0x01, 0xEF, 0x5B, 0x5B },
    { 0x5B, 0xEF, 0xEF, 0x01 },
    { 0xEF, 0x5B, 0x01, 0xEF },
    { 0xEF, 0x01, 0xEF, 0x5B }
}

```

Definition at line 46 of file twofish.c.

### 5.18.4.2 const unsigned char [qord](#)[4][5] [static]

**Initial value:**

```
{
  { 1, 1, 0, 0, 1 },
  { 0, 1, 1, 0, 0 },
  { 0, 0, 0, 1, 1 },
  { 1, 0, 1, 1, 0 }
}
```

Definition at line 62 of file twofish.c.

#### 5.18.4.3 `const unsigned char RS[4][8]` `[static]`

**Initial value:**

```
{
  { 0x01, 0xA4, 0x55, 0x87, 0x5A, 0x58, 0xDB, 0x9E },
  { 0xA4, 0x56, 0x82, 0xF3, 0x1E, 0xC6, 0x68, 0xE5 },
  { 0x02, 0xA1, 0xFC, 0xC1, 0x47, 0xAE, 0x3D, 0x19 },
  { 0xA4, 0x55, 0x87, 0x5A, 0x58, 0xDB, 0x9E, 0x03 }
}
```

Definition at line 54 of file twofish.c.

Referenced by `rs_mult()`.

#### 5.18.4.4 `const struct ltc_cipher_descriptor twofish_desc`

**Initial value:**

```
{
  "twofish",
  7,
  16, 32, 16, 16,
  &twofish_setup,
  &twofish_ecb_encrypt,
  &twofish_ecb_decrypt,
  &twofish_test,
  &twofish_done,
  &twofish_keysize,
  NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}
```

Definition at line 27 of file twofish.c.

Referenced by `yarrow_start()`.



## 5.19 ciphers/twofish/twofish\_tab.c File Reference

### 5.19.1 Detailed Description

Twofish tables, Tom St Denis.

Definition in file [twofish\\_tab.c](#).

This graph shows which files directly or indirectly include this file:

## 5.20 ciphers/xtea.c File Reference

### 5.20.1 Detailed Description

Implementation of XTEA, Tom St Denis.

Definition in file [xtea.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for xtea.c:

### Functions

- int [xtea\\_setup](#) (const unsigned char \*key, int keylen, int num\_rounds, [symmetric\\_key](#) \*skey)
- int [xtea\\_ecb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, [symmetric\\_key](#) \*skey)  
*Encrypts a block of text with XTEA.*
- int [xtea\\_ecb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, [symmetric\\_key](#) \*skey)  
*Decrypts a block of text with XTEA.*
- int [xtea\\_test](#) (void)  
*Performs a self-test of the XTEA block cipher.*
- void [xtea\\_done](#) ([symmetric\\_key](#) \*skey)  
*Terminate the context.*
- int [xtea\\_keysize](#) (int \*keysize)  
*Gets suitable key size.*

### Variables

- const struct [ltc\\_cipher\\_descriptor](#) [xtea\\_desc](#)

### 5.20.2 Function Documentation

#### 5.20.2.1 void [xtea\\_done](#) ([symmetric\\_key](#) \* *skey*)

Terminate the context.

#### Parameters:

*skey* The scheduled key

Definition at line 184 of file xtea.c.

```
185 {  
186 }
```

**5.20.2.2 int xtea\_ecb\_decrypt (const unsigned char \* *ct*, unsigned char \* *pt*, [symmetric\\_key](#) \* *skey*)**

Decrypts a block of text with XTEA.

**Parameters:**

- ct* The input ciphertext (8 bytes)
- pt* The output plaintext (8 bytes)
- skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 112 of file xtea.c.

References LTC\_ARGCHK.

Referenced by xtea\_test().

```

113 {
114     unsigned long y, z;
115     int r;
116
117     LTC_ARGCHK(pt != NULL);
118     LTC_ARGCHK(ct != NULL);
119     LTC_ARGCHK(skey != NULL);
120
121     LOAD32L(y, &ct[0]);
122     LOAD32L(z, &ct[4]);
123     for (r = 31; r >= 0; r -= 4) {
124         z = (z - (((y<<4)^(y>>5)) + y) ^ skey->xtea.B[r])) & 0xFFFFFFFFFUL;
125         y = (y - (((z<<4)^(z>>5)) + z) ^ skey->xtea.A[r])) & 0xFFFFFFFFFUL;
126
127         z = (z - (((y<<4)^(y>>5)) + y) ^ skey->xtea.B[r-1])) & 0xFFFFFFFFFUL;
128         y = (y - (((z<<4)^(z>>5)) + z) ^ skey->xtea.A[r-1])) & 0xFFFFFFFFFUL;
129
130         z = (z - (((y<<4)^(y>>5)) + y) ^ skey->xtea.B[r-2])) & 0xFFFFFFFFFUL;
131         y = (y - (((z<<4)^(z>>5)) + z) ^ skey->xtea.A[r-2])) & 0xFFFFFFFFFUL;
132
133         z = (z - (((y<<4)^(y>>5)) + y) ^ skey->xtea.B[r-3])) & 0xFFFFFFFFFUL;
134         y = (y - (((z<<4)^(z>>5)) + z) ^ skey->xtea.A[r-3])) & 0xFFFFFFFFFUL;
135     }
136     STORE32L(y, &pt[0]);
137     STORE32L(z, &pt[4]);
138     return CRYPT_OK;
139 }
```

**5.20.2.3 int xtea\_ecb\_encrypt (const unsigned char \* *pt*, unsigned char \* *ct*, [symmetric\\_key](#) \* *skey*)**

Encrypts a block of text with XTEA.

**Parameters:**

- pt* The input plaintext (8 bytes)
- ct* The output ciphertext (8 bytes)
- skey* The key as scheduled

**Returns:**

CRYPT\_OK if successful

Definition at line 76 of file xtea.c.

References LTC\_ARGCHK.

Referenced by xtea\_test().

```

77 {
78     unsigned long y, z;
79     int r;
80
81     LTC_ARGCHK(pt != NULL);
82     LTC_ARGCHK(ct != NULL);
83     LTC_ARGCHK(skey != NULL);
84
85     LOAD32L(y, &pt[0]);
86     LOAD32L(z, &pt[4]);
87     for (r = 0; r < 32; r += 4) {
88         y = (y + (((z<<4)^(z>>5)) + z) ^ skey->xtea.A[r])) & 0xFFFFFFFFFUL;
89         z = (z + (((y<<4)^(y>>5)) + y) ^ skey->xtea.B[r])) & 0xFFFFFFFFFUL;
90
91         y = (y + (((z<<4)^(z>>5)) + z) ^ skey->xtea.A[r+1])) & 0xFFFFFFFFFUL;
92         z = (z + (((y<<4)^(y>>5)) + y) ^ skey->xtea.B[r+1])) & 0xFFFFFFFFFUL;
93
94         y = (y + (((z<<4)^(z>>5)) + z) ^ skey->xtea.A[r+2])) & 0xFFFFFFFFFUL;
95         z = (z + (((y<<4)^(y>>5)) + y) ^ skey->xtea.B[r+2])) & 0xFFFFFFFFFUL;
96
97         y = (y + (((z<<4)^(z>>5)) + z) ^ skey->xtea.A[r+3])) & 0xFFFFFFFFFUL;
98         z = (z + (((y<<4)^(y>>5)) + y) ^ skey->xtea.B[r+3])) & 0xFFFFFFFFFUL;
99     }
100     STORE32L(y, &ct[0]);
101     STORE32L(z, &ct[4]);
102     return CRYPT_OK;
103 }

```

#### 5.20.2.4 int xtea\_keysize (int \* keysize)

Gets suitable key size.

##### Parameters:

**keysize** [in/out] The length of the recommended key (in bytes). This function will store the suitable size back in this variable.

##### Returns:

CRYPT\_OK if the input key size is acceptable.

Definition at line 193 of file xtea.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_OK, and LTC\_ARGCHK.

```

194 {
195     LTC_ARGCHK(keysize != NULL);
196     if (*keysize < 16) {
197         return CRYPT_INVALID_KEYSIZE;
198     }
199     *keysize = 16;
200     return CRYPT_OK;
201 }

```

### 5.20.2.5 int xtea\_setup (const unsigned char \*key, int keylen, int num\_rounds, symmetric\_key \*skey)

Definition at line 34 of file xtea.c.

References CRYPT\_INVALID\_KEYSIZE, CRYPT\_INVALID\_ROUNDS, K, and LTC\_ARGCHK.

Referenced by xtea\_test().

```

35 {
36     unsigned long x, sum, K[4];
37
38     LTC_ARGCHK(key != NULL);
39     LTC_ARGCHK(skey != NULL);
40
41     /* check arguments */
42     if (keylen != 16) {
43         return CRYPT_INVALID_KEYSIZE;
44     }
45
46     if (num_rounds != 0 && num_rounds != 32) {
47         return CRYPT_INVALID_ROUNDS;
48     }
49
50     /* load key */
51     LOAD32L(K[0], key+0);
52     LOAD32L(K[1], key+4);
53     LOAD32L(K[2], key+8);
54     LOAD32L(K[3], key+12);
55
56     for (x = sum = 0; x < 32; x++) {
57         skey->xtea.A[x] = (sum + K[sum&3]) & 0xFFFFFFFFFUL;
58         sum = (sum + 0x9E3779B9UL) & 0xFFFFFFFFFUL;
59         skey->xtea.B[x] = (sum + K[(sum>>11)&3]) & 0xFFFFFFFFFUL;
60     }
61
62     #ifdef LTC_CLEAN_STACK
63         zeromem(&K, sizeof(K));
64     #endif
65
66     return CRYPT_OK;
67 }
```

### 5.20.2.6 int xtea\_test (void)

Performs a self-test of the XTEA block cipher.

#### Returns:

CRYPT\_OK if functional, CRYPT\_NOP if self-test has been disabled

Definition at line 145 of file xtea.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, CRYPT\_OK, XMEMCMP, xtea\_ecb\_decrypt(), xtea\_ecb\_encrypt(), and xtea\_setup().

```

146 {
147     #ifndef LTC_TEST
148         return CRYPT_NOP;
149     #else
150         static const unsigned char key[16] =
151             { 0x78, 0x56, 0x34, 0x12, 0xf0, 0xcd, 0xcb, 0x9a,
```

```

152         0x48, 0x37, 0x26, 0x15, 0xc0, 0xbf, 0xae, 0x9d );
153     static const unsigned char pt[8] =
154     { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08 };
155     static const unsigned char ct[8] =
156     { 0x75, 0xd7, 0xc5, 0xbf, 0xcf, 0x58, 0xc9, 0x3f };
157     unsigned char tmp[2][8];
158     symmetric_key skey;
159     int err, y;
160
161     if ((err = xtea_setup(key, 16, 0, &skey)) != CRYPT_OK) {
162         return err;
163     }
164     xtea_ecb_encrypt(pt, tmp[0], &skey);
165     xtea_ecb_decrypt(tmp[0], tmp[1], &skey);
166
167     if (XMEMCMP(tmp[0], ct, 8) != 0 || XMEMCMP(tmp[1], pt, 8) != 0) {
168         return CRYPT_FAIL_TESTVECTOR;
169     }
170
171     /* now see if we can encrypt all zero bytes 1000 times, decrypt and come back where we started */
172     for (y = 0; y < 8; y++) tmp[0][y] = 0;
173     for (y = 0; y < 1000; y++) xtea_ecb_encrypt(tmp[0], tmp[0], &skey);
174     for (y = 0; y < 1000; y++) xtea_ecb_decrypt(tmp[0], tmp[0], &skey);
175     for (y = 0; y < 8; y++) if (tmp[0][y] != 0) return CRYPT_FAIL_TESTVECTOR;
176
177     return CRYPT_OK;
178 #endif
179 }

```

Here is the call graph for this function:

## 5.20.3 Variable Documentation

### 5.20.3.1 const struct [ltc\\_cipher\\_descriptor](#) [xtea\\_desc](#)

**Initial value:**

```

{
    "xtea",
    1,
    16, 16, 8, 32,
    &xtea_setup,
    &xtea_ecb_encrypt,
    &xtea_ecb_decrypt,
    &xtea_test,
    &xtea_done,
    &xtea_keysize,
    NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
}

```

Definition at line 20 of file xtea.c.

Referenced by [yarrow\\_start\(\)](#).

## 5.21 encauth/ccm/ccm\_memory.c File Reference

### 5.21.1 Detailed Description

CCM support, process a block of memory, Tom St Denis.

Definition in file [ccm\\_memory.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ccm\_memory.c:

### Functions

- [int ccm\\_memory](#) (int cipher, const unsigned char \*key, unsigned long keylen, [symmetric\\_key](#) \*uskey, const unsigned char \*nonce, unsigned long noncelen, const unsigned char \*header, unsigned long headerlen, unsigned char \*pt, unsigned long ptlen, unsigned char \*ct, unsigned char \*tag, unsigned long \*taglen, int direction)

*CCM encrypt/decrypt and produce an authentication tag.*

### 5.21.2 Function Documentation

- 5.21.2.1** [int ccm\\_memory](#) (int *cipher*, const unsigned char \* *key*, unsigned long *keylen*, [symmetric\\_key](#) \* *uskey*, const unsigned char \* *nonce*, unsigned long *noncelen*, const unsigned char \* *header*, unsigned long *headerlen*, unsigned char \* *pt*, unsigned long *ptlen*, unsigned char \* *ct*, unsigned char \* *tag*, unsigned long \* *taglen*, int *direction*)

CCM encrypt/decrypt and produce an authentication tag.

#### Parameters:

- cipher* The index of the cipher desired
- key* The secret key to use
- keylen* The length of the secret key (octets)
- uskey* A previously scheduled key [optional can be NULL]
- nonce* The session nonce [use once]
- noncelen* The length of the nonce
- header* The header for the session
- headerlen* The length of the header (octets)
- pt* [out] The plaintext
- ptlen* The length of the plaintext (octets)
- ct* [out] The ciphertext
- tag* [out] The destination tag
- taglen* [in/out] The max size and resulting size of the authentication tag
- direction* Encrypt or Decrypt direction (0 or 1)

#### Returns:

CRYPT\_OK if successful

Definition at line 38 of file ccm\_memory.c.

References `ltc_cipher_descriptor::accel_ccm_memory`, `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_INVALID_CIPHER`, `CRYPT_MEM`, `CRYPT_OK`, `len`, `LTC_ARGCHK`, `ltc_cipher_descriptor::setup`, `XFREE`, and `XMALLOC`.

Referenced by `ccm_test()`.

```

47 {
48     unsigned char  PAD[16], ctr[16], CTRPAD[16], b;
49     symmetric_key *skey;
50     int            err;
51     unsigned long  len, L, x, y, z, CTRlen;
52
53     if (uskey == NULL) {
54         LTC_ARGCHK(key      != NULL);
55     }
56     LTC_ARGCHK(nonce      != NULL);
57     if (headerlen > 0) {
58         LTC_ARGCHK(header  != NULL);
59     }
60     LTC_ARGCHK(pt         != NULL);
61     LTC_ARGCHK(ct         != NULL);
62     LTC_ARGCHK(tag        != NULL);
63     LTC_ARGCHK(taglen     != NULL);
64
65 #ifdef LTC_FAST
66     if (16 % sizeof(LTC_FAST_TYPE)) {
67         return CRYPT_INVALID_ARG;
68     }
69 #endif
70
71     /* check cipher input */
72     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
73         return err;
74     }
75     if (cipher_descriptor[cipher].block_length != 16) {
76         return CRYPT_INVALID_CIPHER;
77     }
78
79     /* make sure the taglen is even and <= 16 */
80     *taglen &= ~1;
81     if (*taglen > 16) {
82         *taglen = 16;
83     }
84
85     /* can't use < 4 */
86     if (*taglen < 4) {
87         return CRYPT_INVALID_ARG;
88     }
89
90     /* is there an accelerator? */
91     if (cipher_descriptor[cipher].accel_ccm_memory != NULL) {
92         return cipher_descriptor[cipher].accel_ccm_memory(
93             key,      keylen,
94             uskey,
95             nonce,    noncelen,
96             header,   headerlen,
97             pt,       ptlen,
98             ct,
99             tag,      taglen,
100             direction);
101     }
102
103     /* let's get the L value */
104     len = ptlen;
105     L   = 0;

```



```

106     while (len) {
107         ++L;
108         len >>= 8;
109     }
110     if (L <= 1) {
111         L = 2;
112     }
113
114     /* increase L to match the nonce len */
115     noncelen = (noncelen > 13) ? 13 : noncelen;
116     if ((15 - noncelen) > L) {
117         L = 15 - noncelen;
118     }
119
120     /* decrease noncelen to match L */
121     if ((noncelen + L) > 15) {
122         noncelen = 15 - L;
123     }
124
125     /* allocate mem for the symmetric key */
126     if (uskey == NULL) {
127         skey = XMALLOC(sizeof(*skey));
128         if (skey == NULL) {
129             return CRYPT_MEM;
130         }
131
132         /* initialize the cipher */
133         if ((err = cipher_descriptor[cipher].setup(key, keylen, 0, skey)) != CRYPT_OK) {
134             XFREE(skey);
135             return err;
136         }
137     } else {
138         skey = uskey;
139     }
140
141     /* form B_0 == flags | Nonce N | l(m) */
142     x = 0;
143     PAD[x++] = ((headerlen > 0) ? (1<<6) : 0) |
144               (((*taglen - 2)>>1)<<3) |
145               (L-1);
146
147     /* nonce */
148     for (y = 0; y < (16 - (L + 1)); y++) {
149         PAD[x++] = nonce[y];
150     }
151
152     /* store len */
153     len = pten;
154
155     /* shift len so the upper bytes of len are the contents of the length */
156     for (y = L; y < 4; y++) {
157         len <<= 8;
158     }
159
160     /* store l(m) (only store 32-bits) */
161     for (y = 0; L > 4 && (L-y)>4; y++) {
162         PAD[x++] = 0;
163     }
164     for (; y < L; y++) {
165         PAD[x++] = (len >> 24) & 255;
166         len <<= 8;
167     }
168
169     /* encrypt PAD */
170     if ((err = cipher_descriptor[cipher].ecb_encrypt(PAD, PAD, skey)) != CRYPT_OK) {
171         goto error;
172     }

```

```

173
174     /* handle header */
175     if (headerlen > 0) {
176         x = 0;
177
178         /* store length */
179         if (headerlen < ((1UL<<16) - (1UL<<8))) {
180             PAD[x++] ^= (headerlen>>8) & 255;
181             PAD[x++] ^= headerlen & 255;
182         } else {
183             PAD[x++] ^= 0xFF;
184             PAD[x++] ^= 0xFE;
185             PAD[x++] ^= (headerlen>>24) & 255;
186             PAD[x++] ^= (headerlen>>16) & 255;
187             PAD[x++] ^= (headerlen>>8) & 255;
188             PAD[x++] ^= headerlen & 255;
189         }
190
191         /* now add the data */
192         for (y = 0; y < headerlen; y++) {
193             if (x == 16) {
194                 /* full block so let's encrypt it */
195                 if ((err = cipher_descriptor[cipher].ecb_encrypt(PAD, PAD, skey)) != CRYPT_OK) {
196                     goto error;
197                 }
198                 x = 0;
199             }
200             PAD[x++] ^= header[y];
201         }
202
203         /* remainder? */
204         if (x != 0) {
205             if ((err = cipher_descriptor[cipher].ecb_encrypt(PAD, PAD, skey)) != CRYPT_OK) {
206                 goto error;
207             }
208         }
209     }
210
211     /* setup the ctr counter */
212     x = 0;
213
214     /* flags */
215     ctr[x++] = L-1;
216
217     /* nonce */
218     for (y = 0; y < (16 - (L+1)); ++y) {
219         ctr[x++] = nonce[y];
220     }
221     /* offset */
222     while (x < 16) {
223         ctr[x++] = 0;
224     }
225
226     x = 0;
227     CTRlen = 16;
228
229     /* now handle the PT */
230     if (ptlen > 0) {
231         y = 0;
232 #ifndef LTC_FAST
233         if (ptlen & ~15) {
234             if (direction == CCM_ENCRYPT) {
235                 for (; y < (ptlen & ~15); y += 16) {
236                     /* increment the ctr? */
237                     for (z = 15; z > 15-L; z--) {
238                         ctr[z] = (ctr[z] + 1) & 255;
239                         if (ctr[z]) break;

```

```

240         }
241         if ((err = cipher_descriptor[cipher].ecb_encrypt(ctr, CTRPAD, skey)) != CRYPT_OK) {
242             goto error;
243         }
244
245         /* xor the PT against the pad first */
246         for (z = 0; z < 16; z += sizeof(LTC_FAST_TYPE)) {
247             *((LTC_FAST_TYPE*) (&PAD[z])) ^= *((LTC_FAST_TYPE*) (&pt[y+z]));
248             *((LTC_FAST_TYPE*) (&ct[y+z])) = *((LTC_FAST_TYPE*) (&pt[y+z])) ^ *((LTC_FAST_TYPE*)
249             )
250             if ((err = cipher_descriptor[cipher].ecb_encrypt(PAD, PAD, skey)) != CRYPT_OK) {
251                 goto error;
252             }
253         }
254     } else {
255         for (; y < (ptlen & ~15); y += 16) {
256             /* increment the ctr? */
257             for (z = 15; z > 15-L; z--) {
258                 ctr[z] = (ctr[z] + 1) & 255;
259                 if (ctr[z]) break;
260             }
261             if ((err = cipher_descriptor[cipher].ecb_encrypt(ctr, CTRPAD, skey)) != CRYPT_OK) {
262                 goto error;
263             }
264
265             /* xor the PT against the pad last */
266             for (z = 0; z < 16; z += sizeof(LTC_FAST_TYPE)) {
267                 *((LTC_FAST_TYPE*) (&pt[y+z])) = *((LTC_FAST_TYPE*) (&ct[y+z])) ^ *((LTC_FAST_TYPE*)
268                 )
269                 *((LTC_FAST_TYPE*) (&PAD[z])) ^= *((LTC_FAST_TYPE*) (&pt[y+z]));
270             }
271             if ((err = cipher_descriptor[cipher].ecb_encrypt(PAD, PAD, skey)) != CRYPT_OK) {
272                 goto error;
273             }
274         }
275     }
276 #endif
277
278     for (; y < ptlen; y++) {
279         /* increment the ctr? */
280         if (CTRlen == 16) {
281             for (z = 15; z > 15-L; z--) {
282                 ctr[z] = (ctr[z] + 1) & 255;
283                 if (ctr[z]) break;
284             }
285             if ((err = cipher_descriptor[cipher].ecb_encrypt(ctr, CTRPAD, skey)) != CRYPT_OK) {
286                 goto error;
287             }
288             CTRlen = 0;
289         }
290
291         /* if we encrypt we add the bytes to the MAC first */
292         if (direction == CCM_ENCRYPT) {
293             b = pt[y];
294             ct[y] = b ^ CTRPAD[CTRlen++];
295         } else {
296             b = ct[y] ^ CTRPAD[CTRlen++];
297             pt[y] = b;
298         }
299
300         if (x == 16) {
301             if ((err = cipher_descriptor[cipher].ecb_encrypt(PAD, PAD, skey)) != CRYPT_OK) {
302                 goto error;
303             }
304             x = 0;
305         }
306         PAD[x++] ^= b;

```

```
307     }
308
309     if (x != 0) {
310         if ((err = cipher_descriptor[cipher].ecb_encrypt(PAD, PAD, skey)) != CRYPT_OK) {
311             goto error;
312         }
313     }
314 }
315
316 /* setup CTR for the TAG (zero the count) */
317 for (y = 15; y > 15 - L; y--) {
318     ctr[y] = 0x00;
319 }
320 if ((err = cipher_descriptor[cipher].ecb_encrypt(ctr, CTRPAD, skey)) != CRYPT_OK) {
321     goto error;
322 }
323
324 if (skey != uskey) {
325     cipher_descriptor[cipher].done(skey);
326 }
327
328 /* store the TAG */
329 for (x = 0; x < 16 && x < *taglen; x++) {
330     tag[x] = PAD[x] ^ CTRPAD[x];
331 }
332 *taglen = x;
333
334 #ifdef LTC_CLEAN_STACK
335     zeromem(skey,    sizeof(*skey));
336     zeromem(PAD,     sizeof(PAD));
337     zeromem(CTRPAD, sizeof(CTRPAD));
338 #endif
339 error:
340     if (skey != uskey) {
341         XFREE(skey);
342     }
343
344     return err;
345 }
```

Here is the call graph for this function:

## 5.22 encauth/ccm/ccm\_test.c File Reference

### 5.22.1 Detailed Description

CCM support, process a block of memory, Tom St Denis.

Definition in file [ccm\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ccm\_test.c:

### Functions

- [int ccm\\_test](#) (void)

### 5.22.2 Function Documentation

#### 5.22.2.1 int ccm\_test (void)

Definition at line 20 of file ccm\_test.c.

References [ccm\\_memory\(\)](#), [cipher\\_descriptor](#), [CRYPT\\_FAIL\\_TESTVECTOR](#), [CRYPT\\_NOP](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::done](#), [find\\_cipher\(\)](#), and [XMEMCMP](#).

```
21 {
22 #ifndef LTC_TEST
23     return CRYPT_NOP;
24 #else
25     static const struct {
26         unsigned char key[16];
27         unsigned char nonce[16];
28         int          noncelen;
29         unsigned char header[64];
30         int          headerlen;
31         unsigned char pt[64];
32         int          ptlen;
33         unsigned char ct[64];
34         unsigned char tag[16];
35         int          taglen;
36     } tests[] = {
37
38 /* 13 byte nonce, 8 byte auth, 23 byte pt */
39 {
40     { 0xC0, 0xC1, 0xC2, 0xC3, 0xC4, 0xC5, 0xC6, 0xC7,
41       0xC8, 0xC9, 0xCA, 0xCB, 0xCC, 0xCD, 0xCE, 0xCF },
42     { 0x00, 0x00, 0x00, 0x03, 0x02, 0x01, 0x00, 0xA0,
43       0xA1, 0xA2, 0xA3, 0xA4, 0xA5 },
44     13,
45     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 },
46     8,
47     { 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
48       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
49       0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E },
50     23,
51     { 0x58, 0x8C, 0x97, 0x9A, 0x61, 0xC6, 0x63, 0xD2,
52       0xF0, 0x66, 0xD0, 0xC2, 0xC0, 0xF9, 0x89, 0x80,
53       0x6D, 0x5F, 0x6B, 0x61, 0xDA, 0xC3, 0x84 },
54     { 0x17, 0xe8, 0xd1, 0x2c, 0xfd, 0xf9, 0x26, 0xe0 },
55     8
56 },
```

```

57
58 /* 13 byte nonce, 12 byte header, 19 byte pt */
59 {
60     { 0xC0, 0xC1, 0xC2, 0xC3, 0xC4, 0xC5, 0xC6, 0xC7,
61       0xC8, 0xC9, 0xCA, 0xCB, 0xCC, 0xCD, 0xCE, 0xCF },
62     { 0x00, 0x00, 0x00, 0x06, 0x05, 0x04, 0x03, 0xA0,
63       0xA1, 0xA2, 0xA3, 0xA4, 0xA5 },
64     13,
65     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
66       0x08, 0x09, 0x0A, 0x0B },
67     12,
68     { 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13,
69       0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1A, 0x1B,
70       0x1C, 0x1D, 0x1E },
71     19,
72     { 0xA2, 0x8C, 0x68, 0x65, 0x93, 0x9A, 0x9A, 0x79,
73       0xFA, 0xAA, 0x5C, 0x4C, 0x2A, 0x9D, 0x4A, 0x91,
74       0xCD, 0xAC, 0x8C },
75     { 0x96, 0xC8, 0x61, 0xB9, 0xC9, 0xE6, 0x1E, 0xF1 },
76     8
77 },
78
79 /* supplied by Brian Gladman */
80 {
81     { 0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
82       0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f },
83     { 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16 },
84     7,
85     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 },
86     8,
87     { 0x20, 0x21, 0x22, 0x23 },
88     4,
89     { 0x71, 0x62, 0x01, 0x5b },
90     { 0x4d, 0xac, 0x25, 0x5d },
91     4
92 },
93
94 {
95     { 0xc9, 0x7c, 0x1f, 0x67, 0xce, 0x37, 0x11, 0x85,
96       0x51, 0x4a, 0x8a, 0x19, 0xf2, 0xbd, 0xd5, 0x2f },
97     { 0x00, 0x50, 0x30, 0xf1, 0x84, 0x44, 0x08, 0xb5,
98       0x03, 0x97, 0x76, 0xe7, 0x0c },
99     13,
100    { 0x08, 0x40, 0x0f, 0xd2, 0xe1, 0x28, 0xa5, 0x7c,
101      0x50, 0x30, 0xf1, 0x84, 0x44, 0x08, 0xab, 0xae,
102      0xa5, 0xb8, 0xfc, 0xba, 0x00, 0x00 },
103    22,
104    { 0xf8, 0xba, 0x1a, 0x55, 0xd0, 0x2f, 0x85, 0xae,
105      0x96, 0x7b, 0xb6, 0x2f, 0xb6, 0xcd, 0xa8, 0xeb,
106      0x7e, 0x78, 0xa0, 0x50 },
107    20,
108    { 0xf3, 0xd0, 0xa2, 0xfe, 0x9a, 0x3d, 0xbf, 0x23,
109      0x42, 0xa6, 0x43, 0xe4, 0x32, 0x46, 0xe8, 0x0c,
110      0x3c, 0x04, 0xd0, 0x19 },
111    { 0x78, 0x45, 0xce, 0x0b, 0x16, 0xf9, 0x76, 0x23 },
112    8
113 },
114
115 };
116 unsigned long taglen, x;
117 unsigned char buf[64], buf2[64], tag2[16], tag[16];
118 int err, idx;
119 symmetric_key skey;
120
121 idx = find_cipher("aes");
122 if (idx == -1) {
123     idx = find_cipher("rijndael");

```

```

124     if (idx == -1) {
125         return CRYPT_NOP;
126     }
127 }
128
129 for (x = 0; x < (sizeof(tests)/sizeof(tests[0])); x++) {
130     taglen = tests[x].taglen;
131     if ((err = cipher_descriptor[idx].setup(tests[x].key, 16, 0, &skey)) != CRYPT_OK) {
132         return err;
133     }
134
135     if ((err = ccm_memory(idx,
136                         tests[x].key, 16,
137                         &skey,
138                         tests[x].nonce, tests[x].noncelen,
139                         tests[x].header, tests[x].headerlen,
140                         (unsigned char*)tests[x].pt, tests[x].ptlen,
141                         buf,
142                         tag, &taglen, 0)) != CRYPT_OK) {
143         return err;
144     }
145
146     if (XMEMCMP(buf, tests[x].ct, tests[x].ptlen)) {
147         return CRYPT_FAIL_TESTVECTOR;
148     }
149     if (XMEMCMP(tag, tests[x].tag, tests[x].taglen)) {
150         return CRYPT_FAIL_TESTVECTOR;
151     }
152
153     if ((err = ccm_memory(idx,
154                         tests[x].key, 16,
155                         NULL,
156                         tests[x].nonce, tests[x].noncelen,
157                         tests[x].header, tests[x].headerlen,
158                         buf2, tests[x].ptlen,
159                         buf,
160                         tag2, &taglen, 1    )) != CRYPT_OK) {
161         return err;
162     }
163
164     if (XMEMCMP(buf2, tests[x].pt, tests[x].ptlen)) {
165         return CRYPT_FAIL_TESTVECTOR;
166     }
167     if (XMEMCMP(tag2, tests[x].tag, tests[x].taglen)) {
168         return CRYPT_FAIL_TESTVECTOR;
169     }
170     cipher_descriptor[idx].done(&skey);
171 }
172 return CRYPT_OK;
173 #endif
174 }

```

Here is the call graph for this function:

## 5.23 encauth/eax/eax\_addheader.c File Reference

### 5.23.1 Detailed Description

EAX implementation, add meta-data, by Tom St Denis.

Definition in file [eax\\_addheader.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `eax_addheader.c`:

### Functions

- `int eax\_addheader (eax_state *eax, const unsigned char *header, unsigned long length)`  
*add header (metadata) to the stream*

### 5.23.2 Function Documentation

#### 5.23.2.1 `int eax\_addheader (eax_state * eax, const unsigned char * header, unsigned long length)`

*add header (metadata) to the stream*

#### Parameters:

- eax* The current EAX state
- header* The header (meta-data) data you wish to add to the state
- length* The length of the header data

#### Returns:

CRYPT\_OK if successful

Definition at line 26 of file `eax_addheader.c`.

References `LTC_ARGCHK`, and `omac_process()`.

```
28 {  
29     LTC_ARGCHK(eax != NULL);  
30     LTC_ARGCHK(header != NULL);  
31     return omac_process(&eax->headeromac, header, length);  
32 }
```

Here is the call graph for this function:



## 5.24 encauth/eax/eax\_decrypt.c File Reference

### 5.24.1 Detailed Description

EAX implementation, decrypt block, by Tom St Denis.

Definition in file [eax\\_decrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `eax_decrypt.c`:

### Functions

- `int eax\_decrypt(eax_state *eax, const unsigned char *ct, unsigned char *pt, unsigned long length)`
- Decrypt data with the EAX protocol.*

### 5.24.2 Function Documentation

#### 5.24.2.1 `int eax\_decrypt(eax_state *eax, const unsigned char *ct, unsigned char *pt, unsigned long length)`

Decrypt data with the EAX protocol.

#### Parameters:

- `eax`* The EAX state
- `ct`* The ciphertext
- `pt`* [out] The plaintext
- `length`* The length (octets) of the ciphertext

#### Returns:

- CRYPT\_OK if successful

Definition at line 28 of file `eax_decrypt.c`.

References `CRYPT_OK`, `ctr_decrypt()`, `LTC_ARGCHK`, and `omac_process()`.

Referenced by `eax_decrypt_verify_memory()`.

```
30 {
31     int err;
32
33     LTC_ARGCHK(eax != NULL);
34     LTC_ARGCHK(pt != NULL);
35     LTC_ARGCHK(ct != NULL);
36
37     /* omac ciphertext */
38     if ((err = omac_process(&eax->ctomac, ct, length)) != CRYPT_OK) {
39         return err;
40     }
41
42     /* decrypt */
43     return ctr_decrypt(ct, pt, length, &eax->ctr);
44 }
```

Here is the call graph for this function:

## 5.25 encauth/eax/eax\_decrypt\_verify\_memory.c File Reference

### 5.25.1 Detailed Description

EAX implementation, decrypt block of memory, by Tom St Denis.

Definition in file [eax\\_decrypt\\_verify\\_memory.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `eax_decrypt_verify_memory.c`:

### Functions

- `int eax\_decrypt\_verify\_memory (int cipher, const unsigned char *key, unsigned long keylen, const unsigned char *nonce, unsigned long noncelen, const unsigned char *header, unsigned long headerlen, const unsigned char *ct, unsigned long ctlen, unsigned char *pt, unsigned char *tag, unsigned long taglen, int *stat)`

*Decrypt a block of memory and verify the provided MAC tag with EAX.*

### 5.25.2 Function Documentation

- 5.25.2.1** `int eax\_decrypt\_verify\_memory (int cipher, const unsigned char *key, unsigned long keylen, const unsigned char *nonce, unsigned long noncelen, const unsigned char *header, unsigned long headerlen, const unsigned char *ct, unsigned long ctlen, unsigned char *pt, unsigned char *tag, unsigned long taglen, int *stat)`

Decrypt a block of memory and verify the provided MAC tag with EAX.

#### Parameters:

***cipher*** The index of the cipher desired  
***key*** The secret key  
***keylen*** The length of the key (octets)  
***nonce*** The nonce data (use once) for the session  
***noncelen*** The length of the nonce data.  
***header*** The session header data  
***headerlen*** The length of the header (octets)  
***ct*** The ciphertext  
***ctlen*** The length of the ciphertext (octets)  
***pt*** [out] The plaintext  
***tag*** The authentication tag provided by the encoder  
***taglen*** [in/out] The length of the tag (octets)  
***stat*** [out] The result of the decryption (1==valid tag, 0==invalid)

#### Returns:

CRYPT\_OK if successful regardless of the resulting tag comparison

Definition at line 37 of file `eax_decrypt_verify_memory.c`.

References `CRYPT_MEM`, `CRYPT_OK`, `eax_decrypt()`, `eax_done()`, `eax_init()`, `LTC_ARGCHK`, `XFREE`, `XMALLOC`, `XMEMCMP`, and `zeromem()`.

```

45 {
46     int            err;
47     eax_state      *eax;
48     unsigned char  *buf;
49     unsigned long   buflen;
50
51     LTC_ARGCHK(stat != NULL);
52     LTC_ARGCHK(key  != NULL);
53     LTC_ARGCHK(pt   != NULL);
54     LTC_ARGCHK(ct   != NULL);
55     LTC_ARGCHK(tag  != NULL);
56
57     /* default to zero */
58     *stat = 0;
59
60     /* allocate ram */
61     buf = XMALLOC(taglen);
62     eax = XMALLOC(sizeof(*eax));
63     if (eax == NULL || buf == NULL) {
64         if (eax != NULL) {
65             XFREE(eax);
66         }
67         if (buf != NULL) {
68             XFREE(buf);
69         }
70         return CRYPT_MEM;
71     }
72
73     if ((err = eax_init(eax, cipher, key, keylen, nonce, noncelen, header, headerlen)) != CRYPT_OK) {
74         goto LBL_ERR;
75     }
76
77     if ((err = eax_decrypt(eax, ct, pt, ctlen)) != CRYPT_OK) {
78         goto LBL_ERR;
79     }
80
81     buflen = taglen;
82     if ((err = eax_done(eax, buf, &buflen)) != CRYPT_OK) {
83         goto LBL_ERR;
84     }
85
86     /* compare tags */
87     if (buflen >= taglen && XMEMCMP(buf, tag, taglen) == 0) {
88         *stat = 1;
89     }
90
91     err = CRYPT_OK;
92 LBL_ERR:
93 #ifdef LTC_CLEAN_STACK
94     zeromem(buf, taglen);
95     zeromem(eax, sizeof(*eax));
96 #endif
97
98     XFREE(eax);
99     XFREE(buf);
100
101     return err;
102 }

```

Here is the call graph for this function:

## 5.26 encauth/eax/eax\_done.c File Reference

### 5.26.1 Detailed Description

EAX implementation, terminate session, by Tom St Denis.

Definition in file [eax\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `eax_done.c`:

### Functions

- `int eax\_done (eax_state *eax, unsigned char *tag, unsigned long *taglen)`

*Terminate an EAX session and get the tag.*

### 5.26.2 Function Documentation

#### 5.26.2.1 `int eax\_done (eax_state * eax, unsigned char * tag, unsigned long * taglen)`

Terminate an EAX session and get the tag.

#### Parameters:

*eax* The EAX state

*tag* [out] The destination of the authentication tag

*taglen* [in/out] The max length and resulting length of the authentication tag

#### Returns:

CRYPT\_OK if successful

Definition at line 27 of file `eax_done.c`.

References `CRYPT_MEM`, `CRYPT_OK`, `ctr_done()`, `len`, `LTC_ARGCHK`, `MAXBLOCKSIZE`, `omac_done()`, `XFREE`, and `XMALLOC`.

Referenced by `eax_decrypt_verify_memory()`, and `eax_encrypt_authenticate_memory()`.

```
28 {
29     int            err;
30     unsigned char  *headermac, *ctmac;
31     unsigned long  x, len;
32
33     LTC_ARGCHK(eax    != NULL);
34     LTC_ARGCHK(tag    != NULL);
35     LTC_ARGCHK(taglen != NULL);
36
37     /* allocate ram */
38     headermac = XMALLOC(MAXBLOCKSIZE);
39     ctmac     = XMALLOC(MAXBLOCKSIZE);
40
41     if (headermac == NULL || ctmac == NULL) {
42         if (headermac != NULL) {
43             XFREE(headermac);
44         }
```

```
45     if (ctmac != NULL) {
46         XFREE(ctmac);
47     }
48     return CRYPT_MEM;
49 }
50
51 /* finish ctomac */
52 len = MAXBLOCKSIZE;
53 if ((err = omac_done(&eax->ctomac, ctmac, &len)) != CRYPT_OK) {
54     goto LBL_ERR;
55 }
56
57 /* finish headeromac */
58
59 /* note we specifically don't reset len so the two lens are minimal */
60
61 if ((err = omac_done(&eax->headeromac, headermac, &len)) != CRYPT_OK) {
62     goto LBL_ERR;
63 }
64
65 /* terminate the CTR chain */
66 if ((err = ctr_done(&eax->ctr)) != CRYPT_OK) {
67     goto LBL_ERR;
68 }
69
70 /* compute N xor H xor C */
71 for (x = 0; x < len && x < *taglen; x++) {
72     tag[x] = eax->N[x] ^ headermac[x] ^ ctmac[x];
73 }
74 *taglen = x;
75
76 err = CRYPT_OK;
77 LBL_ERR:
78 #ifdef LTC_CLEAN_STACK
79     zeromem(ctmac,      MAXBLOCKSIZE);
80     zeromem(headermac, MAXBLOCKSIZE);
81     zeromem(eax,        sizeof(*eax));
82 #endif
83
84     XFREE(ctmac);
85     XFREE(headermac);
86
87     return err;
88 }
```

Here is the call graph for this function:

## 5.27 encauth/eax/eax\_encrypt.c File Reference

### 5.27.1 Detailed Description

EAX implementation, encrypt block by Tom St Denis.

Definition in file [eax\\_encrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `eax_encrypt.c`:

### Functions

- `int eax\_encrypt(eax_state *eax, const unsigned char *pt, unsigned char *ct, unsigned long length)`
- Encrypt with EAX a block of data.*

### 5.27.2 Function Documentation

#### 5.27.2.1 `int eax\_encrypt(eax_state *eax, const unsigned char *pt, unsigned char *ct, unsigned long length)`

Encrypt with EAX a block of data.

#### Parameters:

- `eax`* The EAX state
- `pt`* The plaintext to encrypt
- `ct`* [out] The ciphertext as encrypted
- `length`* The length of the plaintext (octets)

#### Returns:

- CRYPT\_OK if successful

Definition at line 28 of file `eax_encrypt.c`.

References `CRYPT_OK`, `ctr_encrypt()`, `LTC_ARGCHK`, and `omac_process()`.

Referenced by `eax_encrypt_authenticate_memory()`.

```
30 {
31     int err;
32
33     LTC_ARGCHK(eax != NULL);
34     LTC_ARGCHK(pt != NULL);
35     LTC_ARGCHK(ct != NULL);
36
37     /* encrypt */
38     if ((err = ctr_encrypt(pt, ct, length, &eax->ctr)) != CRYPT_OK) {
39         return err;
40     }
41
42     /* omac ciphertext */
43     return omac_process(&eax->ctomac, ct, length);
44 }
```

Here is the call graph for this function:

## 5.28 encauth/eax/eax\_encrypt\_authenticate\_memory.c File Reference

### 5.28.1 Detailed Description

EAX implementation, encrypt a block of memory, by Tom St Denis.

Definition in file [eax\\_encrypt\\_authenticate\\_memory.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `eax_encrypt_authenticate_memory.c`:

### Functions

- [int eax\\_encrypt\\_authenticate\\_memory](#) (int cipher, const unsigned char \*key, unsigned long keylen, const unsigned char \*nonce, unsigned long noncelen, const unsigned char \*header, unsigned long headerlen, const unsigned char \*pt, unsigned long ptlen, unsigned char \*ct, unsigned char \*tag, unsigned long \*taglen)

*EAX encrypt and produce an authentication tag.*

### 5.28.2 Function Documentation

- 5.28.2.1** `int eax_encrypt_authenticate_memory` (int *cipher*, const unsigned char \* *key*, unsigned long *keylen*, const unsigned char \* *nonce*, unsigned long *noncelen*, const unsigned char \* *header*, unsigned long *headerlen*, const unsigned char \* *pt*, unsigned long *ptlen*, unsigned char \* *ct*, unsigned char \* *tag*, unsigned long \* *taglen*)

EAX encrypt and produce an authentication tag.

#### Parameters:

- cipher* The index of the cipher desired
- key* The secret key to use
- keylen* The length of the secret key (octets)
- nonce* The session nonce [use once]
- noncelen* The length of the nonce
- header* The header for the session
- headerlen* The length of the header (octets)
- pt* The plaintext
- ptlen* The length of the plaintext (octets)
- ct* [out] The ciphertext
- tag* [out] The destination tag
- taglen* [in/out] The max size and resulting size of the authentication tag

#### Returns:

CRYPT\_OK if successful

Definition at line 36 of file `eax_encrypt_authenticate_memory.c`.

References `CRYPT_OK`, `eax_done()`, `eax_encrypt()`, `eax_init()`, `LTC_ARGCHK`, `XFREE`, `XMALLOC`, and `zeromem()`.

Referenced by `eax_test()`.

```
43 {
44     int err;
45     eax_state *eax;
46
47     LTC_ARGCHK(key    != NULL);
48     LTC_ARGCHK(pt     != NULL);
49     LTC_ARGCHK(ct     != NULL);
50     LTC_ARGCHK(tag    != NULL);
51     LTC_ARGCHK(taglen != NULL);
52
53     eax = XMALLOC(sizeof(*eax));
54
55     if ((err = eax_init(eax, cipher, key, keylen, nonce, noncelen, header, headerlen)) != CRYPT_OK) {
56         goto LBL_ERR;
57     }
58
59     if ((err = eax_encrypt(eax, pt, ct, ptlen)) != CRYPT_OK) {
60         goto LBL_ERR;
61     }
62
63     if ((err = eax_done(eax, tag, taglen)) != CRYPT_OK) {
64         goto LBL_ERR;
65     }
66
67     err = CRYPT_OK;
68 LBL_ERR:
69 #ifdef LTC_CLEAN_STACK
70     zeromem(eax, sizeof(*eax));
71 #endif
72
73     XFREE(eax);
74
75     return err;
76 }
```

Here is the call graph for this function:



## 5.29 encauth/eax/eax\_init.c File Reference

### 5.29.1 Detailed Description

EAX implementation, initialized EAX state, by Tom St Denis.

Definition in file [eax\\_init.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `eax_init.c`:

### Functions

- `int eax\_init (eax_state *eax, int cipher, const unsigned char *key, unsigned long keylen, const unsigned char *nonce, unsigned long noncelen, const unsigned char *header, unsigned long headerlen)`

*Initialized an EAX state.*

### 5.29.2 Function Documentation

- 5.29.2.1** `int eax\_init (eax_state * eax, int cipher, const unsigned char * key, unsigned long keylen, const unsigned char * nonce, unsigned long noncelen, const unsigned char * header, unsigned long headerlen)`

Initialized an EAX state.

#### Parameters:

*eax* [out] The EAX state to initialize  
*cipher* The index of the desired cipher  
*key* The secret key  
*keylen* The length of the secret key (octets)  
*nonce* The use-once nonce for the session  
*noncelen* The length of the nonce (octets)  
*header* The header for the EAX state  
*headerlen* The header length (octets)

#### Returns:

CRYPT\_OK if successful

Definition at line 32 of file `eax_init.c`.

References `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_MEM`, `CRYPT_OK`, `ctr_start()`, `len`, `LTC_ARGCHK`, `MAXBLOCKSIZE`, `omac_done()`, `omac_init()`, `omac_process()`, `XFREE`, `XMALLOC`, and `zeromem()`.

Referenced by `eax_decrypt_verify_memory()`, and `eax_encrypt_authenticate_memory()`.

```
36 {
37     unsigned char *buf;
38     int          err, blklen;
```

```

39     omac_state      *omac;
40     unsigned long len;
41
42
43     LTC_ARGCHK(eax    != NULL);
44     LTC_ARGCHK(key    != NULL);
45     LTC_ARGCHK(nonce  != NULL);
46     if (headerlen > 0) {
47         LTC_ARGCHK(header != NULL);
48     }
49
50     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
51         return err;
52     }
53     blklen = cipher_descriptor[cipher].block_length;
54
55     /* allocate ram */
56     buf = XMALLOC(MAXBLOCKSIZE);
57     omac = XMALLOC(sizeof(*omac));
58
59     if (buf == NULL || omac == NULL) {
60         if (buf != NULL) {
61             XFREE(buf);
62         }
63         if (omac != NULL) {
64             XFREE(omac);
65         }
66         return CRYPT_MEM;
67     }
68
69     /* N = OMAC_0K(nonce) */
70     zeromem(buf, MAXBLOCKSIZE);
71     if ((err = omac_init(omac, cipher, key, keylen)) != CRYPT_OK) {
72         goto LBL_ERR;
73     }
74
75     /* omac the [0]_n */
76     if ((err = omac_process(omac, buf, blklen)) != CRYPT_OK) {
77         goto LBL_ERR;
78     }
79     /* omac the nonce */
80     if ((err = omac_process(omac, nonce, noncelen)) != CRYPT_OK) {
81         goto LBL_ERR;
82     }
83     /* store result */
84     len = sizeof(eax->N);
85     if ((err = omac_done(omac, eax->N, &len)) != CRYPT_OK) {
86         goto LBL_ERR;
87     }
88
89     /* H = OMAC_1K(header) */
90     zeromem(buf, MAXBLOCKSIZE);
91     buf[blklen - 1] = 1;
92
93     if ((err = omac_init(&eax->headeromac, cipher, key, keylen)) != CRYPT_OK) {
94         goto LBL_ERR;
95     }
96
97     /* omac the [1]_n */
98     if ((err = omac_process(&eax->headeromac, buf, blklen)) != CRYPT_OK) {
99         goto LBL_ERR;
100     }
101     /* omac the header */
102     if (headerlen != 0) {
103         if ((err = omac_process(&eax->headeromac, header, headerlen)) != CRYPT_OK) {
104             goto LBL_ERR;
105         }

```

```
106     }
107
108     /* note we don't finish the headeromac, this allows us to add more header later */
109
110     /* setup the CTR mode */
111     if ((err = ctr_start(cipher, eax->N, key, keylen, 0, CTR_COUNTER_BIG_ENDIAN, &eax->ctr)) != CRYPT_OK)
112         goto LBL_ERR;
113 }
114
115 /* setup the OMAC for the ciphertext */
116 if ((err = omac_init(&eax->ctomac, cipher, key, keylen)) != CRYPT_OK) {
117     goto LBL_ERR;
118 }
119
120 /* omac [2]_n */
121 zeromem(buf, MAXBLOCKSIZE);
122 buf[blklen-1] = 2;
123 if ((err = omac_process(&eax->ctomac, buf, blklen)) != CRYPT_OK) {
124     goto LBL_ERR;
125 }
126
127 err = CRYPT_OK;
128 LBL_ERR:
129 #ifdef LTC_CLEAN_STACK
130     zeromem(buf, MAXBLOCKSIZE);
131     zeromem(omac, sizeof(*omac));
132 #endif
133
134     XFREE(omac);
135     XFREE(buf);
136
137     return err;
138 }
```

Here is the call graph for this function:

## 5.30 encauth/eax/eax\_test.c File Reference

### 5.30.1 Detailed Description

EAX implementation, self-test, by Tom St Denis.

Definition in file [eax\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `eax_test.c`:

### Functions

- `int eax\_test (void)`  
*Test the EAX implementation.*

### 5.30.2 Function Documentation

#### 5.30.2.1 `int eax\_test (void)`

Test the EAX implementation.

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-testing has been disabled

Definition at line 24 of file `eax_test.c`.

References `CRYPT_NOP`, `CRYPT_OK`, `eax_encrypt_authenticate_memory()`, `find_cipher()`, `len`, and `MAXBLOCKSIZE`.

```
25 {
26 #ifndef LTC_TEST
27     return CRYPT_NOP;
28 #else
29     static const struct {
30         int          keylen,
31                 noncelen,
32                 headerlen,
33                 msglen;
34
35         unsigned char    key[MAXBLOCKSIZE],
36                 nonce[MAXBLOCKSIZE],
37                 header[MAXBLOCKSIZE],
38                 plaintext[MAXBLOCKSIZE],
39                 ciphertext[MAXBLOCKSIZE],
40                 tag[MAXBLOCKSIZE];
41     } tests[] = {
42
43     /* NULL message */
44     {
45         16, 0, 0, 0,
46         /* key */
47         { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
48           0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
49         /* nonce */
50         { 0 },
```

```

51  /* header */
52  { 0 },
53  /* plaintext */
54  { 0 },
55  /* ciphertext */
56  { 0 },
57  /* tag */
58  { 0x9a, 0xd0, 0x7e, 0x7d, 0xbf, 0xf3, 0x01, 0xf5,
59    0x05, 0xde, 0x59, 0x6b, 0x96, 0x15, 0xdf, 0xff }
60 },
61
62 /* test with nonce */
63 {
64     16, 16, 0, 0,
65     /* key */
66     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
67       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
68     /* nonce */
69     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
70       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
71     /* header */
72     { 0 },
73     /* plaintext */
74     { 0 },
75     /* ciphertext */
76     { 0 },
77     /* tag */
78     { 0x1c, 0xe1, 0x0d, 0x3e, 0xff, 0xd4, 0xca, 0xdb,
79       0xe2, 0xe4, 0x4b, 0x58, 0xd6, 0x0a, 0xb9, 0xec }
80 },
81
82 /* test with header [no nonce] */
83 {
84     16, 0, 16, 0,
85     /* key */
86     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
87       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
88     /* nonce */
89     { 0 },
90     /* header */
91     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
92       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
93     /* plaintext */
94     { 0 },
95     /* ciphertext */
96     { 0 },
97     /* tag */
98     { 0x3a, 0x69, 0x8f, 0x7a, 0x27, 0x0e, 0x51, 0xb0,
99       0xf6, 0x5b, 0x3d, 0x3e, 0x47, 0x19, 0x3c, 0xff }
100 },
101
102 /* test with header + nonce + plaintext */
103 {
104     16, 16, 16, 32,
105     /* key */
106     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
107       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
108     /* nonce */
109     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
110       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
111     /* header */
112     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
113       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
114     /* plaintext */
115     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
116       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
117       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,

```

```

118     0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f },
119     /* ciphertext */
120     { 0x29, 0xd8, 0x78, 0xd1, 0xa3, 0xbe, 0x85, 0x7b,
121       0x6f, 0xb8, 0xc8, 0xea, 0x59, 0x50, 0xa7, 0x78,
122       0x33, 0x1f, 0xbf, 0x2c, 0xcf, 0x33, 0x98, 0x6f,
123       0x35, 0xe8, 0xcf, 0x12, 0x1d, 0xcb, 0x30, 0xbc },
124     /* tag */
125     { 0x4f, 0xbe, 0x03, 0x38, 0xbe, 0x1c, 0x8c, 0x7e,
126       0x1d, 0x7a, 0xe7, 0xe4, 0x5b, 0x92, 0xc5, 0x87 }
127 },
128
129 /* test with header + nonce + plaintext [not even sizes!] */
130 {
131     16, 15, 14, 29,
132     /* key */
133     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
134       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
135     /* nonce */
136     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
137       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e },
138     /* header */
139     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
140       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d },
141     /* plaintext */
142     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
143       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
144       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
145       0x18, 0x19, 0x1a, 0x1b, 0x1c },
146     /* ciphertext */
147     { 0xdd, 0x25, 0xc7, 0x54, 0xc5, 0xb1, 0x7c, 0x59,
148       0x28, 0xb6, 0x9b, 0x73, 0x15, 0x5f, 0x7b, 0xb8,
149       0x88, 0x8f, 0xaf, 0x37, 0x09, 0x1a, 0xd9, 0x2c,
150       0x8a, 0x24, 0xdb, 0x86, 0x8b },
151     /* tag */
152     { 0x0d, 0x1a, 0x14, 0xe5, 0x22, 0x24, 0xff, 0xd2,
153       0x3a, 0x05, 0xfa, 0x02, 0xcd, 0xef, 0x52, 0xda }
154 },
155
156 /* Vectors from Brian Gladman */
157
158 {
159     16, 16, 8, 0,
160     /* key */
161     { 0x23, 0x39, 0x52, 0xde, 0xe4, 0xd5, 0xed, 0x5f,
162       0x9b, 0x9c, 0x6d, 0x6f, 0xf8, 0x0f, 0xf4, 0x78 },
163     /* nonce */
164     { 0x62, 0xec, 0x67, 0xf9, 0xc3, 0xa4, 0xa4, 0x07,
165       0xfc, 0xb2, 0xa8, 0xc4, 0x90, 0x31, 0xa8, 0xb3 },
166     /* header */
167     { 0x6b, 0xfb, 0x91, 0x4f, 0xd0, 0x7e, 0xae, 0x6b },
168     /* PT */
169     { 0x00 },
170     /* CT */
171     { 0x00 },
172     /* tag */
173     { 0xe0, 0x37, 0x83, 0x0e, 0x83, 0x89, 0xf2, 0x7b,
174       0x02, 0x5a, 0x2d, 0x65, 0x27, 0xe7, 0x9d, 0x01 }
175 },
176
177 {
178     16, 16, 8, 2,
179     /* key */
180     { 0x91, 0x94, 0x5d, 0x3f, 0x4d, 0xcb, 0xee, 0x0b,
181       0xf4, 0x5e, 0xf5, 0x22, 0x55, 0xf0, 0x95, 0xa4 },
182     /* nonce */
183     { 0xbe, 0xca, 0xf0, 0x43, 0xb0, 0xa2, 0x3d, 0x84,
184       0x31, 0x94, 0xba, 0x97, 0x2c, 0x66, 0xde, 0xbd },

```

```

185  /* header */
186  { 0xfa, 0x3b, 0xfd, 0x48, 0x06, 0xeb, 0x53, 0xfa },
187  /* PT */
188  { 0xf7, 0xfb },
189  /* CT */
190  { 0x19, 0xdd },
191  /* tag */
192  { 0x5c, 0x4c, 0x93, 0x31, 0x04, 0x9d, 0x0b, 0xda,
193    0xb0, 0x27, 0x74, 0x08, 0xf6, 0x79, 0x67, 0xe5 }
194 },
195
196 {
197     16, 16, 8, 5,
198     /* key */
199     { 0x01, 0xf7, 0x4a, 0xd6, 0x40, 0x77, 0xf2, 0xe7,
200       0x04, 0xc0, 0xf6, 0x0a, 0xda, 0x3d, 0xd5, 0x23 },
201     /* nonce */
202     { 0x70, 0xc3, 0xdb, 0x4f, 0x0d, 0x26, 0x36, 0x84,
203       0x00, 0xa1, 0x0e, 0xd0, 0x5d, 0x2b, 0xff, 0x5e },
204     /* header */
205     { 0x23, 0x4a, 0x34, 0x63, 0xc1, 0x26, 0x4a, 0xc6 },
206     /* PT */
207     { 0x1a, 0x47, 0xcb, 0x49, 0x33 },
208     /* CT */
209     { 0xd8, 0x51, 0xd5, 0xba, 0xe0 },
210     /* Tag */
211     { 0x3a, 0x59, 0xf2, 0x38, 0xa2, 0x3e, 0x39, 0x19,
212       0x9d, 0xc9, 0x26, 0x66, 0x26, 0xc4, 0x0f, 0x80 }
213 }
214
215 };
216 int err, x, idx, res;
217 unsigned long len;
218 unsigned char outct[MAXBLOCKSIZE], outtag[MAXBLOCKSIZE];
219
220 /* AES can be under rijndael or aes... try to find it */
221 if ((idx = find_cipher("aes")) == -1) {
222     if ((idx = find_cipher("rijndael")) == -1) {
223         return CRYPT_NOP;
224     }
225 }
226
227 for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
228     len = sizeof(outtag);
229     if ((err = eax_encrypt_authenticate_memory(idx, tests[x].key, tests[x].keylen,
230         tests[x].nonce, tests[x].noncelen, tests[x].header, tests[x].headerlen,
231         tests[x].plaintext, tests[x].msglen, outct, outtag, &len)) != CRYPT_OK) {
232         return err;
233     }
234     if (XMEMCMP(outct, tests[x].ciphertext, tests[x].msglen) || XMEMCMP(outtag, tests[x].tag, len))
235 #if 0
236         unsigned long y;
237         printf("\n\nFailure: \nCT:\n");
238         for (y = 0; y < (unsigned long)tests[x].msglen; ) {
239             printf("0x%02x", outct[y]);
240             if (y < (unsigned long)(tests[x].msglen-1)) printf(", ");
241             if (!(++y % 8)) printf("\n");
242         }
243         printf("\nTAG:\n");
244         for (y = 0; y < len; ) {
245             printf("0x%02x", outtag[y]);
246             if (y < len-1) printf(", ");
247             if (!(++y % 8)) printf("\n");
248         }
249 #endif
250     return CRYPT_FAIL_TESTVECTOR;
251 }

```

```
252
253     /* test decrypt */
254     if ((err = eax_decrypt_verify_memory(idx, tests[x].key, tests[x].keylen,
255         tests[x].nonce, tests[x].noncelen, tests[x].header, tests[x].headerlen,
256         outct, tests[x].msglen, outct, outtag, len, &res)) != CRYPT_OK) {
257         return err;
258     }
259     if ((res != 1) || XMEMCMP(outct, tests[x].plaintext, tests[x].msglen)) {
260 #if 0
261         unsigned long y;
262         printf("\n\nFailure (res == %d): \nPT:\n", res);
263         for (y = 0; y < (unsigned long)tests[x].msglen; ) {
264             printf("0x%02x", outct[y]);
265             if (y < (unsigned long)(tests[x].msglen-1)) printf(", ");
266             if (!(++y % 8)) printf("\n");
267         }
268         printf("\n\n");
269 #endif
270         return CRYPT_FAIL_TESTVECTOR;
271     }
272
273     }
274     return CRYPT_OK;
275 #endif /* LTC_TEST */
276 }
```

Here is the call graph for this function:



## 5.31 `encauth/gcm/gcm_add_aad.c` File Reference

### 5.31.1 Detailed Description

GCM implementation, Add AAD data to the stream, by Tom St Denis.

Definition in file [gcm\\_add\\_aad.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `gcm_add_aad.c`:

### Functions

- `int gcm_add_aad(gcm_state *gcm, const unsigned char *adata, unsigned long adatalen)`  
*Add AAD to the GCM state.*

### 5.31.2 Function Documentation

#### 5.31.2.1 `int gcm_add_aad(gcm_state *gcm, const unsigned char *adata, unsigned long adatalen)`

Add AAD to the GCM state.

#### Parameters:

*gcm* The GCM state

*adata* The additional authentication data to add to the GCM state

*adatalen* The length of the AAD data.

#### Returns:

CRYPT\_OK on success

Definition at line 27 of file `gcm_add_aad.c`.

References `cipher_is_valid()`, `CONST64`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `gcm_mult_h()`, `LTC_ARGCHK`, `XMEMCPY`, and `zeromem()`.

Referenced by `gcm_memory()`.

```
29 {
30     unsigned long x;
31     int          err;
32 #ifdef LTC_FAST
33     unsigned long y;
34 #endif
35
36     LTC_ARGCHK(gcm      != NULL);
37     if (adatalen > 0) {
38         LTC_ARGCHK(adata != NULL);
39     }
40
41     if (gcm->buflen > 16 || gcm->buflen < 0) {
42         return CRYPT_INVALID_ARG;
43     }
44
45     if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
```

```

46     return err;
47 }
48
49 /* in IV mode? */
50 if (gcm->mode == GCM_MODE_IV) {
51     /* let's process the IV */
52     if (gcm->ivmode || gcm->buflen != 12) {
53         for (x = 0; x < (unsigned long)gcm->buflen; x++) {
54             gcm->X[x] ^= gcm->buf[x];
55         }
56         if (gcm->buflen) {
57             gcm->totlen += gcm->buflen * CONST64(8);
58             gcm_mult_h(gcm, gcm->X);
59         }
60
61         /* mix in the length */
62         zeromem(gcm->buf, 8);
63         STORE64H(gcm->totlen, gcm->buf+8);
64         for (x = 0; x < 16; x++) {
65             gcm->X[x] ^= gcm->buf[x];
66         }
67         gcm_mult_h(gcm, gcm->X);
68
69         /* copy counter out */
70         XMEMCPY(gcm->Y, gcm->X, 16);
71         zeromem(gcm->X, 16);
72     } else {
73         XMEMCPY(gcm->Y, gcm->buf, 12);
74         gcm->Y[12] = 0;
75         gcm->Y[13] = 0;
76         gcm->Y[14] = 0;
77         gcm->Y[15] = 1;
78     }
79     XMEMCPY(gcm->Y_0, gcm->Y, 16);
80     zeromem(gcm->buf, 16);
81     gcm->buflen = 0;
82     gcm->totlen = 0;
83     gcm->mode = GCM_MODE_AAD;
84 }
85
86 if (gcm->mode != GCM_MODE_AAD || gcm->buflen >= 16) {
87     return CRYPT_INVALID_ARG;
88 }
89
90 x = 0;
91 #ifdef LTC_FAST
92     if (gcm->buflen == 0) {
93         for (x = 0; x < (adatalen & ~15); x += 16) {
94             for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
95                 *((LTC_FAST_TYPE*)(&gcm->X[y])) ^= *((LTC_FAST_TYPE*)(&adata[x + y]));
96             }
97             gcm_mult_h(gcm, gcm->X);
98             gcm->totlen += 128;
99         }
100         adata += x;
101     }
102 #endif
103
104
105 /* start adding AAD data to the state */
106 for (; x < adatalen; x++) {
107     gcm->X[gcm->buflen++] ^= *adata++;
108
109     if (gcm->buflen == 16) {
110         /* GF mult it */
111         gcm_mult_h(gcm, gcm->X);
112         gcm->buflen = 0;

```

```
113         gcm->totlen += 128;
114     }
115 }
116
117 return CRYPT_OK;
118 }
```

Here is the call graph for this function:

## 5.32 encauth/gcm/gcm\_add\_iv.c File Reference

### 5.32.1 Detailed Description

GCM implementation, add IV data to the state, by Tom St Denis.

Definition in file [gcm\\_add\\_iv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for gcm\_add\_iv.c:

### Functions

- `int gcm_add_iv(gcm_state *gcm, const unsigned char *IV, unsigned long IVlen)`  
*Add IV data to the GCM state.*

### 5.32.2 Function Documentation

#### 5.32.2.1 `int gcm_add_iv(gcm_state *gcm, const unsigned char *IV, unsigned long IVlen)`

Add IV data to the GCM state.

#### Parameters:

- gcm* The GCM state
- IV* The initial value data to add
- IVlen* The length of the IV

#### Returns:

- CRYPT\_OK on success

Definition at line 27 of file gcm\_add\_iv.c.

References [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [gcm\\_mult\\_h\(\)](#), and [LTC\\_ARGCHK](#).

Referenced by [gcm\\_memory\(\)](#).

```
29 {
30     unsigned long x, y;
31     int          err;
32
33     LTC_ARGCHK(gcm != NULL);
34     if (IVlen > 0) {
35         LTC_ARGCHK(IV != NULL);
36     }
37
38     /* must be in IV mode */
39     if (gcm->mode != GCM_MODE_IV) {
40         return CRYPT_INVALID_ARG;
41     }
42
43     if (gcm->buflen >= 16 || gcm->buflen < 0) {
44         return CRYPT_INVALID_ARG;
45     }
```

```
46
47     if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
48         return err;
49     }
50
51
52     /* trip the ivmode flag */
53     if (IVlen + gcm->buflen > 12) {
54         gcm->ivmode |= 1;
55     }
56
57     x = 0;
58 #ifdef LTC_FAST
59     if (gcm->buflen == 0) {
60         for (x = 0; x < (IVlen & ~15); x += 16) {
61             for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
62                 *((LTC_FAST_TYPE*) (&gcm->X[y])) ^= *((LTC_FAST_TYPE*) (&IV[x + y]));
63             }
64             gcm_mult_h(gcm, gcm->X);
65             gcm->totlen += 128;
66         }
67         IV += x;
68     }
69 #endif
70
71     /* start adding IV data to the state */
72     for (; x < IVlen; x++) {
73         gcm->buf[gcm->buflen++] = *IV++;
74
75         if (gcm->buflen == 16) {
76             /* GF mult it */
77             for (y = 0; y < 16; y++) {
78                 gcm->X[y] ^= gcm->buf[y];
79             }
80             gcm_mult_h(gcm, gcm->X);
81             gcm->buflen = 0;
82             gcm->totlen += 128;
83         }
84     }
85
86     return CRYPT_OK;
87 }
```

Here is the call graph for this function:

## 5.33 encauth/gcm/gcm\_done.c File Reference

### 5.33.1 Detailed Description

GCM implementation, Terminate the stream, by Tom St Denis.

Definition in file [gcm\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for gcm\_done.c:

### Functions

- int [gcm\\_done](#) (gcm\_state \*gcm, unsigned char \*tag, unsigned long \*taglen)

*Terminate a GCM stream.*

### 5.33.2 Function Documentation

#### 5.33.2.1 int gcm\_done (gcm\_state \* gcm, unsigned char \* tag, unsigned long \* taglen)

Terminate a GCM stream.

#### Parameters:

*gcm* The GCM state

*tag* [out] The destination for the MAC tag

*taglen* [in/out] The length of the MAC tag

#### Returns:

CRYPT\_OK on success

Definition at line 27 of file gcm\_done.c.

References [cipher\\_is\\_valid\(\)](#), [CONST64](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [gcm\\_mult\\_h\(\)](#), and [LTC\\_ARGCHK](#).

Referenced by [gcm\\_memory\(\)](#).

```

29 {
30     unsigned long x;
31     int err;
32
33     LTC_ARGCHK(gcm      != NULL);
34     LTC_ARGCHK(tag      != NULL);
35     LTC_ARGCHK(taglen   != NULL);
36
37     if (gcm->buflen > 16 || gcm->buflen < 0) {
38         return CRYPT_INVALID_ARG;
39     }
40
41     if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
42         return err;
43     }
44
45
```

```
46     if (gcm->mode != GCM_MODE_TEXT) {
47         return CRYPT_INVALID_ARG;
48     }
49
50     /* handle remaining ciphertext */
51     if (gcm->buflen) {
52         gcm->pttotlen += gcm->buflen * CONST64(8);
53         gcm_mult_h(gcm, gcm->X);
54     }
55
56     /* length */
57     STORE64H(gcm->totlen, gcm->buf);
58     STORE64H(gcm->pttotlen, gcm->buf+8);
59     for (x = 0; x < 16; x++) {
60         gcm->X[x] ^= gcm->buf[x];
61     }
62     gcm_mult_h(gcm, gcm->X);
63
64     /* encrypt original counter */
65     if ((err = cipher_descriptor[gcm->cipher].ecb_encrypt(gcm->Y_0, gcm->buf, &gcm->K)) != CRYPT_OK) {
66         return err;
67     }
68     for (x = 0; x < 16 && x < *taglen; x++) {
69         tag[x] = gcm->buf[x] ^ gcm->X[x];
70     }
71     *taglen = x;
72
73     cipher_descriptor[gcm->cipher].done(&gcm->K);
74
75     return CRYPT_OK;
76 }
```

Here is the call graph for this function:

## 5.34 encauth/gcm/gcm\_gf\_mult.c File Reference

### 5.34.1 Detailed Description

GCM implementation, do the GF mult, by Tom St Denis.

Definition in file [gcm\\_gf\\_mult.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for gcm\_gf\_mult.c:

### Functions

- static void [gcm\\_rightshift](#) (unsigned char \*a)
- void [gcm\\_gf\\_mult](#) (const unsigned char \*a, const unsigned char \*b, unsigned char \*c)  
*GCM GF multiplier (internal use only) bitserial.*

### Variables

- const unsigned char [gcm\\_shift\\_table](#) [256 \*2]
- static const unsigned char [mask](#) [] = { 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 }
- static const unsigned char [poly](#) [] = { 0x00, 0xE1 }

### 5.34.2 Function Documentation

#### 5.34.2.1 void gcm\_gf\_mult (const unsigned char \* a, const unsigned char \* b, unsigned char \* c)

GCM GF multiplier (internal use only) bitserial.

#### Parameters:

- a** First value
- b** Second value
- c** Destination for a \* b

Definition at line 83 of file gcm\_gf\_mult.c.

References [mask](#), [XMEMCPY](#), and [zeromem\(\)](#).

Referenced by [gcm\\_init\(\)](#), and [lrw\\_start\(\)](#).

```
84 {
85     unsigned char Z[16], V[16];
86     unsigned x, y, z;
87
88     zeromem(Z, 16);
89     XMEMCPY(V, a, 16);
90     for (x = 0; x < 128; x++) {
91         if (b[x>>3] & mask[x&7]) {
92             for (y = 0; y < 16; y++) {
93                 Z[y] ^= V[y];
94             }
95         }
96     }
```



```
96      z      = V[15] & 0x01;
97      gcm_rightshift(V);
98      V[0] ^= poly[z];
99  }
100  XMEMCPY(c, z, 16);
101 }
```

Here is the call graph for this function:

#### 5.34.2.2 static void gcm\_rightshift (unsigned char \*a) [static]

Definition at line 63 of file gcm\_gf\_mult.c.

```
64 {
65     int x;
66     for (x = 15; x > 0; x--) {
67         a[x] = (a[x]>>1) | ((a[x-1]<<7)&0x80);
68     }
69     a[0] >>= 1;
70 }
```

### 5.34.3 Variable Documentation

#### 5.34.3.1 const unsigned char gcm\_shift\_table[256\*2]

Definition at line 22 of file gcm\_gf\_mult.c.

#### 5.34.3.2 const unsigned char mask[] = { 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 } [static]

Definition at line 73 of file gcm\_gf\_mult.c.

Referenced by der\_encode\_object\_identifier(), gcm\_gf\_mult(), omac\_init(), pkcs\_1\_oaep\_decode(), pkcs\_1\_oaep\_encode(), pkcs\_1\_pss\_decode(), and pkcs\_1\_pss\_encode().

#### 5.34.3.3 const unsigned char poly[] = { 0x00, 0xE1 } [static]

Definition at line 74 of file gcm\_gf\_mult.c.

Referenced by ocb\_init(), and pmac\_init().

## 5.35 encauth/gcm/gcm\_init.c File Reference

### 5.35.1 Detailed Description

GCM implementation, initialize state, by Tom St Denis.

Definition in file [gcm\\_init.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for gcm\_init.c:

### Functions

- [int gcm\\_init](#) (gcm\_state \*gcm, int cipher, const unsigned char \*key, int keylen)  
*Initialize a GCM state.*

### 5.35.2 Function Documentation

#### 5.35.2.1 int gcm\_init (gcm\_state \*gcm, int cipher, const unsigned char \*key, int keylen)

Initialize a GCM state.

#### Parameters:

- gcm** The GCM state to initialize
- cipher** The index of the cipher to use
- key** The secret key
- keylen** The length of the secret key

#### Returns:

CRYPT\_OK on success

Definition at line 28 of file gcm\_init.c.

References [B](#), [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_INVALID\\_CIPHER](#), [CRYPT\\_OK](#), [ecb\\_encrypt\(\)](#), [gcm\\_gf\\_mult\(\)](#), [LTC\\_ARGCHK](#), and [zeromem\(\)](#).

Referenced by [gcm\\_memory\(\)](#).

```
30 {
31     int          err;
32     unsigned char B[16];
33 #ifdef GCM_TABLES
34     int          x, y, z, t;
35 #endif
36
37     LTC_ARGCHK(gcm != NULL);
38     LTC_ARGCHK(key != NULL);
39
40 #ifdef LTC_FAST
41     if (16 % sizeof(LTC_FAST_TYPE)) {
42         return CRYPT_INVALID_ARG;
43     }
44 #endif
45 }
```

```

46  /* is cipher valid? */
47  if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
48      return err;
49  }
50  if (cipher_descriptor[cipher].block_length != 16) {
51      return CRYPT_INVALID_CIPHER;
52  }
53
54  /* schedule key */
55  if ((err = cipher_descriptor[cipher].setup(key, keylen, 0, &gcm->K)) != CRYPT_OK) {
56      return err;
57  }
58
59  /* H = E(0) */
60  zeromem(B, 16);
61  if ((err = cipher_descriptor[cipher].ecb_encrypt(B, gcm->H, &gcm->K)) != CRYPT_OK) {
62      return err;
63  }
64
65  /* setup state */
66  zeromem(gcm->buf, sizeof(gcm->buf));
67  zeromem(gcm->X, sizeof(gcm->X));
68  gcm->cipher = cipher;
69  gcm->mode = GCM_MODE_IV;
70  gcm->ivmode = 0;
71  gcm->buflen = 0;
72  gcm->totlen = 0;
73  gcm->pttotlen = 0;
74
75  #ifdef GCM_TABLES
76  /* setup tables */
77
78  /* generate the first table as it has no shifting (from which we make the other tables) */
79  zeromem(B, 16);
80  for (y = 0; y < 256; y++) {
81      B[0] = y;
82      gcm_gf_mult(gcm->H, B, &gcm->PC[0][y][0]);
83  }
84
85  /* now generate the rest of the tables based the previous table */
86  for (x = 1; x < 16; x++) {
87      for (y = 0; y < 256; y++) {
88          /* now shift it right by 8 bits */
89          t = gcm->PC[x-1][y][15];
90          for (z = 15; z > 0; z--) {
91              gcm->PC[x][y][z] = gcm->PC[x-1][y][z-1];
92          }
93          gcm->PC[x][y][0] = gcm_shift_table[t<<1];
94          gcm->PC[x][y][1] ^= gcm_shift_table[(t<<1)+1];
95      }
96  }
97
98  #endif
99
100  return CRYPT_OK;
101 }

```

Here is the call graph for this function:

## 5.36 encauth/gcm/gcm\_memory.c File Reference

### 5.36.1 Detailed Description

GCM implementation, process a packet, by Tom St Denis.

Definition in file [gcm\\_memory.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for gcm\_memory.c:

### Functions

- int [gcm\\_memory](#) (int cipher, const unsigned char \*key, unsigned long keylen, const unsigned char \*IV, unsigned long IVlen, const unsigned char \*adata, unsigned long adatalen, unsigned char \*pt, unsigned long ptlen, unsigned char \*ct, unsigned char \*tag, unsigned long \*taglen, int direction)

*Process an entire GCM packet in one call.*

### 5.36.2 Function Documentation

- 5.36.2.1** int [gcm\\_memory](#) (int *cipher*, const unsigned char \* *key*, unsigned long *keylen*, const unsigned char \* *IV*, unsigned long *IVlen*, const unsigned char \* *adata*, unsigned long *adatalen*, unsigned char \* *pt*, unsigned long *ptlen*, unsigned char \* *ct*, unsigned char \* *tag*, unsigned long \* *taglen*, int *direction*)

Process an entire GCM packet in one call.

#### Parameters:

*cipher* Index of cipher to use

*key* The secret key

*keylen* The length of the secret key

*IV* The initial vector

*IVlen* The length of the initial vector

*adata* The additional authentication data (header)

*adatalen* The length of the adata

*pt* The plaintext

*ptlen* The length of the plaintext (ciphertext length is the same)

*ct* The ciphertext

*tag* [out] The MAC tag

*taglen* [in/out] The MAC tag length

*direction* Encrypt or Decrypt mode (GCM\_ENCRYPT or GCM\_DECRYPT)

#### Returns:

CRYPT\_OK on success

Definition at line 37 of file gcm\_memory.c.

References `ltc_cipher_descriptor::accel_gcm_memory`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_MEM`, `CRYPT_OK`, `gcm_add_aad()`, `gcm_add_iv()`, `gcm_done()`, `gcm_init()`, `gcm_process()`, `XFREE`, and `XMALLOC`.

Referenced by `gcm_test()`.

```

45 {
46     void      *orig;
47     gcm_state *gcm;
48     int       err;
49
50     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
51         return err;
52     }
53
54     if (cipher_descriptor[cipher].accel_gcm_memory != NULL) {
55         return
56             cipher_descriptor[cipher].accel_gcm_memory
57                 (key,    keylen,
58                  IV,     IVlen,
59                  adata,  adatalen,
60                  pt,     ptlen,
61                  ct,
62                  tag,    taglen,
63                  direction);
64     }
65
66
67
68 #ifndef GCM_TABLES_SSE2
69     orig = gcm = XMALLOC(sizeof(*gcm));
70 #else
71     orig = gcm = XMALLOC(sizeof(*gcm) + 16);
72 #endif
73     if (gcm == NULL) {
74         return CRYPT_MEM;
75     }
76
77     /* Force GCM to be on a multiple of 16 so we can use 128-bit aligned operations
78      * note that we only modify gcm and keep orig intact. This code is not portable
79      * but again it's only for SSE2 anyways, so who cares?
80      */
81 #ifdef GCM_TABLES_SSE2
82     if ((unsigned long)gcm & 15) {
83         gcm = (gcm_state *)((unsigned long)gcm + (16 - ((unsigned long)gcm & 15)));
84     }
85 #endif
86
87     if ((err = gcm_init(gcm, cipher, key, keylen)) != CRYPT_OK) {
88         goto LTC_ERR;
89     }
90     if ((err = gcm_add_iv(gcm, IV, IVlen)) != CRYPT_OK) {
91         goto LTC_ERR;
92     }
93     if ((err = gcm_add_aad(gcm, adata, adatalen)) != CRYPT_OK) {
94         goto LTC_ERR;
95     }
96     if ((err = gcm_process(gcm, pt, ptlen, ct, direction)) != CRYPT_OK) {
97         goto LTC_ERR;
98     }
99     err = gcm_done(gcm, tag, taglen);
100 LTC_ERR:
101     XFREE(orig);
102     return err;
103 }

```

Here is the call graph for this function:

## 5.37 encauth/gcm/gcm\_mult\_h.c File Reference

### 5.37.1 Detailed Description

GCM implementation, do the GF mult, by Tom St Denis.

Definition in file [gcm\\_mult\\_h.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for gcm\_mult\_h.c:

### Functions

- void [gcm\\_mult\\_h](#) (gcm\_state \*gcm, unsigned char \*I)  
*GCM multiply by H.*

### 5.37.2 Function Documentation

#### 5.37.2.1 void gcm\_mult\_h (gcm\_state \*gcm, unsigned char \*I)

GCM multiply by H.

#### Parameters:

*gcm* The GCM state which holds the H value

*I* The value to multiply H by

Definition at line 24 of file gcm\_mult\_h.c.

Referenced by gcm\_add\_aad(), gcm\_add\_iv(), gcm\_done(), and gcm\_process().

```
25 {
26     unsigned char T[16];
27 #ifdef GCM_TABLES
28     int x, y;
29 #ifdef GCM_TABLES_SSE2
30     asm("movdqa (%0), %%xmm0"::"r" (&gcm->PC[0][I[0]][0]));
31     for (x = 1; x < 16; x++) {
32         asm("pxor (%0), %%xmm0"::"r" (&gcm->PC[x][I[x]][0]));
33     }
34     asm("movdqa %%xmm0, (%0)"::"r" (&T));
35 #else
36     XMEMCPY(T, &gcm->PC[0][I[0]][0], 16);
37     for (x = 1; x < 16; x++) {
38 #ifdef LTC_FAST
39         for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
40             *((LTC_FAST_TYPE *) (T + y)) ^= *((LTC_FAST_TYPE *) (&gcm->PC[x][I[x]][y]));
41         }
42 #else
43         for (y = 0; y < 16; y++) {
44             T[y] ^= gcm->PC[x][I[x]][y];
45         }
46 #endif /* LTC_FAST */
47     }
48 #endif /* GCM_TABLES_SSE2 */
49 #else
50     gcm_gf_mult(gcm->H, I, T);
```

```
51 #endif
52     XMEMCPY(I, T, 16);
53 }
```



## 5.38 encauth/gcm/gcm\_process.c File Reference

### 5.38.1 Detailed Description

GCM implementation, process message data, by Tom St Denis.

Definition in file [gcm\\_process.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for gcm\_process.c:

### Functions

- `int gcm_process` (`gcm_state *gcm`, `unsigned char *pt`, `unsigned long ptlen`, `unsigned char *ct`, `int direction`)

*Process plaintext/ciphertext through GCM.*

### 5.38.2 Function Documentation

#### 5.38.2.1 `int gcm_process` (`gcm_state *gcm`, `unsigned char *pt`, `unsigned long ptlen`, `unsigned char *ct`, `int direction`)

Process plaintext/ciphertext through GCM.

#### Parameters:

*gcm* The GCM state

*pt* The plaintext

*ptlen* The plaintext length (ciphertext length is the same)

*ct* The ciphertext

*direction* Encrypt or Decrypt mode (GCM\_ENCRYPT or GCM\_DECRYPT)

#### Returns:

CRYPT\_OK on success

Definition at line 29 of file gcm\_process.c.

References `cipher_is_valid()`, `CONST64`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `gcm_mult_h()`, and `LTC_ARGCHK`.

Referenced by `gcm_memory()`.

```
33 {
34     unsigned long x;
35     int          y, err;
36     unsigned char b;
37
38     LTC_ARGCHK(gcm != NULL);
39     if (ptlen > 0) {
40         LTC_ARGCHK(pt != NULL);
41         LTC_ARGCHK(ct != NULL);
42     }
43 }
```

```

44     if (gcm->buflen > 16 || gcm->buflen < 0) {
45         return CRYPT_INVALID_ARG;
46     }
47
48     if ((err = cipher_is_valid(gcm->cipher)) != CRYPT_OK) {
49         return err;
50     }
51
52     /* in AAD mode? */
53     if (gcm->mode == GCM_MODE_AAD) {
54         /* let's process the AAD */
55         if (gcm->buflen) {
56             gcm->totlen += gcm->buflen * CONST64(8);
57             gcm_mult_h(gcm, gcm->X);
58         }
59
60         /* increment counter */
61         for (y = 15; y >= 0; y--) {
62             if (++gcm->Y[y] & 255) { break; }
63         }
64         /* encrypt the counter */
65         if ((err = cipher_descriptor[gcm->cipher].ecb_encrypt(gcm->Y, gcm->buf, &gcm->K)) != CRYPT_OK) {
66             return err;
67         }
68
69         gcm->buflen = 0;
70         gcm->mode = GCM_MODE_TEXT;
71     }
72
73     if (gcm->mode != GCM_MODE_TEXT) {
74         return CRYPT_INVALID_ARG;
75     }
76
77     x = 0;
78 #ifdef LTC_FAST
79     if (gcm->buflen == 0) {
80         if (direction == GCM_ENCRYPT) {
81             for (x = 0; x < (ptlen & ~15); x += 16) {
82                 /* ctr encrypt */
83                 for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
84                     *((LTC_FAST_TYPE*)(&ct[x + y])) = *((LTC_FAST_TYPE*)(&pt[x+y])) ^ *((LTC_FAST_TYPE*)(&gcm->X[y]));
85                     *((LTC_FAST_TYPE*)(&gcm->X[y])) ^= *((LTC_FAST_TYPE*)(&ct[x+y]));
86                 }
87                 /* GMAC it */
88                 gcm->pttotlen += 128;
89                 gcm_mult_h(gcm, gcm->X);
90                 /* increment counter */
91                 for (y = 15; y >= 0; y--) {
92                     if (++gcm->Y[y] & 255) { break; }
93                 }
94                 if ((err = cipher_descriptor[gcm->cipher].ecb_encrypt(gcm->Y, gcm->buf, &gcm->K)) != CRYPT_OK) {
95                     return err;
96                 }
97             }
98         } else {
99             for (x = 0; x < (ptlen & ~15); x += 16) {
100                 /* ctr encrypt */
101                 for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
102                     *((LTC_FAST_TYPE*)(&gcm->X[y])) ^= *((LTC_FAST_TYPE*)(&ct[x+y]));
103                     *((LTC_FAST_TYPE*)(&pt[x + y])) = *((LTC_FAST_TYPE*)(&ct[x+y])) ^ *((LTC_FAST_TYPE*)(&gcm->X[y]));
104                 }
105                 /* GMAC it */
106                 gcm->pttotlen += 128;
107                 gcm_mult_h(gcm, gcm->X);
108                 /* increment counter */
109                 for (y = 15; y >= 0; y--) {
110                     if (++gcm->Y[y] & 255) { break; }

```

```

111         }
112         if ((err = cipher_descriptor[gcm->cipher].ecb_encrypt(gcm->Y, gcm->buf, &gcm->K)) != CRYPT_OK)
113             return err;
114         }
115     }
116 }
117 }
118 #endif
119
120 /* process text */
121 for (; x < ptext; x++) {
122     if (gcm->buflen == 16) {
123         gcm->pttotlen += 128;
124         gcm_mult_h(gcm, gcm->X);
125
126         /* increment counter */
127         for (y = 15; y >= 0; y--) {
128             if (++gcm->Y[y] & 255) { break; }
129         }
130         if ((err = cipher_descriptor[gcm->cipher].ecb_encrypt(gcm->Y, gcm->buf, &gcm->K)) != CRYPT_OK)
131             return err;
132     }
133     gcm->buflen = 0;
134 }
135
136 if (direction == GCM_ENCRYPT) {
137     b = ct[x] = pt[x] ^ gcm->buf[gcm->buflen];
138 } else {
139     b = ct[x];
140     pt[x] = ct[x] ^ gcm->buf[gcm->buflen];
141 }
142 gcm->X[gcm->buflen++] ^= b;
143 }
144
145 return CRYPT_OK;
146 }

```

Here is the call graph for this function:

## 5.39 encauth/gcm/gcm\_reset.c File Reference

### 5.39.1 Detailed Description

GCM implementation, reset a used state so it can accept IV data, by Tom St Denis.

Definition in file [gcm\\_reset.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for gcm\_reset.c:

### Functions

- [int gcm\\_reset](#) (gcm\_state \*gcm)  
*Reset a GCM state to as if you just called [gcm\\_init\(\)](#).*

### 5.39.2 Function Documentation

#### 5.39.2.1 int gcm\_reset (gcm\_state \* gcm)

Reset a GCM state to as if you just called [gcm\\_init\(\)](#).

This saves the initialization time.

#### Parameters:

*gcm* The GCM state to reset

#### Returns:

CRYPT\_OK on success

Definition at line 25 of file gcm\_reset.c.

References CRYPT\_OK, LTC\_ARGCHK, and zeromem().

```
26 {
27     LTC_ARGCHK(gcm != NULL);
28
29     zeromem(gcm->buf, sizeof(gcm->buf));
30     zeromem(gcm->X, sizeof(gcm->X));
31     gcm->mode = GCM_MODE_IV;
32     gcm->ivmode = 0;
33     gcm->buflen = 0;
34     gcm->totlen = 0;
35     gcm->pttotlen = 0;
36
37     return CRYPT_OK;
38 }
```

Here is the call graph for this function:

## 5.40 encauth/gcm/gcm\_test.c File Reference

### 5.40.1 Detailed Description

GCM implementation, testing, by Tom St Denis.

Definition in file [gcm\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for gcm\_test.c:

### Functions

- [int gcm\\_test](#) (void)  
*Test the GCM code.*

### 5.40.2 Function Documentation

#### 5.40.2.1 int gcm\_test (void)

Test the GCM code.

#### Returns:

CRYPT\_OK on success

Definition at line 24 of file gcm\_test.c.

References CRYPT\_NOP, CRYPT\_OK, find\_cipher(), gcm\_memory(), and K.

```
25 {
26 #ifndef LTC_TEST
27     return CRYPT_NOP;
28 #else
29     static const struct {
30         unsigned char K[32];
31         int          keylen;
32         unsigned char P[64];
33         unsigned long ptlen;
34         unsigned char A[64];
35         unsigned long alen;
36         unsigned char IV[64];
37         unsigned long IVlen;
38         unsigned char C[64];
39         unsigned char T[16];
40     } tests[] = {
41
42 /* test case #1 */
43 {
44     /* key */
45     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
46       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
47     16,
48
49     /* plaintext */
50     { 0 },
51     0,
52 }
```

```

53  /* AAD data */
54  { 0 },
55  0,
56
57  /* IV */
58  { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
59    0x00, 0x00, 0x00, 0x00 },
60  12,
61
62  /* ciphertext */
63  { 0 },
64
65  /* tag */
66  { 0x58, 0xe2, 0xfc, 0xce, 0xfa, 0x7e, 0x30, 0x61,
67    0x36, 0x7f, 0x1d, 0x57, 0xa4, 0xe7, 0x45, 0x5a }
68 },
69
70 /* test case #2 */
71 {
72   /* key */
73   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
74     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
75   16,
76
77   /* PT */
78   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
79     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
80   16,
81
82   /* ADATA */
83   { 0 },
84   0,
85
86   /* IV */
87   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
88     0x00, 0x00, 0x00, 0x00 },
89   12,
90
91   /* CT */
92   { 0x03, 0x88, 0xda, 0xce, 0x60, 0xb6, 0xa3, 0x92,
93     0xf3, 0x28, 0xc2, 0xb9, 0x71, 0xb2, 0xfe, 0x78 },
94
95   /* TAG */
96   { 0xab, 0x6e, 0x47, 0xd4, 0x2c, 0xec, 0x13, 0xbd,
97     0xf5, 0x3a, 0x67, 0xb2, 0x12, 0x57, 0xbd, 0xdf }
98 },
99
100 /* test case #3 */
101 {
102   /* key */
103   { 0xfe, 0xff, 0xe9, 0x92, 0x86, 0x65, 0x73, 0x1c,
104     0x6d, 0x6a, 0x8f, 0x94, 0x67, 0x30, 0x83, 0x08 },
105   16,
106
107   /* PT */
108   { 0xd9, 0x31, 0x32, 0x25, 0xf8, 0x84, 0x06, 0xe5,
109     0xa5, 0x59, 0x09, 0xc5, 0xaf, 0xf5, 0x26, 0x9a,
110     0x86, 0xa7, 0xa9, 0x53, 0x15, 0x34, 0xf7, 0xda,
111     0x2e, 0x4c, 0x30, 0x3d, 0x8a, 0x31, 0x8a, 0x72,
112     0x1c, 0x3c, 0x0c, 0x95, 0x95, 0x68, 0x09, 0x53,
113     0x2f, 0xcf, 0x0e, 0x24, 0x49, 0xa6, 0xb5, 0x25,
114     0xb1, 0x6a, 0xed, 0xf5, 0xaa, 0x0d, 0xe6, 0x57,
115     0xba, 0x63, 0x7b, 0x39, 0x1a, 0xaf, 0xd2, 0x55 },
116   64,
117
118   /* ADATA */
119   { 0 },

```

```
120 0,
121
122 /* IV */
123 { 0xca, 0xfe, 0xba, 0xbe, 0xfa, 0xce, 0xdb, 0xad,
124   0xde, 0xca, 0xf8, 0x88, },
125 12,
126
127 /* CT */
128 { 0x42, 0x83, 0x1e, 0xc2, 0x21, 0x77, 0x74, 0x24,
129   0x4b, 0x72, 0x21, 0xb7, 0x84, 0xd0, 0xd4, 0x9c,
130   0xe3, 0xaa, 0x21, 0x2f, 0x2c, 0x02, 0xa4, 0xe0,
131   0x35, 0xc1, 0x7e, 0x23, 0x29, 0xac, 0xa1, 0x2e,
132   0x21, 0xd5, 0x14, 0xb2, 0x54, 0x66, 0x93, 0x1c,
133   0x7d, 0x8f, 0x6a, 0x5a, 0xac, 0x84, 0xaa, 0x05,
134   0x1b, 0xa3, 0x0b, 0x39, 0x6a, 0x0a, 0xac, 0x97,
135   0x3d, 0x58, 0xe0, 0x91, 0x47, 0x3f, 0x59, 0x85, },
136
137 /* TAG */
138 { 0x4d, 0x5c, 0x2a, 0xf3, 0x27, 0xcd, 0x64, 0xa6,
139   0x2c, 0xf3, 0x5a, 0xbd, 0x2b, 0xa6, 0xfa, 0xb4, }
140 },
141
142 /* test case #4 */
143 {
144   /* key */
145   { 0xfe, 0xff, 0xe9, 0x92, 0x86, 0x65, 0x73, 0x1c,
146     0x6d, 0x6a, 0x8f, 0x94, 0x67, 0x30, 0x83, 0x08, },
147   16,
148
149   /* PT */
150   { 0xd9, 0x31, 0x32, 0x25, 0xf8, 0x84, 0x06, 0xe5,
151     0xa5, 0x59, 0x09, 0xc5, 0xaf, 0xf5, 0x26, 0x9a,
152     0x86, 0xa7, 0xa9, 0x53, 0x15, 0x34, 0xf7, 0xda,
153     0x2e, 0x4c, 0x30, 0x3d, 0x8a, 0x31, 0x8a, 0x72,
154     0x1c, 0x3c, 0x0c, 0x95, 0x95, 0x68, 0x09, 0x53,
155     0x2f, 0xcf, 0x0e, 0x24, 0x49, 0xa6, 0xb5, 0x25,
156     0xb1, 0x6a, 0xed, 0xf5, 0xaa, 0x0d, 0xe6, 0x57,
157     0xba, 0x63, 0x7b, 0x39, },
158   60,
159
160   /* ADATA */
161   { 0xfe, 0xed, 0xfa, 0xce, 0xde, 0xad, 0xbe, 0xef,
162     0xfe, 0xed, 0xfa, 0xce, 0xde, 0xad, 0xbe, 0xef,
163     0xab, 0xad, 0xda, 0xd2, },
164   20,
165
166   /* IV */
167   { 0xca, 0xfe, 0xba, 0xbe, 0xfa, 0xce, 0xdb, 0xad,
168     0xde, 0xca, 0xf8, 0x88, },
169   12,
170
171   /* CT */
172   { 0x42, 0x83, 0x1e, 0xc2, 0x21, 0x77, 0x74, 0x24,
173     0x4b, 0x72, 0x21, 0xb7, 0x84, 0xd0, 0xd4, 0x9c,
174     0xe3, 0xaa, 0x21, 0x2f, 0x2c, 0x02, 0xa4, 0xe0,
175     0x35, 0xc1, 0x7e, 0x23, 0x29, 0xac, 0xa1, 0x2e,
176     0x21, 0xd5, 0x14, 0xb2, 0x54, 0x66, 0x93, 0x1c,
177     0x7d, 0x8f, 0x6a, 0x5a, 0xac, 0x84, 0xaa, 0x05,
178     0x1b, 0xa3, 0x0b, 0x39, 0x6a, 0x0a, 0xac, 0x97,
179     0x3d, 0x58, 0xe0, 0x91, },
180
181   /* TAG */
182   { 0x5b, 0xc9, 0x4f, 0xbc, 0x32, 0x21, 0xa5, 0xdb,
183     0x94, 0xfa, 0xe9, 0x5a, 0xe7, 0x12, 0x1a, 0x47, }
184 },
185 },
186
```

```
187 /* test case #5 */
188 {
189     /* key */
190     { 0xfe, 0xff, 0xe9, 0x92, 0x86, 0x65, 0x73, 0x1c,
191       0x6d, 0x6a, 0x8f, 0x94, 0x67, 0x30, 0x83, 0x08, },
192     16,
193
194     /* PT */
195     { 0xd9, 0x31, 0x32, 0x25, 0xf8, 0x84, 0x06, 0xe5,
196       0xa5, 0x59, 0x09, 0xc5, 0xaf, 0xf5, 0x26, 0x9a,
197       0x86, 0xa7, 0xa9, 0x53, 0x15, 0x34, 0xf7, 0xda,
198       0x2e, 0x4c, 0x30, 0x3d, 0x8a, 0x31, 0x8a, 0x72,
199       0x1c, 0x3c, 0x0c, 0x95, 0x95, 0x68, 0x09, 0x53,
200       0x2f, 0xcf, 0x0e, 0x24, 0x49, 0xa6, 0xb5, 0x25,
201       0xb1, 0x6a, 0xed, 0xf5, 0xaa, 0x0d, 0xe6, 0x57,
202       0xba, 0x63, 0x7b, 0x39, },
203     60,
204
205     /* ADATA */
206     { 0xfe, 0xed, 0xfa, 0xce, 0xde, 0xad, 0xbe, 0xef,
207       0xfe, 0xed, 0xfa, 0xce, 0xde, 0xad, 0xbe, 0xef,
208       0xab, 0xad, 0xda, 0xd2, },
209     20,
210
211     /* IV */
212     { 0xca, 0xfe, 0xba, 0xbe, 0xfa, 0xce, 0xdb, 0xad, },
213     8,
214
215     /* CT */
216     { 0x61, 0x35, 0x3b, 0x4c, 0x28, 0x06, 0x93, 0x4a,
217       0x77, 0x7f, 0xf5, 0x1f, 0xa2, 0x2a, 0x47, 0x55,
218       0x69, 0x9b, 0x2a, 0x71, 0x4f, 0xcd, 0xc6, 0xf8,
219       0x37, 0x66, 0xe5, 0xf9, 0x7b, 0x6c, 0x74, 0x23,
220       0x73, 0x80, 0x69, 0x00, 0xe4, 0x9f, 0x24, 0xb2,
221       0x2b, 0x09, 0x75, 0x44, 0xd4, 0x89, 0x6b, 0x42,
222       0x49, 0x89, 0xb5, 0xe1, 0xeb, 0xac, 0x0f, 0x07,
223       0xc2, 0x3f, 0x45, 0x98, },
224
225     /* TAG */
226     { 0x36, 0x12, 0xd2, 0xe7, 0x9e, 0x3b, 0x07, 0x85,
227       0x56, 0x1b, 0xe1, 0x4a, 0xac, 0xa2, 0xfc, 0xcb, }
228 },
229
230 /* test case #6 */
231 {
232     /* key */
233     { 0xfe, 0xff, 0xe9, 0x92, 0x86, 0x65, 0x73, 0x1c,
234       0x6d, 0x6a, 0x8f, 0x94, 0x67, 0x30, 0x83, 0x08, },
235     16,
236
237     /* PT */
238     { 0xd9, 0x31, 0x32, 0x25, 0xf8, 0x84, 0x06, 0xe5,
239       0xa5, 0x59, 0x09, 0xc5, 0xaf, 0xf5, 0x26, 0x9a,
240       0x86, 0xa7, 0xa9, 0x53, 0x15, 0x34, 0xf7, 0xda,
241       0x2e, 0x4c, 0x30, 0x3d, 0x8a, 0x31, 0x8a, 0x72,
242       0x1c, 0x3c, 0x0c, 0x95, 0x95, 0x68, 0x09, 0x53,
243       0x2f, 0xcf, 0x0e, 0x24, 0x49, 0xa6, 0xb5, 0x25,
244       0xb1, 0x6a, 0xed, 0xf5, 0xaa, 0x0d, 0xe6, 0x57,
245       0xba, 0x63, 0x7b, 0x39, },
246     60,
247
248     /* ADATA */
249     { 0xfe, 0xed, 0xfa, 0xce, 0xde, 0xad, 0xbe, 0xef,
250       0xfe, 0xed, 0xfa, 0xce, 0xde, 0xad, 0xbe, 0xef,
251       0xab, 0xad, 0xda, 0xd2, },
252     20,
253 }
```



```

254  /* IV */
255  { 0x93, 0x13, 0x22, 0x5d, 0xf8, 0x84, 0x06, 0xe5,
256    0x55, 0x90, 0x9c, 0x5a, 0xff, 0x52, 0x69, 0xaa,
257    0x6a, 0x7a, 0x95, 0x38, 0x53, 0x4f, 0x7d, 0xa1,
258    0xe4, 0xc3, 0x03, 0xd2, 0xa3, 0x18, 0xa7, 0x28,
259    0xc3, 0xc0, 0xc9, 0x51, 0x56, 0x80, 0x95, 0x39,
260    0xfc, 0xf0, 0xe2, 0x42, 0x9a, 0x6b, 0x52, 0x54,
261    0x16, 0xae, 0xdb, 0xf5, 0xa0, 0xde, 0x6a, 0x57,
262    0xa6, 0x37, 0xb3, 0x9b, },
263  60,
264
265  /* CT */
266  { 0x8c, 0xe2, 0x49, 0x98, 0x62, 0x56, 0x15, 0xb6,
267    0x03, 0xa0, 0x33, 0xac, 0xa1, 0x3f, 0xb8, 0x94,
268    0xbe, 0x91, 0x12, 0xa5, 0xc3, 0xa2, 0x11, 0xa8,
269    0xba, 0x26, 0x2a, 0x3c, 0xca, 0x7e, 0x2c, 0xa7,
270    0x01, 0xe4, 0xa9, 0xa4, 0xfb, 0xa4, 0x3c, 0x90,
271    0xcc, 0xdc, 0xb2, 0x81, 0xd4, 0x8c, 0x7c, 0x6f,
272    0xd6, 0x28, 0x75, 0xd2, 0xac, 0xa4, 0x17, 0x03,
273    0x4c, 0x34, 0xae, 0xe5, },
274
275  /* TAG */
276  { 0x61, 0x9c, 0xc5, 0xae, 0xff, 0xfe, 0x0b, 0xfa,
277    0x46, 0x2a, 0xf4, 0x3c, 0x16, 0x99, 0xd0, 0x50, }
278 },
279
280 #if 0
281
282 /* test case #10 */
283 {
284   { 0xdb, 0xbc, 0x85, 0x66, 0xd6, 0xf5, 0xb1, 0x58,
285     0xda, 0x99, 0xa2, 0xff, 0x2e, 0x01, 0xdd, 0xa6,
286     0x29, 0xb8, 0x9c, 0x34, 0xad, 0x1e, 0x5f, 0xeb,
287     0xa7, 0x0e, 0x7a, 0xae, 0x43, 0x28, 0x28, 0x9c },
288   32,
289
290   { 0xce, 0x20, 0x27, 0xb4, 0x7a, 0x84, 0x32, 0x52,
291     0x01, 0x34, 0x65, 0x83, 0x4d, 0x75, 0xfd, 0x0f },
292   16,
293
294   { 0 },
295   0,
296
297   { 0xcf, 0xc0, 0x6e, 0x72, 0x2b, 0xe9, 0x87, 0xb3,
298     0x76, 0x7f, 0x70, 0xa7, 0xb8, 0x56, 0xb7, 0x74 },
299   16,
300
301   { 0x03, 0x30, 0xea, 0x65, 0xb1, 0xf4, 0x8a, 0xd7,
302     0x18, 0xc3, 0xf1, 0xf3, 0xdc, 0xef, 0xe4, 0x20 },
303
304   { 0xe9, 0xef, 0xa9, 0x97, 0xd0, 0xae, 0x82, 0x42,
305     0x90, 0xbb, 0x5a, 0x66, 0x95, 0xff, 0x2c, 0x7a }
306 }
307
308 #endif
309
310
311 /* rest of test cases are the same except AES key size changes... ignored... */
312 };
313
314 int      idx, err;
315 unsigned long x, y;
316 unsigned char out[2][64], T[2][16];
317
318 /* find aes */
319 idx = find_cipher("aes");
320 if (idx == -1) {
321     idx = find_cipher("rijndael");

```

```

321     if (idx == -1) {
322         return CRYPT_NOP;
323     }
324 }
325
326 for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
327     y = sizeof(T[0]);
328     if ((err = gcm_memory(idx, tests[x].K, tests[x].keylen,
329                          tests[x].IV, tests[x].IVlen,
330                          tests[x].A, tests[x].alen,
331                          (unsigned char*)tests[x].P, tests[x].ptlen,
332                          out[0], T[0], &y, GCM_ENCRYPT)) != CRYPT_OK) {
333         return err;
334     }
335
336     if (XMEMCMP(out[0], tests[x].C, tests[x].ptlen)) {
337 #if 0
338         printf("\nCiphertext wrong %lu\n", x);
339         for (y = 0; y < tests[x].ptlen; y++) {
340             printf("%02x", out[0][y] & 255);
341         }
342         printf("\n");
343 #endif
344         return CRYPT_FAIL_TESTVECTOR;
345     }
346
347     if (XMEMCMP(T[0], tests[x].T, 16)) {
348 #if 0
349         printf("\nTag on plaintext wrong %lu\n", x);
350         for (y = 0; y < 16; y++) {
351             printf("%02x", T[0][y] & 255);
352         }
353         printf("\n");
354 #endif
355         return CRYPT_FAIL_TESTVECTOR;
356     }
357
358     y = sizeof(T[1]);
359     if ((err = gcm_memory(idx, tests[x].K, tests[x].keylen,
360                          tests[x].IV, tests[x].IVlen,
361                          tests[x].A, tests[x].alen,
362                          out[1], tests[x].ptlen,
363                          out[0], T[1], &y, GCM_DECRYPT)) != CRYPT_OK) {
364         return err;
365     }
366
367     if (XMEMCMP(out[1], tests[x].P, tests[x].ptlen)) {
368 #if 0
369         printf("\nplaintext wrong %lu\n", x);
370         for (y = 0; y < tests[x].ptlen; y++) {
371             printf("%02x", out[0][y] & 255);
372         }
373         printf("\n");
374 #endif
375         return CRYPT_FAIL_TESTVECTOR;
376     }
377
378     if (XMEMCMP(T[1], tests[x].T, 16)) {
379 #if 0
380         printf("\nTag on ciphertext wrong %lu\n", x);
381         for (y = 0; y < 16; y++) {
382             printf("%02x", T[1][y] & 255);
383         }
384         printf("\n");
385 #endif
386         return CRYPT_FAIL_TESTVECTOR;
387     }

```

```
388
389     }
390     return CRYPT_OK;
391 #endif
392 }
```

Here is the call graph for this function:

## 5.41 encauth/ocb/ocb\_decrypt.c File Reference

### 5.41.1 Detailed Description

OCB implementation, decrypt data, by Tom St Denis.

Definition in file [ocb\\_decrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ocb\_decrypt.c:

### Functions

- `int ocb_decrypt` (`ocb_state *ocb`, `const unsigned char *ct`, `unsigned char *pt`)  
*Decrypt a block with OCB.*

### 5.41.2 Function Documentation

#### 5.41.2.1 `int ocb_decrypt` (`ocb_state *ocb`, `const unsigned char *ct`, `unsigned char *pt`)

Decrypt a block with OCB.

#### Parameters:

- ocb* The OCB state
- ct* The ciphertext (length of the block size of the block cipher)
- pt* [out] The plaintext (length of ct)

#### Returns:

CRYPT\_OK if successful

Definition at line 27 of file ocb\_decrypt.c.

References `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `ltc_cipher_descriptor::ecb_decrypt`, `LTC_ARGCHK`, `MAXBLOCKSIZE`, and `ocb_shift_xor()`.

Referenced by `ocb_decrypt_verify_memory()`.

```
28 {
29     unsigned char Z[MAXBLOCKSIZE], tmp[MAXBLOCKSIZE];
30     int err, x;
31
32     LTC_ARGCHK(ocb != NULL);
33     LTC_ARGCHK(pt != NULL);
34     LTC_ARGCHK(ct != NULL);
35
36     /* check if valid cipher */
37     if ((err = cipher_is_valid(ocb->cipher)) != CRYPT_OK) {
38         return err;
39     }
40     LTC_ARGCHK(cipher_descriptor[ocb->cipher].ecb_decrypt != NULL);
41
42     /* check length */
43     if (ocb->block_len != cipher_descriptor[ocb->cipher].block_length) {
```

```
44     return CRYPT_INVALID_ARG;
45 }
46
47 /* Get Z[i] value */
48 ocb_shift_xor(ocb, Z);
49
50 /* xor ct in, encrypt, xor Z out */
51 for (x = 0; x < ocb->block_len; x++) {
52     tmp[x] = ct[x] ^ Z[x];
53 }
54 if ((err = cipher_descriptor[ocb->cipher].ecb_decrypt(tmp, pt, &ocb->key)) != CRYPT_OK) {
55     return err;
56 }
57 for (x = 0; x < ocb->block_len; x++) {
58     pt[x] ^= Z[x];
59 }
60
61 /* compute checksum */
62 for (x = 0; x < ocb->block_len; x++) {
63     ocb->checksum[x] ^= pt[x];
64 }
65
66
67 #ifdef LTC_CLEAN_STACK
68     zeromem(Z, sizeof(Z));
69     zeromem(tmp, sizeof(tmp));
70 #endif
71     return CRYPT_OK;
72 }
```

Here is the call graph for this function:

## 5.42 encauth/ocb/ocb\_decrypt\_verify\_memory.c File Reference

### 5.42.1 Detailed Description

OCB implementation, helper to decrypt block of memory, by Tom St Denis.

Definition in file [ocb\\_decrypt\\_verify\\_memory.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ocb\_decrypt\_verify\_memory.c:

### Functions

- [int ocb\\_decrypt\\_verify\\_memory](#) (int cipher, const unsigned char \*key, unsigned long keylen, const unsigned char \*nonce, const unsigned char \*ct, unsigned long ctlen, unsigned char \*pt, const unsigned char \*tag, unsigned long taglen, int \*stat)

*Decrypt and compare the tag with OCB.*

### 5.42.2 Function Documentation

- 5.42.2.1 int ocb\_decrypt\_verify\_memory** (int *cipher*, const unsigned char \* *key*, unsigned long *keylen*, const unsigned char \* *nonce*, const unsigned char \* *ct*, unsigned long *ctlen*, unsigned char \* *pt*, const unsigned char \* *tag*, unsigned long *taglen*, int \* *stat*)

Decrypt and compare the tag with OCB.

#### Parameters:

*cipher* The index of the cipher desired  
*key* The secret key  
*keylen* The length of the secret key (octets)  
*nonce* The session nonce (length of the block size of the block cipher)  
*ct* The ciphertext  
*ctlen* The length of the ciphertext (octets)  
*pt* [out] The plaintext  
*tag* The tag to compare against  
*taglen* The length of the tag (octets)  
*stat* [out] The result of the tag comparison (1==valid, 0==invalid)

#### Returns:

CRYPT\_OK if successful regardless of the tag comparison

Definition at line 34 of file ocb\_decrypt\_verify\_memory.c.

References CRYPT\_MEM, CRYPT\_OK, LTC\_ARGCHK, ocb\_decrypt(), ocb\_done\_decrypt(), ocb\_init(), XFREE, XMALLOC, and zeromem().

```
41 {
42     int err;
43     ocb_state *ocb;
```

```
44
45     LTC_ARGCHK(key      != NULL);
46     LTC_ARGCHK(nonce    != NULL);
47     LTC_ARGCHK(pt       != NULL);
48     LTC_ARGCHK(ct       != NULL);
49     LTC_ARGCHK(tag      != NULL);
50     LTC_ARGCHK(stat     != NULL);
51
52     /* allocate memory */
53     ocb = XMALLOC(sizeof(ocb_state));
54     if (ocb == NULL) {
55         return CRYPT_MEM;
56     }
57
58     if ((err = ocb_init(ocb, cipher, key, keylen, nonce)) != CRYPT_OK) {
59         goto LBL_ERR;
60     }
61
62     while (ctlen > (unsigned long)ocb->block_len) {
63         if ((err = ocb_decrypt(ocb, ct, pt)) != CRYPT_OK) {
64             goto LBL_ERR;
65         }
66         ctlen  -= ocb->block_len;
67         pt     += ocb->block_len;
68         ct     += ocb->block_len;
69     }
70
71     err = ocb_done_decrypt(ocb, ct, ctlen, pt, tag, taglen, stat);
72 LBL_ERR:
73 #ifdef LTC_CLEAN_STACK
74     zeromem(ocb, sizeof(ocb_state));
75 #endif
76
77     XFREE(ocb);
78
79     return err;
80 }
```

Here is the call graph for this function:

## 5.43 encauth/ocb/ocb\_done\_decrypt.c File Reference

### 5.43.1 Detailed Description

OCB implementation, terminate decryption, by Tom St Denis.

Definition in file [ocb\\_done\\_decrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ocb\_done\_decrypt.c:

### Functions

- `int ocb_done_decrypt` (`ocb_state *ocb`, `const unsigned char *ct`, `unsigned long ctlen`, `unsigned char *pt`, `const unsigned char *tag`, `unsigned long taglen`, `int *stat`)

*Terminate a decrypting OCB state.*

### 5.43.2 Function Documentation

#### 5.43.2.1 `int ocb_done_decrypt (ocb_state * ocb, const unsigned char * ct, unsigned long ctlen, unsigned char * pt, const unsigned char * tag, unsigned long taglen, int * stat)`

Terminate a decrypting OCB state.

#### Parameters:

- ocb* The OCB state
- ct* The ciphertext (if any)
- ctlen* The length of the ciphertext (octets)
- pt* [out] The plaintext
- tag* The authentication tag (to compare against)
- taglen* The length of the authentication tag provided
- stat* [out] The result of the tag comparison

#### Returns:

- CRYPT\_OK if the process was successful regardless if the tag is valid

Definition at line 31 of file ocb\_done\_decrypt.c.

References CRYPT\_MEM, CRYPT\_OK, LTC\_ARGCHK, MAXBLOCKSIZE, s\_ocb\_done(), XFREE, XMALLOC, XMEMCMP, and zeromem().

Referenced by ocb\_decrypt\_verify\_memory().

```
35 {
36     int err;
37     unsigned char *tagbuf;
38     unsigned long tagbuflen;
39
40     LTC_ARGCHK(ocb != NULL);
41     LTC_ARGCHK(pt != NULL);
42     LTC_ARGCHK(ct != NULL);
```



```
43     LTC_ARGCHK(tag != NULL);
44     LTC_ARGCHK(stat != NULL);
45
46     /* default to failed */
47     *stat = 0;
48
49     /* allocate memory */
50     tagbuf = XMALLOC(MAXBLOCKSIZE);
51     if (tagbuf == NULL) {
52         return CRYPT_MEM;
53     }
54
55     tagbuflen = MAXBLOCKSIZE;
56     if ((err = s_ocb_done(ocb, ct, ctlen, pt, tagbuf, &tagbuflen, 1)) != CRYPT_OK) {
57         goto LBL_ERR;
58     }
59
60     if (taglen <= tagbuflen && XMEMCMP(tagbuf, tag, taglen) == 0) {
61         *stat = 1;
62     }
63
64     err = CRYPT_OK;
65 LBL_ERR:
66 #ifdef LTC_CLEAN_STACK
67     zeromem(tagbuf, MAXBLOCKSIZE);
68 #endif
69
70     XFREE(tagbuf);
71
72     return err;
73 }
```

Here is the call graph for this function:

## 5.44 encauth/ocb/ocb\_done\_encrypt.c File Reference

### 5.44.1 Detailed Description

OCB implementation, terminate encryption, by Tom St Denis.

Definition in file [ocb\\_done\\_encrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ocb\_done\_encrypt.c:

### Functions

- `int ocb_done_encrypt` (`ocb_state *ocb`, `const unsigned char *pt`, `unsigned long ptlen`, `unsigned char *ct`, `unsigned char *tag`, `unsigned long *taglen`)

*Terminate an encryption OCB state.*

### 5.44.2 Function Documentation

#### 5.44.2.1 `int ocb_done_encrypt` (`ocb_state *ocb`, `const unsigned char *pt`, `unsigned long ptlen`, `unsigned char *ct`, `unsigned char *tag`, `unsigned long *taglen`)

Terminate an encryption OCB state.

#### Parameters:

- ocb* The OCB state
- pt* Remaining plaintext (if any)
- ptlen* The length of the plaintext (octets)
- ct* [out] The ciphertext (if any)
- tag* [out] The tag for the OCB stream
- taglen* [in/out] The max size and resulting size of the tag

#### Returns:

- CRYPT\_OK if successful

Definition at line 30 of file `ocb_done_encrypt.c`.

References `LTC_ARGCHK`, and `s_ocb_done()`.

Referenced by `ocb_encrypt_authenticate_memory()`.

```
32 {
33     LTC_ARGCHK(ocb      != NULL);
34     LTC_ARGCHK(pt       != NULL);
35     LTC_ARGCHK(ct       != NULL);
36     LTC_ARGCHK(tag      != NULL);
37     LTC_ARGCHK(taglen   != NULL);
38     return s_ocb_done(ocb, pt, ptlen, ct, tag, taglen, 0);
39 }
```

Here is the call graph for this function:

## 5.45 encauth/ocb/ocb\_encrypt.c File Reference

### 5.45.1 Detailed Description

OCB implementation, encrypt data, by Tom St Denis.

Definition in file [ocb\\_encrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ocb\_encrypt.c:

### Functions

- `int ocb_encrypt(ocb_state *ocb, const unsigned char *pt, unsigned char *ct)`  
*Encrypt a block of data with OCB.*

### 5.45.2 Function Documentation

#### 5.45.2.1 `int ocb_encrypt(ocb_state *ocb, const unsigned char *pt, unsigned char *ct)`

Encrypt a block of data with OCB.

#### Parameters:

- ocb* The OCB state
- pt* The plaintext (length of the block size of the block cipher)
- ct* [out] The ciphertext (same size as the pt)

#### Returns:

CRYPT\_OK if successful

Definition at line 27 of file ocb\_encrypt.c.

References `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `LTC_ARGCHK`, and `MAXBLOCKSIZE`.

Referenced by `ocb_encrypt_authenticate_memory()`.

```
28 {
29     unsigned char Z[MAXBLOCKSIZE], tmp[MAXBLOCKSIZE];
30     int err, x;
31
32     LTC_ARGCHK(ocb != NULL);
33     LTC_ARGCHK(pt != NULL);
34     LTC_ARGCHK(ct != NULL);
35     if ((err = cipher_is_valid(ocb->cipher)) != CRYPT_OK) {
36         return err;
37     }
38     if (ocb->block_len != cipher_descriptor[ocb->cipher].block_length) {
39         return CRYPT_INVALID_ARG;
40     }
41
42     /* compute checksum */
43     for (x = 0; x < ocb->block_len; x++) {
44         ocb->checksum[x] ^= pt[x];
45     }
```

```
45     }
46
47     /* Get Z[i] value */
48     ocb_shift_xor(ocb, Z);
49
50     /* xor pt in, encrypt, xor Z out */
51     for (x = 0; x < ocb->block_len; x++) {
52         tmp[x] = pt[x] ^ Z[x];
53     }
54     if ((err = cipher_descriptor[ocb->cipher].ecb_encrypt(tmp, ct, &ocb->key)) != CRYPT_OK) {
55         return err;
56     }
57     for (x = 0; x < ocb->block_len; x++) {
58         ct[x] ^= Z[x];
59     }
60
61 #ifdef LTC_CLEAN_STACK
62     zeromem(Z, sizeof(Z));
63     zeromem(tmp, sizeof(tmp));
64 #endif
65     return CRYPT_OK;
66 }
```

Here is the call graph for this function:

## 5.46 `enauth/ocb/ocb_encrypt_authenticate_memory.c` File Reference

### 5.46.1 Detailed Description

OCB implementation, encrypt block of memory, by Tom St Denis.

Definition in file [ocb\\_encrypt\\_authenticate\\_memory.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `ocb_encrypt_authenticate_memory.c`:

### Functions

- [int ocb\\_encrypt\\_authenticate\\_memory](#) (int cipher, const unsigned char \*key, unsigned long keylen, const unsigned char \*nonce, const unsigned char \*pt, unsigned long ptlen, unsigned char \*ct, unsigned char \*tag, unsigned long \*taglen)

*Encrypt and generate an authentication code for a buffer of memory.*

### 5.46.2 Function Documentation

**5.46.2.1** `int ocb_encrypt_authenticate_memory` (int *cipher*, const unsigned char \* *key*, unsigned long *keylen*, const unsigned char \* *nonce*, const unsigned char \* *pt*, unsigned long *ptlen*, unsigned char \* *ct*, unsigned char \* *tag*, unsigned long \* *taglen*)

Encrypt and generate an authentication code for a buffer of memory.

#### Parameters:

*cipher* The index of the cipher desired

*key* The secret key

*keylen* The length of the secret key (octets)

*nonce* The session nonce (length of the block ciphers block size)

*pt* The plaintext

*ptlen* The length of the plaintext (octets)

*ct* [out] The ciphertext

*tag* [out] The authentication tag

*taglen* [in/out] The max size and resulting size of the authentication tag

#### Returns:

CRYPT\_OK if successful

Definition at line 33 of file `ocb_encrypt_authenticate_memory.c`.

References CRYPT\_MEM, CRYPT\_OK, LTC\_ARGCHK, `ocb_done_encrypt()`, `ocb_encrypt()`, `ocb_init()`, XFREE, XMALLOC, and `zeromem()`.

Referenced by `ocb_test()`.

```
39 {
40     int err;
41     ocb_state *ocb;
42
43     LTC_ARGCHK(key    != NULL);
44     LTC_ARGCHK(nonce  != NULL);
45     LTC_ARGCHK(pt     != NULL);
46     LTC_ARGCHK(ct     != NULL);
47     LTC_ARGCHK(tag    != NULL);
48     LTC_ARGCHK(taglen != NULL);
49
50     /* allocate ram */
51     ocb = XMALLOC(sizeof(ocb_state));
52     if (ocb == NULL) {
53         return CRYPT_MEM;
54     }
55
56     if ((err = ocb_init(ocb, cipher, key, keylen, nonce)) != CRYPT_OK) {
57         goto LBL_ERR;
58     }
59
60     while (ptlen > (unsigned long)ocb->block_len) {
61         if ((err = ocb_encrypt(ocb, pt, ct)) != CRYPT_OK) {
62             goto LBL_ERR;
63         }
64         ptlen -= ocb->block_len;
65         pt    += ocb->block_len;
66         ct    += ocb->block_len;
67     }
68
69     err = ocb_done_encrypt(ocb, pt, ptlen, ct, tag, taglen);
70 LBL_ERR:
71 #ifdef LTC_CLEAN_STACK
72     zeromem(ocb, sizeof(ocb_state));
73 #endif
74
75     XFREE(ocb);
76
77     return err;
78 }
```

Here is the call graph for this function:

## 5.47 encauth/ocb/ocb\_init.c File Reference

### 5.47.1 Detailed Description

OCB implementation, initialize state, by Tom St Denis.

Definition in file [ocb\\_init.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ocb\_init.c:

### Functions

- `int ocb_init (ocb_state *ocb, int cipher, const unsigned char *key, unsigned long keylen, const unsigned char *nonce)`

*Initialize an OCB context.*

### Variables

- struct {  
     int `len`  
     unsigned char `poly_div` [MAXBLOCKSIZE]  
     unsigned char `poly_mul` [MAXBLOCKSIZE]  
     int `code`  
     int `value`  
     } `polys` []

### 5.47.2 Function Documentation

#### 5.47.2.1 `int ocb_init (ocb_state * ocb, int cipher, const unsigned char * key, unsigned long keylen, const unsigned char * nonce)`

Initialize an OCB context.

#### Parameters:

- ocb*** [out] The destination of the OCB state
- cipher*** The index of the desired cipher
- key*** The secret key
- keylen*** The length of the secret key (octets)
- nonce*** The session nonce (length of the block size of the cipher)

#### Returns:

CRYPT\_OK if successful

Definition at line 47 of file ocb\_init.c.

References `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_OK`, `len`, `LTC_ARGCHK`, `poly`, and `polys`.

Referenced by `ocb_decrypt_verify_memory()`, and `ocb_encrypt_authenticate_memory()`.

```

49 {
50     int poly, x, y, m, err;
51
52     LTC_ARGCHK(ocb != NULL);
53     LTC_ARGCHK(key != NULL);
54     LTC_ARGCHK(nonce != NULL);
55
56     /* valid cipher? */
57     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
58         return err;
59     }
60
61     /* determine which polys to use */
62     ocb->block_len = cipher_descriptor[cipher].block_length;
63     for (poly = 0; poly < (int)(sizeof(polys)/sizeof(polys[0])); poly++) {
64         if (polys[poly].len == ocb->block_len) {
65             break;
66         }
67     }
68     if (polys[poly].len != ocb->block_len) {
69         return CRYPT_INVALID_ARG;
70     }
71
72     /* schedule the key */
73     if ((err = cipher_descriptor[cipher].setup(key, keylen, 0, &ocb->key)) != CRYPT_OK) {
74         return err;
75     }
76
77     /* find L = E[0] */
78     zeromem(ocb->L, ocb->block_len);
79     if ((err = cipher_descriptor[cipher].ecb_encrypt(ocb->L, ocb->L, &ocb->key)) != CRYPT_OK) {
80         return err;
81     }
82
83     /* find R = E[N xor L] */
84     for (x = 0; x < ocb->block_len; x++) {
85         ocb->R[x] = ocb->L[x] ^ nonce[x];
86     }
87     if ((err = cipher_descriptor[cipher].ecb_encrypt(ocb->R, ocb->R, &ocb->key)) != CRYPT_OK) {
88         return err;
89     }
90
91     /* find Ls[i] = L << i for i == 0..31 */
92     XMEMCPY(ocb->Ls[0], ocb->L, ocb->block_len);
93     for (x = 1; x < 32; x++) {
94         m = ocb->Ls[x-1][0] >> 7;
95         for (y = 0; y < ocb->block_len-1; y++) {
96             ocb->Ls[x][y] = ((ocb->Ls[x-1][y] << 1) | (ocb->Ls[x-1][y+1] >> 7)) & 255;
97         }
98         ocb->Ls[x][ocb->block_len-1] = (ocb->Ls[x-1][ocb->block_len-1] << 1) & 255;
99
100         if (m == 1) {
101             for (y = 0; y < ocb->block_len; y++) {
102                 ocb->Ls[x][y] ^= polys[poly].poly_mul[y];
103             }
104         }
105     }
106
107     /* find Lr = L / x */
108     m = ocb->L[ocb->block_len-1] & 1;
109
110     /* shift right */
111     for (x = ocb->block_len - 1; x > 0; x--) {
112         ocb->Lr[x] = ((ocb->L[x] >> 1) | (ocb->L[x-1] << 7)) & 255;
113     }
114     ocb->Lr[0] = ocb->L[0] >> 1;
115

```



```

116     if (m == 1) {
117         for (x = 0; x < ocb->block_len; x++) {
118             ocb->Lr[x] ^= polys[poly].poly_div[x];
119         }
120     }
121
122     /* set Li, checksum */
123     zeromem(ocb->Li, ocb->block_len);
124     zeromem(ocb->checksum, ocb->block_len);
125
126     /* set other params */
127     ocb->block_index = 1;
128     ocb->cipher = cipher;
129
130     return CRYPT_OK;
131 }

```

Here is the call graph for this function:

### 5.47.3 Variable Documentation

#### 5.47.3.1 int [len](#)

Definition at line 21 of file ocb\_init.c.

Referenced by ccm\_memory(), der\_decode\_ia5\_string(), der\_decode\_object\_identifier(), der\_decode\_octet\_string(), der\_decode\_printable\_string(), der\_decode\_sequence\_flexi(), der\_decode\_short\_integer(), der\_encode\_bit\_string(), der\_encode\_ia5\_string(), der\_encode\_octet\_string(), der\_encode\_printable\_string(), der\_encode\_short\_integer(), der\_length\_integer(), der\_length\_short\_integer(), eax\_done(), eax\_init(), eax\_test(), ocb\_init(), ocb\_test(), omac\_init(), omac\_test(), pmac\_init(), pmac\_test(), sober128\_test(), and whirlpool\_test().

#### 5.47.3.2 unsigned char [poly\\_div](#)[MAXBLOCKSIZE]

Definition at line 22 of file ocb\_init.c.

#### 5.47.3.3 unsigned char [poly\\_mul](#)[MAXBLOCKSIZE]

Definition at line 22 of file ocb\_init.c.

#### 5.47.3.4 const { ... } [polys](#)[ ] [static]

Referenced by ocb\_init(), and pmac\_init().

## 5.48 encauth/ocb/ocb\_ntz.c File Reference

### 5.48.1 Detailed Description

OCB implementation, internal function, by Tom St Denis.

Definition in file [ocb\\_ntz.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ocb\_ntz.c:

### Functions

- [int ocb\\_ntz](#) (unsigned long x)  
*Returns the number of leading zero bits [from lsb up].*

### 5.48.2 Function Documentation

#### 5.48.2.1 int ocb\_ntz (unsigned long x)

Returns the number of leading zero bits [from lsb up].

#### Parameters:

*x* The 32-bit value to observe

#### Returns:

The number of bits [from the lsb up] that are zero

Definition at line 26 of file ocb\_ntz.c.

References [c](#).

Referenced by [ocb\\_shift\\_xor\(\)](#).

```
27 {
28     int c;
29     x &= 0xFFFFFFFFFUL;
30     c = 0;
31     while ((x & 1) == 0) {
32         ++c;
33         x >>= 1;
34     }
35     return c;
36 }
```

## 5.49 encauth/ocb/ocb\_shift\_xor.c File Reference

### 5.49.1 Detailed Description

OCB implementation, internal function, by Tom St Denis.

Definition in file [ocb\\_shift\\_xor.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ocb\_shift\_xor.c:

### Functions

- void [ocb\\_shift\\_xor](#) (ocb\_state \*ocb, unsigned char \*Z)  
*Compute the shift/xor for OCB (internal function).*

### 5.49.2 Function Documentation

#### 5.49.2.1 void ocb\_shift\_xor (ocb\_state \*ocb, unsigned char \*Z)

Compute the shift/xor for OCB (internal function).

#### Parameters:

*ocb* The OCB state

*Z* The destination of the shift

Definition at line 25 of file ocb\_shift\_xor.c.

References [ocb\\_ntz\(\)](#).

Referenced by [ocb\\_decrypt\(\)](#), and [s\\_ocb\\_done\(\)](#).

```
26 {
27     int x, y;
28     y = ocb_ntz(ocb->block_index++);
29     for (x = 0; x < ocb->block_len; x++) {
30         ocb->Li[x] ^= ocb->Is[y][x];
31         Z[x]       = ocb->Li[x] ^ ocb->R[x];
32     }
33 }
```

Here is the call graph for this function:

## 5.50 encauth/ocb/ocb\_test.c File Reference

### 5.50.1 Detailed Description

OCB implementation, self-test by Tom St Denis.

Definition in file [ocb\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ocb\_test.c:

### Functions

- [int ocb\\_test](#) (void)  
*Test the OCB protocol.*

### 5.50.2 Function Documentation

#### 5.50.2.1 int ocb\_test (void)

Test the OCB protocol.

#### Returns:

CRYPT\_OK if successful

Definition at line 24 of file ocb\_test.c.

References [CRYPT\\_NOP](#), [CRYPT\\_OK](#), [find\\_cipher\(\)](#), [len](#), [MAXBLOCKSIZE](#), and [ocb\\_encrypt\\_authenticate\\_memory\(\)](#).

```
25 {
26 #ifndef LTC_TEST
27     return CRYPT_NOP;
28 #else
29     static const struct {
30         int pten;
31         unsigned char key[16], nonce[16], pt[34], ct[34], tag[16];
32     } tests[] = {
33
34         /* OCB-AES-128-0B */
35     {
36         0,
37         /* key */
38         { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
39           0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
40         /* nonce */
41         { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
42           0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 },
43         /* pt */
44         { 0 },
45         /* ct */
46         { 0 },
47         /* tag */
48         { 0x15, 0xd3, 0x7d, 0xd7, 0xc8, 0x90, 0xd5, 0xd6,
49           0xac, 0xab, 0x92, 0x7b, 0xc0, 0xdc, 0x60, 0xee },
50     },
```

```
51
52
53 /* OCB-AES-128-3B */
54 {
55     3,
56     /* key */
57     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
58       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
59     /* nonce */
60     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
61       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 },
62     /* pt */
63     { 0x00, 0x01, 0x02 },
64     /* ct */
65     { 0xfc, 0xd3, 0x7d },
66     /* tag */
67     { 0x02, 0x25, 0x47, 0x39, 0xa5, 0xe3, 0x56, 0x5a,
68       0xe2, 0xdc, 0xd6, 0x2c, 0x65, 0x97, 0x46, 0xba },
69 },
70
71 /* OCB-AES-128-16B */
72 {
73     16,
74     /* key */
75     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
76       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
77     /* nonce */
78     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
79       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 },
80     /* pt */
81     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
82       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
83     /* ct */
84     { 0x37, 0xdf, 0x8c, 0xe1, 0x5b, 0x48, 0x9b, 0xf3,
85       0x1d, 0x0f, 0xc4, 0x4d, 0xa1, 0xfa, 0xf6, 0xd6 },
86     /* tag */
87     { 0xdf, 0xb7, 0x63, 0xeb, 0xdb, 0x5f, 0x0e, 0x71,
88       0x9c, 0x7b, 0x41, 0x61, 0x80, 0x80, 0x04, 0xdf },
89 },
90
91 /* OCB-AES-128-20B */
92 {
93     20,
94     /* key */
95     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
96       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
97     /* nonce */
98     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
99       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 },
100     /* pt */
101     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
102       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
103       0x10, 0x11, 0x12, 0x13 },
104     /* ct */
105     { 0x01, 0xa0, 0x75, 0xf0, 0xd8, 0x15, 0xb1, 0xa4,
106       0xe9, 0xc8, 0x81, 0xa1, 0xbc, 0xff, 0xc3, 0xeb,
107       0x70, 0x03, 0xeb, 0x55 },
108     /* tag */
109     { 0x75, 0x30, 0x84, 0x14, 0x4e, 0xb6, 0x3b, 0x77,
110       0x0b, 0x06, 0x3c, 0x2e, 0x23, 0xcd, 0xa0, 0xbb },
111 },
112
113 /* OCB-AES-128-32B */
114 {
115     32,
116     /* key */
117     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
```

```

118     0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
119     /* nonce */
120     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
121       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 },
122     /* pt */
123     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
124       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
125       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
126       0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f },
127     /* ct */
128     { 0x01, 0xa0, 0x75, 0xf0, 0xd8, 0x15, 0xb1, 0xa4,
129       0xe9, 0xc8, 0x81, 0xa1, 0xbc, 0xff, 0xc3, 0xeb,
130       0x4a, 0xfc, 0xbb, 0x7f, 0xed, 0xc0, 0x8c, 0xa8,
131       0x65, 0x4c, 0x6d, 0x30, 0x4d, 0x16, 0x12, 0xfa },
132
133     /* tag */
134     { 0xc1, 0x4c, 0xbf, 0x2c, 0x1a, 0x1f, 0x1c, 0x3c,
135       0x13, 0x7e, 0xad, 0xea, 0x1f, 0x2f, 0x2f, 0xcf },
136 },
137
138     /* OCB-AES-128-34B */
139 {
140     34,
141     /* key */
142     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
143       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
144     /* nonce */
145     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
146       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 },
147     /* pt */
148     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
149       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
150       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
151       0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
152       0x20, 0x21 },
153     /* ct */
154     { 0x01, 0xa0, 0x75, 0xf0, 0xd8, 0x15, 0xb1, 0xa4,
155       0xe9, 0xc8, 0x81, 0xa1, 0xbc, 0xff, 0xc3, 0xeb,
156       0xd4, 0x90, 0x3d, 0xd0, 0x02, 0x5b, 0xa4, 0xaa,
157       0x83, 0x7c, 0x74, 0xf1, 0x21, 0xb0, 0x26, 0x0f,
158       0xa9, 0x5d },
159
160     /* tag */
161     { 0xcf, 0x83, 0x41, 0xbb, 0x10, 0x82, 0x0c, 0xcf,
162       0x14, 0xbd, 0xec, 0x56, 0xb8, 0xd7, 0xd6, 0xab },
163 },
164
165 };
166
167 int err, x, idx, res;
168 unsigned long len;
169 unsigned char outct[MAXBLOCKSIZE], outtag[MAXBLOCKSIZE];
170
171     /* AES can be under rijndael or aes... try to find it */
172     if ((idx = find_cipher("aes")) == -1) {
173         if ((idx = find_cipher("rijndael")) == -1) {
174             return CRYPT_NOP;
175         }
176     }
177
178     for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
179         len = sizeof(outtag);
180         if ((err = ocb_encrypt_authenticate_memory(idx, tests[x].key, 16,
181           tests[x].nonce, tests[x].pt, tests[x].ptlen, outct, outtag, &len)) != CRYPT_OK) {
182             return err;
183         }
184     }

```

```

185         if (XMEMCMP(outtag, tests[x].tag, len) || XMEMCMP(outct, tests[x].ct, tests[x].ptlen)) {
186 #if 0
187         unsigned long y;
188         printf("\n\nFailure: \nCT:\n");
189         for (y = 0; y < (unsigned long)tests[x].ptlen; ) {
190             printf("0x%02x", outct[y]);
191             if (y < (unsigned long)(tests[x].ptlen-1)) printf(", ");
192             if (!(++y % 8)) printf("\n");
193         }
194         printf("\nTAG:\n");
195         for (y = 0; y < len; ) {
196             printf("0x%02x", outtag[y]);
197             if (y < len-1) printf(", ");
198             if (!(++y % 8)) printf("\n");
199         }
200 #endif
201         return CRYPT_FAIL_TESTVECTOR;
202     }
203
204     if ((err = ocb_decrypt_verify_memory(idx, tests[x].key, 16, tests[x].nonce, outct, tests[x].ptlen,
205     outct, tests[x].tag, len, &res)) != CRYPT_OK) {
206         return err;
207     }
208     if ((res != 1) || XMEMCMP(tests[x].pt, outct, tests[x].ptlen)) {
209 #if 0
210         unsigned long y;
211         printf("\n\nFailure-decrypt: \nPT:\n");
212         for (y = 0; y < (unsigned long)tests[x].ptlen; ) {
213             printf("0x%02x", outct[y]);
214             if (y < (unsigned long)(tests[x].ptlen-1)) printf(", ");
215             if (!(++y % 8)) printf("\n");
216         }
217         printf("\nres = %d\n\n", res);
218 #endif
219     }
220 }
221 return CRYPT_OK;
222 #endif /* LTC_TEST */
223 }

```

Here is the call graph for this function:

## 5.51 encauth/ocb/s\_ocb\_done.c File Reference

### 5.51.1 Detailed Description

OCB implementation, internal helper, by Tom St Denis.

Definition in file [s\\_ocb\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for s\_ocb\_done.c:

### Functions

- [int s\\_ocb\\_done](#) (ocb\_state \*ocb, const unsigned char \*pt, unsigned long ptlen, unsigned char \*ct, unsigned char \*tag, unsigned long \*taglen, int mode)

*Shared code to finish an OCB stream.*

### 5.51.2 Function Documentation

#### 5.51.2.1 int s\_ocb\_done (ocb\_state \*ocb, const unsigned char \*pt, unsigned long ptlen, unsigned char \*ct, unsigned char \*tag, unsigned long \*taglen, int mode)

Shared code to finish an OCB stream.

#### Parameters:

*ocb* The OCB state

*pt* The remaining plaintext [or input]

*ptlen* The length of the input (octets)

*ct* [out] The output buffer

*tag* [out] The destination for the authentication tag

*taglen* [in/out] The max size and resulting size of the authentication tag

*mode* The mode we are terminating, 0==encrypt, 1==decrypt

#### Returns:

CRYPT\_OK if successful

Definition at line 39 of file s\_ocb\_done.c.

References `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_MEM`, `CRYPT_OK`, `LTC_ARGCHK`, `MAXBLOCKSIZE`, `ocb_shift_xor()`, `XFREE`, `XMALLOC`, and `XMEMCPY`.

Referenced by `ocb_done_decrypt()`, and `ocb_done_encrypt()`.

```
42 {
43     unsigned char *Z, *Y, *X;
44     int err, x;
45
46     LTC_ARGCHK(ocb    != NULL);
47     LTC_ARGCHK(pt     != NULL);
48     LTC_ARGCHK(ct     != NULL);
```



```

49 LTC_ARGCHK(tag != NULL);
50 LTC_ARGCHK(taglen != NULL);
51 if ((err = cipher_is_valid(ocb->cipher)) != CRYPT_OK) {
52     return err;
53 }
54 if (ocb->block_len != cipher_descriptor[ocb->cipher].block_length ||
55     (int)ptlen > ocb->block_len || (int)ptlen < 0) {
56     return CRYPT_INVALID_ARG;
57 }
58
59 /* allocate ram */
60 Z = XMALLOC(MAXBLOCKSIZE);
61 Y = XMALLOC(MAXBLOCKSIZE);
62 X = XMALLOC(MAXBLOCKSIZE);
63 if (X == NULL || Y == NULL || Z == NULL) {
64     if (X != NULL) {
65         XFREE(X);
66     }
67     if (Y != NULL) {
68         XFREE(Y);
69     }
70     if (Z != NULL) {
71         XFREE(Z);
72     }
73     return CRYPT_MEM;
74 }
75
76 /* compute X[m] = len(pt[m]) XOR Lr XOR Z[m] */
77 ocb_shift_xor(ocb, X);
78 XMEMCPY(Z, X, ocb->block_len);
79
80 X[ocb->block_len-1] ^= (ptlen*8)&255;
81 X[ocb->block_len-2] ^= ((ptlen*8)>>8)&255;
82 for (x = 0; x < ocb->block_len; x++) {
83     X[x] ^= ocb->Lr[x];
84 }
85
86 /* Y[m] = E(X[m]) */
87 if ((err = cipher_descriptor[ocb->cipher].ecb_encrypt(X, Y, &ocb->key)) != CRYPT_OK) {
88     goto error;
89 }
90
91 if (mode == 1) {
92     /* decrypt mode, so let's xor it first */
93     /* xor C[m] into checksum */
94     for (x = 0; x < (int)ptlen; x++) {
95         ocb->checksum[x] ^= ct[x];
96     }
97 }
98
99 /* C[m] = P[m] xor Y[m] */
100 for (x = 0; x < (int)ptlen; x++) {
101     ct[x] = pt[x] ^ Y[x];
102 }
103
104 if (mode == 0) {
105     /* encrypt mode */
106     /* xor C[m] into checksum */
107     for (x = 0; x < (int)ptlen; x++) {
108         ocb->checksum[x] ^= ct[x];
109     }
110 }
111
112 /* xor Y[m] and Z[m] into checksum */
113 for (x = 0; x < ocb->block_len; x++) {
114     ocb->checksum[x] ^= Y[x] ^ Z[x];
115 }

```

```
116
117     /* encrypt checksum, er... tag!! */
118     if ((err = cipher_descriptor[ocb->cipher].ecb_encrypt(ocb->checksum, X, &ocb->key)) != CRYPT_OK) {
119         goto error;
120     }
121     cipher_descriptor[ocb->cipher].done(&ocb->key);
122
123     /* now store it */
124     for (x = 0; x < ocb->block_len && x < (int)*taglen; x++) {
125         tag[x] = X[x];
126     }
127     *taglen = x;
128
129 #ifdef LTC_CLEAN_STACK
130     zeromem(X, MAXBLOCKSIZE);
131     zeromem(Y, MAXBLOCKSIZE);
132     zeromem(Z, MAXBLOCKSIZE);
133     zeromem(ocb, sizeof(*ocb));
134 #endif
135 error:
136     XFREE(X);
137     XFREE(Y);
138     XFREE(Z);
139
140     return err;
141 }
```

Here is the call graph for this function:

## 5.52 hashes/chc/chc.c File Reference

### 5.52.1 Detailed Description

CHC support.

(Tom St Denis)

Definition in file [chc.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for chc.c:

### Defines

- #define [UNDEFED\\_HASH](#) -17

### Functions

- int [chc\\_register](#) (int cipher)  
*Initialize the CHC state with a given cipher.*
- int [chc\\_init](#) (hash\_state \*md)  
*Initialize the hash state.*
- static int [chc\\_compress](#) (hash\_state \*md, unsigned char \*buf)
- int [\\_chc\\_process](#) (hash\_state \*md, const unsigned char \*buf, unsigned long len)
- [HASH\\_PROCESS](#) (\_chc\_process, chc\_compress, chc,(unsigned long) [cipher\\_blocksize](#)) int chc\_  
process(hash\_state \*md)  
*Process a block of memory through the hash.*

### Variables

- static int [cipher\\_idx](#) = UNDEFED\_HASH
- static int [cipher\\_blocksize](#)
- const struct [ltc\\_hash\\_descriptor](#) [chc\\_desc](#)
- const unsigned char \* [in](#)

### 5.52.2 Define Documentation

#### 5.52.2.1 #define UNDEFED\_HASH -17

Definition at line 21 of file chc.c.

### 5.52.3 Function Documentation

**5.52.3.1** `int _chc_process (hash_state * md, const unsigned char * buf, unsigned long len)`

**5.52.3.2** `static int chc_compress (hash_state * md, unsigned char * buf) [static]`

Definition at line 132 of file chc.c.

References cipher\_blocksize, cipher\_descriptor, cipher\_idx, CRYPT\_MEM, CRYPT\_OK, ltc\_cipher\_descriptor::ecb\_encrypt, MAXBLOCKSIZE, XFREE, XMALLOC, and XMEMCPY.

```

133 {
134     unsigned char  T[2][MAXBLOCKSIZE];
135     symmetric_key *key;
136     int            err, x;
137
138     if ((key = XMALLOC(sizeof(*key))) == NULL) {
139         return CRYPT_MEM;
140     }
141     if ((err = cipher_descriptor[cipher_idx].setup(md->chc.state, cipher_blocksize, 0, key)) != CRYPT_OK) {
142         XFREE(key);
143         return err;
144     }
145     XMEMCPY(T[1], buf, cipher_blocksize);
146     cipher_descriptor[cipher_idx].ecb_encrypt(buf, T[0], key);
147     for (x = 0; x < cipher_blocksize; x++) {
148         md->chc.state[x] ^= T[0][x] ^ T[1][x];
149     }
150     XFREE(key);
151 #ifdef LTC_CLEAN_STACK
152     zeromem(T, sizeof(T));
153     zeromem(&key, sizeof(key));
154 #endif
155     return CRYPT_OK;
156 }
```

**5.52.3.3** `int chc_init (hash_state * md)`

Initialize the hash state.

#### Parameters:

*md* The hash state you wish to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 87 of file chc.c.

References cipher\_blocksize, cipher\_descriptor, cipher\_idx, cipher\_is\_valid(), CRYPT\_INVALID\_CIPHER, CRYPT\_MEM, CRYPT\_OK, ltc\_cipher\_descriptor::ecb\_encrypt, LTC\_ARGCHK, MAXBLOCKSIZE, XFREE, XMALLOC, and zeromem().

```

88 {
89     symmetric_key *key;
90     unsigned char  buf[MAXBLOCKSIZE];
91     int            err;
92
93     LTC_ARGCHK(md != NULL);
94 }
```

```

95  /* is the cipher valid? */
96  if ((err = cipher_is_valid(cipher_idx)) != CRYPT_OK) {
97      return err;
98  }
99
100  if (cipher_blocksize != cipher_descriptor[cipher_idx].block_length) {
101      return CRYPT_INVALID_CIPHER;
102  }
103
104  if ((key = XMALLOC(sizeof(*key))) == NULL) {
105      return CRYPT_MEM;
106  }
107
108  /* zero key and what not */
109  zeromem(buf, cipher_blocksize);
110  if ((err = cipher_descriptor[cipher_idx].setup(buf, cipher_blocksize, 0, key)) != CRYPT_OK) {
111      XFREE(key);
112      return err;
113  }
114
115  /* encrypt zero block */
116  cipher_descriptor[cipher_idx].ecb_encrypt(buf, md->chc.state, key);
117
118  /* zero other members */
119  md->chc.length = 0;
120  md->chc.curlen = 0;
121  zeromem(md->chc.buf, sizeof(md->chc.buf));
122  XFREE(key);
123  return CRYPT_OK;
124 }

```

Here is the call graph for this function:

### 5.52.3.4 int chc\_register (int cipher)

Initialize the CHC state with a given cipher.

#### Parameters:

***cipher*** The index of the cipher you wish to bind

#### Returns:

CRYPT\_OK if successful

Definition at line 42 of file chc.c.

References ltc\_cipher\_descriptor::block\_length, ltc\_hash\_descriptor::blocksize, cipher\_blocksize, cipher\_descriptor, cipher\_idx, cipher\_is\_valid(), CRYPT\_INVALID\_CIPHER, CRYPT\_OK, find\_hash(), hash\_descriptor, hash\_is\_valid(), and ltc\_hash\_descriptor::hashsize.

```

43 {
44     int err, kl, idx;
45
46     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
47         return err;
48     }
49
50     /* will it be valid? */
51     kl = cipher_descriptor[cipher].block_length;
52
53     /* must be >64 bit block */
54     if (kl <= 8) {

```

```

55     return CRYPT_INVALID_CIPHER;
56 }
57
58 /* can we use the ideal keysize? */
59 if ((err = cipher_descriptor[cipher].keysize(&kl)) != CRYPT_OK) {
60     return err;
61 }
62 /* we require that key size == block size be a valid choice */
63 if (kl != cipher_descriptor[cipher].block_length) {
64     return CRYPT_INVALID_CIPHER;
65 }
66
67 /* determine if chc_hash has been register_hash'ed already */
68 if ((err = hash_is_valid(idx = find_hash("chc_hash"))) != CRYPT_OK) {
69     return err;
70 }
71
72 /* store into descriptor */
73 hash_descriptor[idx].hashsize =
74 hash_descriptor[idx].blocksize = cipher_descriptor[cipher].block_length;
75
76 /* store the idx and block size */
77 cipher_idx = cipher;
78 cipher_blocksize = cipher_descriptor[cipher].block_length;
79 return CRYPT_OK;
80 }

```

Here is the call graph for this function:

#### 5.52.3.5 HASH\_PROCESS (*\_chc\_process*, *chc\_compress*, *chc*, (unsigned long) *cipher\_blocksize*)

Process a block of memory through the hash.

##### Parameters:

- md* The hash state
- in* The data to hash
- inlen* The length of the data (octets)

##### Returns:

CRYPT\_OK if successful

#### 5.52.4 Variable Documentation

##### 5.52.4.1 `const struct ltc_hash_descriptor chc_desc`

##### Initial value:

```

{
    "chc_hash", 12, 0, 0, { 0 }, 0,
    &chc_init,
    &chc_process,
    &chc_done,
    &chc_test,
    NULL
}

```

Definition at line 28 of file chc.c.

**5.52.4.2** `int cipher_blocksize` `[static]`

Definition at line 24 of file chc.c.

Referenced by `chc_compress()`, `chc_init()`, and `chc_register()`.

**5.52.4.3** `int cipher_idx = UNDEFED_HASH` `[static]`

Definition at line 24 of file chc.c.

Referenced by `chc_compress()`, `chc_init()`, and `chc_register()`.

**5.52.4.4** `const unsigned char* in`

Definition at line 169 of file chc.c.

Referenced by `f9_file()`, `hash_file()`, `hmac_file()`, `omac_file()`, `pmac_file()`, and `xcbc_file()`.

## 5.53 hashes/helper/hash\_file.c File Reference

### 5.53.1 Detailed Description

Hash a file, Tom St Denis.

Definition in file [hash\\_file.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for hash\_file.c:

### Functions

- [int hash\\_file](#) (int hash, const char \*fname, unsigned char \*out, unsigned long \*outlen)

### 5.53.2 Function Documentation

#### 5.53.2.1 int hash\_file (int *hash*, const char \* *fname*, unsigned char \* *out*, unsigned long \* *outlen*)

##### Parameters:

*hash* The index of the hash desired

*fname* The name of the file you wish to hash

*out* [out] The destination of the digest

*outlen* [in/out] The max size and resulting size of the message digest

##### Returns:

CRYPT\_OK if successful

Definition at line 25 of file hash\_file.c.

References [CRYPT\\_ERROR](#), [CRYPT\\_FILE\\_NOTFOUND](#), [CRYPT\\_NOP](#), [CRYPT\\_OK](#), [hash\\_filehandle\(\)](#), [hash\\_is\\_valid\(\)](#), [in](#), and [LTC\\_ARGCHK](#).

```

26 {
27 #ifdef LTC_NO_FILE
28     return CRYPT_NOP;
29 #else
30     FILE *in;
31     int err;
32     LTC_ARGCHK(fname != NULL);
33     LTC_ARGCHK(out != NULL);
34     LTC_ARGCHK(outlen != NULL);
35
36     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
37         return err;
38     }
39
40     in = fopen(fname, "rb");
41     if (in == NULL) {
42         return CRYPT_FILE_NOTFOUND;
43     }
44
45     err = hash_filehandle(hash, in, out, outlen);
46     if (fclose(in) != 0) {
47         return CRYPT_ERROR;
48     }

```



```
49
50     return err;
51 #endif
52 }
```

Here is the call graph for this function:

## 5.54 hashes/helper/hash\_filehandle.c File Reference

### 5.54.1 Detailed Description

Hash open files, Tom St Denis.

Definition in file [hash\\_filehandle.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for hash\_filehandle.c:

### Functions

- [int hash\\_filehandle](#) (int hash, FILE \**in*, unsigned char \*out, unsigned long \*outlen)  
*Hash data from an open file handle.*

### 5.54.2 Function Documentation

#### 5.54.2.1 int hash\_filehandle (int hash, FILE \*in, unsigned char \*out, unsigned long \*outlen)

Hash data from an open file handle.

#### Parameters:

- hash* The index of the hash you want to use
- in* The FILE\* handle of the file you want to hash
- out* [out] The destination of the digest
- outlen* [in/out] The max size and resulting size of the digest

#### Returns:

CRYPT\_OK if successful

Definition at line 26 of file hash\_filehandle.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_NOP, CRYPT\_OK, ltc\_hash\_descriptor::done, hash\_descriptor, hash\_is\_valid(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, and zeromem().

Referenced by hash\_file().

```
27 {
28 #ifdef LTC_NO_FILE
29     return CRYPT_NOP;
30 #else
31     hash_state md;
32     unsigned char buf[512];
33     size_t x;
34     int err;
35
36     LTC_ARGCHK(out != NULL);
37     LTC_ARGCHK(outlen != NULL);
38     LTC_ARGCHK(in != NULL);
39
40     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
41         return err;
42     }
```

```
43
44     if (*outlen < hash_descriptor[hash].hashsize) {
45         *outlen = hash_descriptor[hash].hashsize;
46         return CRYPT_BUFFER_OVERFLOW;
47     }
48     if ((err = hash_descriptor[hash].init(&md)) != CRYPT_OK) {
49         return err;
50     }
51
52     *outlen = hash_descriptor[hash].hashsize;
53     do {
54         x = fread(buf, 1, sizeof(buf), in);
55         if ((err = hash_descriptor[hash].process(&md, buf, x)) != CRYPT_OK) {
56             return err;
57         }
58     } while (x == sizeof(buf));
59     err = hash_descriptor[hash].done(&md, out);
60
61 #ifdef LTC_CLEAN_STACK
62     zeromem(buf, sizeof(buf));
63 #endif
64     return err;
65 #endif
66 }
```

Here is the call graph for this function:

## 5.55 hashes/helper/hash\_memory.c File Reference

### 5.55.1 Detailed Description

Hash memory helper, Tom St Denis.

Definition in file [hash\\_memory.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for hash\_memory.c:

### Functions

- [int hash\\_memory](#) (int hash, const unsigned char \*in, unsigned long inlen, unsigned char \*out, unsigned long \*outlen)

*Hash a block of memory and store the digest.*

### 5.55.2 Function Documentation

#### 5.55.2.1 int hash\_memory (int hash, const unsigned char \*in, unsigned long inlen, unsigned char \*out, unsigned long \*outlen)

Hash a block of memory and store the digest.

#### Parameters:

- hash** The index of the hash you wish to use
- in** The data you wish to hash
- inlen** The length of the data to hash (octets)
- out** [out] Where to store the digest
- outlen** [in/out] Max size and resulting size of the digest

#### Returns:

- CRYPT\_OK if successful

Definition at line 27 of file hash\_memory.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_MEM, CRYPT\_OK, ltc\_hash\_descriptor::done, hash\_descriptor, hash\_is\_valid(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, XFREE, XMALLOC, and zeromem().

Referenced by dsa\_decrypt\_key(), dsa\_encrypt\_key(), ecc\_decrypt\_key(), ecc\_encrypt\_key(), hmac\_init(), pkcs\_1\_oaep\_encode(), and pkcs\_5\_alg1().

```
28 {
29     hash_state *md;
30     int err;
31
32     LTC_ARGCHK(in != NULL);
33     LTC_ARGCHK(out != NULL);
34     LTC_ARGCHK(outlen != NULL);
35
36     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
```

```
37         return err;
38     }
39
40     if (*outlen < hash_descriptor[hash].hashsize) {
41         *outlen = hash_descriptor[hash].hashsize;
42         return CRYPT_BUFFER_OVERFLOW;
43     }
44
45     md = XMALLOC(sizeof(hash_state));
46     if (md == NULL) {
47         return CRYPT_MEM;
48     }
49
50     if ((err = hash_descriptor[hash].init(md)) != CRYPT_OK) {
51         goto LBL_ERR;
52     }
53     if ((err = hash_descriptor[hash].process(md, in, inlen)) != CRYPT_OK) {
54         goto LBL_ERR;
55     }
56     err = hash_descriptor[hash].done(md, out);
57     *outlen = hash_descriptor[hash].hashsize;
58 LBL_ERR:
59 #ifdef LTC_CLEAN_STACK
60     zeromem(md, sizeof(hash_state));
61 #endif
62     XFREE(md);
63
64     return err;
65 }
```

Here is the call graph for this function:

## 5.56 hashes/helper/hash\_memory\_multi.c File Reference

### 5.56.1 Detailed Description

Hash (multiple buffers) memory helper, Tom St Denis.

Definition in file [hash\\_memory\\_multi.c](#).

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for hash\_memory\_multi.c:

### Functions

- [int hash\\_memory\\_multi](#) (int hash, unsigned char \*out, unsigned long \*outlen, const unsigned char \*in, unsigned long inlen,...)

*Hash multiple (non-adjacent) blocks of memory at once.*

### 5.56.2 Function Documentation

#### 5.56.2.1 int hash\_memory\_multi (int hash, unsigned char \* out, unsigned long \* outlen, const unsigned char \* in, unsigned long inlen, ...)

Hash multiple (non-adjacent) blocks of memory at once.

#### Parameters:

**hash** The index of the hash you wish to use

**out** [out] Where to store the digest

**outlen** [in/out] Max size and resulting size of the digest

**in** The data you wish to hash

**inlen** The length of the data to hash (octets)

... tuples of (data,len) pairs to hash, terminated with a (NULL,x) (x=don't care)

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file hash\_memory\_multi.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_MEM, CRYPT\_OK, hash\_descriptor, hash\_is\_valid(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, and XMALLOC.

```
30 {
31     hash_state      *md;
32     int             err;
33     va_list         args;
34     const unsigned char *curptr;
35     unsigned long    curlen;
36
37     LTC_ARGCHK(in    != NULL);
38     LTC_ARGCHK(out   != NULL);
39     LTC_ARGCHK(outlen != NULL);
```

```
40
41     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
42         return err;
43     }
44
45     if (*outlen < hash_descriptor[hash].hashsize) {
46         *outlen = hash_descriptor[hash].hashsize;
47         return CRYPT_BUFFER_OVERFLOW;
48     }
49
50     md = XMALLOC(sizeof(hash_state));
51     if (md == NULL) {
52         return CRYPT_MEM;
53     }
54
55     if ((err = hash_descriptor[hash].init(md)) != CRYPT_OK) {
56         goto LBL_ERR;
57     }
58
59     va_start(args, inlen);
60     curptr = in;
61     curlen = inlen;
62     for (;;) {
63         /* process buf */
64         if ((err = hash_descriptor[hash].process(md, curptr, curlen)) != CRYPT_OK) {
65             goto LBL_ERR;
66         }
67         /* step to next */
68         curptr = va_arg(args, const unsigned char*);
69         if (curptr == NULL) {
70             break;
71         }
72         curlen = va_arg(args, unsigned long);
73     }
74     err = hash_descriptor[hash].done(md, out);
75     *outlen = hash_descriptor[hash].hashsize;
76 LBL_ERR:
77 #ifdef LTC_CLEAN_STACK
78     zeromem(md, sizeof(hash_state));
79 #endif
80     XFREE(md);
81     va_end(args);
82     return err;
83 }
```

Here is the call graph for this function:

## 5.57 hashes/md2.c File Reference

```
#include "tomcrypt.h"
```

Include dependency graph for md2.c:

### Functions

- static void [md2\\_update\\_chksum](#) ([hash\\_state](#) \*md)
- static void [md2\\_compress](#) ([hash\\_state](#) \*md)
- int [md2\\_init](#) ([hash\\_state](#) \*md)  
*Initialize the hash state.*
- int [md2\\_process](#) ([hash\\_state](#) \*md, const unsigned char \*in, unsigned long inlen)  
*Process a block of memory through the hash.*
- int [md2\\_done](#) ([hash\\_state](#) \*md, unsigned char \*out)  
*Terminate the hash to get the digest.*
- int [md2\\_test](#) (void)  
*Self-test the hash.*

### Variables

- const struct [ltc\\_hash\\_descriptor](#) [md2\\_desc](#)
- static const unsigned char [PI\\_SUBST](#) [256]

### 5.57.1 Function Documentation

#### 5.57.1.1 static void [md2\\_compress](#) ([hash\\_state](#) \*md) [static]

Definition at line 74 of file md2.c.

Referenced by [md2\\_done\(\)](#), and [md2\\_process\(\)](#).

```
75 {
76     int j, k;
77     unsigned char t;
78
79     /* copy block */
80     for (j = 0; j < 16; j++) {
81         md->md2.X[16+j] = md->md2.buf[j];
82         md->md2.X[32+j] = md->md2.X[j] ^ md->md2.X[16+j];
83     }
84
85     t = (unsigned char)0;
86
87     /* do 18 rounds */
88     for (j = 0; j < 18; j++) {
89         for (k = 0; k < 48; k++) {
90             t = (md->md2.X[k] ^= PI_SUBST[(int)(t & 255)]);
91         }
92         t = (t + (unsigned char)j) & 255;
93     }
94 }
```



**5.57.1.2** `int md2_done (hash_state * md, unsigned char * out)`

Terminate the hash to get the digest.

**Parameters:**

- md* The hash state
- out* [out] The destination of the hash (16 bytes)

**Returns:**

CRYPT\_OK if successful

Definition at line 151 of file md2.c.

References CRYPT\_INVALID\_ARG, CRYPT\_OK, LTC\_ARGCHK, md2\_compress(), md2\_update\_checksum(), XMEMCPY, and zeromem().

Referenced by md2\_test().

```

152 {
153     unsigned long i, k;
154
155     LTC_ARGCHK(md != NULL);
156     LTC_ARGCHK(out != NULL);
157
158     if (md->md2.curlen >= sizeof(md->md2.buf)) {
159         return CRYPT_INVALID_ARG;
160     }
161
162
163     /* pad the message */
164     k = 16 - md->md2.curlen;
165     for (i = md->md2.curlen; i < 16; i++) {
166         md->md2.buf[i] = (unsigned char)k;
167     }
168
169     /* hash and update */
170     md2_compress(md);
171     md2_update_checksum(md);
172
173     /* hash checksum */
174     XMEMCPY(md->md2.buf, md->md2.chksum, 16);
175     md2_compress(md);
176
177     /* output is lower 16 bytes of X */
178     XMEMCPY(out, md->md2.X, 16);
179
180 #ifdef LTC_CLEAN_STACK
181     zeromem(md, sizeof(hash_state));
182 #endif
183     return CRYPT_OK;
184 }
```

Here is the call graph for this function:

**5.57.1.3** `int md2_init (hash_state * md)`

Initialize the hash state.

**Parameters:**

- md* The hash state you wish to initialize

**Returns:**

CRYPT\_OK if successful

Definition at line 101 of file md2.c.

References CRYPT\_OK, LTC\_ARGCHK, and zeromem().

Referenced by md2\_test().

```

102 {
103     LTC_ARGCHK(md != NULL);
104
105     /* MD2 uses a zero'ed state... */
106     zeromem(md->md2.X, sizeof(md->md2.X));
107     zeromem(md->md2.chksum, sizeof(md->md2.chksum));
108     zeromem(md->md2.buf, sizeof(md->md2.buf));
109     md->md2.curlen = 0;
110     return CRYPT_OK;
111 }
```

Here is the call graph for this function:

#### 5.57.1.4 int md2\_process (hash\_state \* md, const unsigned char \* in, unsigned long inlen)

Process a block of memory through the hash.

**Parameters:**

*md* The hash state

*in* The data to hash

*inlen* The length of the data (octets)

**Returns:**

CRYPT\_OK if successful

Definition at line 120 of file md2.c.

References CRYPT\_INVALID\_ARG, CRYPT\_OK, LTC\_ARGCHK, md2\_compress(), md2\_update\_chksum(), MIN, and XMEMCPY.

Referenced by md2\_test().

```

121 {
122     unsigned long n;
123     LTC_ARGCHK(md != NULL);
124     LTC_ARGCHK(in != NULL);
125     if (md->md2.curlen > sizeof(md->md2.buf)) {
126         return CRYPT_INVALID_ARG;
127     }
128     while (inlen > 0) {
129         n = MIN(inlen, (16 - md->md2.curlen));
130         XMEMCPY(md->md2.buf + md->md2.curlen, in, (size_t)n);
131         md->md2.curlen += n;
132         in             += n;
133         inlen           -= n;
134
135         /* is 16 bytes full? */
136         if (md->md2.curlen == 16) {
137             md2_compress(md);
138             md2_update_chksum(md);

```

```

139         md->md2.curlen = 0;
140     }
141 }
142 return CRYPT_OK;
143 }

```

Here is the call graph for this function:

#### 5.57.1.5 int md2\_test (void)

Self-test the hash.

##### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 190 of file md2.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, md2\_done(), md2\_init(), md2\_process(), and XMEMCMP.

```

191 {
192     #ifndef LTC_TEST
193         return CRYPT_NOP;
194     #else
195         static const struct {
196             char *msg;
197             unsigned char md[16];
198         } tests[] = {
199             { "",
200               {0x83,0x50,0xe5,0xa3,0xe2,0x4c,0x15,0x3d,
201                0xf2,0x27,0x5c,0x9f,0x80,0x69,0x27,0x73
202               },
203             { "a",
204               {0x32,0xec,0x01,0xec,0x4a,0x6d,0xac,0x72,
205                0xc0,0xab,0x96,0xfb,0x34,0xc0,0xb5,0xd1
206               },
207             { "message digest",
208               {0xab,0x4f,0x49,0x6b,0xfb,0x2a,0x53,0x0b,
209                0x21,0x9f,0xf3,0x30,0x31,0xfe,0x06,0xb0
210               },
211             { "abcdefghijklmnopqrstuvwxyz",
212               {0x4e,0x8d,0xdf,0xf3,0x65,0x02,0x92,0xab,
213                0x5a,0x41,0x08,0xc3,0xaa,0x47,0x94,0x0b
214               },
215             { "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789",
216               {0xda,0x33,0xde,0xf2,0xa4,0x2d,0xf1,0x39,
217                0x75,0x35,0x28,0x46,0xc3,0x03,0x38,0xcd
218               },
219             { "123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890",
220               {0xd5,0x97,0x6f,0x79,0xd8,0x3d,0x3a,0x0d,
221                0xc9,0x80,0x6c,0x3c,0x66,0xf3,0xef,0xd8
222               },
223             },
224         };
225     int i;
226     hash_state md;
227     unsigned char buf[16];
228 }
229 };
230 int i;
231 hash_state md;
232 unsigned char buf[16];
233 }

```

```

234     for (i = 0; i < (int)(sizeof(tests) / sizeof(tests[0])); i++) {
235         md2_init(&md);
236         md2_process(&md, (unsigned char*)tests[i].msg, (unsigned long)strlen(tests[i].msg));
237         md2_done(&md, buf);
238         if (XMEMCMP(buf, tests[i].md, 16) != 0) {
239             return CRYPT_FAIL_TESTVECTOR;
240         }
241     }
242     return CRYPT_OK;
243 #endif
244 }

```

Here is the call graph for this function:

#### 5.57.1.6 static void md2\_update\_chksum ([hash\\_state](#) \* *md*) [static]

Definition at line 60 of file md2.c.

References `PI_SUBST`.

Referenced by `md2_done()`, and `md2_process()`.

```

61 {
62     int j;
63     unsigned char L;
64     L = md->md2.chksum[15];
65     for (j = 0; j < 16; j++) {
66
67 /* caution, the RFC says its "C[j] = S[M[i*16+j] xor L]" but the reference source code [and test vector
68     otherwise.
69 */
70         L = (md->md2.chksum[j] ^= PI_SUBST[(int)(md->md2.buf[j] ^ L)] & 255);
71     }
72 }

```

## 5.57.2 Variable Documentation

### 5.57.2.1 const struct [ltc\\_hash\\_descriptor](#) md2\_desc

**Initial value:**

```

{
    "md2",
    7,
    16,
    16,

    { 1, 2, 840, 113549, 2, 2, },
    6,

    &md2_init,
    &md2_process,
    &md2_done,
    &md2_test,
    NULL
}

```

**Parameters:**

[md2.c](#) MD2 (RFC 1319) hash function implementation by Tom St Denis

Definition at line 20 of file md2.c.

Referenced by `yarrow_start()`.

#### 5.57.2.2 `const unsigned char PI_SUBST[256]` `[static]`

**Initial value:**

```
{
  41, 46, 67, 201, 162, 216, 124, 1, 61, 54, 84, 161, 236, 240, 6,
  19, 98, 167, 5, 243, 192, 199, 115, 140, 152, 147, 43, 217, 188,
  76, 130, 202, 30, 155, 87, 60, 253, 212, 224, 22, 103, 66, 111, 24,
  138, 23, 229, 18, 190, 78, 196, 214, 218, 158, 222, 73, 160, 251,
  245, 142, 187, 47, 238, 122, 169, 104, 121, 145, 21, 178, 7, 63,
  148, 194, 16, 137, 11, 34, 95, 33, 128, 127, 93, 154, 90, 144, 50,
  39, 53, 62, 204, 231, 191, 247, 151, 3, 255, 25, 48, 179, 72, 165,
  181, 209, 215, 94, 146, 42, 172, 86, 170, 198, 79, 184, 56, 210,
  150, 164, 125, 182, 118, 252, 107, 226, 156, 116, 4, 241, 69, 157,
  112, 89, 100, 113, 135, 32, 134, 91, 207, 101, 230, 45, 168, 2, 27,
  96, 37, 173, 174, 176, 185, 246, 28, 70, 97, 105, 52, 64, 126, 15,
  85, 71, 163, 35, 221, 81, 175, 58, 195, 92, 249, 206, 186, 197,
  234, 38, 44, 83, 13, 110, 133, 40, 132, 9, 211, 223, 205, 244, 65,
  129, 77, 82, 106, 220, 55, 200, 108, 193, 171, 250, 36, 225, 123,
  8, 12, 189, 177, 74, 120, 136, 149, 139, 227, 99, 232, 109, 233,
  203, 213, 254, 59, 0, 29, 57, 242, 239, 183, 14, 102, 88, 208, 228,
  166, 119, 114, 248, 235, 117, 75, 10, 49, 68, 80, 180, 143, 237,
  31, 26, 219, 153, 141, 51, 159, 17, 131, 20
}
```

Definition at line 38 of file md2.c.

Referenced by `md2_update_chksum()`.

## 5.58 hashes/md4.c File Reference

```
#include "tomcrypt.h"
```

Include dependency graph for md4.c:

### Defines

- `#define S11` 3
- `#define S12` 7
- `#define S13` 11
- `#define S14` 19
- `#define S21` 3
- `#define S22` 5
- `#define S23` 9
- `#define S24` 13
- `#define S31` 3
- `#define S32` 9
- `#define S33` 11
- `#define S34` 15
- `#define F(x, y, z)`  $(z \wedge (x \& (y \wedge z)))$
- `#define G(x, y, z)`  $((x \& y) \mid (z \& (x \mid y)))$
- `#define H(x, y, z)`  $((x) \wedge (y) \wedge (z))$
- `#define ROTATE_LEFT(x, n)`  $\text{ROLc}(x, n)$
- `#define FF(a, b, c, d, x, s)`
- `#define GG(a, b, c, d, x, s)`
- `#define HH(a, b, c, d, x, s)`

### Functions

- `static int md4_compress` (`hash_state` \*md, unsigned char \*buf)
- `int md4_init` (`hash_state` \*md)  
*Initialize the hash state.*
- `int md4_done` (`hash_state` \*md, unsigned char \*out)  
*Terminate the hash to get the digest.*
- `int md4_test` (void)  
*Self-test the hash.*

### Variables

- `const struct ltc_hash_descriptor md4_desc`

#### 5.58.1 Define Documentation

##### 5.58.1.1 `#define F(x, y, z) (z ^ (x & (y ^ z)))`

Definition at line 52 of file md4.c.

**5.58.1.2 #define FF(a, b, c, d, x, s)****Value:**

```
{ \
    (a) += F ((b), (c), (d)) + (x); \
    (a) = ROTATE_LEFT ((a), (s)); \
}
```

Definition at line 62 of file md4.c.

**5.58.1.3 #define G(x, y, z) ((x & y) | (z & (x | y)))**

Definition at line 53 of file md4.c.

**5.58.1.4 #define GG(a, b, c, d, x, s)****Value:**

```
{ \
    (a) += G ((b), (c), (d)) + (x) + 0x5a827999UL; \
    (a) = ROTATE_LEFT ((a), (s)); \
}
```

Definition at line 66 of file md4.c.

Referenced by ecc\_test().

**5.58.1.5 #define H(x, y, z) ((x) ^ (y) ^ (z))**

Definition at line 54 of file md4.c.

**5.58.1.6 #define HH(a, b, c, d, x, s)****Value:**

```
{ \
    (a) += H ((b), (c), (d)) + (x) + 0x6ed9eba1UL; \
    (a) = ROTATE_LEFT ((a), (s)); \
}
```

Definition at line 70 of file md4.c.

**5.58.1.7 #define ROTATE\_LEFT(x, n) ROLc(x, n)**

Definition at line 57 of file md4.c.

**5.58.1.8 #define S11 3**

Definition at line 38 of file md4.c.

**5.58.1.9 #define S12 7**

Definition at line 39 of file md4.c.

**5.58.1.10 #define S13 11**

Definition at line 40 of file md4.c.

**5.58.1.11 #define S14 19**

Definition at line 41 of file md4.c.

**5.58.1.12 #define S21 3**

Definition at line 42 of file md4.c.

**5.58.1.13 #define S22 5**

Definition at line 43 of file md4.c.

**5.58.1.14 #define S23 9**

Definition at line 44 of file md4.c.

**5.58.1.15 #define S24 13**

Definition at line 45 of file md4.c.

**5.58.1.16 #define S31 3**

Definition at line 46 of file md4.c.

**5.58.1.17 #define S32 9**

Definition at line 47 of file md4.c.

**5.58.1.18 #define S33 11**

Definition at line 48 of file md4.c.

**5.58.1.19 #define S34 15**

Definition at line 49 of file md4.c.



## 5.58.2 Function Documentation

### 5.58.2.1 static int md4\_compress ([hash\\_state](#) \*md, unsigned char \*buf) [static]

Definition at line 78 of file md4.c.

References c.

Referenced by md4\_done().

```

80 {
81     ulong32 x[16], a, b, c, d;
82     int i;
83
84     /* copy state */
85     a = md->md4.state[0];
86     b = md->md4.state[1];
87     c = md->md4.state[2];
88     d = md->md4.state[3];
89
90     /* copy the state into 512-bits into W[0..15] */
91     for (i = 0; i < 16; i++) {
92         LOAD32L(x[i], buf + (4*i));
93     }
94
95     /* Round 1 */
96     FF (a, b, c, d, x[ 0], S11); /* 1 */
97     FF (d, a, b, c, x[ 1], S12); /* 2 */
98     FF (c, d, a, b, x[ 2], S13); /* 3 */
99     FF (b, c, d, a, x[ 3], S14); /* 4 */
100    FF (a, b, c, d, x[ 4], S11); /* 5 */
101    FF (d, a, b, c, x[ 5], S12); /* 6 */
102    FF (c, d, a, b, x[ 6], S13); /* 7 */
103    FF (b, c, d, a, x[ 7], S14); /* 8 */
104    FF (a, b, c, d, x[ 8], S11); /* 9 */
105    FF (d, a, b, c, x[ 9], S12); /* 10 */
106    FF (c, d, a, b, x[10], S13); /* 11 */
107    FF (b, c, d, a, x[11], S14); /* 12 */
108    FF (a, b, c, d, x[12], S11); /* 13 */
109    FF (d, a, b, c, x[13], S12); /* 14 */
110    FF (c, d, a, b, x[14], S13); /* 15 */
111    FF (b, c, d, a, x[15], S14); /* 16 */
112
113    /* Round 2 */
114    GG (a, b, c, d, x[ 0], S21); /* 17 */
115    GG (d, a, b, c, x[ 4], S22); /* 18 */
116    GG (c, d, a, b, x[ 8], S23); /* 19 */
117    GG (b, c, d, a, x[12], S24); /* 20 */
118    GG (a, b, c, d, x[ 1], S21); /* 21 */
119    GG (d, a, b, c, x[ 5], S22); /* 22 */
120    GG (c, d, a, b, x[ 9], S23); /* 23 */
121    GG (b, c, d, a, x[13], S24); /* 24 */
122    GG (a, b, c, d, x[ 2], S21); /* 25 */
123    GG (d, a, b, c, x[ 6], S22); /* 26 */
124    GG (c, d, a, b, x[10], S23); /* 27 */
125    GG (b, c, d, a, x[14], S24); /* 28 */
126    GG (a, b, c, d, x[ 3], S21); /* 29 */
127    GG (d, a, b, c, x[ 7], S22); /* 30 */
128    GG (c, d, a, b, x[11], S23); /* 31 */
129    GG (b, c, d, a, x[15], S24); /* 32 */
130
131    /* Round 3 */
132    HH (a, b, c, d, x[ 0], S31); /* 33 */
133    HH (d, a, b, c, x[ 8], S32); /* 34 */
134    HH (c, d, a, b, x[ 4], S33); /* 35 */
135    HH (b, c, d, a, x[12], S34); /* 36 */

```

```

136     HH (a, b, c, d, x[ 2], S31); /* 37 */
137     HH (d, a, b, c, x[10], S32); /* 38 */
138     HH (c, d, a, b, x[ 6], S33); /* 39 */
139     HH (b, c, d, a, x[14], S34); /* 40 */
140     HH (a, b, c, d, x[ 1], S31); /* 41 */
141     HH (d, a, b, c, x[ 9], S32); /* 42 */
142     HH (c, d, a, b, x[ 5], S33); /* 43 */
143     HH (b, c, d, a, x[13], S34); /* 44 */
144     HH (a, b, c, d, x[ 3], S31); /* 45 */
145     HH (d, a, b, c, x[11], S32); /* 46 */
146     HH (c, d, a, b, x[ 7], S33); /* 47 */
147     HH (b, c, d, a, x[15], S34); /* 48 */
148
149
150     /* Update our state */
151     md->md4.state[0] = md->md4.state[0] + a;
152     md->md4.state[1] = md->md4.state[1] + b;
153     md->md4.state[2] = md->md4.state[2] + c;
154     md->md4.state[3] = md->md4.state[3] + d;
155
156     return CRYPT_OK;
157 }

```

### 5.58.2.2 int md4\_done (hash\_state \*md, unsigned char \*out)

Terminate the hash to get the digest.

#### Parameters:

*md* The hash state

*out* [out] The destination of the hash (16 bytes)

#### Returns:

CRYPT\_OK if successful

Definition at line 201 of file md4.c.

References CRYPT\_INVALID\_ARG, LTC\_ARGCHK, and md4\_compress().

Referenced by md4\_test().

```

202 {
203     int i;
204
205     LTC_ARGCHK(md != NULL);
206     LTC_ARGCHK(out != NULL);
207
208     if (md->md4.curlen >= sizeof(md->md4.buf)) {
209         return CRYPT_INVALID_ARG;
210     }
211
212     /* increase the length of the message */
213     md->md4.length += md->md4.curlen * 8;
214
215     /* append the '1' bit */
216     md->md4.buf[md->md4.curlen++] = (unsigned char)0x80;
217
218     /* if the length is currently above 56 bytes we append zeros
219      * then compress.  Then we can fall back to padding zeros and length
220      * encoding like normal.
221      */
222     if (md->md4.curlen > 56) {

```

```

223     while (md->md4.curlen < 64) {
224         md->md4.buf[md->md4.curlen++] = (unsigned char)0;
225     }
226     md4_compress(md, md->md4.buf);
227     md->md4.curlen = 0;
228 }
229
230 /* pad upto 56 bytes of zeroes */
231 while (md->md4.curlen < 56) {
232     md->md4.buf[md->md4.curlen++] = (unsigned char)0;
233 }
234
235 /* store length */
236 STORE64L(md->md4.length, md->md4.buf+56);
237 md4_compress(md, md->md4.buf);
238
239 /* copy output */
240 for (i = 0; i < 4; i++) {
241     STORE32L(md->md4.state[i], out+(4*i));
242 }
243 #ifdef LTC_CLEAN_STACK
244     zeromem(md, sizeof(hash_state));
245 #endif
246     return CRYPT_OK;
247 }

```

Here is the call graph for this function:

### 5.58.2.3 int md4\_init (hash\_state \* md)

Initialize the hash state.

#### Parameters:

*md* The hash state you wish to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 174 of file md4.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by md4\_test().

```

175 {
176     LTC_ARGCHK(md != NULL);
177     md->md4.state[0] = 0x67452301UL;
178     md->md4.state[1] = 0xefcdab89UL;
179     md->md4.state[2] = 0x98badcfeUL;
180     md->md4.state[3] = 0x10325476UL;
181     md->md4.length = 0;
182     md->md4.curlen = 0;
183     return CRYPT_OK;
184 }

```

### 5.58.2.4 int md4\_test (void)

Self-test the hash.

**Returns:**

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 253 of file md4.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, md4\_done(), md4\_init(), and XMEMCMP.

```

254 {
255     #ifndef LTC_TEST
256         return CRYPT_NOP;
257     #else
258         static const struct md4_test_case {
259             char *input;
260             unsigned char digest[16];
261         } cases[] = {
262             { "",
263               {0x31, 0xd6, 0xcf, 0xe0, 0xd1, 0x6a, 0xe9, 0x31,
264                0xb7, 0x3c, 0x59, 0xd7, 0xe0, 0xc0, 0x89, 0xc0} },
265             { "a",
266               {0xbd, 0xe5, 0x2c, 0xb3, 0x1d, 0xe3, 0x3e, 0x46,
267                0x24, 0x5e, 0x05, 0xfb, 0xdb, 0xd6, 0xfb, 0x24} },
268             { "abc",
269               {0xa4, 0x48, 0x01, 0x7a, 0xaf, 0x21, 0xd8, 0x52,
270                0x5f, 0xc1, 0x0a, 0xe8, 0x7a, 0xa6, 0x72, 0x9d} },
271             { "message digest",
272               {0xd9, 0x13, 0x0a, 0x81, 0x64, 0x54, 0x9f, 0xe8,
273                0x18, 0x87, 0x48, 0x06, 0xe1, 0xc7, 0x01, 0x4b} },
274             { "abcdefghijklmnopqrstuvwxyz",
275               {0xd7, 0x9e, 0x1c, 0x30, 0x8a, 0xa5, 0xbb, 0xcd,
276                0xee, 0xa8, 0xed, 0x63, 0xdf, 0x41, 0x2d, 0xa9} },
277             { "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789",
278               {0x04, 0x3f, 0x85, 0x82, 0xf2, 0x41, 0xdb, 0x35,
279                0x1c, 0xe6, 0x27, 0xe1, 0x53, 0xe7, 0xf0, 0xe4} },
280             { "123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890",
281               {0xe3, 0x3b, 0x4d, 0xdc, 0x9c, 0x38, 0xf2, 0x19,
282                0x9c, 0x3e, 0x7b, 0x16, 0x4f, 0xcc, 0x05, 0x36} },
283         };
284         int i;
285         hash_state md;
286         unsigned char digest[16];
287
288         for(i = 0; i < (int)(sizeof(cases) / sizeof(cases[0])); i++) {
289             md4_init(&md);
290             md4_process(&md, (unsigned char *)cases[i].input, (unsigned long)strlen(cases[i].input));
291             md4_done(&md, digest);
292             if (XMEMCMP(digest, cases[i].digest, 16) != 0) {
293                 return CRYPT_FAIL_TESTVECTOR;
294             }
295         }
296         return CRYPT_OK;
297     #endif
298 }

```

Here is the call graph for this function:

### 5.58.3 Variable Documentation

#### 5.58.3.1 `const struct ltc_hash_descriptor md4_desc`

**Initial value:**

```
{
```

```
"md4",
6,
16,
64,

{ 1, 2, 840, 113549, 2, 4, },
6,

&md4_init,
&md4_process,
&md4_done,
&md4_test,
NULL
}
```

**Parameters:**

[\*md4.c\*](#) Submitted by Dobes Vandermeer ([dobes@smar tt .com](mailto:dobes@smar tt .com))

Definition at line 20 of file md4.c.

Referenced by yarrow\_start().

## 5.59 hashes/md5.c File Reference

### 5.59.1 Detailed Description

MD5 hash function by Tom St Denis.

Definition in file [md5.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for md5.c:

#### Defines

- `#define F(x, y, z) (z ^ (x & (y ^ z)))`
- `#define G(x, y, z) (y ^ (z & (y ^ x)))`
- `#define H(x, y, z) (x ^ y ^ z)`
- `#define I(x, y, z) (y ^ (x | (~z)))`
- `#define FF(a, b, c, d, M, s, t) a = (a + F(b,c,d) + M + t); a = ROLc(a, s) + b;`
- `#define GG(a, b, c, d, M, s, t) a = (a + G(b,c,d) + M + t); a = ROLc(a, s) + b;`
- `#define HH(a, b, c, d, M, s, t) a = (a + H(b,c,d) + M + t); a = ROLc(a, s) + b;`
- `#define II(a, b, c, d, M, s, t) a = (a + I(b,c,d) + M + t); a = ROLc(a, s) + b;`

#### Functions

- `static int md5_compress (hash_state *md, unsigned char *buf)`
- `int md5_init (hash_state *md)`  
*Initialize the hash state.*
- `int md5_done (hash_state *md, unsigned char *out)`  
*Terminate the hash to get the digest.*
- `int md5_test (void)`  
*Self-test the hash.*

#### Variables

- `const struct ltc_hash_descriptor md5_desc`

### 5.59.2 Define Documentation

#### 5.59.2.1 `#define F(x, y, z) (z ^ (x & (y ^ z)))`

Definition at line 39 of file md5.c.

#### 5.59.2.2 `#define FF(a, b, c, d, M, s, t) a = (a + F(b,c,d) + M + t); a = ROLc(a, s) + b;`

Definition at line 85 of file md5.c.

**5.59.2.3 #define G(x, y, z) (y ^ (z & (y ^ x)))**

Definition at line 40 of file md5.c.

**5.59.2.4 #define GG(a, b, c, d, M, s, t) a = (a + G(b,c,d) + M + t); a = ROLc(a, s) + b;**

Definition at line 88 of file md5.c.

**5.59.2.5 #define H(x, y, z) (x^y^z)**

Definition at line 41 of file md5.c.

**5.59.2.6 #define HH(a, b, c, d, M, s, t) a = (a + H(b,c,d) + M + t); a = ROLc(a, s) + b;**

Definition at line 91 of file md5.c.

**5.59.2.7 #define I(x, y, z) (y^(x|(~z)))**

Definition at line 42 of file md5.c.

Referenced by FI(), FII(), and FIII().

**5.59.2.8 #define II(a, b, c, d, M, s, t) a = (a + I(b,c,d) + M + t); a = ROLc(a, s) + b;**

Definition at line 94 of file md5.c.

**5.59.3 Function Documentation****5.59.3.1 static int md5\_compress (hash\_state \* md, unsigned char \* buf) [static]**

Definition at line 103 of file md5.c.

References c.

Referenced by md5\_done().

```

105 {
106     ulong32 i, W[16], a, b, c, d;
107     #ifdef LTC_SMALL_CODE
108         ulong32 t;
109     #endif
110
111     /* copy the state into 512-bits into W[0..15] */
112     for (i = 0; i < 16; i++) {
113         LOAD32L(W[i], buf + (4*i));
114     }
115
116     /* copy state */
117     a = md->md5.state[0];
118     b = md->md5.state[1];
119     c = md->md5.state[2];
120     d = md->md5.state[3];
121
122     #ifdef LTC_SMALL_CODE

```

```
123     for (i = 0; i < 16; ++i) {
124         FF(a,b,c,d,W[Worder[i]],Rorder[i],Korder[i]);
125         t = d; d = c; c = b; b = a; a = t;
126     }
127
128     for (; i < 32; ++i) {
129         GG(a,b,c,d,W[Worder[i]],Rorder[i],Korder[i]);
130         t = d; d = c; c = b; b = a; a = t;
131     }
132
133     for (; i < 48; ++i) {
134         HH(a,b,c,d,W[Worder[i]],Rorder[i],Korder[i]);
135         t = d; d = c; c = b; b = a; a = t;
136     }
137
138     for (; i < 64; ++i) {
139         II(a,b,c,d,W[Worder[i]],Rorder[i],Korder[i]);
140         t = d; d = c; c = b; b = a; a = t;
141     }
142
143 #else
144     FF(a,b,c,d,W[0],7,0xd76aa478UL)
145     FF(d,a,b,c,W[1],12,0xe8c7b756UL)
146     FF(c,d,a,b,W[2],17,0x242070dbUL)
147     FF(b,c,d,a,W[3],22,0xc1bdceeeUL)
148     FF(a,b,c,d,W[4],7,0xf57c0fafUL)
149     FF(d,a,b,c,W[5],12,0x4787c62aUL)
150     FF(c,d,a,b,W[6],17,0xa8304613UL)
151     FF(b,c,d,a,W[7],22,0xfd469501UL)
152     FF(a,b,c,d,W[8],7,0x698098d8UL)
153     FF(d,a,b,c,W[9],12,0x8b44f7afUL)
154     FF(c,d,a,b,W[10],17,0xfffff5bb1UL)
155     FF(b,c,d,a,W[11],22,0x895cd7beUL)
156     FF(a,b,c,d,W[12],7,0x6b901122UL)
157     FF(d,a,b,c,W[13],12,0xfd987193UL)
158     FF(c,d,a,b,W[14],17,0xa679438eUL)
159     FF(b,c,d,a,W[15],22,0x49b40821UL)
160     GG(a,b,c,d,W[1],5,0xf61e2562UL)
161     GG(d,a,b,c,W[6],9,0xc040b340UL)
162     GG(c,d,a,b,W[11],14,0x265e5a51UL)
163     GG(b,c,d,a,W[0],20,0xe9b6c7aaUL)
164     GG(a,b,c,d,W[5],5,0xd62f105dUL)
165     GG(d,a,b,c,W[10],9,0x02441453UL)
166     GG(c,d,a,b,W[15],14,0xd8a1e681UL)
167     GG(b,c,d,a,W[4],20,0xe7d3fbc8UL)
168     GG(a,b,c,d,W[9],5,0x21e1cde6UL)
169     GG(d,a,b,c,W[14],9,0xc33707d6UL)
170     GG(c,d,a,b,W[3],14,0xf4d50d87UL)
171     GG(b,c,d,a,W[8],20,0x455a14edUL)
172     GG(a,b,c,d,W[13],5,0xa9e3e905UL)
173     GG(d,a,b,c,W[2],9,0xfcefa3f8UL)
174     GG(c,d,a,b,W[7],14,0x676f02d9UL)
175     GG(b,c,d,a,W[12],20,0x8d2a4c8aUL)
176     HH(a,b,c,d,W[5],4,0xfffffa3942UL)
177     HH(d,a,b,c,W[8],11,0x8771f681UL)
178     HH(c,d,a,b,W[11],16,0x6d9d6122UL)
179     HH(b,c,d,a,W[14],23,0xfde5380cUL)
180     HH(a,b,c,d,W[1],4,0xa4beea44UL)
181     HH(d,a,b,c,W[4],11,0x4bdecfa9UL)
182     HH(c,d,a,b,W[7],16,0xf6bb4b60UL)
183     HH(b,c,d,a,W[10],23,0xbebfb70UL)
184     HH(a,b,c,d,W[13],4,0x289b7ec6UL)
185     HH(d,a,b,c,W[0],11,0xea127faUL)
186     HH(c,d,a,b,W[3],16,0xd4ef3085UL)
187     HH(b,c,d,a,W[6],23,0x04881d05UL)
188     HH(a,b,c,d,W[9],4,0xd9d4d039UL)
189     HH(d,a,b,c,W[12],11,0xe6db99e5UL)
```



```

190     HH(c,d,a,b,W[15],16,0x1fa27cf8UL)
191     HH(b,c,d,a,W[2],23,0xc4ac5665UL)
192     II(a,b,c,d,W[0],6,0xf4292244UL)
193     II(d,a,b,c,W[7],10,0x432aff97UL)
194     II(c,d,a,b,W[14],15,0xab9423a7UL)
195     II(b,c,d,a,W[5],21,0xfc93a039UL)
196     II(a,b,c,d,W[12],6,0x655b59c3UL)
197     II(d,a,b,c,W[3],10,0x8f0ccc92UL)
198     II(c,d,a,b,W[10],15,0xffeff47dUL)
199     II(b,c,d,a,W[1],21,0x85845dd1UL)
200     II(a,b,c,d,W[8],6,0x6fa87e4fUL)
201     II(d,a,b,c,W[15],10,0xfe2ce6e0UL)
202     II(c,d,a,b,W[6],15,0xa3014314UL)
203     II(b,c,d,a,W[13],21,0x4e0811a1UL)
204     II(a,b,c,d,W[4],6,0xf7537e82UL)
205     II(d,a,b,c,W[11],10,0xbd3af235UL)
206     II(c,d,a,b,W[2],15,0x2ad7d2bbUL)
207     II(b,c,d,a,W[9],21,0xeb86d391UL)
208 #endif
209
210     md->md5.state[0] = md->md5.state[0] + a;
211     md->md5.state[1] = md->md5.state[1] + b;
212     md->md5.state[2] = md->md5.state[2] + c;
213     md->md5.state[3] = md->md5.state[3] + d;
214
215     return CRYPT_OK;
216 }

```

### 5.59.3.2 int md5\_done (hash\_state \* md, unsigned char \* out)

Terminate the hash to get the digest.

#### Parameters:

- md* The hash state
- out* [out] The destination of the hash (16 bytes)

#### Returns:

CRYPT\_OK if successful

Definition at line 260 of file md5.c.

References CRYPT\_INVALID\_ARG, LTC\_ARGCHK, and md5\_compress().

Referenced by md5\_test().

```

261 {
262     int i;
263
264     LTC_ARGCHK(md != NULL);
265     LTC_ARGCHK(out != NULL);
266
267     if (md->md5.curlen >= sizeof(md->md5.buf)) {
268         return CRYPT_INVALID_ARG;
269     }
270
271
272     /* increase the length of the message */
273     md->md5.length += md->md5.curlen * 8;
274
275     /* append the '1' bit */
276     md->md5.buf[md->md5.curlen++] = (unsigned char)0x80;

```

```

277
278     /* if the length is currently above 56 bytes we append zeros
279     * then compress. Then we can fall back to padding zeros and length
280     * encoding like normal.
281     */
282     if (md->md5.curlen > 56) {
283         while (md->md5.curlen < 64) {
284             md->md5.buf[md->md5.curlen++] = (unsigned char)0;
285         }
286         md5_compress(md, md->md5.buf);
287         md->md5.curlen = 0;
288     }
289
290     /* pad upto 56 bytes of zeroes */
291     while (md->md5.curlen < 56) {
292         md->md5.buf[md->md5.curlen++] = (unsigned char)0;
293     }
294
295     /* store length */
296     STORE64L(md->md5.length, md->md5.buf+56);
297     md5_compress(md, md->md5.buf);
298
299     /* copy output */
300     for (i = 0; i < 4; i++) {
301         STORE32L(md->md5.state[i], out+(4*i));
302     }
303 #ifdef LTC_CLEAN_STACK
304     zeromem(md, sizeof(hash_state));
305 #endif
306     return CRYPT_OK;
307 }

```

Here is the call graph for this function:

### 5.59.3.3 int md5\_init ([hash\\_state](#) \* *md*)

Initialize the hash state.

#### Parameters:

*md* The hash state you wish to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 233 of file md5.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by md5\_test().

```

234 {
235     LTC_ARGCHK(md != NULL);
236     md->md5.state[0] = 0x67452301UL;
237     md->md5.state[1] = 0xefcdab89UL;
238     md->md5.state[2] = 0x98badcfeUL;
239     md->md5.state[3] = 0x10325476UL;
240     md->md5.curlen = 0;
241     md->md5.length = 0;
242     return CRYPT_OK;
243 }

```

**5.59.3.4 int md5\_test (void)**

Self-test the hash.

**Returns:**

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 313 of file md5.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, md5\_done(), md5\_init(), and XMEMCMP.

```

314 {
315     #ifndef LTC_TEST
316         return CRYPT_NOP;
317     #else
318         static const struct {
319             char *msg;
320             unsigned char hash[16];
321         } tests[] = {
322             { "",
323               { 0xd4, 0x1d, 0x8c, 0xd9, 0x8f, 0x00, 0xb2, 0x04,
324                 0xe9, 0x80, 0x09, 0x98, 0xec, 0xf8, 0x42, 0x7e } },
325             { "a",
326               { 0x0c, 0xc1, 0x75, 0xb9, 0xc0, 0xf1, 0xb6, 0xa8,
327                 0x31, 0xc3, 0x99, 0xe2, 0x69, 0x77, 0x26, 0x61 } },
328             { "abc",
329               { 0x90, 0x01, 0x50, 0x98, 0x3c, 0xd2, 0x4f, 0xb0,
330                 0xd6, 0x96, 0x3f, 0x7d, 0x28, 0xe1, 0x7f, 0x72 } },
331             { "message digest",
332               { 0xf9, 0x6b, 0x69, 0x7d, 0x7c, 0xb7, 0x93, 0x8d,
333                 0x52, 0x5a, 0x2f, 0x31, 0xaa, 0xf1, 0x61, 0xd0 } },
334             { "abcdefghijklmnopqrstuvwxyz",
335               { 0xc3, 0xfc, 0xd3, 0xd7, 0x61, 0x92, 0xe4, 0x00,
336                 0x7d, 0xfb, 0x49, 0x6c, 0xca, 0x67, 0xe1, 0x3b } },
337             { "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789",
338               { 0xd1, 0x74, 0xab, 0x98, 0xd2, 0x77, 0xd9, 0xf5,
339                 0xa5, 0x61, 0x1c, 0x2c, 0x9f, 0x41, 0x9d, 0x9f } },
340             { "12345678901234567890123456789012345678901234567890123456789012345678901234567890",
341               { 0x57, 0xed, 0xf4, 0xa2, 0x2b, 0xe3, 0xc9, 0x55,
342                 0xac, 0x49, 0xda, 0x2e, 0x21, 0x07, 0xb6, 0x7a } },
343             { NULL, { 0 } }
344         };
345
346         int i;
347         unsigned char tmp[16];
348         hash_state md;
349
350         for (i = 0; tests[i].msg != NULL; i++) {
351             md5_init(&md);
352             md5_process(&md, (unsigned char *)tests[i].msg, (unsigned long)strlen(tests[i].msg));
353             md5_done(&md, tmp);
354             if (XMEMCMP(tmp, tests[i].hash, 16) != 0) {
355                 return CRYPT_FAIL_TESTVECTOR;
356             }
357         }
358         return CRYPT_OK;
359     #endif
360 }

```

Here is the call graph for this function:

## 5.59.4 Variable Documentation

### 5.59.4.1 const struct [ltc\\_hash\\_descriptor](#) md5\_desc

**Initial value:**

```
{
    "md5",
    3,
    16,
    64,

    { 1, 2, 840, 113549, 2, 5, },
    6,

    &md5_init,
    &md5_process,
    &md5_done,
    &md5_test,
    NULL
}
```

Definition at line 21 of file md5.c.

Referenced by yarrow\_start().

## 5.60 hashes/rmd128.c File Reference

```
#include "tomcrypt.h"
```

Include dependency graph for rmd128.c:

### Defines

- `#define F(x, y, z) ((x) ^ (y) ^ (z))`
- `#define G(x, y, z) (((x) & (y)) | (~x & (z)))`
- `#define H(x, y, z) (((x) | ~y) ^ (z))`
- `#define I(x, y, z) (((x) & (z)) | ((y) & ~z))`
- `#define FF(a, b, c, d, x, s)`
- `#define GG(a, b, c, d, x, s)`
- `#define HH(a, b, c, d, x, s)`
- `#define II(a, b, c, d, x, s)`
- `#define FFF(a, b, c, d, x, s)`
- `#define GGG(a, b, c, d, x, s)`
- `#define HHH(a, b, c, d, x, s)`
- `#define III(a, b, c, d, x, s)`

### Functions

- `static int rmd128_compress (hash_state *md, unsigned char *buf)`
- `int rmd128_init (hash_state *md)`  
*Initialize the hash state.*
- `int rmd128_done (hash_state *md, unsigned char *out)`  
*Terminate the hash to get the digest.*
- `int rmd128_test (void)`  
*Self-test the hash.*

### Variables

- `const struct ltc_hash_descriptor rmd128_desc`

#### 5.60.1 Define Documentation

##### 5.60.1.1 `#define F(x, y, z) ((x) ^ (y) ^ (z))`

Definition at line 45 of file rmd128.c.

**5.60.1.2 #define FF(a, b, c, d, x, s)****Value:**

```
(a) += F((b), (c), (d)) + (x); \
(a) = ROTLc((a), (s));
```

Definition at line 51 of file rmd128.c.

**5.60.1.3 #define FFF(a, b, c, d, x, s)****Value:**

```
(a) += F((b), (c), (d)) + (x); \
(a) = ROTLc((a), (s));
```

Definition at line 67 of file rmd128.c.

**5.60.1.4 #define G(x, y, z) (((x) & (y)) | (~(x) & (z)))**

Definition at line 46 of file rmd128.c.

**5.60.1.5 #define GG(a, b, c, d, x, s)****Value:**

```
(a) += G((b), (c), (d)) + (x) + 0x5a827999UL; \
(a) = ROTLc((a), (s));
```

Definition at line 55 of file rmd128.c.

**5.60.1.6 #define GGG(a, b, c, d, x, s)****Value:**

```
(a) += G((b), (c), (d)) + (x) + 0x6d703ef3UL; \
(a) = ROTLc((a), (s));
```

Definition at line 71 of file rmd128.c.

**5.60.1.7 #define H(x, y, z) (((x) | ~(y)) ^ (z))**

Definition at line 47 of file rmd128.c.

**5.60.1.8 #define HH(a, b, c, d, x, s)****Value:**

```
(a) += H((b), (c), (d)) + (x) + 0x6ed9eba1UL; \
(a) = ROTLc((a), (s));
```

Definition at line 59 of file rmd128.c.

**5.60.1.9 #define HHH(a, b, c, d, x, s)****Value:**

```
(a) += H((b), (c), (d)) + (x) + 0x5c4dd124UL;\
(a) = ROTL((a), (s));
```

Definition at line 75 of file rmd128.c.

**5.60.1.10 #define I(x, y, z) (((x) & (z)) | ((y) & ~(z)))**

Definition at line 48 of file rmd128.c.

**5.60.1.11 #define II(a, b, c, d, x, s)****Value:**

```
(a) += I((b), (c), (d)) + (x) + 0x8f1bbcdcUL;\
(a) = ROTL((a), (s));
```

Definition at line 63 of file rmd128.c.

**5.60.1.12 #define III(a, b, c, d, x, s)****Value:**

```
(a) += I((b), (c), (d)) + (x) + 0x50a28be6UL;\
(a) = ROTL((a), (s));
```

Definition at line 79 of file rmd128.c.

**5.60.2 Function Documentation****5.60.2.1 static int rmd128\_compress (hash\_state \*md, unsigned char \*buf) [static]**

Definition at line 86 of file rmd128.c.

Referenced by rmd128\_done().

```
88 {
89     ulong32 aa,bb,cc,dd,aaa,bbb,ccc,ddd,X[16];
90     int i;
91
92     /* load words X */
93     for (i = 0; i < 16; i++){
94         LOAD32L(X[i], buf + (4 * i));
95     }
96
97     /* load state */
98     aa = aaa = md->rmd128.state[0];
99     bb = bbb = md->rmd128.state[1];
100     cc = ccc = md->rmd128.state[2];
101     dd = ddd = md->rmd128.state[3];
102 }
```

```
103  /* round 1 */
104  FF(aa, bb, cc, dd, X[ 0], 11);
105  FF(dd, aa, bb, cc, X[ 1], 14);
106  FF(cc, dd, aa, bb, X[ 2], 15);
107  FF(bb, cc, dd, aa, X[ 3], 12);
108  FF(aa, bb, cc, dd, X[ 4], 5);
109  FF(dd, aa, bb, cc, X[ 5], 8);
110  FF(cc, dd, aa, bb, X[ 6], 7);
111  FF(bb, cc, dd, aa, X[ 7], 9);
112  FF(aa, bb, cc, dd, X[ 8], 11);
113  FF(dd, aa, bb, cc, X[ 9], 13);
114  FF(cc, dd, aa, bb, X[10], 14);
115  FF(bb, cc, dd, aa, X[11], 15);
116  FF(aa, bb, cc, dd, X[12], 6);
117  FF(dd, aa, bb, cc, X[13], 7);
118  FF(cc, dd, aa, bb, X[14], 9);
119  FF(bb, cc, dd, aa, X[15], 8);
120
121  /* round 2 */
122  GG(aa, bb, cc, dd, X[ 7], 7);
123  GG(dd, aa, bb, cc, X[ 4], 6);
124  GG(cc, dd, aa, bb, X[13], 8);
125  GG(bb, cc, dd, aa, X[ 1], 13);
126  GG(aa, bb, cc, dd, X[10], 11);
127  GG(dd, aa, bb, cc, X[ 6], 9);
128  GG(cc, dd, aa, bb, X[15], 7);
129  GG(bb, cc, dd, aa, X[ 3], 15);
130  GG(aa, bb, cc, dd, X[12], 7);
131  GG(dd, aa, bb, cc, X[ 0], 12);
132  GG(cc, dd, aa, bb, X[ 9], 15);
133  GG(bb, cc, dd, aa, X[ 5], 9);
134  GG(aa, bb, cc, dd, X[ 2], 11);
135  GG(dd, aa, bb, cc, X[14], 7);
136  GG(cc, dd, aa, bb, X[11], 13);
137  GG(bb, cc, dd, aa, X[ 8], 12);
138
139  /* round 3 */
140  HH(aa, bb, cc, dd, X[ 3], 11);
141  HH(dd, aa, bb, cc, X[10], 13);
142  HH(cc, dd, aa, bb, X[14], 6);
143  HH(bb, cc, dd, aa, X[ 4], 7);
144  HH(aa, bb, cc, dd, X[ 9], 14);
145  HH(dd, aa, bb, cc, X[15], 9);
146  HH(cc, dd, aa, bb, X[ 8], 13);
147  HH(bb, cc, dd, aa, X[ 1], 15);
148  HH(aa, bb, cc, dd, X[ 2], 14);
149  HH(dd, aa, bb, cc, X[ 7], 8);
150  HH(cc, dd, aa, bb, X[ 0], 13);
151  HH(bb, cc, dd, aa, X[ 6], 6);
152  HH(aa, bb, cc, dd, X[13], 5);
153  HH(dd, aa, bb, cc, X[11], 12);
154  HH(cc, dd, aa, bb, X[ 5], 7);
155  HH(bb, cc, dd, aa, X[12], 5);
156
157  /* round 4 */
158  II(aa, bb, cc, dd, X[ 1], 11);
159  II(dd, aa, bb, cc, X[ 9], 12);
160  II(cc, dd, aa, bb, X[11], 14);
161  II(bb, cc, dd, aa, X[10], 15);
162  II(aa, bb, cc, dd, X[ 0], 14);
163  II(dd, aa, bb, cc, X[ 8], 15);
164  II(cc, dd, aa, bb, X[12], 9);
165  II(bb, cc, dd, aa, X[ 4], 8);
166  II(aa, bb, cc, dd, X[13], 9);
167  II(dd, aa, bb, cc, X[ 3], 14);
168  II(cc, dd, aa, bb, X[ 7], 5);
169  II(bb, cc, dd, aa, X[15], 6);
```



```
170     II(aa, bb, cc, dd, X[14], 8);
171     II(dd, aa, bb, cc, X[ 5], 6);
172     II(cc, dd, aa, bb, X[ 6], 5);
173     II(bb, cc, dd, aa, X[ 2], 12);
174
175     /* parallel round 1 */
176     III(aaa, bbb, ccc, ddd, X[ 5], 8);
177     III(ddd, aaa, bbb, ccc, X[14], 9);
178     III(ccc, ddd, aaa, bbb, X[ 7], 9);
179     III(bbb, ccc, ddd, aaa, X[ 0], 11);
180     III(aaa, bbb, ccc, ddd, X[ 9], 13);
181     III(ddd, aaa, bbb, ccc, X[ 2], 15);
182     III(ccc, ddd, aaa, bbb, X[11], 15);
183     III(bbb, ccc, ddd, aaa, X[ 4], 5);
184     III(aaa, bbb, ccc, ddd, X[13], 7);
185     III(ddd, aaa, bbb, ccc, X[ 6], 7);
186     III(ccc, ddd, aaa, bbb, X[15], 8);
187     III(bbb, ccc, ddd, aaa, X[ 8], 11);
188     III(aaa, bbb, ccc, ddd, X[ 1], 14);
189     III(ddd, aaa, bbb, ccc, X[10], 14);
190     III(ccc, ddd, aaa, bbb, X[ 3], 12);
191     III(bbb, ccc, ddd, aaa, X[12], 6);
192
193     /* parallel round 2 */
194     HHH(aaa, bbb, ccc, ddd, X[ 6], 9);
195     HHH(ddd, aaa, bbb, ccc, X[11], 13);
196     HHH(ccc, ddd, aaa, bbb, X[ 3], 15);
197     HHH(bbb, ccc, ddd, aaa, X[ 7], 7);
198     HHH(aaa, bbb, ccc, ddd, X[ 0], 12);
199     HHH(ddd, aaa, bbb, ccc, X[13], 8);
200     HHH(ccc, ddd, aaa, bbb, X[ 5], 9);
201     HHH(bbb, ccc, ddd, aaa, X[10], 11);
202     HHH(aaa, bbb, ccc, ddd, X[14], 7);
203     HHH(ddd, aaa, bbb, ccc, X[15], 7);
204     HHH(ccc, ddd, aaa, bbb, X[ 8], 12);
205     HHH(bbb, ccc, ddd, aaa, X[12], 7);
206     HHH(aaa, bbb, ccc, ddd, X[ 4], 6);
207     HHH(ddd, aaa, bbb, ccc, X[ 9], 15);
208     HHH(ccc, ddd, aaa, bbb, X[ 1], 13);
209     HHH(bbb, ccc, ddd, aaa, X[ 2], 11);
210
211     /* parallel round 3 */
212     GGG(aaa, bbb, ccc, ddd, X[15], 9);
213     GGG(ddd, aaa, bbb, ccc, X[ 5], 7);
214     GGG(ccc, ddd, aaa, bbb, X[ 1], 15);
215     GGG(bbb, ccc, ddd, aaa, X[ 3], 11);
216     GGG(aaa, bbb, ccc, ddd, X[ 7], 8);
217     GGG(ddd, aaa, bbb, ccc, X[14], 6);
218     GGG(ccc, ddd, aaa, bbb, X[ 6], 6);
219     GGG(bbb, ccc, ddd, aaa, X[ 9], 14);
220     GGG(aaa, bbb, ccc, ddd, X[11], 12);
221     GGG(ddd, aaa, bbb, ccc, X[ 8], 13);
222     GGG(ccc, ddd, aaa, bbb, X[12], 5);
223     GGG(bbb, ccc, ddd, aaa, X[ 2], 14);
224     GGG(aaa, bbb, ccc, ddd, X[10], 13);
225     GGG(ddd, aaa, bbb, ccc, X[ 0], 13);
226     GGG(ccc, ddd, aaa, bbb, X[ 4], 7);
227     GGG(bbb, ccc, ddd, aaa, X[13], 5);
228
229     /* parallel round 4 */
230     FFF(aaa, bbb, ccc, ddd, X[ 8], 15);
231     FFF(ddd, aaa, bbb, ccc, X[ 6], 5);
232     FFF(ccc, ddd, aaa, bbb, X[ 4], 8);
233     FFF(bbb, ccc, ddd, aaa, X[ 1], 11);
234     FFF(aaa, bbb, ccc, ddd, X[ 3], 14);
235     FFF(ddd, aaa, bbb, ccc, X[11], 14);
236     FFF(ccc, ddd, aaa, bbb, X[15], 6);
```

```

237     FFF(bbb, ccc, ddd, aaa, X[ 0], 14);
238     FFF(aaa, bbb, ccc, ddd, X[ 5],  6);
239     FFF(ddd, aaa, bbb, ccc, X[12],  9);
240     FFF(ccc, ddd, aaa, bbb, X[ 2], 12);
241     FFF(bbb, ccc, ddd, aaa, X[13],  9);
242     FFF(aaa, bbb, ccc, ddd, X[ 9], 12);
243     FFF(ddd, aaa, bbb, ccc, X[ 7],  5);
244     FFF(ccc, ddd, aaa, bbb, X[10], 15);
245     FFF(bbb, ccc, ddd, aaa, X[14],  8);
246
247     /* combine results */
248     ddd += cc + md->rmdbl28.state[1];          /* final result for MDbuf[0] */
249     md->rmdbl28.state[1] = md->rmdbl28.state[2] + dd + aaa;
250     md->rmdbl28.state[2] = md->rmdbl28.state[3] + aa + bbb;
251     md->rmdbl28.state[3] = md->rmdbl28.state[0] + bb + ccc;
252     md->rmdbl28.state[0] = ddd;
253
254     return CRYPT_OK;
255 }

```

### 5.60.2.2 int rmd128\_done (hash\_state \* md, unsigned char \* out)

Terminate the hash to get the digest.

#### Parameters:

*md* The hash state

*out* [out] The destination of the hash (16 bytes)

#### Returns:

CRYPT\_OK if successful

Definition at line 299 of file rmd128.c.

References CRYPT\_INVALID\_ARG, LTC\_ARGCHK, and rmd128\_compress().

Referenced by rmd128\_test().

```

300 {
301     int i;
302
303     LTC_ARGCHK(md != NULL);
304     LTC_ARGCHK(out != NULL);
305
306     if (md->rmdbl28.curlen >= sizeof(md->rmdbl28.buf)) {
307         return CRYPT_INVALID_ARG;
308     }
309
310
311     /* increase the length of the message */
312     md->rmdbl28.length += md->rmdbl28.curlen * 8;
313
314     /* append the '1' bit */
315     md->rmdbl28.buf[md->rmdbl28.curlen++] = (unsigned char)0x80;
316
317     /* if the length is currently above 56 bytes we append zeros
318      * then compress. Then we can fall back to padding zeros and length
319      * encoding like normal.
320      */
321     if (md->rmdbl28.curlen > 56) {
322         while (md->rmdbl28.curlen < 64) {
323             md->rmdbl28.buf[md->rmdbl28.curlen++] = (unsigned char)0;

```

```

324     }
325     rmd128_compress(md, md->rmd128.buf);
326     md->rmd128.curlen = 0;
327 }
328
329 /* pad upto 56 bytes of zeroes */
330 while (md->rmd128.curlen < 56) {
331     md->rmd128.buf[md->rmd128.curlen++] = (unsigned char)0;
332 }
333
334 /* store length */
335 STORE64L(md->rmd128.length, md->rmd128.buf+56);
336 rmd128_compress(md, md->rmd128.buf);
337
338 /* copy output */
339 for (i = 0; i < 4; i++) {
340     STORE32L(md->rmd128.state[i], out+(4*i));
341 }
342 #ifdef LTC_CLEAN_STACK
343     zeromem(md, sizeof(hash_state));
344 #endif
345     return CRYPT_OK;
346 }

```

Here is the call graph for this function:

#### 5.60.2.3 int rmd128\_init (hash\_state \* md)

Initialize the hash state.

##### Parameters:

*md* The hash state you wish to initialize

##### Returns:

CRYPT\_OK if successful

Definition at line 272 of file rmd128.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by rmd128\_test().

```

273 {
274     LTC_ARGCHK(md != NULL);
275     md->rmd128.state[0] = 0x67452301UL;
276     md->rmd128.state[1] = 0xefcdab89UL;
277     md->rmd128.state[2] = 0x98badcfeUL;
278     md->rmd128.state[3] = 0x10325476UL;
279     md->rmd128.curlen = 0;
280     md->rmd128.length = 0;
281     return CRYPT_OK;
282 }

```

#### 5.60.2.4 int rmd128\_test (void)

Self-test the hash.

##### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 352 of file rmd128.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, rmd128\_done(), rmd128\_init(), and XMEMCMP.

```

353 {
354 #ifndef LTC_TEST
355     return CRYPT_NOP;
356 #else
357     static const struct {
358         char *msg;
359         unsigned char md[16];
360     } tests[] = {
361     { "",
362       { 0xcd, 0xf2, 0x62, 0x13, 0xa1, 0x50, 0xdc, 0x3e,
363         0xcb, 0x61, 0x0f, 0x18, 0xf6, 0xb3, 0x8b, 0x46 }
364     },
365     { "a",
366       { 0x86, 0xbe, 0x7a, 0xfa, 0x33, 0x9d, 0x0f, 0xc7,
367         0xcf, 0xc7, 0x85, 0xe7, 0x2f, 0x57, 0x8d, 0x33 }
368     },
369     { "abc",
370       { 0xc1, 0x4a, 0x12, 0x19, 0x9c, 0x66, 0xe4, 0xba,
371         0x84, 0x63, 0x6b, 0x0f, 0x69, 0x14, 0x4c, 0x77 }
372     },
373     { "message digest",
374       { 0x9e, 0x32, 0x7b, 0x3d, 0x6e, 0x52, 0x30, 0x62,
375         0xaf, 0xc1, 0x13, 0x2d, 0x7d, 0xf9, 0xd1, 0xb8 }
376     },
377     { "abcdefghijklmnopqrstuvwxyz",
378       { 0xfd, 0x2a, 0xa6, 0x07, 0xf7, 0x1d, 0xc8, 0xf5,
379         0x10, 0x71, 0x49, 0x22, 0xb3, 0x71, 0x83, 0x4e }
380     },
381     { "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789",
382       { 0xd1, 0xe9, 0x59, 0xeb, 0x17, 0x9c, 0x91, 0x1f,
383         0xae, 0xa4, 0x62, 0x4c, 0x60, 0xc5, 0xc7, 0x02 }
384     }
385     };
386     int x;
387     unsigned char buf[16];
388     hash_state md;
389
390     for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
391         rmd128_init(&md);
392         rmd128_process(&md, (unsigned char *)tests[x].msg, strlen(tests[x].msg));
393         rmd128_done(&md, buf);
394         if (XMEMCMP(buf, tests[x].md, 16) != 0) {
395             #if 0
396             printf("Failed test %d\n", x);
397             #endif
398             return CRYPT_FAIL_TESTVECTOR;
399         }
400     }
401     return CRYPT_OK;
402 #endif
403 }

```

Here is the call graph for this function:

## 5.60.3 Variable Documentation

### 5.60.3.1 const struct `ltc_hash_descriptor` `rmd128_desc`

Initial value:

```
{
    "rmd128",
    8,
    16,
    64,

    { 1, 0, 10118, 3, 0, 50 },
    6,

    &rmd128_init,
    &rmd128_process,
    &rmd128_done,
    &rmd128_test,
    NULL
}
```

**Parameters:**

[\*rmd128.c\*](#) RMD128 Hash function

Definition at line 26 of file rmd128.c.

Referenced by yarrow\_start().

## 5.61 hashes/rmd160.c File Reference

### 5.61.1 Detailed Description

RMD160 hash function.

Definition in file [rmd160.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for rmd160.c:

### Defines

- `#define F(x, y, z) ((x) ^ (y) ^ (z))`
- `#define G(x, y, z) (((x) & (y)) | (~ (x) & (z)))`
- `#define H(x, y, z) (((x) | ~ (y)) ^ (z))`
- `#define I(x, y, z) (((x) & (z)) | ((y) & ~ (z)))`
- `#define J(x, y, z) ((x) ^ ((y) | ~ (z)))`
- `#define FF(a, b, c, d, e, x, s)`
- `#define GG(a, b, c, d, e, x, s)`
- `#define HH(a, b, c, d, e, x, s)`
- `#define II(a, b, c, d, e, x, s)`
- `#define JJ(a, b, c, d, e, x, s)`
- `#define FFF(a, b, c, d, e, x, s)`
- `#define GGG(a, b, c, d, e, x, s)`
- `#define HHH(a, b, c, d, e, x, s)`
- `#define III(a, b, c, d, e, x, s)`
- `#define JJJ(a, b, c, d, e, x, s)`

### Functions

- static int [rmd160\\_compress](#) ([hash\\_state](#) \*md, unsigned char \*buf)
- int [rmd160\\_init](#) ([hash\\_state](#) \*md)  
*Initialize the hash state.*
- int [rmd160\\_done](#) ([hash\\_state](#) \*md, unsigned char \*out)  
*Terminate the hash to get the digest.*
- int [rmd160\\_test](#) (void)  
*Self-test the hash.*

### Variables

- const struct [ltc\\_hash\\_descriptor](#) [rmd160\\_desc](#)

## 5.61.2 Define Documentation

### 5.61.2.1 #define F(x, y, z) ((x) ^ (y) ^ (z))

Definition at line 45 of file rmd160.c.

### 5.61.2.2 #define FF(a, b, c, d, e, x, s)

**Value:**

```
(a) += F((b), (c), (d)) + (x); \
(a) = ROTLc((a), (s)) + (e); \
(c) = ROTLc((c), 10);
```

Definition at line 52 of file rmd160.c.

### 5.61.2.3 #define FFF(a, b, c, d, e, x, s)

**Value:**

```
(a) += F((b), (c), (d)) + (x); \
(a) = ROTLc((a), (s)) + (e); \
(c) = ROTLc((c), 10);
```

Definition at line 77 of file rmd160.c.

### 5.61.2.4 #define G(x, y, z) (((x) & (y)) | (~(x) & (z)))

Definition at line 46 of file rmd160.c.

### 5.61.2.5 #define GG(a, b, c, d, e, x, s)

**Value:**

```
(a) += G((b), (c), (d)) + (x) + 0x5a827999UL; \
(a) = ROTLc((a), (s)) + (e); \
(c) = ROTLc((c), 10);
```

Definition at line 57 of file rmd160.c.

### 5.61.2.6 #define GGG(a, b, c, d, e, x, s)

**Value:**

```
(a) += G((b), (c), (d)) + (x) + 0x7a6d76e9UL; \
(a) = ROTLc((a), (s)) + (e); \
(c) = ROTLc((c), 10);
```

Definition at line 82 of file rmd160.c.

**5.61.2.7 #define H(x, y, z) (((x) | ~(y)) ^ (z))**

Definition at line 47 of file rmd160.c.

**5.61.2.8 #define HH(a, b, c, d, e, x, s)**

**Value:**

```
(a) += H((b), (c), (d)) + (x) + 0x6ed9eba1UL;\
(a) = ROTLc((a), (s)) + (e);\
(c) = ROTLc((c), 10);
```

Definition at line 62 of file rmd160.c.

**5.61.2.9 #define HHH(a, b, c, d, e, x, s)**

**Value:**

```
(a) += H((b), (c), (d)) + (x) + 0x6d703ef3UL;\
(a) = ROTLc((a), (s)) + (e);\
(c) = ROTLc((c), 10);
```

Definition at line 87 of file rmd160.c.

**5.61.2.10 #define I(x, y, z) (((x) & (z)) | ((y) & ~(z)))**

Definition at line 48 of file rmd160.c.

**5.61.2.11 #define II(a, b, c, d, e, x, s)**

**Value:**

```
(a) += I((b), (c), (d)) + (x) + 0x8f1bbcdcUL;\
(a) = ROTLc((a), (s)) + (e);\
(c) = ROTLc((c), 10);
```

Definition at line 67 of file rmd160.c.

**5.61.2.12 #define III(a, b, c, d, e, x, s)**

**Value:**

```
(a) += I((b), (c), (d)) + (x) + 0x5c4dd124UL;\
(a) = ROTLc((a), (s)) + (e);\
(c) = ROTLc((c), 10);
```

Definition at line 92 of file rmd160.c.

**5.61.2.13 #define J(x, y, z) ((x) ^ ((y) | ~(z)))**

Definition at line 49 of file rmd160.c.



**5.61.2.14 #define JJ(a, b, c, d, e, x, s)****Value:**

```
(a) += J((b), (c), (d)) + (x) + 0xa953fd4eUL;\
(a) = ROLc((a), (s)) + (e);\
(c) = ROLc((c), 10);
```

Definition at line 72 of file rmd160.c.

**5.61.2.15 #define JJJ(a, b, c, d, e, x, s)****Value:**

```
(a) += J((b), (c), (d)) + (x) + 0x50a28be6UL;\
(a) = ROLc((a), (s)) + (e);\
(c) = ROLc((c), 10);
```

Definition at line 97 of file rmd160.c.

**5.61.3 Function Documentation****5.61.3.1 static int rmd160\_compress (hash\_state \*md, unsigned char \*buf) [static]**

Definition at line 106 of file rmd160.c.

Referenced by rmd160\_done().

```
108 {
109     ulong32 aa,bb,cc,dd,ee,aaa,bbb,ccc,ddd,eee,X[16];
110     int i;
111
112     /* load words X */
113     for (i = 0; i < 16; i++){
114         LOAD32L(X[i], buf + (4 * i));
115     }
116
117     /* load state */
118     aa = aaa = md->rmd160.state[0];
119     bb = bbb = md->rmd160.state[1];
120     cc = ccc = md->rmd160.state[2];
121     dd = ddd = md->rmd160.state[3];
122     ee = eee = md->rmd160.state[4];
123
124     /* round 1 */
125     FF(aa, bb, cc, dd, ee, X[ 0], 11);
126     FF(ee, aa, bb, cc, dd, X[ 1], 14);
127     FF(dd, ee, aa, bb, cc, X[ 2], 15);
128     FF(cc, dd, ee, aa, bb, X[ 3], 12);
129     FF(bb, cc, dd, ee, aa, X[ 4],  5);
130     FF(aa, bb, cc, dd, ee, X[ 5],  8);
131     FF(ee, aa, bb, cc, dd, X[ 6],  7);
132     FF(dd, ee, aa, bb, cc, X[ 7],  9);
133     FF(cc, dd, ee, aa, bb, X[ 8], 11);
134     FF(bb, cc, dd, ee, aa, X[ 9], 13);
135     FF(aa, bb, cc, dd, ee, X[10], 14);
136     FF(ee, aa, bb, cc, dd, X[11], 15);
137     FF(dd, ee, aa, bb, cc, X[12],  6);
138     FF(cc, dd, ee, aa, bb, X[13],  7);
```

```
139 FF(bb, cc, dd, ee, aa, X[14], 9);
140 FF(aa, bb, cc, dd, ee, X[15], 8);
141
142 /* round 2 */
143 GG(ee, aa, bb, cc, dd, X[ 7], 7);
144 GG(dd, ee, aa, bb, cc, X[ 4], 6);
145 GG(cc, dd, ee, aa, bb, X[13], 8);
146 GG(bb, cc, dd, ee, aa, X[ 1], 13);
147 GG(aa, bb, cc, dd, ee, X[10], 11);
148 GG(ee, aa, bb, cc, dd, X[ 6], 9);
149 GG(dd, ee, aa, bb, cc, X[15], 7);
150 GG(cc, dd, ee, aa, bb, X[ 3], 15);
151 GG(bb, cc, dd, ee, aa, X[12], 7);
152 GG(aa, bb, cc, dd, ee, X[ 0], 12);
153 GG(ee, aa, bb, cc, dd, X[ 9], 15);
154 GG(dd, ee, aa, bb, cc, X[ 5], 9);
155 GG(cc, dd, ee, aa, bb, X[ 2], 11);
156 GG(bb, cc, dd, ee, aa, X[14], 7);
157 GG(aa, bb, cc, dd, ee, X[11], 13);
158 GG(ee, aa, bb, cc, dd, X[ 8], 12);
159
160 /* round 3 */
161 HH(dd, ee, aa, bb, cc, X[ 3], 11);
162 HH(cc, dd, ee, aa, bb, X[10], 13);
163 HH(bb, cc, dd, ee, aa, X[14], 6);
164 HH(aa, bb, cc, dd, ee, X[ 4], 7);
165 HH(ee, aa, bb, cc, dd, X[ 9], 14);
166 HH(dd, ee, aa, bb, cc, X[15], 9);
167 HH(cc, dd, ee, aa, bb, X[ 8], 13);
168 HH(bb, cc, dd, ee, aa, X[ 1], 15);
169 HH(aa, bb, cc, dd, ee, X[ 2], 14);
170 HH(ee, aa, bb, cc, dd, X[ 7], 8);
171 HH(dd, ee, aa, bb, cc, X[ 0], 13);
172 HH(cc, dd, ee, aa, bb, X[ 6], 6);
173 HH(bb, cc, dd, ee, aa, X[13], 5);
174 HH(aa, bb, cc, dd, ee, X[11], 12);
175 HH(ee, aa, bb, cc, dd, X[ 5], 7);
176 HH(dd, ee, aa, bb, cc, X[12], 5);
177
178 /* round 4 */
179 II(cc, dd, ee, aa, bb, X[ 1], 11);
180 II(bb, cc, dd, ee, aa, X[ 9], 12);
181 II(aa, bb, cc, dd, ee, X[11], 14);
182 II(ee, aa, bb, cc, dd, X[10], 15);
183 II(dd, ee, aa, bb, cc, X[ 0], 14);
184 II(cc, dd, ee, aa, bb, X[ 8], 15);
185 II(bb, cc, dd, ee, aa, X[12], 9);
186 II(aa, bb, cc, dd, ee, X[ 4], 8);
187 II(ee, aa, bb, cc, dd, X[13], 9);
188 II(dd, ee, aa, bb, cc, X[ 3], 14);
189 II(cc, dd, ee, aa, bb, X[ 7], 5);
190 II(bb, cc, dd, ee, aa, X[15], 6);
191 II(aa, bb, cc, dd, ee, X[14], 8);
192 II(ee, aa, bb, cc, dd, X[ 5], 6);
193 II(dd, ee, aa, bb, cc, X[ 6], 5);
194 II(cc, dd, ee, aa, bb, X[ 2], 12);
195
196 /* round 5 */
197 JJ(bb, cc, dd, ee, aa, X[ 4], 9);
198 JJ(aa, bb, cc, dd, ee, X[ 0], 15);
199 JJ(ee, aa, bb, cc, dd, X[ 5], 5);
200 JJ(dd, ee, aa, bb, cc, X[ 9], 11);
201 JJ(cc, dd, ee, aa, bb, X[ 7], 6);
202 JJ(bb, cc, dd, ee, aa, X[12], 8);
203 JJ(aa, bb, cc, dd, ee, X[ 2], 13);
204 JJ(ee, aa, bb, cc, dd, X[10], 12);
205 JJ(dd, ee, aa, bb, cc, X[14], 5);
```

```
206 JJ(cc, dd, ee, aa, bb, X[ 1], 12);
207 JJ(bb, cc, dd, ee, aa, X[ 3], 13);
208 JJ(aa, bb, cc, dd, ee, X[ 8], 14);
209 JJ(ee, aa, bb, cc, dd, X[11], 11);
210 JJ(dd, ee, aa, bb, cc, X[ 6], 8);
211 JJ(cc, dd, ee, aa, bb, X[15], 5);
212 JJ(bb, cc, dd, ee, aa, X[13], 6);
213
214 /* parallel round 1 */
215 JJJ(aaa, bbb, ccc, ddd, eee, X[ 5], 8);
216 JJJ(eee, aaa, bbb, ccc, ddd, X[14], 9);
217 JJJ(ddd, eee, aaa, bbb, ccc, X[ 7], 9);
218 JJJ(ccc, ddd, eee, aaa, bbb, X[ 0], 11);
219 JJJ(bbb, ccc, ddd, eee, aaa, X[ 9], 13);
220 JJJ(aaa, bbb, ccc, ddd, eee, X[ 2], 15);
221 JJJ(eee, aaa, bbb, ccc, ddd, X[11], 15);
222 JJJ(ddd, eee, aaa, bbb, ccc, X[ 4], 5);
223 JJJ(ccc, ddd, eee, aaa, bbb, X[13], 7);
224 JJJ(bbb, ccc, ddd, eee, aaa, X[ 6], 7);
225 JJJ(aaa, bbb, ccc, ddd, eee, X[15], 8);
226 JJJ(eee, aaa, bbb, ccc, ddd, X[ 8], 11);
227 JJJ(ddd, eee, aaa, bbb, ccc, X[ 1], 14);
228 JJJ(ccc, ddd, eee, aaa, bbb, X[10], 14);
229 JJJ(bbb, ccc, ddd, eee, aaa, X[ 3], 12);
230 JJJ(aaa, bbb, ccc, ddd, eee, X[12], 6);
231
232 /* parallel round 2 */
233 III(eee, aaa, bbb, ccc, ddd, X[ 6], 9);
234 III(ddd, eee, aaa, bbb, ccc, X[11], 13);
235 III(ccc, ddd, eee, aaa, bbb, X[ 3], 15);
236 III(bbb, ccc, ddd, eee, aaa, X[ 7], 7);
237 III(aaa, bbb, ccc, ddd, eee, X[ 0], 12);
238 III(eee, aaa, bbb, ccc, ddd, X[13], 8);
239 III(ddd, eee, aaa, bbb, ccc, X[ 5], 9);
240 III(ccc, ddd, eee, aaa, bbb, X[10], 11);
241 III(bbb, ccc, ddd, eee, aaa, X[14], 7);
242 III(aaa, bbb, ccc, ddd, eee, X[15], 7);
243 III(eee, aaa, bbb, ccc, ddd, X[ 8], 12);
244 III(ddd, eee, aaa, bbb, ccc, X[12], 7);
245 III(ccc, ddd, eee, aaa, bbb, X[ 4], 6);
246 III(bbb, ccc, ddd, eee, aaa, X[ 9], 15);
247 III(aaa, bbb, ccc, ddd, eee, X[ 1], 13);
248 III(eee, aaa, bbb, ccc, ddd, X[ 2], 11);
249
250 /* parallel round 3 */
251 HHH(ddd, eee, aaa, bbb, ccc, X[15], 9);
252 HHH(ccc, ddd, eee, aaa, bbb, X[ 5], 7);
253 HHH(bbb, ccc, ddd, eee, aaa, X[ 1], 15);
254 HHH(aaa, bbb, ccc, ddd, eee, X[ 3], 11);
255 HHH(eee, aaa, bbb, ccc, ddd, X[ 7], 8);
256 HHH(ddd, eee, aaa, bbb, ccc, X[14], 6);
257 HHH(ccc, ddd, eee, aaa, bbb, X[ 6], 6);
258 HHH(bbb, ccc, ddd, eee, aaa, X[ 9], 14);
259 HHH(aaa, bbb, ccc, ddd, eee, X[11], 12);
260 HHH(eee, aaa, bbb, ccc, ddd, X[ 8], 13);
261 HHH(ddd, eee, aaa, bbb, ccc, X[12], 5);
262 HHH(ccc, ddd, eee, aaa, bbb, X[ 2], 14);
263 HHH(bbb, ccc, ddd, eee, aaa, X[10], 13);
264 HHH(aaa, bbb, ccc, ddd, eee, X[ 0], 13);
265 HHH(eee, aaa, bbb, ccc, ddd, X[ 4], 7);
266 HHH(ddd, eee, aaa, bbb, ccc, X[13], 5);
267
268 /* parallel round 4 */
269 GGG(ccc, ddd, eee, aaa, bbb, X[ 8], 15);
270 GGG(bbb, ccc, ddd, eee, aaa, X[ 6], 5);
271 GGG(aaa, bbb, ccc, ddd, eee, X[ 4], 8);
272 GGG(eee, aaa, bbb, ccc, ddd, X[ 1], 11);
```

```

273     GGG(ddd, eee, aaa, bbb, ccc, X[ 3], 14);
274     GGG(ccc, ddd, eee, aaa, bbb, X[11], 14);
275     GGG(bbb, ccc, ddd, eee, aaa, X[15],  6);
276     GGG(aaa, bbb, ccc, ddd, eee, X[ 0], 14);
277     GGG(eee, aaa, bbb, ccc, ddd, X[ 5],  6);
278     GGG(ddd, eee, aaa, bbb, ccc, X[12],  9);
279     GGG(ccc, ddd, eee, aaa, bbb, X[ 2], 12);
280     GGG(bbb, ccc, ddd, eee, aaa, X[13],  9);
281     GGG(aaa, bbb, ccc, ddd, eee, X[ 9], 12);
282     GGG(eee, aaa, bbb, ccc, ddd, X[ 7],  5);
283     GGG(ddd, eee, aaa, bbb, ccc, X[10], 15);
284     GGG(ccc, ddd, eee, aaa, bbb, X[14],  8);
285
286     /* parallel round 5 */
287     FFF(bbb, ccc, ddd, eee, aaa, X[12] ,  8);
288     FFF(aaa, bbb, ccc, ddd, eee, X[15] ,  5);
289     FFF(eee, aaa, bbb, ccc, ddd, X[10] , 12);
290     FFF(ddd, eee, aaa, bbb, ccc, X[ 4] ,  9);
291     FFF(ccc, ddd, eee, aaa, bbb, X[ 1] , 12);
292     FFF(bbb, ccc, ddd, eee, aaa, X[ 5] ,  5);
293     FFF(aaa, bbb, ccc, ddd, eee, X[ 8] , 14);
294     FFF(eee, aaa, bbb, ccc, ddd, X[ 7] ,  6);
295     FFF(ddd, eee, aaa, bbb, ccc, X[ 6] ,  8);
296     FFF(ccc, ddd, eee, aaa, bbb, X[ 2] , 13);
297     FFF(bbb, ccc, ddd, eee, aaa, X[13] ,  6);
298     FFF(aaa, bbb, ccc, ddd, eee, X[14] ,  5);
299     FFF(eee, aaa, bbb, ccc, ddd, X[ 0] , 15);
300     FFF(ddd, eee, aaa, bbb, ccc, X[ 3] , 13);
301     FFF(ccc, ddd, eee, aaa, bbb, X[ 9] , 11);
302     FFF(bbb, ccc, ddd, eee, aaa, X[11] , 11);
303
304     /* combine results */
305     ddd += cc + md->rmd160.state[1];          /* final result for md->rmd160.state[0] */
306     md->rmd160.state[1] = md->rmd160.state[2] + dd + eee;
307     md->rmd160.state[2] = md->rmd160.state[3] + ee + aaa;
308     md->rmd160.state[3] = md->rmd160.state[4] + aa + bbb;
309     md->rmd160.state[4] = md->rmd160.state[0] + bb + ccc;
310     md->rmd160.state[0] = ddd;
311
312     return CRYPT_OK;
313 }

```

### 5.61.3.2 int rmd160\_done ([hash\\_state](#) \* *md*, unsigned char \* *out*)

Terminate the hash to get the digest.

#### Parameters:

*md* The hash state

*out* [out] The destination of the hash (20 bytes)

#### Returns:

CRYPT\_OK if successful

Definition at line 358 of file rmd160.c.

References CRYPT\_INVALID\_ARG, LTC\_ARGCHK, and rmd160\_compress().

Referenced by rmd160\_test().

```

359 {
360     int i;

```

```

361
362     LTC_ARGCHK(md != NULL);
363     LTC_ARGCHK(out != NULL);
364
365     if (md->rmd160.curlen >= sizeof(md->rmd160.buf)) {
366         return CRYPT_INVALID_ARG;
367     }
368
369
370     /* increase the length of the message */
371     md->rmd160.length += md->rmd160.curlen * 8;
372
373     /* append the '1' bit */
374     md->rmd160.buf[md->rmd160.curlen++] = (unsigned char)0x80;
375
376     /* if the length is currently above 56 bytes we append zeros
377      * then compress.  Then we can fall back to padding zeros and length
378      * encoding like normal.
379      */
380     if (md->rmd160.curlen > 56) {
381         while (md->rmd160.curlen < 64) {
382             md->rmd160.buf[md->rmd160.curlen++] = (unsigned char)0;
383         }
384         rmd160_compress(md, md->rmd160.buf);
385         md->rmd160.curlen = 0;
386     }
387
388     /* pad upto 56 bytes of zeroes */
389     while (md->rmd160.curlen < 56) {
390         md->rmd160.buf[md->rmd160.curlen++] = (unsigned char)0;
391     }
392
393     /* store length */
394     STORE64L(md->rmd160.length, md->rmd160.buf+56);
395     rmd160_compress(md, md->rmd160.buf);
396
397     /* copy output */
398     for (i = 0; i < 5; i++) {
399         STORE32L(md->rmd160.state[i], out+(4*i));
400     }
401 #ifdef LTC_CLEAN_STACK
402     zeromem(md, sizeof(hash_state));
403 #endif
404     return CRYPT_OK;
405 }

```

Here is the call graph for this function:

### 5.61.3.3 int rmd160\_init (hash\_state \* md)

Initialize the hash state.

#### Parameters:

*md* The hash state you wish to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 330 of file rmd160.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by rmd160\_test().

```

331 {
332     LTC_ARGCHK(md != NULL);
333     md->rmd160.state[0] = 0x67452301UL;
334     md->rmd160.state[1] = 0xefcdab89UL;
335     md->rmd160.state[2] = 0x98badcfeUL;
336     md->rmd160.state[3] = 0x10325476UL;
337     md->rmd160.state[4] = 0xc3d2e1f0UL;
338     md->rmd160.curlen = 0;
339     md->rmd160.length = 0;
340     return CRYPT_OK;
341 }

```

### 5.61.3.4 int rmd160\_test (void)

Self-test the hash.

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 411 of file rmd160.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, rmd160\_done(), rmd160\_init(), and XMEMPMP.

```

412 {
413 #ifndef LTC_TEST
414     return CRYPT_NOP;
415 #else
416     static const struct {
417         char *msg;
418         unsigned char md[20];
419     } tests[] = {
420     { "",
421       { 0x9c, 0x11, 0x85, 0xa5, 0xc5, 0xe9, 0xfc, 0x54, 0x61, 0x28,
422         0x08, 0x97, 0x7e, 0xe8, 0xf5, 0x48, 0xb2, 0x25, 0x8d, 0x31 }
423     },
424     { "a",
425       { 0x0b, 0xdc, 0x9d, 0x2d, 0x25, 0x6b, 0x3e, 0xe9, 0xda, 0xae,
426         0x34, 0x7b, 0xe6, 0xf4, 0xdc, 0x83, 0x5a, 0x46, 0x7f, 0xfe }
427     },
428     { "abc",
429       { 0x8e, 0xb2, 0x08, 0xf7, 0xe0, 0x5d, 0x98, 0x7a, 0x9b, 0x04,
430         0x4a, 0x8e, 0x98, 0xc6, 0xb0, 0x87, 0xf1, 0x5a, 0x0b, 0xfc }
431     },
432     { "message digest",
433       { 0x5d, 0x06, 0x89, 0xef, 0x49, 0xd2, 0xfa, 0xe5, 0x72, 0xb8,
434         0x81, 0xb1, 0x23, 0xa8, 0x5f, 0xfa, 0x21, 0x59, 0x5f, 0x36 }
435     },
436     { "abcdefghijklmnopqrstuvwxyz",
437       { 0xf7, 0x1c, 0x27, 0x10, 0x9c, 0x69, 0x2c, 0x1b, 0x56, 0xbb,
438         0xdc, 0xeb, 0x5b, 0x9d, 0x28, 0x65, 0xb3, 0x70, 0x8d, 0xbc }
439     },
440     { "abdcdbcdcedfdefgefghfghighijhijkijklklmklmnlmnomnopnopq",
441       { 0x12, 0xa0, 0x53, 0x38, 0x4a, 0x9c, 0x0c, 0x88, 0xe4, 0x05,
442         0xa0, 0x6c, 0x27, 0xdc, 0xf4, 0x9a, 0xda, 0x62, 0xeb, 0x2b }
443     }
444     };
445     int x;
446     unsigned char buf[20];
447     hash_state md;
448
449     for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
450         rmd160_init(&md);

```

```
451     rmd160_process(&md, (unsigned char *)tests[x].msg, strlen(tests[x].msg));
452     rmd160_done(&md, buf);
453     if (XMEMCMP(buf, tests[x].md, 20) != 0) {
454 #if 0
455         printf("Failed test %d\n", x);
456 #endif
457         return CRYPT_FAIL_TESTVECTOR;
458     }
459 }
460 return CRYPT_OK;
461 #endif
462 }
```

Here is the call graph for this function:

### 5.61.4 Variable Documentation

#### 5.61.4.1 const struct [ltc\\_hash\\_descriptor](#) [rmd160\\_desc](#)

**Initial value:**

```
{
    "rmd160",
    9,
    20,
    64,

    { 1, 3, 36, 3, 2, 1, },
    6,

    &rmd160_init,
    &rmd160_process,
    &rmd160_done,
    &rmd160_test,
    NULL
}
```

Definition at line 26 of file rmd160.c.

Referenced by [yarrow\\_start\(\)](#).

## 5.62 hashes/rmd256.c File Reference

```
#include "tomcrypt.h"
```

Include dependency graph for rmd256.c:

### Defines

- `#define F(x, y, z) ((x) ^ (y) ^ (z))`
- `#define G(x, y, z) (((x) & (y)) | (~x & (z)))`
- `#define H(x, y, z) (((x) | ~y) ^ (z))`
- `#define I(x, y, z) (((x) & (z)) | ((y) & ~z))`
- `#define FF(a, b, c, d, x, s)`
- `#define GG(a, b, c, d, x, s)`
- `#define HH(a, b, c, d, x, s)`
- `#define II(a, b, c, d, x, s)`
- `#define FFF(a, b, c, d, x, s)`
- `#define GGG(a, b, c, d, x, s)`
- `#define HHH(a, b, c, d, x, s)`
- `#define III(a, b, c, d, x, s)`

### Functions

- `static int rmd256_compress (hash_state *md, unsigned char *buf)`
- `int rmd256_init (hash_state *md)`  
*Initialize the hash state.*
- `int rmd256_done (hash_state *md, unsigned char *out)`  
*Terminate the hash to get the digest.*
- `int rmd256_test (void)`  
*Self-test the hash.*

### Variables

- `const struct ltc_hash_descriptor rmd256_desc`

#### 5.62.1 Define Documentation

##### 5.62.1.1 `#define F(x, y, z) ((x) ^ (y) ^ (z))`

Definition at line 39 of file rmd256.c.



**5.62.1.2 #define FF(a, b, c, d, x, s)****Value:**

```
(a) += F((b), (c), (d)) + (x); \
(a) = ROTLc((a), (s));
```

Definition at line 45 of file rmd256.c.

**5.62.1.3 #define FFF(a, b, c, d, x, s)****Value:**

```
(a) += F((b), (c), (d)) + (x); \
(a) = ROTLc((a), (s));
```

Definition at line 61 of file rmd256.c.

**5.62.1.4 #define G(x, y, z) (((x) & (y)) | (~(x) & (z)))**

Definition at line 40 of file rmd256.c.

**5.62.1.5 #define GG(a, b, c, d, x, s)****Value:**

```
(a) += G((b), (c), (d)) + (x) + 0x5a827999UL; \
(a) = ROTLc((a), (s));
```

Definition at line 49 of file rmd256.c.

**5.62.1.6 #define GGG(a, b, c, d, x, s)****Value:**

```
(a) += G((b), (c), (d)) + (x) + 0x6d703ef3UL; \
(a) = ROTLc((a), (s));
```

Definition at line 65 of file rmd256.c.

**5.62.1.7 #define H(x, y, z) (((x) | ~(y)) ^ (z))**

Definition at line 41 of file rmd256.c.

**5.62.1.8 #define HH(a, b, c, d, x, s)****Value:**

```
(a) += H((b), (c), (d)) + (x) + 0x6ed9eba1UL; \
(a) = ROTLc((a), (s));
```

Definition at line 53 of file rmd256.c.

**5.62.1.9 #define HHH(a, b, c, d, x, s)****Value:**

```
(a) += H((b), (c), (d)) + (x) + 0x5c4dd124UL;\
(a) = ROTL((a), (s));
```

Definition at line 69 of file rmd256.c.

**5.62.1.10 #define I(x, y, z) (((x) & (z)) | ((y) & ~(z)))**

Definition at line 42 of file rmd256.c.

**5.62.1.11 #define II(a, b, c, d, x, s)****Value:**

```
(a) += I((b), (c), (d)) + (x) + 0x8f1bbcdcUL;\
(a) = ROTL((a), (s));
```

Definition at line 57 of file rmd256.c.

**5.62.1.12 #define III(a, b, c, d, x, s)****Value:**

```
(a) += I((b), (c), (d)) + (x) + 0x50a28be6UL;\
(a) = ROTL((a), (s));
```

Definition at line 73 of file rmd256.c.

**5.62.2 Function Documentation****5.62.2.1 static int rmd256\_compress (hash\_state \*md, unsigned char \*buf) [static]**

Definition at line 80 of file rmd256.c.

Referenced by rmd256\_done().

```
82 {
83     ulong32 aa,bb,cc,dd,aaa,bbb,ccc,ddd,tmp,X[16];
84     int i;
85
86     /* load words X */
87     for (i = 0; i < 16; i++){
88         LOAD32L(X[i], buf + (4 * i));
89     }
90
91     /* load state */
92     aa = md->rmd256.state[0];
93     bb = md->rmd256.state[1];
94     cc = md->rmd256.state[2];
95     dd = md->rmd256.state[3];
96     aaa = md->rmd256.state[4];
```

```
97 bbb = md->rmd256.state[5];
98 ccc = md->rmd256.state[6];
99 ddd = md->rmd256.state[7];
100
101 /* round 1 */
102 FF(aa, bb, cc, dd, X[ 0], 11);
103 FF(dd, aa, bb, cc, X[ 1], 14);
104 FF(cc, dd, aa, bb, X[ 2], 15);
105 FF(bb, cc, dd, aa, X[ 3], 12);
106 FF(aa, bb, cc, dd, X[ 4], 5);
107 FF(dd, aa, bb, cc, X[ 5], 8);
108 FF(cc, dd, aa, bb, X[ 6], 7);
109 FF(bb, cc, dd, aa, X[ 7], 9);
110 FF(aa, bb, cc, dd, X[ 8], 11);
111 FF(dd, aa, bb, cc, X[ 9], 13);
112 FF(cc, dd, aa, bb, X[10], 14);
113 FF(bb, cc, dd, aa, X[11], 15);
114 FF(aa, bb, cc, dd, X[12], 6);
115 FF(dd, aa, bb, cc, X[13], 7);
116 FF(cc, dd, aa, bb, X[14], 9);
117 FF(bb, cc, dd, aa, X[15], 8);
118
119 /* parallel round 1 */
120 III(aaa, bbb, ccc, ddd, X[ 5], 8);
121 III(ddd, aaa, bbb, ccc, X[14], 9);
122 III(ccc, ddd, aaa, bbb, X[ 7], 9);
123 III(bbb, ccc, ddd, aaa, X[ 0], 11);
124 III(aaa, bbb, ccc, ddd, X[ 9], 13);
125 III(ddd, aaa, bbb, ccc, X[ 2], 15);
126 III(ccc, ddd, aaa, bbb, X[11], 15);
127 III(bbb, ccc, ddd, aaa, X[ 4], 5);
128 III(aaa, bbb, ccc, ddd, X[13], 7);
129 III(ddd, aaa, bbb, ccc, X[ 6], 7);
130 III(ccc, ddd, aaa, bbb, X[15], 8);
131 III(bbb, ccc, ddd, aaa, X[ 8], 11);
132 III(aaa, bbb, ccc, ddd, X[ 1], 14);
133 III(ddd, aaa, bbb, ccc, X[10], 14);
134 III(ccc, ddd, aaa, bbb, X[ 3], 12);
135 III(bbb, ccc, ddd, aaa, X[12], 6);
136
137 tmp = aa; aa = aaa; aaa = tmp;
138
139 /* round 2 */
140 GG(aa, bb, cc, dd, X[ 7], 7);
141 GG(dd, aa, bb, cc, X[ 4], 6);
142 GG(cc, dd, aa, bb, X[13], 8);
143 GG(bb, cc, dd, aa, X[ 1], 13);
144 GG(aa, bb, cc, dd, X[10], 11);
145 GG(dd, aa, bb, cc, X[ 6], 9);
146 GG(cc, dd, aa, bb, X[15], 7);
147 GG(bb, cc, dd, aa, X[ 3], 15);
148 GG(aa, bb, cc, dd, X[12], 7);
149 GG(dd, aa, bb, cc, X[ 0], 12);
150 GG(cc, dd, aa, bb, X[ 9], 15);
151 GG(bb, cc, dd, aa, X[ 5], 9);
152 GG(aa, bb, cc, dd, X[ 2], 11);
153 GG(dd, aa, bb, cc, X[14], 7);
154 GG(cc, dd, aa, bb, X[11], 13);
155 GG(bb, cc, dd, aa, X[ 8], 12);
156
157 /* parallel round 2 */
158 HHH(aaa, bbb, ccc, ddd, X[ 6], 9);
159 HHH(ddd, aaa, bbb, ccc, X[11], 13);
160 HHH(ccc, ddd, aaa, bbb, X[ 3], 15);
161 HHH(bbb, ccc, ddd, aaa, X[ 7], 7);
162 HHH(aaa, bbb, ccc, ddd, X[ 0], 12);
163 HHH(ddd, aaa, bbb, ccc, X[13], 8);
```

```
164   HHH(ccc, ddd, aaa, bbb, X[ 5], 9);
165   HHH(bbb, ccc, ddd, aaa, X[10], 11);
166   HHH(aaa, bbb, ccc, ddd, X[14], 7);
167   HHH(ddd, aaa, bbb, ccc, X[15], 7);
168   HHH(ccc, ddd, aaa, bbb, X[ 8], 12);
169   HHH(bbb, ccc, ddd, aaa, X[12], 7);
170   HHH(aaa, bbb, ccc, ddd, X[ 4], 6);
171   HHH(ddd, aaa, bbb, ccc, X[ 9], 15);
172   HHH(ccc, ddd, aaa, bbb, X[ 1], 13);
173   HHH(bbb, ccc, ddd, aaa, X[ 2], 11);
174
175   tmp = bb; bb = bbb; bbb = tmp;
176
177   /* round 3 */
178   HH(aa, bb, cc, dd, X[ 3], 11);
179   HH(dd, aa, bb, cc, X[10], 13);
180   HH(cc, dd, aa, bb, X[14], 6);
181   HH(bb, cc, dd, aa, X[ 4], 7);
182   HH(aa, bb, cc, dd, X[ 9], 14);
183   HH(dd, aa, bb, cc, X[15], 9);
184   HH(cc, dd, aa, bb, X[ 8], 13);
185   HH(bb, cc, dd, aa, X[ 1], 15);
186   HH(aa, bb, cc, dd, X[ 2], 14);
187   HH(dd, aa, bb, cc, X[ 7], 8);
188   HH(cc, dd, aa, bb, X[ 0], 13);
189   HH(bb, cc, dd, aa, X[ 6], 6);
190   HH(aa, bb, cc, dd, X[13], 5);
191   HH(dd, aa, bb, cc, X[11], 12);
192   HH(cc, dd, aa, bb, X[ 5], 7);
193   HH(bb, cc, dd, aa, X[12], 5);
194
195   /* parallel round 3 */
196   GGG(aaa, bbb, ccc, ddd, X[15], 9);
197   GGG(ddd, aaa, bbb, ccc, X[ 5], 7);
198   GGG(ccc, ddd, aaa, bbb, X[ 1], 15);
199   GGG(bbb, ccc, ddd, aaa, X[ 3], 11);
200   GGG(aaa, bbb, ccc, ddd, X[ 7], 8);
201   GGG(ddd, aaa, bbb, ccc, X[14], 6);
202   GGG(ccc, ddd, aaa, bbb, X[ 6], 6);
203   GGG(bbb, ccc, ddd, aaa, X[ 9], 14);
204   GGG(aaa, bbb, ccc, ddd, X[11], 12);
205   GGG(ddd, aaa, bbb, ccc, X[ 8], 13);
206   GGG(ccc, ddd, aaa, bbb, X[12], 5);
207   GGG(bbb, ccc, ddd, aaa, X[ 2], 14);
208   GGG(aaa, bbb, ccc, ddd, X[10], 13);
209   GGG(ddd, aaa, bbb, ccc, X[ 0], 13);
210   GGG(ccc, ddd, aaa, bbb, X[ 4], 7);
211   GGG(bbb, ccc, ddd, aaa, X[13], 5);
212
213   tmp = cc; cc = ccc; ccc = tmp;
214
215   /* round 4 */
216   II(aa, bb, cc, dd, X[ 1], 11);
217   II(dd, aa, bb, cc, X[ 9], 12);
218   II(cc, dd, aa, bb, X[11], 14);
219   II(bb, cc, dd, aa, X[10], 15);
220   II(aa, bb, cc, dd, X[ 0], 14);
221   II(dd, aa, bb, cc, X[ 8], 15);
222   II(cc, dd, aa, bb, X[12], 9);
223   II(bb, cc, dd, aa, X[ 4], 8);
224   II(aa, bb, cc, dd, X[13], 9);
225   II(dd, aa, bb, cc, X[ 3], 14);
226   II(cc, dd, aa, bb, X[ 7], 5);
227   II(bb, cc, dd, aa, X[15], 6);
228   II(aa, bb, cc, dd, X[14], 8);
229   II(dd, aa, bb, cc, X[ 5], 6);
230   II(cc, dd, aa, bb, X[ 6], 5);
```

```

231     II(bb, cc, dd, aa, X[ 2], 12);
232
233     /* parallel round 4 */
234     FFF(aaa, bbb, ccc, ddd, X[ 8], 15);
235     FFF(ddd, aaa, bbb, ccc, X[ 6],  5);
236     FFF(ccc, ddd, aaa, bbb, X[ 4],  8);
237     FFF(bbb, ccc, ddd, aaa, X[ 1], 11);
238     FFF(aaa, bbb, ccc, ddd, X[ 3], 14);
239     FFF(ddd, aaa, bbb, ccc, X[11], 14);
240     FFF(ccc, ddd, aaa, bbb, X[15],  6);
241     FFF(bbb, ccc, ddd, aaa, X[ 0], 14);
242     FFF(aaa, bbb, ccc, ddd, X[ 5],  6);
243     FFF(ddd, aaa, bbb, ccc, X[12],  9);
244     FFF(ccc, ddd, aaa, bbb, X[ 2], 12);
245     FFF(bbb, ccc, ddd, aaa, X[13],  9);
246     FFF(aaa, bbb, ccc, ddd, X[ 9], 12);
247     FFF(ddd, aaa, bbb, ccc, X[ 7],  5);
248     FFF(ccc, ddd, aaa, bbb, X[10], 15);
249     FFF(bbb, ccc, ddd, aaa, X[14],  8);
250
251     tmp = dd; dd = ddd; ddd = tmp;
252
253     /* combine results */
254     md->rmd256.state[0] += aa;
255     md->rmd256.state[1] += bb;
256     md->rmd256.state[2] += cc;
257     md->rmd256.state[3] += dd;
258     md->rmd256.state[4] += aaa;
259     md->rmd256.state[5] += bbb;
260     md->rmd256.state[6] += ccc;
261     md->rmd256.state[7] += ddd;
262
263     return CRYPT_OK;
264 }

```

### 5.62.2.2 int rmd256\_done (hash\_state \* md, unsigned char \* out)

Terminate the hash to get the digest.

#### Parameters:

- md* The hash state
- out* [out] The destination of the hash (16 bytes)

#### Returns:

- CRYPT\_OK if successful

Definition at line 312 of file rmd256.c.

References CRYPT\_INVALID\_ARG, LTC\_ARGCHK, and rmd256\_compress().

Referenced by rmd256\_test().

```

313 {
314     int i;
315
316     LTC_ARGCHK(md != NULL);
317     LTC_ARGCHK(out != NULL);
318
319     if (md->rmd256.curlen >= sizeof(md->rmd256.buf)) {
320         return CRYPT_INVALID_ARG;
321     }

```

```

322
323
324     /* increase the length of the message */
325     md->rm256.length += md->rm256.curlen * 8;
326
327     /* append the '1' bit */
328     md->rm256.buf[md->rm256.curlen++] = (unsigned char)0x80;
329
330     /* if the length is currently above 56 bytes we append zeros
331      * then compress.  Then we can fall back to padding zeros and length
332      * encoding like normal.
333      */
334     if (md->rm256.curlen > 56) {
335         while (md->rm256.curlen < 64) {
336             md->rm256.buf[md->rm256.curlen++] = (unsigned char)0;
337         }
338         rmd256_compress(md, md->rm256.buf);
339         md->rm256.curlen = 0;
340     }
341
342     /* pad upto 56 bytes of zeroes */
343     while (md->rm256.curlen < 56) {
344         md->rm256.buf[md->rm256.curlen++] = (unsigned char)0;
345     }
346
347     /* store length */
348     STORE64L(md->rm256.length, md->rm256.buf+56);
349     rmd256_compress(md, md->rm256.buf);
350
351     /* copy output */
352     for (i = 0; i < 8; i++) {
353         STORE32L(md->rm256.state[i], out+(4*i));
354     }
355     #ifdef LTC_CLEAN_STACK
356         zeromem(md, sizeof(hash_state));
357     #endif
358     return CRYPT_OK;
359 }

```

Here is the call graph for this function:

### 5.62.2.3 int rmd256\_init (hash\_state \* md)

Initialize the hash state.

#### Parameters:

*md* The hash state you wish to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 281 of file rmd256.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by rmd256\_test().

```

282 {
283     LTC_ARGCHK(md != NULL);
284     md->rm256.state[0] = 0x67452301UL;
285     md->rm256.state[1] = 0xefcdab89UL;
286     md->rm256.state[2] = 0x98badcfeUL;

```

```

287     md->rmd256.state[3] = 0x10325476UL;
288     md->rmd256.state[4] = 0x76543210UL;
289     md->rmd256.state[5] = 0xfedcba98UL;
290     md->rmd256.state[6] = 0x89abcdefUL;
291     md->rmd256.state[7] = 0x01234567UL;
292     md->rmd256.curlen   = 0;
293     md->rmd256.length   = 0;
294     return CRYPT_OK;
295 }

```

#### 5.62.2.4 int rmd256\_test (void)

Self-test the hash.

##### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 365 of file rmd256.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, rmd256\_done(), rmd256\_init(), and XMEMP\_CMP.

```

366 {
367     #ifndef LTC_TEST
368         return CRYPT_NOP;
369     #else
370         static const struct {
371             char *msg;
372             unsigned char md[32];
373         } tests[] = {
374             { "",
375               { 0x02, 0xba, 0x4c, 0x4e, 0x5f, 0x8e, 0xcd, 0x18,
376                 0x77, 0xfc, 0x52, 0xd6, 0x4d, 0x30, 0xe3, 0x7a,
377                 0x2d, 0x97, 0x74, 0xfb, 0x1e, 0x5d, 0x02, 0x63,
378                 0x80, 0xae, 0x01, 0x68, 0xe3, 0xc5, 0x52, 0x2d }
379             },
380             { "a",
381               { 0xf9, 0x33, 0x3e, 0x45, 0xd8, 0x57, 0xf5, 0xd9,
382                 0x0a, 0x91, 0xba, 0xb7, 0x0a, 0x1e, 0xba, 0x0c,
383                 0xfb, 0x1b, 0xe4, 0xb0, 0x78, 0x3c, 0x9a, 0xcf,
384                 0xcd, 0x88, 0x3a, 0x91, 0x34, 0x69, 0x29, 0x25 }
385             },
386             { "abc",
387               { 0xaf, 0xbd, 0x6e, 0x22, 0x8b, 0x9d, 0x8c, 0xbb,
388                 0xce, 0xf5, 0xca, 0x2d, 0x03, 0xe6, 0xdb, 0xa1,
389                 0x0a, 0xc0, 0xbc, 0x7d, 0xcb, 0xe4, 0x68, 0x0e,
390                 0x1e, 0x42, 0xd2, 0xe9, 0x75, 0x45, 0x9b, 0x65 }
391             },
392             { "message digest",
393               { 0x87, 0xe9, 0x71, 0x75, 0x9a, 0x1c, 0xe4, 0x7a,
394                 0x51, 0x4d, 0x5c, 0x91, 0x4c, 0x39, 0x2c, 0x90,
395                 0x18, 0xc7, 0xc4, 0x6b, 0xc1, 0x44, 0x65, 0x55,
396                 0x4a, 0xfc, 0xdf, 0x54, 0xa5, 0x07, 0x0c, 0x0e }
397             },
398             { "abcdefghijklmnopqrstuvwxyz",
399               { 0x64, 0x9d, 0x30, 0x34, 0x75, 0x1e, 0xa2, 0x16,
400                 0x77, 0x6b, 0xf9, 0xa1, 0x8a, 0xcc, 0x81, 0xbc,
401                 0x78, 0x96, 0x11, 0x8a, 0x51, 0x97, 0x96, 0x87,
402                 0x82, 0xdd, 0x1f, 0xd9, 0x7d, 0x8d, 0x51, 0x33 }
403             },
404             { "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789",
405               { 0x57, 0x40, 0xa4, 0x08, 0xac, 0x16, 0xb7, 0x20,
406                 0xb8, 0x44, 0x24, 0xae, 0x93, 0x1c, 0xbb, 0x1f,

```

```

407         0xe3, 0x63, 0xd1, 0xd0, 0xbf, 0x40, 0x17, 0xf1,
408         0xa8, 0x9f, 0x7e, 0xa6, 0xde, 0x77, 0xa0, 0xb8 }
409     };
410 };
411 int x;
412 unsigned char buf[32];
413 hash_state md;
414
415 for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
416     rmd256_init(&md);
417     rmd256_process(&md, (unsigned char *)tests[x].msg, strlen(tests[x].msg));
418     rmd256_done(&md, buf);
419     if (XMEMCMP(buf, tests[x].md, 32) != 0) {
420         #if 0
421             printf("Failed test %d\n", x);
422         #endif
423         return CRYPT_FAIL_TESTVECTOR;
424     }
425 }
426 return CRYPT_OK;
427 #endif
428 }

```

Here is the call graph for this function:

### 5.62.3 Variable Documentation

#### 5.62.3.1 `const struct ltc_hash_descriptor rmd256_desc`

**Initial value:**

```

{
    "rmd256",
    8,
    16,
    64,

    { 1, 3, 36, 3, 2, 3 },
    6,

    &rmd256_init,
    &rmd256_process,
    &rmd256_done,
    &rmd256_test,
    NULL
}

```

**Parameters:**

[\*rmd256.c\*](#) RMD256 Hash function

Definition at line 20 of file rmd256.c.

Referenced by yarrow\_start().



## 5.63 hashes/rmd320.c File Reference

### 5.63.1 Detailed Description

RMD320 hash function.

Definition in file [rmd320.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for rmd320.c:

### Defines

- `#define F(x, y, z) ((x) ^ (y) ^ (z))`
- `#define G(x, y, z) (((x) & (y)) | (~ (x) & (z)))`
- `#define H(x, y, z) (((x) | ~ (y)) ^ (z))`
- `#define I(x, y, z) (((x) & (z)) | ((y) & ~ (z)))`
- `#define J(x, y, z) ((x) ^ ((y) | ~ (z)))`
- `#define FF(a, b, c, d, e, x, s)`
- `#define GG(a, b, c, d, e, x, s)`
- `#define HH(a, b, c, d, e, x, s)`
- `#define II(a, b, c, d, e, x, s)`
- `#define JJ(a, b, c, d, e, x, s)`
- `#define FFF(a, b, c, d, e, x, s)`
- `#define GGG(a, b, c, d, e, x, s)`
- `#define HHH(a, b, c, d, e, x, s)`
- `#define III(a, b, c, d, e, x, s)`
- `#define JJJ(a, b, c, d, e, x, s)`

### Functions

- static int [rmd320\\_compress](#) ([hash\\_state](#) \*md, unsigned char \*buf)
- int [rmd320\\_init](#) ([hash\\_state](#) \*md)  
*Initialize the hash state.*
- int [rmd320\\_done](#) ([hash\\_state](#) \*md, unsigned char \*out)  
*Terminate the hash to get the digest.*
- int [rmd320\\_test](#) (void)  
*Self-test the hash.*

### Variables

- const struct [ltc\\_hash\\_descriptor](#) [rmd320\\_desc](#)

## 5.63.2 Define Documentation

### 5.63.2.1 #define F(x, y, z) ((x) ^ (y) ^ (z))

Definition at line 39 of file rmd320.c.

### 5.63.2.2 #define FF(a, b, c, d, e, x, s)

**Value:**

```
(a) += F((b), (c), (d)) + (x); \
(a) = ROTLc((a), (s)) + (e); \
(c) = ROTLc((c), 10);
```

Definition at line 46 of file rmd320.c.

### 5.63.2.3 #define FFF(a, b, c, d, e, x, s)

**Value:**

```
(a) += F((b), (c), (d)) + (x); \
(a) = ROTLc((a), (s)) + (e); \
(c) = ROTLc((c), 10);
```

Definition at line 71 of file rmd320.c.

### 5.63.2.4 #define G(x, y, z) (((x) & (y)) | (~ (x) & (z)))

Definition at line 40 of file rmd320.c.

### 5.63.2.5 #define GG(a, b, c, d, e, x, s)

**Value:**

```
(a) += G((b), (c), (d)) + (x) + 0x5a827999UL; \
(a) = ROTLc((a), (s)) + (e); \
(c) = ROTLc((c), 10);
```

Definition at line 51 of file rmd320.c.

### 5.63.2.6 #define GGG(a, b, c, d, e, x, s)

**Value:**

```
(a) += G((b), (c), (d)) + (x) + 0x7a6d76e9UL; \
(a) = ROTLc((a), (s)) + (e); \
(c) = ROTLc((c), 10);
```

Definition at line 76 of file rmd320.c.

**5.63.2.7 #define H(x, y, z) (((x) | ~(y)) ^ (z))**

Definition at line 41 of file rmd320.c.

**5.63.2.8 #define HH(a, b, c, d, e, x, s)**

**Value:**

```
(a) += H((b), (c), (d)) + (x) + 0x6ed9eba1UL;\n      (a) = ROTL((a), (s)) + (e);\n      (c) = ROTL((c), 10);
```

Definition at line 56 of file rmd320.c.

**5.63.2.9 #define HHH(a, b, c, d, e, x, s)**

**Value:**

```
(a) += H((b), (c), (d)) + (x) + 0x6d703ef3UL;\n      (a) = ROTL((a), (s)) + (e);\n      (c) = ROTL((c), 10);
```

Definition at line 81 of file rmd320.c.

**5.63.2.10 #define I(x, y, z) (((x) & (z)) | ((y) & ~(z)))**

Definition at line 42 of file rmd320.c.

**5.63.2.11 #define II(a, b, c, d, e, x, s)**

**Value:**

```
(a) += I((b), (c), (d)) + (x) + 0x8f1bbcdcUL;\n      (a) = ROTL((a), (s)) + (e);\n      (c) = ROTL((c), 10);
```

Definition at line 61 of file rmd320.c.

**5.63.2.12 #define III(a, b, c, d, e, x, s)**

**Value:**

```
(a) += I((b), (c), (d)) + (x) + 0x5c4dd124UL;\n      (a) = ROTL((a), (s)) + (e);\n      (c) = ROTL((c), 10);
```

Definition at line 86 of file rmd320.c.

**5.63.2.13 #define J(x, y, z) ((x) ^ ((y) | ~(z)))**

Definition at line 43 of file rmd320.c.

**5.63.2.14 #define JJ(a, b, c, d, e, x, s)****Value:**

```
(a) += J((b), (c), (d)) + (x) + 0xa953fd4eUL;\
(a) = ROLc((a), (s)) + (e);\
(c) = ROLc((c), 10);
```

Definition at line 66 of file rmd320.c.

**5.63.2.15 #define JJJ(a, b, c, d, e, x, s)****Value:**

```
(a) += J((b), (c), (d)) + (x) + 0x50a28be6UL;\
(a) = ROLc((a), (s)) + (e);\
(c) = ROLc((c), 10);
```

Definition at line 91 of file rmd320.c.

**5.63.3 Function Documentation****5.63.3.1 static int rmd320\_compress (hash\_state \*md, unsigned char \*buf) [static]**

Definition at line 100 of file rmd320.c.

Referenced by rmd320\_done().

```
102 {
103     ulong32 aa,bb,cc,dd,ee,aaa,bbb,ccc,ddd,eee,tmp,X[16];
104     int i;
105
106     /* load words X */
107     for (i = 0; i < 16; i++){
108         LOAD32L(X[i], buf + (4 * i));
109     }
110
111     /* load state */
112     aa = md->rmd320.state[0];
113     bb = md->rmd320.state[1];
114     cc = md->rmd320.state[2];
115     dd = md->rmd320.state[3];
116     ee = md->rmd320.state[4];
117     aaa = md->rmd320.state[5];
118     bbb = md->rmd320.state[6];
119     ccc = md->rmd320.state[7];
120     ddd = md->rmd320.state[8];
121     eee = md->rmd320.state[9];
122
123     /* round 1 */
124     FF(aa, bb, cc, dd, ee, X[ 0], 11);
125     FF(ee, aa, bb, cc, dd, X[ 1], 14);
126     FF(dd, ee, aa, bb, cc, X[ 2], 15);
127     FF(cc, dd, ee, aa, bb, X[ 3], 12);
128     FF(bb, cc, dd, ee, aa, X[ 4],  5);
129     FF(aa, bb, cc, dd, ee, X[ 5],  8);
130     FF(ee, aa, bb, cc, dd, X[ 6],  7);
131     FF(dd, ee, aa, bb, cc, X[ 7],  9);
132     FF(cc, dd, ee, aa, bb, X[ 8], 11);
```

```
133     FF(bb, cc, dd, ee, aa, X[ 9], 13);
134     FF(aa, bb, cc, dd, ee, X[10], 14);
135     FF(ee, aa, bb, cc, dd, X[11], 15);
136     FF(dd, ee, aa, bb, cc, X[12], 6);
137     FF(cc, dd, ee, aa, bb, X[13], 7);
138     FF(bb, cc, dd, ee, aa, X[14], 9);
139     FF(aa, bb, cc, dd, ee, X[15], 8);
140
141     /* parallel round 1 */
142     JJJ(aaa, bbb, ccc, ddd, eee, X[ 5], 8);
143     JJJ(eee, aaa, bbb, ccc, ddd, X[14], 9);
144     JJJ(ddd, eee, aaa, bbb, ccc, X[ 7], 9);
145     JJJ(ccc, ddd, eee, aaa, bbb, X[ 0], 11);
146     JJJ(bbb, ccc, ddd, eee, aaa, X[ 9], 13);
147     JJJ(aaa, bbb, ccc, ddd, eee, X[ 2], 15);
148     JJJ(eee, aaa, bbb, ccc, ddd, X[11], 15);
149     JJJ(ddd, eee, aaa, bbb, ccc, X[ 4], 5);
150     JJJ(ccc, ddd, eee, aaa, bbb, X[13], 7);
151     JJJ(bbb, ccc, ddd, eee, aaa, X[ 6], 7);
152     JJJ(aaa, bbb, ccc, ddd, eee, X[15], 8);
153     JJJ(eee, aaa, bbb, ccc, ddd, X[ 8], 11);
154     JJJ(ddd, eee, aaa, bbb, ccc, X[ 1], 14);
155     JJJ(ccc, ddd, eee, aaa, bbb, X[10], 14);
156     JJJ(bbb, ccc, ddd, eee, aaa, X[ 3], 12);
157     JJJ(aaa, bbb, ccc, ddd, eee, X[12], 6);
158
159     tmp = aa; aa = aaa; aaa = tmp;
160
161     /* round 2 */
162     GG(ee, aa, bb, cc, dd, X[ 7], 7);
163     GG(dd, ee, aa, bb, cc, X[ 4], 6);
164     GG(cc, dd, ee, aa, bb, X[13], 8);
165     GG(bb, cc, dd, ee, aa, X[ 1], 13);
166     GG(aa, bb, cc, dd, ee, X[10], 11);
167     GG(ee, aa, bb, cc, dd, X[ 6], 9);
168     GG(dd, ee, aa, bb, cc, X[15], 7);
169     GG(cc, dd, ee, aa, bb, X[ 3], 15);
170     GG(bb, cc, dd, ee, aa, X[12], 7);
171     GG(aa, bb, cc, dd, ee, X[ 0], 12);
172     GG(ee, aa, bb, cc, dd, X[ 9], 15);
173     GG(dd, ee, aa, bb, cc, X[ 5], 9);
174     GG(cc, dd, ee, aa, bb, X[ 2], 11);
175     GG(bb, cc, dd, ee, aa, X[14], 7);
176     GG(aa, bb, cc, dd, ee, X[11], 13);
177     GG(ee, aa, bb, cc, dd, X[ 8], 12);
178
179     /* parallel round 2 */
180     III(eee, aaa, bbb, ccc, ddd, X[ 6], 9);
181     III(ddd, eee, aaa, bbb, ccc, X[11], 13);
182     III(ccc, ddd, eee, aaa, bbb, X[ 3], 15);
183     III(bbb, ccc, ddd, eee, aaa, X[ 7], 7);
184     III(aaa, bbb, ccc, ddd, eee, X[ 0], 12);
185     III(eee, aaa, bbb, ccc, ddd, X[13], 8);
186     III(ddd, eee, aaa, bbb, ccc, X[ 5], 9);
187     III(ccc, ddd, eee, aaa, bbb, X[10], 11);
188     III(bbb, ccc, ddd, eee, aaa, X[14], 7);
189     III(aaa, bbb, ccc, ddd, eee, X[15], 7);
190     III(eee, aaa, bbb, ccc, ddd, X[ 8], 12);
191     III(ddd, eee, aaa, bbb, ccc, X[12], 7);
192     III(ccc, ddd, eee, aaa, bbb, X[ 4], 6);
193     III(bbb, ccc, ddd, eee, aaa, X[ 9], 15);
194     III(aaa, bbb, ccc, ddd, eee, X[ 1], 13);
195     III(eee, aaa, bbb, ccc, ddd, X[ 2], 11);
196
197     tmp = bb; bb = bbb; bbb = tmp;
198
199     /* round 3 */
```

```
200 HH(dd, ee, aa, bb, cc, X[ 3], 11);
201 HH(cc, dd, ee, aa, bb, X[10], 13);
202 HH(bb, cc, dd, ee, aa, X[14], 6);
203 HH(aa, bb, cc, dd, ee, X[ 4], 7);
204 HH(ee, aa, bb, cc, dd, X[ 9], 14);
205 HH(dd, ee, aa, bb, cc, X[15], 9);
206 HH(cc, dd, ee, aa, bb, X[ 8], 13);
207 HH(bb, cc, dd, ee, aa, X[ 1], 15);
208 HH(aa, bb, cc, dd, ee, X[ 2], 14);
209 HH(ee, aa, bb, cc, dd, X[ 7], 8);
210 HH(dd, ee, aa, bb, cc, X[ 0], 13);
211 HH(cc, dd, ee, aa, bb, X[ 6], 6);
212 HH(bb, cc, dd, ee, aa, X[13], 5);
213 HH(aa, bb, cc, dd, ee, X[11], 12);
214 HH(ee, aa, bb, cc, dd, X[ 5], 7);
215 HH(dd, ee, aa, bb, cc, X[12], 5);
216
217 /* parallel round 3 */
218 HHH(ddd, eee, aaa, bbb, ccc, X[15], 9);
219 HHH(ccc, ddd, eee, aaa, bbb, X[ 5], 7);
220 HHH(bbb, ccc, ddd, eee, aaa, X[ 1], 15);
221 HHH(aaa, bbb, ccc, ddd, eee, X[ 3], 11);
222 HHH(eee, aaa, bbb, ccc, ddd, X[ 7], 8);
223 HHH(ddd, eee, aaa, bbb, ccc, X[14], 6);
224 HHH(ccc, ddd, eee, aaa, bbb, X[ 6], 6);
225 HHH(bbb, ccc, ddd, eee, aaa, X[ 9], 14);
226 HHH(aaa, bbb, ccc, ddd, eee, X[11], 12);
227 HHH(eee, aaa, bbb, ccc, ddd, X[ 8], 13);
228 HHH(ddd, eee, aaa, bbb, ccc, X[12], 5);
229 HHH(ccc, ddd, eee, aaa, bbb, X[ 2], 14);
230 HHH(bbb, ccc, ddd, eee, aaa, X[10], 13);
231 HHH(aaa, bbb, ccc, ddd, eee, X[ 0], 13);
232 HHH(eee, aaa, bbb, ccc, ddd, X[ 4], 7);
233 HHH(ddd, eee, aaa, bbb, ccc, X[13], 5);
234
235 tmp = cc; cc = ccc; ccc = tmp;
236
237 /* round 4 */
238 II(cc, dd, ee, aa, bb, X[ 1], 11);
239 II(bb, cc, dd, ee, aa, X[ 9], 12);
240 II(aa, bb, cc, dd, ee, X[11], 14);
241 II(ee, aa, bb, cc, dd, X[10], 15);
242 II(dd, ee, aa, bb, cc, X[ 0], 14);
243 II(cc, dd, ee, aa, bb, X[ 8], 15);
244 II(bb, cc, dd, ee, aa, X[12], 9);
245 II(aa, bb, cc, dd, ee, X[ 4], 8);
246 II(ee, aa, bb, cc, dd, X[13], 9);
247 II(dd, ee, aa, bb, cc, X[ 3], 14);
248 II(cc, dd, ee, aa, bb, X[ 7], 5);
249 II(bb, cc, dd, ee, aa, X[15], 6);
250 II(aa, bb, cc, dd, ee, X[14], 8);
251 II(ee, aa, bb, cc, dd, X[ 5], 6);
252 II(dd, ee, aa, bb, cc, X[ 6], 5);
253 II(cc, dd, ee, aa, bb, X[ 2], 12);
254
255 /* parallel round 4 */
256 GGG(ccc, ddd, eee, aaa, bbb, X[ 8], 15);
257 GGG(bbb, ccc, ddd, eee, aaa, X[ 6], 5);
258 GGG(aaa, bbb, ccc, ddd, eee, X[ 4], 8);
259 GGG(eee, aaa, bbb, ccc, ddd, X[ 1], 11);
260 GGG(ddd, eee, aaa, bbb, ccc, X[ 3], 14);
261 GGG(ccc, ddd, eee, aaa, bbb, X[11], 14);
262 GGG(bbb, ccc, ddd, eee, aaa, X[15], 6);
263 GGG(aaa, bbb, ccc, ddd, eee, X[ 0], 14);
264 GGG(eee, aaa, bbb, ccc, ddd, X[ 5], 6);
265 GGG(ddd, eee, aaa, bbb, ccc, X[12], 9);
266 GGG(ccc, ddd, eee, aaa, bbb, X[ 2], 12);
```

```

267     GGG(bbb, ccc, ddd, eee, aaa, X[13], 9);
268     GGG(aaa, bbb, ccc, ddd, eee, X[ 9], 12);
269     GGG(eee, aaa, bbb, ccc, ddd, X[ 7], 5);
270     GGG(ddd, eee, aaa, bbb, ccc, X[10], 15);
271     GGG(ccc, ddd, eee, aaa, bbb, X[14], 8);
272
273     tmp = dd; dd = ddd; ddd = tmp;
274
275     /* round 5 */
276     JJ(bb, cc, dd, ee, aa, X[ 4], 9);
277     JJ(aa, bb, cc, dd, ee, X[ 0], 15);
278     JJ(ee, aa, bb, cc, dd, X[ 5], 5);
279     JJ(dd, ee, aa, bb, cc, X[ 9], 11);
280     JJ(cc, dd, ee, aa, bb, X[ 7], 6);
281     JJ(bb, cc, dd, ee, aa, X[12], 8);
282     JJ(aa, bb, cc, dd, ee, X[ 2], 13);
283     JJ(ee, aa, bb, cc, dd, X[10], 12);
284     JJ(dd, ee, aa, bb, cc, X[14], 5);
285     JJ(cc, dd, ee, aa, bb, X[ 1], 12);
286     JJ(bb, cc, dd, ee, aa, X[ 3], 13);
287     JJ(aa, bb, cc, dd, ee, X[ 8], 14);
288     JJ(ee, aa, bb, cc, dd, X[11], 11);
289     JJ(dd, ee, aa, bb, cc, X[ 6], 8);
290     JJ(cc, dd, ee, aa, bb, X[15], 5);
291     JJ(bb, cc, dd, ee, aa, X[13], 6);
292
293     /* parallel round 5 */
294     FFF(bbb, ccc, ddd, eee, aaa, X[12], 8);
295     FFF(aaa, bbb, ccc, ddd, eee, X[15], 5);
296     FFF(eee, aaa, bbb, ccc, ddd, X[10], 12);
297     FFF(ddd, eee, aaa, bbb, ccc, X[ 4], 9);
298     FFF(ccc, ddd, eee, aaa, bbb, X[ 1], 12);
299     FFF(bbb, ccc, ddd, eee, aaa, X[ 5], 5);
300     FFF(aaa, bbb, ccc, ddd, eee, X[ 8], 14);
301     FFF(eee, aaa, bbb, ccc, ddd, X[ 7], 6);
302     FFF(ddd, eee, aaa, bbb, ccc, X[ 6], 8);
303     FFF(ccc, ddd, eee, aaa, bbb, X[ 2], 13);
304     FFF(bbb, ccc, ddd, eee, aaa, X[13], 6);
305     FFF(aaa, bbb, ccc, ddd, eee, X[14], 5);
306     FFF(eee, aaa, bbb, ccc, ddd, X[ 0], 15);
307     FFF(ddd, eee, aaa, bbb, ccc, X[ 3], 13);
308     FFF(ccc, ddd, eee, aaa, bbb, X[ 9], 11);
309     FFF(bbb, ccc, ddd, eee, aaa, X[11], 11);
310
311     tmp = ee; ee = eee; eee = tmp;
312
313     /* combine results */
314     md->rmd320.state[0] += aa;
315     md->rmd320.state[1] += bb;
316     md->rmd320.state[2] += cc;
317     md->rmd320.state[3] += dd;
318     md->rmd320.state[4] += ee;
319     md->rmd320.state[5] += aaa;
320     md->rmd320.state[6] += bbb;
321     md->rmd320.state[7] += ccc;
322     md->rmd320.state[8] += ddd;
323     md->rmd320.state[9] += eee;
324
325     return CRYPT_OK;
326 }

```

### 5.63.3.2 int rmd320\_done (hash\_state \* md, unsigned char \* out)

Terminate the hash to get the digest.

**Parameters:**

*md* The hash state

*out* [out] The destination of the hash (20 bytes)

**Returns:**

CRYPT\_OK if successful

Definition at line 376 of file rmd320.c.

References CRYPT\_INVALID\_ARG, LTC\_ARGCHK, and rmd320\_compress().

Referenced by rmd320\_test().

```
377 {
378     int i;
379
380     LTC_ARGCHK(md != NULL);
381     LTC_ARGCHK(out != NULL);
382
383     if (md->rmd320	curlen >= sizeof(md->rmd320.buf)) {
384         return CRYPT_INVALID_ARG;
385     }
386
387     /* increase the length of the message */
388     md->rmd320.length += md->rmd320	curlen * 8;
389
390     /* append the '1' bit */
391     md->rmd320.buf[md->rmd320	curlen++] = (unsigned char)0x80;
392
393     /* if the length is currently above 56 bytes we append zeros
394      * then compress.  Then we can fall back to padding zeros and length
395      * encoding like normal.
396      */
397     if (md->rmd320	curlen > 56) {
398         while (md->rmd320	curlen < 64) {
399             md->rmd320.buf[md->rmd320	curlen++] = (unsigned char)0;
400         }
401         rmd320_compress(md, md->rmd320.buf);
402         md->rmd320	curlen = 0;
403     }
404
405     /* pad upto 56 bytes of zeroes */
406     while (md->rmd320	curlen < 56) {
407         md->rmd320.buf[md->rmd320	curlen++] = (unsigned char)0;
408     }
409
410     /* store length */
411     STORE64L(md->rmd320.length, md->rmd320.buf+56);
412     rmd320_compress(md, md->rmd320.buf);
413
414     /* copy output */
415     for (i = 0; i < 10; i++) {
416         STORE32L(md->rmd320.state[i], out+(4*i));
417     }
418
419     #ifdef LTC_CLEAN_STACK
420     zeromem(md, sizeof(hash_state));
421     #endif
422     return CRYPT_OK;
423 }
```

Here is the call graph for this function:



**5.63.3.3 int rmd320\_init (hash\_state \* md)**

Initialize the hash state.

**Parameters:**

*md* The hash state you wish to initialize

**Returns:**

CRYPT\_OK if successful

Definition at line 343 of file rmd320.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by rmd320\_test().

```

344 {
345     LTC_ARGCHK(md != NULL);
346     md->rmd320.state[0] = 0x67452301UL;
347     md->rmd320.state[1] = 0xefcdab89UL;
348     md->rmd320.state[2] = 0x98badcfeUL;
349     md->rmd320.state[3] = 0x10325476UL;
350     md->rmd320.state[4] = 0xc3d2e1f0UL;
351     md->rmd320.state[5] = 0x76543210UL;
352     md->rmd320.state[6] = 0xfedcba98UL;
353     md->rmd320.state[7] = 0x89abcdefUL;
354     md->rmd320.state[8] = 0x01234567UL;
355     md->rmd320.state[9] = 0x3c2d1e0fUL;
356     md->rmd320.curlen = 0;
357     md->rmd320.length = 0;
358     return CRYPT_OK;
359 }
```

**5.63.3.4 int rmd320\_test (void)**

Self-test the hash.

**Returns:**

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 429 of file rmd320.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, rmd320\_done(), rmd320\_init(), and XMEMCMP.

```

430 {
431 #ifndef LTC_TEST
432     return CRYPT_NOP;
433 #else
434     static const struct {
435         char *msg;
436         unsigned char md[40];
437     } tests[] = {
438     { "",
439       { 0x22, 0xd6, 0x5d, 0x56, 0x61, 0x53, 0x6c, 0xdc, 0x75, 0xc1,
440         0xfd, 0xf5, 0xc6, 0xde, 0x7b, 0x41, 0xb9, 0xf2, 0x73, 0x25,
441         0xeb, 0xc6, 0x1e, 0x85, 0x57, 0x17, 0x7d, 0x70, 0x5a, 0x0e,
442         0xc8, 0x80, 0x15, 0x1c, 0x3a, 0x32, 0xa0, 0x08, 0x99, 0xb8 }
443     },
```



```
{ 0 },  
0,  
  
    &rmd320_init,  
    &rmd320_process,  
    &rmd320_done,  
    &rmd320_test,  
    NULL  
}
```

Definition at line 20 of file rmd320.c.

Referenced by yarrow\_start().

## 5.64 hashes/sha1.c File Reference

### 5.64.1 Detailed Description

SHA1 code by Tom St Denis.

Definition in file [sha1.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for sha1.c:

### Defines

- `#define F0(x, y, z) (z ^ (x & (y ^ z)))`
- `#define F1(x, y, z) (x ^ y ^ z)`
- `#define F2(x, y, z) ((x & y) | (z & (x | y)))`
- `#define F3(x, y, z) (x ^ y ^ z)`
- `#define FF0(a, b, c, d, e, i) e = (ROLc(a, 5) + F0(b,c,d) + e + W[i] + 0x5a827999UL); b = ROLc(b, 30);`
- `#define FF1(a, b, c, d, e, i) e = (ROLc(a, 5) + F1(b,c,d) + e + W[i] + 0x6ed9eba1UL); b = ROLc(b, 30);`
- `#define FF2(a, b, c, d, e, i) e = (ROLc(a, 5) + F2(b,c,d) + e + W[i] + 0x8f1bbcdcUL); b = ROLc(b, 30);`
- `#define FF3(a, b, c, d, e, i) e = (ROLc(a, 5) + F3(b,c,d) + e + W[i] + 0xca62c1d6UL); b = ROLc(b, 30);`

### Functions

- `static int sha1_compress (hash_state *md, unsigned char *buf)`
- `int sha1_init (hash_state *md)`  
*Initialize the hash state.*
- `int sha1_done (hash_state *md, unsigned char *out)`  
*Terminate the hash to get the digest.*
- `int sha1_test (void)`  
*Self-test the hash.*

### Variables

- `const struct ltc_hash_descriptor sha1_desc`

### 5.64.2 Define Documentation

#### 5.64.2.1 `#define F0(x, y, z) (z ^ (x & (y ^ z)))`

Definition at line 39 of file sha1.c.

**5.64.2.2 #define F1(x, y, z) (x ^ y ^ z)**

Definition at line 40 of file sha1.c.

**5.64.2.3 #define F2(x, y, z) ((x & y) | (z & (x | y)))**

Definition at line 41 of file sha1.c.

**5.64.2.4 #define F3(x, y, z) (x ^ y ^ z)**

Definition at line 42 of file sha1.c.

**5.64.2.5 #define FF0(a, b, c, d, e, i) e = (ROLC(a, 5) + F0(b,c,d) + e + W[i] + 0x5a827999UL); b = ROLC(b, 30);****5.64.2.6 #define FF1(a, b, c, d, e, i) e = (ROLC(a, 5) + F1(b,c,d) + e + W[i] + 0x6ed9eba1UL); b = ROLC(b, 30);****5.64.2.7 #define FF2(a, b, c, d, e, i) e = (ROLC(a, 5) + F2(b,c,d) + e + W[i] + 0x8f1bbcdcUL); b = ROLC(b, 30);****5.64.2.8 #define FF3(a, b, c, d, e, i) e = (ROLC(a, 5) + F3(b,c,d) + e + W[i] + 0xca62c1d6UL); b = ROLC(b, 30);****5.64.3 Function Documentation****5.64.3.1 static int sha1\_compress (hash\_state \* md, unsigned char \* buf) [static]**

Definition at line 47 of file sha1.c.

References c.

Referenced by sha1\_done().

```

49 {
50     ulong32 a,b,c,d,e,W[80],i;
51 #ifdef LTC_SMALL_CODE
52     ulong32 t;
53 #endif
54
55     /* copy the state into 512-bits into W[0..15] */
56     for (i = 0; i < 16; i++) {
57         LOAD32H(W[i], buf + (4*i));
58     }
59
60     /* copy state */
61     a = md->shal.state[0];
62     b = md->shal.state[1];
63     c = md->shal.state[2];
64     d = md->shal.state[3];
65     e = md->shal.state[4];
66
67     /* expand it */
68     for (i = 16; i < 80; i++) {
69         W[i] = ROL(W[i-3] ^ W[i-8] ^ W[i-14] ^ W[i-16], 1);
70     }
71

```

```

72  /* compress */
73  /* round one */
74  #define FF0(a,b,c,d,e,i) e = (ROlc(a, 5) + F0(b,c,d) + e + W[i] + 0x5a827999UL); b = ROlc(b, 30);
75  #define FF1(a,b,c,d,e,i) e = (ROlc(a, 5) + F1(b,c,d) + e + W[i] + 0x6ed9eba1UL); b = ROlc(b, 30);
76  #define FF2(a,b,c,d,e,i) e = (ROlc(a, 5) + F2(b,c,d) + e + W[i] + 0x8f1bbcdcUL); b = ROlc(b, 30);
77  #define FF3(a,b,c,d,e,i) e = (ROlc(a, 5) + F3(b,c,d) + e + W[i] + 0xca62c1d6UL); b = ROlc(b, 30);
78
79  #ifdef LTC_SMALL_CODE
80
81      for (i = 0; i < 20; ) {
82          FF0(a,b,c,d,e,i++); t = e; e = d; d = c; c = b; b = a; a = t;
83      }
84
85      for (; i < 40; ) {
86          FF1(a,b,c,d,e,i++); t = e; e = d; d = c; c = b; b = a; a = t;
87      }
88
89      for (; i < 60; ) {
90          FF2(a,b,c,d,e,i++); t = e; e = d; d = c; c = b; b = a; a = t;
91      }
92
93      for (; i < 80; ) {
94          FF3(a,b,c,d,e,i++); t = e; e = d; d = c; c = b; b = a; a = t;
95      }
96
97  #else
98
99      for (i = 0; i < 20; ) {
100          FF0(a,b,c,d,e,i++);
101          FF0(e,a,b,c,d,i++);
102          FF0(d,e,a,b,c,i++);
103          FF0(c,d,e,a,b,i++);
104          FF0(b,c,d,e,a,i++);
105      }
106
107      /* round two */
108      for (; i < 40; ) {
109          FF1(a,b,c,d,e,i++);
110          FF1(e,a,b,c,d,i++);
111          FF1(d,e,a,b,c,i++);
112          FF1(c,d,e,a,b,i++);
113          FF1(b,c,d,e,a,i++);
114      }
115
116      /* round three */
117      for (; i < 60; ) {
118          FF2(a,b,c,d,e,i++);
119          FF2(e,a,b,c,d,i++);
120          FF2(d,e,a,b,c,i++);
121          FF2(c,d,e,a,b,i++);
122          FF2(b,c,d,e,a,i++);
123      }
124
125      /* round four */
126      for (; i < 80; ) {
127          FF3(a,b,c,d,e,i++);
128          FF3(e,a,b,c,d,i++);
129          FF3(d,e,a,b,c,i++);
130          FF3(c,d,e,a,b,i++);
131          FF3(b,c,d,e,a,i++);
132      }
133  #endif
134
135  #undef FF0
136  #undef FF1
137  #undef FF2
138  #undef FF3

```

```

139
140     /* store */
141     md->sha1.state[0] = md->sha1.state[0] + a;
142     md->sha1.state[1] = md->sha1.state[1] + b;
143     md->sha1.state[2] = md->sha1.state[2] + c;
144     md->sha1.state[3] = md->sha1.state[3] + d;
145     md->sha1.state[4] = md->sha1.state[4] + e;
146
147     return CRYPT_OK;
148 }

```

### 5.64.3.2 int sha1\_done (hash\_state \* md, unsigned char \* out)

Terminate the hash to get the digest.

#### Parameters:

*md* The hash state

*out* [out] The destination of the hash (20 bytes)

#### Returns:

CRYPT\_OK if successful

Definition at line 193 of file sha1.c.

References CRYPT\_INVALID\_ARG, LTC\_ARGCHK, and sha1\_compress().

Referenced by sha1\_test().

```

194 {
195     int i;
196
197     LTC_ARGCHK(md != NULL);
198     LTC_ARGCHK(out != NULL);
199
200     if (md->sha1.curlen >= sizeof(md->sha1.buf)) {
201         return CRYPT_INVALID_ARG;
202     }
203
204     /* increase the length of the message */
205     md->sha1.length += md->sha1.curlen * 8;
206
207     /* append the '1' bit */
208     md->sha1.buf[md->sha1.curlen++] = (unsigned char)0x80;
209
210     /* if the length is currently above 56 bytes we append zeros
211      * then compress.  Then we can fall back to padding zeros and length
212      * encoding like normal.
213      */
214     if (md->sha1.curlen > 56) {
215         while (md->sha1.curlen < 64) {
216             md->sha1.buf[md->sha1.curlen++] = (unsigned char)0;
217         }
218         sha1_compress(md, md->sha1.buf);
219         md->sha1.curlen = 0;
220     }
221
222     /* pad upto 56 bytes of zeroes */
223     while (md->sha1.curlen < 56) {
224         md->sha1.buf[md->sha1.curlen++] = (unsigned char)0;
225     }
226

```

```

227     /* store length */
228     STORE64H(md->shal.length, md->shal.buf+56);
229     sha1_compress(md, md->shal.buf);
230
231     /* copy output */
232     for (i = 0; i < 5; i++) {
233         STORE32H(md->shal.state[i], out+(4*i));
234     }
235 #ifdef LTC_CLEAN_STACK
236     zeromem(md, sizeof(hash_state));
237 #endif
238     return CRYPT_OK;
239 }

```

Here is the call graph for this function:

### 5.64.3.3 int sha1\_init ([hash\\_state](#) \* *md*)

Initialize the hash state.

#### Parameters:

*md* The hash state you wish to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 165 of file sha1.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by sha1\_test().

```

166 {
167     LTC_ARGCHK(md != NULL);
168     md->shal.state[0] = 0x67452301UL;
169     md->shal.state[1] = 0xefcdab89UL;
170     md->shal.state[2] = 0x98badcfeUL;
171     md->shal.state[3] = 0x10325476UL;
172     md->shal.state[4] = 0xc3d2e1f0UL;
173     md->shal.curlen = 0;
174     md->shal.length = 0;
175     return CRYPT_OK;
176 }

```

### 5.64.3.4 int sha1\_test (void)

Self-test the hash.

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 245 of file sha1.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, sha1\_done(), sha1\_init(), and XMEMCMP.

```

246 {
247     #ifndef LTC_TEST

```



```

248     return CRYPT_NOP;
249 #else
250     static const struct {
251         char *msg;
252         unsigned char hash[20];
253     } tests[] = {
254         { "abc",
255           { 0xa9, 0x99, 0x3e, 0x36, 0x47, 0x06, 0x81, 0x6a,
256             0xba, 0x3e, 0x25, 0x71, 0x78, 0x50, 0xc2, 0x6c,
257             0x9c, 0xd0, 0xd8, 0x9d }
258         },
259         { "abcdcbcdcedefdefgefghfghighijhiijkijkljklmklmnlmnomnopnopq",
260           { 0x84, 0x98, 0x3E, 0x44, 0x1C, 0x3B, 0xD2, 0x6E,
261             0xBA, 0xAE, 0x4A, 0xA1, 0xF9, 0x51, 0x29, 0xE5,
262             0xE5, 0x46, 0x70, 0xF1 }
263         }
264     };
265
266     int i;
267     unsigned char tmp[20];
268     hash_state md;
269
270     for (i = 0; i < (int)(sizeof(tests) / sizeof(tests[0])); i++) {
271         sha1_init(&md);
272         sha1_process(&md, (unsigned char*)tests[i].msg, (unsigned long)strlen(tests[i].msg));
273         sha1_done(&md, tmp);
274         if (XMEMCMP(tmp, tests[i].hash, 20) != 0) {
275             return CRYPT_FAIL_TESTVECTOR;
276         }
277     }
278     return CRYPT_OK;
279 #endif
280 }

```

Here is the call graph for this function:

## 5.64.4 Variable Documentation

### 5.64.4.1 `const struct ltc_hash_descriptor sha1_desc`

**Initial value:**

```

{
    "sha1",
    2,
    20,
    64,

    { 1, 3, 14, 3, 2, 26, },
    6,

    &sha1_init,
    &sha1_process,
    &sha1_done,
    &sha1_test,
    NULL
}

```

Definition at line 21 of file sha1.c.

Referenced by `yarrow_start()`.

## 5.65 hashes/sha2/sha224.c File Reference

This graph shows which files directly or indirectly include this file:

### Functions

- int `sha224_init` (`hash_state *md`)  
*Initialize the hash state.*
- int `sha224_done` (`hash_state *md`, unsigned char \*out)  
*Terminate the hash to get the digest.*
- int `sha224_test` (void)  
*Self-test the hash.*

### Variables

- const struct `ltc_hash_descriptor sha224_desc`

### 5.65.1 Function Documentation

#### 5.65.1.1 int sha224\_done (`hash_state *md`, unsigned char \*out)

Terminate the hash to get the digest.

##### Parameters:

- md* The hash state  
*out* [out] The destination of the hash (28 bytes)

##### Returns:

CRYPT\_OK if successful

Definition at line 63 of file sha224.c.

References LTC\_ARGCHK, sha256\_done(), XMEMCPY, and zeromem().

Referenced by sha224\_test().

```
64 {  
65     unsigned char buf[32];  
66     int err;  
67  
68     LTC_ARGCHK(md != NULL);  
69     LTC_ARGCHK(out != NULL);  
70  
71     err = sha256_done(md, buf);  
72     XMEMCPY(out, buf, 28);  
73 #ifdef LTC_CLEAN_STACK  
74     zeromem(buf, sizeof(buf));  
75 #endif  
76     return err;  
77 }
```

Here is the call graph for this function:

### 5.65.1.2 int sha224\_init (hash\_state \* md)

Initialize the hash state.

**Parameters:**

*md* The hash state you wish to initialize

**Returns:**

CRYPT\_OK if successful

Definition at line 40 of file sha224.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by sha224\_test().

```
41 {
42     LTC_ARGCHK(md != NULL);
43
44     md->sha256.curlen = 0;
45     md->sha256.length = 0;
46     md->sha256.state[0] = 0xc1059ed8UL;
47     md->sha256.state[1] = 0x367cd507UL;
48     md->sha256.state[2] = 0x3070dd17UL;
49     md->sha256.state[3] = 0xf70e5939UL;
50     md->sha256.state[4] = 0xffc00b31UL;
51     md->sha256.state[5] = 0x68581511UL;
52     md->sha256.state[6] = 0x64f98fa7UL;
53     md->sha256.state[7] = 0xbefa4fa4UL;
54     return CRYPT_OK;
55 }
```

### 5.65.1.3 int sha224\_test (void)

Self-test the hash.

**Returns:**

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 83 of file sha224.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, sha224\_done(), sha224\_init(), and XMEMCMP.

```
84 {
85     #ifndef LTC_TEST
86         return CRYPT_NOP;
87     #else
88         static const struct {
89             char *msg;
90             unsigned char hash[28];
91         } tests[] = {
92             { "abc",
93               { 0x23, 0x09, 0x7d, 0x22, 0x34, 0x05, 0xd8,
94                 0x22, 0x86, 0x42, 0xa4, 0x77, 0xbd, 0xa2,
95                 0x55, 0xb3, 0x2a, 0xad, 0xbc, 0xe4, 0xbd,
96                 0xa0, 0xb3, 0xf7, 0xe3, 0x6c, 0x9d, 0xa7 }
97             },
98             { "abcdcbdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq",
```

```

99      { 0x75, 0x38, 0x8b, 0x16, 0x51, 0x27, 0x76,
100        0xcc, 0x5d, 0xba, 0x5d, 0xa1, 0xfd, 0x89,
101        0x01, 0x50, 0xb0, 0xc6, 0x45, 0x5c, 0xb4,
102        0xf5, 0x8b, 0x19, 0x52, 0x52, 0x25, 0x25 }
103    },
104    };
105
106    int i;
107    unsigned char tmp[28];
108    hash_state md;
109
110    for (i = 0; i < (int)(sizeof(tests) / sizeof(tests[0])); i++) {
111        sha224_init(&md);
112        sha224_process(&md, (unsigned char*)tests[i].msg, (unsigned long)strlen(tests[i].msg));
113        sha224_done(&md, tmp);
114        if (XMEMCMP(tmp, tests[i].hash, 28) != 0) {
115            return CRYPT_FAIL_TESTVECTOR;
116        }
117    }
118    return CRYPT_OK;
119 #endif
120 }

```

Here is the call graph for this function:

## 5.65.2 Variable Documentation

### 5.65.2.1 `const struct ltc_hash_descriptor sha224_desc`

**Initial value:**

```

{
    "sha224",
    10,
    28,
    64,

    { 2, 16, 840, 1, 101, 3, 4, 2, 4, },
    9,

    &sha224_init,
    &sha256_process,
    &sha224_done,
    &sha224_test,
    NULL
}

```

**Parameters:**

[\*sha224.c\*](#) SHA-224 new NIST standard based off of SHA-256 truncated to 224 bits (Tom St Denis)

Definition at line 16 of file sha224.c.

## 5.66 hashes/sha2/sha256.c File Reference

### 5.66.1 Detailed Description

SHA256 by Tom St Denis.

Definition in file [sha256.c](#).

```
#include "tomcrypt.h"
```

```
#include "sha224.c"
```

Include dependency graph for sha256.c:

### Defines

- #define [Ch](#)(x, y, z) ( $z \wedge (x \& (y \wedge z))$ )
- #define [Maj](#)(x, y, z) ( $((x \mid y) \& z) \mid (x \& y)$ )
- #define [S](#)(x, n) RORc((x),(n))
- #define [R](#)(x, n) (((x)&0xFFFFFFFFFUL)>>(n))
- #define [Sigma0](#)(x) ( $S(x, 2) \wedge S(x, 13) \wedge S(x, 22)$ )
- #define [Sigma1](#)(x) ( $S(x, 6) \wedge S(x, 11) \wedge S(x, 25)$ )
- #define [Gamma0](#)(x) ( $S(x, 7) \wedge S(x, 18) \wedge R(x, 3)$ )
- #define [Gamma1](#)(x) ( $S(x, 17) \wedge S(x, 19) \wedge R(x, 10)$ )
- #define [RND](#)(a, b, c, d, e, f, g, h, i, ki)

### Functions

- static int [sha256\\_compress](#) ([hash\\_state](#) \*md, unsigned char \*buf)
- int [sha256\\_init](#) ([hash\\_state](#) \*md)  
*Initialize the hash state.*
- int [sha256\\_done](#) ([hash\\_state](#) \*md, unsigned char \*out)  
*Terminate the hash to get the digest.*
- int [sha256\\_test](#) (void)  
*Self-test the hash.*

### Variables

- const struct [ltc\\_hash\\_descriptor](#) [sha256\\_desc](#)

### 5.66.2 Define Documentation

#### 5.66.2.1 #define Ch(x, y, z) ( $z \wedge (x \& (y \wedge z))$ )

Definition at line 58 of file sha256.c.

**5.66.2.2 #define Gamma0(x) (S(x, 7) ^ S(x, 18) ^ R(x, 3))**

Definition at line 64 of file sha256.c.

**5.66.2.3 #define Gamma1(x) (S(x, 17) ^ S(x, 19) ^ R(x, 10))**

Definition at line 65 of file sha256.c.

**5.66.2.4 #define Maj(x, y, z) (((x | y) & z) | (x & y))**

Definition at line 59 of file sha256.c.

**5.66.2.5 #define R(x, n) (((x)&0xFFFFFFFFFUL)>>(n))**

Definition at line 61 of file sha256.c.

**5.66.2.6 #define RND(a, b, c, d, e, f, g, h, i, ki)**

**Value:**

```
t0 = h + Sigma1(e) + Ch(e, f, g) + ki + W[i];    \
    t1 = Sigma0(a) + Maj(a, b, c);                \
    d += t0;                                       \
    h  = t0 + t1;
```

**5.66.2.7 #define S(x, n) RORc((x),(n))**

Definition at line 60 of file sha256.c.

Referenced by khazad\_setup(), rc5\_setup(), rc6\_setup(), sha256\_compress(), sha512\_compress(), and twofish\_setup().

**5.66.2.8 #define Sigma0(x) (S(x, 2) ^ S(x, 13) ^ S(x, 22))**

Definition at line 62 of file sha256.c.

**5.66.2.9 #define Sigma1(x) (S(x, 6) ^ S(x, 11) ^ S(x, 25))**

Definition at line 63 of file sha256.c.

**5.66.3 Function Documentation****5.66.3.1 static int sha256\_compress (hash\_state \*md, unsigned char \*buf) [static]**

Definition at line 71 of file sha256.c.

References S, and t1.

Referenced by sha256\_done().

```

73 {
74     ulong32 S[8], W[64], t0, t1;
75 #ifdef LTC_SMALL_CODE
76     ulong32 t;
77 #endif
78     int i;
79
80     /* copy state into S */
81     for (i = 0; i < 8; i++) {
82         S[i] = md->sha256.state[i];
83     }
84
85     /* copy the state into 512-bits into W[0..15] */
86     for (i = 0; i < 16; i++) {
87         LOAD32H(W[i], buf + (4*i));
88     }
89
90     /* fill W[16..63] */
91     for (i = 16; i < 64; i++) {
92         W[i] = Gamma1(W[i - 2]) + W[i - 7] + Gamma0(W[i - 15]) + W[i - 16];
93     }
94
95     /* Compress */
96 #ifdef LTC_SMALL_CODE
97 #define RND(a,b,c,d,e,f,g,h,i) \
98     t0 = h + Sigma1(e) + Ch(e, f, g) + K[i] + W[i]; \
99     t1 = Sigma0(a) + Maj(a, b, c); \
100     d += t0; \
101     h = t0 + t1; \
102
103     for (i = 0; i < 64; ++i) {
104         RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],i);
105         t = S[7]; S[7] = S[6]; S[6] = S[5]; S[5] = S[4];
106         S[4] = S[3]; S[3] = S[2]; S[2] = S[1]; S[1] = S[0]; S[0] = t;
107     }
108 #else
109 #define RND(a,b,c,d,e,f,g,h,i,ki) \
110     t0 = h + Sigma1(e) + Ch(e, f, g) + ki + W[i]; \
111     t1 = Sigma0(a) + Maj(a, b, c); \
112     d += t0; \
113     h = t0 + t1; \
114
115     RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],0,0x428a2f98);
116     RND(S[7],S[0],S[1],S[2],S[3],S[4],S[5],S[6],1,0x71374491);
117     RND(S[6],S[7],S[0],S[1],S[2],S[3],S[4],S[5],2,0xb5c0fbcf);
118     RND(S[5],S[6],S[7],S[0],S[1],S[2],S[3],S[4],3,0xe9b5dba5);
119     RND(S[4],S[5],S[6],S[7],S[0],S[1],S[2],S[3],4,0x3956c25b);
120     RND(S[3],S[4],S[5],S[6],S[7],S[0],S[1],S[2],5,0x59f111f1);
121     RND(S[2],S[3],S[4],S[5],S[6],S[7],S[0],S[1],6,0x923f82a4);
122     RND(S[1],S[2],S[3],S[4],S[5],S[6],S[7],S[0],7,0xab1c5ed5);
123     RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],8,0xd807aa98);
124     RND(S[7],S[0],S[1],S[2],S[3],S[4],S[5],S[6],9,0x12835b01);
125     RND(S[6],S[7],S[0],S[1],S[2],S[3],S[4],S[5],10,0x243185be);
126     RND(S[5],S[6],S[7],S[0],S[1],S[2],S[3],S[4],11,0x550c7dc3);
127     RND(S[4],S[5],S[6],S[7],S[0],S[1],S[2],S[3],12,0x72be5d74);
128     RND(S[3],S[4],S[5],S[6],S[7],S[0],S[1],S[2],13,0x80deb1fe);
129     RND(S[2],S[3],S[4],S[5],S[6],S[7],S[0],S[1],14,0x9bdc06a7);
130     RND(S[1],S[2],S[3],S[4],S[5],S[6],S[7],S[0],15,0xc19bf174);
131     RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],16,0xe49b69c1);
132     RND(S[7],S[0],S[1],S[2],S[3],S[4],S[5],S[6],17,0xefbe4786);
133     RND(S[6],S[7],S[0],S[1],S[2],S[3],S[4],S[5],18,0x0fc19dc6);
134     RND(S[5],S[6],S[7],S[0],S[1],S[2],S[3],S[4],19,0x240ca1cc);
135     RND(S[4],S[5],S[6],S[7],S[0],S[1],S[2],S[3],20,0x2de92c6f);
136     RND(S[3],S[4],S[5],S[6],S[7],S[0],S[1],S[2],21,0x4a7484aa);
137     RND(S[2],S[3],S[4],S[5],S[6],S[7],S[0],S[1],22,0x5cb0a9dc);
138     RND(S[1],S[2],S[3],S[4],S[5],S[6],S[7],S[0],23,0x76f988da);
139     RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],24,0x983e5152);

```

```

140     RND(S[7],S[0],S[1],S[2],S[3],S[4],S[5],S[6],25,0xa831c66d);
141     RND(S[6],S[7],S[0],S[1],S[2],S[3],S[4],S[5],26,0xb00327c8);
142     RND(S[5],S[6],S[7],S[0],S[1],S[2],S[3],S[4],27,0xbf597fc7);
143     RND(S[4],S[5],S[6],S[7],S[0],S[1],S[2],S[3],28,0xc6e00bf3);
144     RND(S[3],S[4],S[5],S[6],S[7],S[0],S[1],S[2],29,0xd5a79147);
145     RND(S[2],S[3],S[4],S[5],S[6],S[7],S[0],S[1],30,0x06ca6351);
146     RND(S[1],S[2],S[3],S[4],S[5],S[6],S[7],S[0],31,0x14292967);
147     RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],32,0x27b70a85);
148     RND(S[7],S[0],S[1],S[2],S[3],S[4],S[5],S[6],33,0x2e1b2138);
149     RND(S[6],S[7],S[0],S[1],S[2],S[3],S[4],S[5],34,0x4d2c6dfc);
150     RND(S[5],S[6],S[7],S[0],S[1],S[2],S[3],S[4],35,0x53380d13);
151     RND(S[4],S[5],S[6],S[7],S[0],S[1],S[2],S[3],36,0x650a7354);
152     RND(S[3],S[4],S[5],S[6],S[7],S[0],S[1],S[2],37,0x766a0abb);
153     RND(S[2],S[3],S[4],S[5],S[6],S[7],S[0],S[1],38,0x81c2c92e);
154     RND(S[1],S[2],S[3],S[4],S[5],S[6],S[7],S[0],39,0x92722c85);
155     RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],40,0xa2bfe8a1);
156     RND(S[7],S[0],S[1],S[2],S[3],S[4],S[5],S[6],41,0xa81a664b);
157     RND(S[6],S[7],S[0],S[1],S[2],S[3],S[4],S[5],42,0xc24b8b70);
158     RND(S[5],S[6],S[7],S[0],S[1],S[2],S[3],S[4],43,0xc76c51a3);
159     RND(S[4],S[5],S[6],S[7],S[0],S[1],S[2],S[3],44,0xd192e819);
160     RND(S[3],S[4],S[5],S[6],S[7],S[0],S[1],S[2],45,0xd6990624);
161     RND(S[2],S[3],S[4],S[5],S[6],S[7],S[0],S[1],46,0xf40e3585);
162     RND(S[1],S[2],S[3],S[4],S[5],S[6],S[7],S[0],47,0x106aa070);
163     RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],48,0x19a4c116);
164     RND(S[7],S[0],S[1],S[2],S[3],S[4],S[5],S[6],49,0x1e376c08);
165     RND(S[6],S[7],S[0],S[1],S[2],S[3],S[4],S[5],50,0x2748774c);
166     RND(S[5],S[6],S[7],S[0],S[1],S[2],S[3],S[4],51,0x34b0bcb5);
167     RND(S[4],S[5],S[6],S[7],S[0],S[1],S[2],S[3],52,0x391c0cb3);
168     RND(S[3],S[4],S[5],S[6],S[7],S[0],S[1],S[2],53,0x4ed8aa4a);
169     RND(S[2],S[3],S[4],S[5],S[6],S[7],S[0],S[1],54,0x5b9cca4f);
170     RND(S[1],S[2],S[3],S[4],S[5],S[6],S[7],S[0],55,0x682e6ff3);
171     RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],56,0x748f82ee);
172     RND(S[7],S[0],S[1],S[2],S[3],S[4],S[5],S[6],57,0x78a5636f);
173     RND(S[6],S[7],S[0],S[1],S[2],S[3],S[4],S[5],58,0x84c87814);
174     RND(S[5],S[6],S[7],S[0],S[1],S[2],S[3],S[4],59,0x8cc70208);
175     RND(S[4],S[5],S[6],S[7],S[0],S[1],S[2],S[3],60,0x90bfeffa);
176     RND(S[3],S[4],S[5],S[6],S[7],S[0],S[1],S[2],61,0xa4506ceb);
177     RND(S[2],S[3],S[4],S[5],S[6],S[7],S[0],S[1],62,0xbef9a3f7);
178     RND(S[1],S[2],S[3],S[4],S[5],S[6],S[7],S[0],63,0xc67178f2);
179
180 #undef RND
181
182 #endif
183
184     /* feedback */
185     for (i = 0; i < 8; i++) {
186         md->sha256.state[i] = md->sha256.state[i] + S[i];
187     }
188     return CRYPT_OK;
189 }

```

### 5.66.3.2 int sha256\_done (hash\_state \* md, unsigned char \* out)

Terminate the hash to get the digest.

#### Parameters:

*md* The hash state

*out* [out] The destination of the hash (32 bytes)

#### Returns:

CRYPT\_OK if successful



Definition at line 238 of file sha256.c.

References CRYPT\_INVALID\_ARG, LTC\_ARGCHK, and sha256\_compress().

Referenced by fortuna\_done(), fortuna\_export(), fortuna\_reseed(), fortuna\_start(), sha224\_done(), and sha256\_test().

```

239 {
240     int i;
241
242     LTC_ARGCHK(md != NULL);
243     LTC_ARGCHK(out != NULL);
244
245     if (md->sha256.curlen >= sizeof(md->sha256.buf)) {
246         return CRYPT_INVALID_ARG;
247     }
248
249
250     /* increase the length of the message */
251     md->sha256.length += md->sha256.curlen * 8;
252
253     /* append the '1' bit */
254     md->sha256.buf[md->sha256.curlen++] = (unsigned char)0x80;
255
256     /* if the length is currently above 56 bytes we append zeros
257      * then compress.  Then we can fall back to padding zeros and length
258      * encoding like normal.
259      */
260     if (md->sha256.curlen > 56) {
261         while (md->sha256.curlen < 64) {
262             md->sha256.buf[md->sha256.curlen++] = (unsigned char)0;
263         }
264         sha256_compress(md, md->sha256.buf);
265         md->sha256.curlen = 0;
266     }
267
268     /* pad upto 56 bytes of zeroes */
269     while (md->sha256.curlen < 56) {
270         md->sha256.buf[md->sha256.curlen++] = (unsigned char)0;
271     }
272
273     /* store length */
274     STORE64H(md->sha256.length, md->sha256.buf+56);
275     sha256_compress(md, md->sha256.buf);
276
277     /* copy output */
278     for (i = 0; i < 8; i++) {
279         STORE32H(md->sha256.state[i], out+(4*i));
280     }
281 #ifdef LTC_CLEAN_STACK
282     zeromem(md, sizeof(hash_state));
283 #endif
284     return CRYPT_OK;
285 }

```

Here is the call graph for this function:

### 5.66.3.3 int sha256\_init ([hash\\_state](#) \* *md*)

Initialize the hash state.

#### Parameters:

*md* The hash state you wish to initialize

**Returns:**

CRYPT\_OK if successful

Definition at line 206 of file sha256.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by fortuna\_reseed(), fortuna\_start(), and sha256\_test().

```

207 {
208     LTC_ARGCHK(md != NULL);
209
210     md->sha256.curlen = 0;
211     md->sha256.length = 0;
212     md->sha256.state[0] = 0x6A09E667UL;
213     md->sha256.state[1] = 0xBB67AE85UL;
214     md->sha256.state[2] = 0x3C6EF372UL;
215     md->sha256.state[3] = 0xA54FF53AUL;
216     md->sha256.state[4] = 0x510E527FUL;
217     md->sha256.state[5] = 0x9B05688CUL;
218     md->sha256.state[6] = 0x1F83D9ABUL;
219     md->sha256.state[7] = 0x5BE0CD19UL;
220     return CRYPT_OK;
221 }
```

**5.66.3.4 int sha256\_test (void)**

Self-test the hash.

**Returns:**

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 291 of file sha256.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, sha256\_done(), sha256\_init(), and XMEMCMP.

Referenced by fortuna\_test().

```

292 {
293     #ifndef LTC_TEST
294         return CRYPT_NOP;
295     #else
296         static const struct {
297             char *msg;
298             unsigned char hash[32];
299         } tests[] = {
300             { "abc",
301               { 0xba, 0x78, 0x16, 0xbf, 0x8f, 0x01, 0xcf, 0xea,
302                 0x41, 0x41, 0x40, 0xde, 0x5d, 0xae, 0x22, 0x23,
303                 0xb0, 0x03, 0x61, 0xa3, 0x96, 0x17, 0x7a, 0x9c,
304                 0xb4, 0x10, 0xff, 0x61, 0xf2, 0x00, 0x15, 0xad }
305             },
306             { "abdcdbcdcedefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq",
307               { 0x24, 0x8d, 0x6a, 0x61, 0xd2, 0x06, 0x38, 0xb8,
308                 0xe5, 0xc0, 0x26, 0x93, 0x0c, 0x3e, 0x60, 0x39,
309                 0xa3, 0x3c, 0xe4, 0x59, 0x64, 0xff, 0x21, 0x67,
310                 0xf6, 0xec, 0xed, 0xd4, 0x19, 0xdb, 0x06, 0xc1 }
311             },
312         };
313 }
```

```
314     int i;
315     unsigned char tmp[32];
316     hash_state md;
317
318     for (i = 0; i < (int)(sizeof(tests) / sizeof(tests[0])); i++) {
319         sha256_init(&md);
320         sha256_process(&md, (unsigned char*)tests[i].msg, (unsigned long)strlen(tests[i].msg));
321         sha256_done(&md, tmp);
322         if (XMEMCMP(tmp, tests[i].hash, 32) != 0) {
323             return CRYPT_FAIL_TESTVECTOR;
324         }
325     }
326     return CRYPT_OK;
327 #endif
328 }
```

Here is the call graph for this function:

## 5.66.4 Variable Documentation

### 5.66.4.1 const struct [ltc\\_hash\\_descriptor sha256\\_desc](#)

**Initial value:**

```
{
    "sha256",
    0,
    32,
    64,

    { 2, 16, 840, 1, 101, 3, 4, 2, 1, },
    9,

    &sha256_init,
    &sha256_process,
    &sha256_done,
    &sha256_test,
    NULL
}
```

Definition at line 20 of file sha256.c.

Referenced by `yarrow_start()`.

## 5.67 hashes/sha2/sha384.c File Reference

This graph shows which files directly or indirectly include this file:

### Functions

- int [sha384\\_init](#) ([hash\\_state](#) \*md)  
*Initialize the hash state.*
- int [sha384\\_done](#) ([hash\\_state](#) \*md, unsigned char \*out)  
*Terminate the hash to get the digest.*
- int [sha384\\_test](#) (void)  
*Self-test the hash.*

### Variables

- const struct [ltc\\_hash\\_descriptor](#) [sha384\\_desc](#)

### 5.67.1 Function Documentation

#### 5.67.1.1 int sha384\_done ([hash\\_state](#) \* md, unsigned char \* out)

Terminate the hash to get the digest.

#### Parameters:

- md* The hash state
- out* [out] The destination of the hash (48 bytes)

#### Returns:

CRYPT\_OK if successful

Definition at line 62 of file sha384.c.

References [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), [sha512\\_done\(\)](#), [XMEMCPY](#), and [zeromem\(\)](#).

Referenced by [sha384\\_test\(\)](#).

```

63 {
64     unsigned char buf[64];
65
66     LTC_ARGCHK(md != NULL);
67     LTC_ARGCHK(out != NULL);
68
69     if (md->sha512.curlen >= sizeof(md->sha512.buf)) {
70         return CRYPT_INVALID_ARG;
71     }
72
73     sha512_done(md, buf);
74     XMEMCPY(out, buf, 48);
75 #ifdef LTC_CLEAN_STACK

```

```
76     zeromem(buf, sizeof(buf));
77 #endif
78     return CRYPT_OK;
79 }
```

Here is the call graph for this function:

#### 5.67.1.2 int sha384\_init (hash\_state \* md)

Initialize the hash state.

##### Parameters:

*md* The hash state you wish to initialize

##### Returns:

CRYPT\_OK if successful

Definition at line 39 of file sha384.c.

References CONST64, CRYPT\_OK, and LTC\_ARGCHK.

Referenced by sha384\_test().

```
40 {
41     LTC_ARGCHK(md != NULL);
42
43     md->sha512.curlen = 0;
44     md->sha512.length = 0;
45     md->sha512.state[0] = CONST64(0xcbbb9d5dc1059ed8);
46     md->sha512.state[1] = CONST64(0x629a292a367cd507);
47     md->sha512.state[2] = CONST64(0x9159015a3070dd17);
48     md->sha512.state[3] = CONST64(0x152fec8d8f70e5939);
49     md->sha512.state[4] = CONST64(0x67332667ffc00b31);
50     md->sha512.state[5] = CONST64(0x8eb44a8768581511);
51     md->sha512.state[6] = CONST64(0xdb0c2e0d64f98fa7);
52     md->sha512.state[7] = CONST64(0x47b5481dbefa4fa4);
53     return CRYPT_OK;
54 }
```

#### 5.67.1.3 int sha384\_test (void)

Self-test the hash.

##### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 85 of file sha384.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, sha384\_done(), sha384\_init(), and XMEMP\_CMP.

```
86 {
87     #ifndef LTC_TEST
88         return CRYPT_NOP;
89     #else
90         static const struct {
91             char *msg;
```

```

92     unsigned char hash[48];
93 } tests[] = {
94     { "abc",
95       { 0xcb, 0x00, 0x75, 0x3f, 0x45, 0xa3, 0x5e, 0x8b,
96         0xb5, 0xa0, 0x3d, 0x69, 0x9a, 0xc6, 0x50, 0x07,
97         0x27, 0x2c, 0x32, 0xab, 0x0e, 0xde, 0xd1, 0x63,
98         0x1a, 0x8b, 0x60, 0x5a, 0x43, 0xff, 0x5b, 0xed,
99         0x80, 0x86, 0x07, 0x2b, 0xa1, 0xe7, 0xcc, 0x23,
100        0x58, 0xba, 0xec, 0xa1, 0x34, 0xc8, 0x25, 0xa7 }
101     },
102     { "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz",
103       { 0x09, 0x33, 0x0c, 0x33, 0xf7, 0x11, 0x47, 0xe8,
104         0x3d, 0x19, 0x2f, 0xc7, 0x82, 0xcd, 0x1b, 0x47,
105         0x53, 0x11, 0x1b, 0x17, 0x3b, 0x3b, 0x05, 0xd2,
106         0x2f, 0xa0, 0x80, 0x86, 0xe3, 0xb0, 0xf7, 0x12,
107         0xfc, 0xc7, 0xc7, 0x1a, 0x55, 0x7e, 0x2d, 0xb9,
108         0x66, 0xc3, 0xe9, 0xfa, 0x91, 0x74, 0x60, 0x39 }
109     },
110 };
111
112 int i;
113 unsigned char tmp[48];
114 hash_state md;
115
116 for (i = 0; i < (int)(sizeof(tests) / sizeof(tests[0])); i++) {
117     sha384_init(&md);
118     sha384_process(&md, (unsigned char*)tests[i].msg, (unsigned long)strlen(tests[i].msg));
119     sha384_done(&md, tmp);
120     if (XMEMCMP(tmp, tests[i].hash, 48) != 0) {
121         return CRYPT_FAIL_TESTVECTOR;
122     }
123 }
124 return CRYPT_OK;
125 #endif
126 }

```

Here is the call graph for this function:

## 5.67.2 Variable Documentation

### 5.67.2.1 const struct `ltc_hash_descriptor sha384_desc`

**Initial value:**

```

{
    "sha384",
    4,
    48,
    128,

    { 2, 16, 840, 1, 101, 3, 4, 2, 2, },
    9,

    &sha384_init,
    &sha512_process,
    &sha384_done,
    &sha384_test,
    NULL
}

```

**Parameters:**

[sha384.c](#) SHA384 hash included in [sha512.c](#), Tom St Denis

Definition at line 16 of file sha384.c.

## 5.68 hashes/sha2/sha512.c File Reference

```
#include "tomcrypt.h"
```

```
#include "sha384.c"
```

Include dependency graph for sha512.c:

### Defines

- #define **Ch**(x, y, z) ( $z \wedge (x \& (y \wedge z))$ )
- #define **Maj**(x, y, z) ( $((x \mid y) \& z) \mid (x \& y)$ )
- #define **S**(x, n) ROR64c(x, n)
- #define **R**(x, n) ( $((x) \& \text{CONST64}(0xFFFFFFFFFFFFFFFF)) \gg ((\text{ulong64})n)$ )
- #define **Sigma0**(x) ( $S(x, 28) \wedge S(x, 34) \wedge S(x, 39)$ )
- #define **Sigma1**(x) ( $S(x, 14) \wedge S(x, 18) \wedge S(x, 41)$ )
- #define **Gamma0**(x) ( $S(x, 1) \wedge S(x, 8) \wedge R(x, 7)$ )
- #define **Gamma1**(x) ( $S(x, 19) \wedge S(x, 61) \wedge R(x, 6)$ )
- #define **RND**(a, b, c, d, e, f, g, h, i)

### Functions

- static int **sha512\_compress** (**hash\_state** \*md, unsigned char \*buf)
- int **sha512\_init** (**hash\_state** \*md)  
*Initialize the hash state.*
- int **sha512\_done** (**hash\_state** \*md, unsigned char \*out)  
*Terminate the hash to get the digest.*
- int **sha512\_test** (void)  
*Self-test the hash.*

### Variables

- const struct **ltc\_hash\_descriptor** sha512\_desc
- static const **ulong64** K [80]

#### 5.68.1 Define Documentation

##### 5.68.1.1 #define Ch(x, y, z) ( $z \wedge (x \& (y \wedge z))$ )

Definition at line 83 of file sha512.c.

##### 5.68.1.2 #define Gamma0(x) ( $S(x, 1) \wedge S(x, 8) \wedge R(x, 7)$ )

Definition at line 89 of file sha512.c.



**5.68.1.3 #define Gamma1(x) (S(x, 19) ^ S(x, 61) ^ R(x, 6))**

Definition at line 90 of file sha512.c.

**5.68.1.4 #define Maj(x, y, z) (((x | y) & z) | (x & y))**

Definition at line 84 of file sha512.c.

**5.68.1.5 #define R(x, n) (((x)&CONST64(0xFFFFFFFFFFFFFFFF))>>((ulong64)n))**

Definition at line 86 of file sha512.c.

**5.68.1.6 #define RND(a, b, c, d, e, f, g, h, i)**

**Value:**

```
t0 = h + Sigma1(e) + Ch(e, f, g) + K[i] + W[i];    \
    t1 = Sigma0(a) + Maj(a, b, c);                \
    d += t0;                                       \
    h  = t0 + t1;
```

**5.68.1.7 #define S(x, n) ROR64c(x, n)**

Definition at line 85 of file sha512.c.

**5.68.1.8 #define Sigma0(x) (S(x, 28) ^ S(x, 34) ^ S(x, 39))**

Definition at line 87 of file sha512.c.

**5.68.1.9 #define Sigma1(x) (S(x, 14) ^ S(x, 18) ^ S(x, 41))**

Definition at line 88 of file sha512.c.

**5.68.2 Function Documentation****5.68.2.1 static int sha512\_compress (hash\_state \*md, unsigned char \*buf) [static]**

Definition at line 96 of file sha512.c.

References S, and t1.

Referenced by sha512\_done().

```
98 {
99     ulong64 S[8], W[80], t0, t1;
100     int i;
101
102     /* copy state into S */
103     for (i = 0; i < 8; i++) {
104         S[i] = md->sha512.state[i];
105     }
```

```

106
107     /* copy the state into 1024-bits into W[0..15] */
108     for (i = 0; i < 16; i++) {
109         LOAD64H(W[i], buf + (8*i));
110     }
111
112     /* fill W[16..79] */
113     for (i = 16; i < 80; i++) {
114         W[i] = Gamma1(W[i - 2]) + W[i - 7] + Gamma0(W[i - 15]) + W[i - 16];
115     }
116
117     /* Compress */
118 #ifdef LTC_SMALL_CODE
119     for (i = 0; i < 80; i++) {
120         t0 = S[7] + Sigma1(S[4]) + Ch(S[4], S[5], S[6]) + K[i] + W[i];
121         t1 = Sigma0(S[0]) + Maj(S[0], S[1], S[2]);
122         S[7] = S[6];
123         S[6] = S[5];
124         S[5] = S[4];
125         S[4] = S[3] + t0;
126         S[3] = S[2];
127         S[2] = S[1];
128         S[1] = S[0];
129         S[0] = t0 + t1;
130     }
131 #else
132 #define RND(a,b,c,d,e,f,g,h,i) \
133     t0 = h + Sigma1(e) + Ch(e, f, g) + K[i] + W[i]; \
134     t1 = Sigma0(a) + Maj(a, b, c); \
135     d += t0; \
136     h = t0 + t1;
137
138     for (i = 0; i < 80; i += 8) {
139         RND(S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],i+0);
140         RND(S[7],S[0],S[1],S[2],S[3],S[4],S[5],S[6],i+1);
141         RND(S[6],S[7],S[0],S[1],S[2],S[3],S[4],S[5],i+2);
142         RND(S[5],S[6],S[7],S[0],S[1],S[2],S[3],S[4],i+3);
143         RND(S[4],S[5],S[6],S[7],S[0],S[1],S[2],S[3],i+4);
144         RND(S[3],S[4],S[5],S[6],S[7],S[0],S[1],S[2],i+5);
145         RND(S[2],S[3],S[4],S[5],S[6],S[7],S[0],S[1],i+6);
146         RND(S[1],S[2],S[3],S[4],S[5],S[6],S[7],S[0],i+7);
147     }
148 #endif
149
150
151     /* feedback */
152     for (i = 0; i < 8; i++) {
153         md->sha512.state[i] = md->sha512.state[i] + S[i];
154     }
155
156     return CRYPT_OK;
157 }

```

### 5.68.2.2 int sha512\_done ([hash\\_state](#) \* *md*, unsigned char \* *out*)

Terminate the hash to get the digest.

#### Parameters:

- md* The hash state
- out* [out] The destination of the hash (64 bytes)

#### Returns:

CRYPT\_OK if successful

Definition at line 206 of file sha512.c.

References `CONST64`, `CRYPT_INVALID_ARG`, `LTC_ARGCHK`, and `sha512_compress()`.

Referenced by `sha384_done()`, and `sha512_test()`.

```

207 {
208     int i;
209
210     LTC_ARGCHK(md != NULL);
211     LTC_ARGCHK(out != NULL);
212
213     if (md->sha512.curlen >= sizeof(md->sha512.buf)) {
214         return CRYPT_INVALID_ARG;
215     }
216
217     /* increase the length of the message */
218     md->sha512.length += md->sha512.curlen * CONST64(8);
219
220     /* append the '1' bit */
221     md->sha512.buf[md->sha512.curlen++] = (unsigned char)0x80;
222
223     /* if the length is currently above 112 bytes we append zeros
224      * then compress.  Then we can fall back to padding zeros and length
225      * encoding like normal.
226      */
227     if (md->sha512.curlen > 112) {
228         while (md->sha512.curlen < 128) {
229             md->sha512.buf[md->sha512.curlen++] = (unsigned char)0;
230         }
231         sha512_compress(md, md->sha512.buf);
232         md->sha512.curlen = 0;
233     }
234
235     /* pad upto 120 bytes of zeroes
236      * note: that from 112 to 120 is the 64 MSB of the length.  We assume that you won't hash
237      * > 2^64 bits of data... :-)
238      */
239     while (md->sha512.curlen < 120) {
240         md->sha512.buf[md->sha512.curlen++] = (unsigned char)0;
241     }
242
243     /* store length */
244     STORE64H(md->sha512.length, md->sha512.buf+120);
245     sha512_compress(md, md->sha512.buf);
246
247     /* copy output */
248     for (i = 0; i < 8; i++) {
249         STORE64H(md->sha512.state[i], out+(8*i));
250     }
251 #ifdef LTC_CLEAN_STACK
252     zeromem(md, sizeof(hash_state));
253 #endif
254     return CRYPT_OK;
255 }

```

Here is the call graph for this function:

### 5.68.2.3 int sha512\_init (hash\_state \* md)

Initialize the hash state.

#### Parameters:

*md* The hash state you wish to initialize

**Returns:**

CRYPT\_OK if successful

Definition at line 175 of file sha512.c.

References CONST64, CRYPT\_OK, and LTC\_ARGCHK.

Referenced by sha512\_test().

```

176 {
177     LTC_ARGCHK(md != NULL);
178     md->sha512.curlen = 0;
179     md->sha512.length = 0;
180     md->sha512.state[0] = CONST64(0x6a09e667f3bcc908);
181     md->sha512.state[1] = CONST64(0xbb67ae8584caa73b);
182     md->sha512.state[2] = CONST64(0x3c6ef372fe94f82b);
183     md->sha512.state[3] = CONST64(0xa54ff53a5f1d36f1);
184     md->sha512.state[4] = CONST64(0x510e527fade682d1);
185     md->sha512.state[5] = CONST64(0x9b05688c2b3e6c1f);
186     md->sha512.state[6] = CONST64(0x1f83d9abfb41bd6b);
187     md->sha512.state[7] = CONST64(0x5be0cd19137e2179);
188     return CRYPT_OK;
189 }
```

**5.68.2.4 int sha512\_test (void)**

Self-test the hash.

**Returns:**

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 261 of file sha512.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, sha512\_done(), sha512\_init(), and XMEMCMP.

```

262 {
263     #ifndef LTC_TEST
264         return CRYPT_NOP;
265     #else
266         static const struct {
267             char *msg;
268             unsigned char hash[64];
269         } tests[] = {
270             { "abc",
271               { 0xdd, 0xaf, 0x35, 0xa1, 0x93, 0x61, 0x7a, 0xba,
272                 0xcc, 0x41, 0x73, 0x49, 0xae, 0x20, 0x41, 0x31,
273                 0x12, 0xe6, 0xfa, 0x4e, 0x89, 0xa9, 0x7e, 0xa2,
274                 0x0a, 0x9e, 0xee, 0xe6, 0x4b, 0x55, 0xd3, 0x9a,
275                 0x21, 0x92, 0x99, 0x2a, 0x27, 0x4f, 0xc1, 0xa8,
276                 0x36, 0xba, 0x3c, 0x23, 0xa3, 0xfe, 0xeb, 0xbd,
277                 0x45, 0x4d, 0x44, 0x23, 0x64, 0x3c, 0xe8, 0x0e,
278                 0x2a, 0x9a, 0xc9, 0x4f, 0xa5, 0x4c, 0xa4, 0x9f }
279             },
280             { "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz",
281               { 0x8e, 0x95, 0x9b, 0x75, 0xda, 0xe3, 0x13, 0xda,
282                 0x8c, 0xf4, 0xf7, 0x28, 0x14, 0xfc, 0x14, 0x3f,
283                 0x8f, 0x77, 0x79, 0xc6, 0xeb, 0x9f, 0x7f, 0xa1,
284                 0x72, 0x99, 0xae, 0xad, 0xb6, 0x88, 0x90, 0x18,
285                 0x50, 0x1d, 0x28, 0x9e, 0x49, 0x00, 0xf7, 0xe4,
286                 0x33, 0x1b, 0x99, 0xde, 0xc4, 0xb5, 0x43, 0x3a,
```

```

287         0xc7, 0xd3, 0x29, 0xee, 0xb6, 0xdd, 0x26, 0x54,
288         0x5e, 0x96, 0xe5, 0x5b, 0x87, 0x4b, 0xe9, 0x09 }
289     },
290 };
291
292     int i;
293     unsigned char tmp[64];
294     hash_state md;
295
296     for (i = 0; i < (int)(sizeof(tests) / sizeof(tests[0])); i++) {
297         sha512_init(&md);
298         sha512_process(&md, (unsigned char *)tests[i].msg, (unsigned long)strlen(tests[i].msg));
299         sha512_done(&md, tmp);
300         if (XMEMCMP(tmp, tests[i].hash, 64) != 0) {
301             return CRYPT_FAIL_TESTVECTOR;
302         }
303     }
304     return CRYPT_OK;
305 #endif
306 }

```

Here is the call graph for this function:

### 5.68.3 Variable Documentation

#### 5.68.3.1 `const ulong64 K[80]` `[static]`

Definition at line 39 of file sha512.c.

Referenced by `f9_test()`, `gcm_test()`, `pelican_test()`, `rc5_ecb_decrypt()`, `rc5_ecb_encrypt()`, `rc6_ecb_decrypt()`, `rc6_ecb_encrypt()`, `whirlpool_compress()`, `xcbc_test()`, and `xtea_setup()`.

#### 5.68.3.2 `const struct ltc\_hash\_descriptor sha512_desc`

**Initial value:**

```

{
    "sha512",
    5,
    64,
    128,

    { 2, 16, 840, 1, 101, 3, 4, 2, 3, },
    9,

    &sha512_init,
    &sha512_process,
    &sha512_done,
    &sha512_test,
    NULL
}

```

**Parameters:**

[sha512.c](#) SHA512 by Tom St Denis

Definition at line 20 of file sha512.c.

Referenced by `yarrow_start()`.

## 5.69 hashes/tiger.c File Reference

### 5.69.1 Detailed Description

Tiger hash function, Tom St Denis.

Definition in file [tiger.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for tiger.c:

### Defines

- `#define` [t1](#) ([table](#))
- `#define` [t2](#) ([table](#)+256)
- `#define` [t3](#) ([table](#)+256\*2)
- `#define` [t4](#) ([table](#)+256\*3)
- `#define` [INLINE](#)

### Functions

- static `INLINE` void [tiger\\_round](#) ([ulong64](#) \*a, [ulong64](#) \*b, [ulong64](#) \*c, [ulong64](#) x, int mul)
- static void [pass](#) ([ulong64](#) \*a, [ulong64](#) \*b, [ulong64](#) \*c, [ulong64](#) \*x, int mul)
- static void [key\\_schedule](#) ([ulong64](#) \*x)
- static int [tiger\\_compress](#) ([hash\\_state](#) \*md, unsigned char \*buf)
- int [tiger\\_init](#) ([hash\\_state](#) \*md)

*Initialize the hash state.*

- int [tiger\\_done](#) ([hash\\_state](#) \*md, unsigned char \*out)

*Terminate the hash to get the digest.*

- int [tiger\\_test](#) (void)

*Self-test the hash.*

### Variables

- const struct [ltc\\_hash\\_descriptor](#) [tiger\\_desc](#)
- static const [ulong64](#) [table](#) [4 \*256]

### 5.69.2 Define Documentation

#### 5.69.2.1 `#define` [INLINE](#)

Definition at line 561 of file tiger.c.

**5.69.2.2 #define t1 (table)**

Definition at line 39 of file tiger.c.

Referenced by ECB\_ENC(), four\_rounds(), is\_point(), ltc\_ecc\_map(), ltc\_ecc\_projective\_add\_point(), ltc\_ecc\_projective\_dbl\_point(), sha256\_compress(), sha512\_compress(), tiger\_round(), twofish\_ecb\_decrypt(), and twofish\_ecb\_encrypt().

**5.69.2.3 #define t2 (table+256)**

Definition at line 40 of file tiger.c.

Referenced by ECB\_ENC(), four\_rounds(), is\_point(), ltc\_ecc\_map(), ltc\_ecc\_projective\_add\_point(), ltc\_ecc\_projective\_dbl\_point(), tiger\_round(), twofish\_ecb\_decrypt(), and twofish\_ecb\_encrypt().

**5.69.2.4 #define t3 (table+256\*2)**

Definition at line 41 of file tiger.c.

Referenced by ECB\_ENC(), four\_rounds(), and tiger\_round().

**5.69.2.5 #define t4 (table+256\*3)**

Definition at line 42 of file tiger.c.

Referenced by tiger\_round().

**5.69.3 Function Documentation****5.69.3.1 static void key\_schedule (ulong64 \* x) [static]**

Definition at line 592 of file tiger.c.

References CONST64.

```

593 {
594     x[0] -= x[7] ^ CONST64(0xA5A5A5A5A5A5A5A5);
595     x[1] ^= x[0];
596     x[2] += x[1];
597     x[3] -= x[2] ^ ((~x[1]) << 19);
598     x[4] ^= x[3];
599     x[5] += x[4];
600     x[6] -= x[5] ^ ((~x[4]) >> 23);
601     x[7] ^= x[6];
602     x[0] += x[7];
603     x[1] -= x[0] ^ ((~x[7]) << 19);
604     x[2] ^= x[1];
605     x[3] += x[2];
606     x[4] -= x[3] ^ ((~x[2]) >> 23);
607     x[5] ^= x[4];
608     x[6] += x[5];
609     x[7] -= x[6] ^ CONST64(0x0123456789ABCDEF);
610 }
```

### 5.69.3.2 static void pass (ulong64 \* a, ulong64 \* b, ulong64 \* c, ulong64 \* x, int mul) [static]

Definition at line 579 of file tiger.c.

References tiger\_round().

```

580 {
581     tiger_round(a,b,c,x[0],mul);
582     tiger_round(b,c,a,x[1],mul);
583     tiger_round(c,a,b,x[2],mul);
584     tiger_round(a,b,c,x[3],mul);
585     tiger_round(b,c,a,x[4],mul);
586     tiger_round(c,a,b,x[5],mul);
587     tiger_round(a,b,c,x[6],mul);
588     tiger_round(b,c,a,x[7],mul);
589 }
```

Here is the call graph for this function:

### 5.69.3.3 static int tiger\_compress (hash\_state \* md, unsigned char \* buf) [static]

Definition at line 615 of file tiger.c.

References c.

Referenced by tiger\_done().

```

617 {
618     ulong64 a, b, c, x[8];
619     unsigned long i;
620
621     /* load words */
622     for (i = 0; i < 8; i++) {
623         LOAD64L(x[i], &buf[8*i]);
624     }
625     a = md->tiger.state[0];
626     b = md->tiger.state[1];
627     c = md->tiger.state[2];
628
629     pass(&a, &b, &c, x, 5);
630     key_schedule(x);
631     pass(&c, &a, &b, x, 7);
632     key_schedule(x);
633     pass(&b, &c, &a, x, 9);
634
635     /* store state */
636     md->tiger.state[0] = a ^ md->tiger.state[0];
637     md->tiger.state[1] = b - md->tiger.state[1];
638     md->tiger.state[2] = c + md->tiger.state[2];
639
640     return CRYPT_OK;
641 }
```

### 5.69.3.4 int tiger\_done (hash\_state \* md, unsigned char \* out)

Terminate the hash to get the digest.

#### Parameters:

*md* The hash state



**out** [out] The destination of the hash (24 bytes)

**Returns:**

CRYPT\_OK if successful

Definition at line 684 of file tiger.c.

References CRYPT\_INVALID\_ARG, CRYPT\_OK, LTC\_ARGCHK, tiger\_compress(), and zeromem().

Referenced by tiger\_test().

```

685 {
686     LTC_ARGCHK(md != NULL);
687     LTC_ARGCHK(out != NULL);
688
689     if (md->tiger.curlen >= sizeof(md->tiger.buf)) {
690         return CRYPT_INVALID_ARG;
691     }
692
693     /* increase the length of the message */
694     md->tiger.length += md->tiger.curlen * 8;
695
696     /* append the '1' bit */
697     md->tiger.buf[md->tiger.curlen++] = (unsigned char)0x01;
698
699     /* if the length is currently above 56 bytes we append zeros
700      * then compress.  Then we can fall back to padding zeros and length
701      * encoding like normal. */
702     if (md->tiger.curlen > 56) {
703         while (md->tiger.curlen < 64) {
704             md->tiger.buf[md->tiger.curlen++] = (unsigned char)0;
705         }
706         tiger_compress(md, md->tiger.buf);
707         md->tiger.curlen = 0;
708     }
709
710     /* pad upto 56 bytes of zeroes */
711     while (md->tiger.curlen < 56) {
712         md->tiger.buf[md->tiger.curlen++] = (unsigned char)0;
713     }
714
715     /* store length */
716     STORE64L(md->tiger.length, md->tiger.buf+56);
717     tiger_compress(md, md->tiger.buf);
718
719     /* copy output */
720     STORE64L(md->tiger.state[0], &out[0]);
721     STORE64L(md->tiger.state[1], &out[8]);
722     STORE64L(md->tiger.state[2], &out[16]);
723 #ifdef LTC_CLEAN_STACK
724     zeromem(md, sizeof(hash_state));
725 #endif
726
727     return CRYPT_OK;
728 }
```

Here is the call graph for this function:

### 5.69.3.5 int tiger\_init (hash\_state \* md)

Initialize the hash state.

**Parameters:**

**md** The hash state you wish to initialize

**Returns:**

CRYPT\_OK if successful

Definition at line 658 of file tiger.c.

References CONST64, CRYPT\_OK, and LTC\_ARGCHK.

Referenced by tiger\_test().

```

659 {
660     LTC_ARGCHK(md != NULL);
661     md->tiger.state[0] = CONST64(0x0123456789ABCDEF);
662     md->tiger.state[1] = CONST64(0xFEDCBA9876543210);
663     md->tiger.state[2] = CONST64(0xF096A5B4C3B2E187);
664     md->tiger.curlen = 0;
665     md->tiger.length = 0;
666     return CRYPT_OK;
667 }
```

### 5.69.3.6 static inline void tiger\_round (ulong64 \* a, ulong64 \* b, ulong64 \* c, ulong64 x, int mul) [static]

Definition at line 565 of file tiger.c.

References byte, t1, t2, t3, and t4.

Referenced by pass().

```

566 {
567     ulong64 tmp;
568     tmp = (*c ^= x);
569     *a -= t1[byte(tmp, 0)] ^ t2[byte(tmp, 2)] ^ t3[byte(tmp, 4)] ^ t4[byte(tmp, 6)];
570     tmp = (*b += t4[byte(tmp, 1)] ^ t3[byte(tmp, 3)] ^ t2[byte(tmp, 5)] ^ t1[byte(tmp, 7)]);
571     switch (mul) {
572         case 5: *b = (tmp << 2) + tmp; break;
573         case 7: *b = (tmp << 3) - tmp; break;
574         case 9: *b = (tmp << 3) + tmp; break;
575     }
576 }
```

### 5.69.3.7 int tiger\_test (void)

Self-test the hash.

**Returns:**

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 734 of file tiger.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, tiger\_done(), tiger\_init(), and XMEMCMP.

```

735 {
736     #ifndef LTC_TEST
737         return CRYPT_NOP;
738     #else
739     static const struct {
740         char *msg;
741         unsigned char hash[24];
742     } testvec;
```

```

742     } tests[] = {
743     { "",
744       { 0x32, 0x93, 0xac, 0x63, 0x0c, 0x13, 0xf0, 0x24,
745         0x5f, 0x92, 0xbb, 0xb1, 0x76, 0x6e, 0x16, 0x16,
746         0x7a, 0x4e, 0x58, 0x49, 0x2d, 0xde, 0x73, 0xf3 }
747     },
748     { "abc",
749       { 0x2a, 0xab, 0x14, 0x84, 0xe8, 0xc1, 0x58, 0xf2,
750         0xbf, 0xb8, 0xc5, 0xff, 0x41, 0xb5, 0x7a, 0x52,
751         0x51, 0x29, 0x13, 0x1c, 0x95, 0x7b, 0x5f, 0x93 }
752     },
753     { "Tiger",
754       { 0xdd, 0x00, 0x23, 0x07, 0x99, 0xf5, 0x00, 0x9f,
755         0xec, 0x6d, 0xeb, 0xc8, 0x38, 0xbb, 0x6a, 0x27,
756         0xdf, 0x2b, 0x9d, 0x6f, 0x11, 0x0c, 0x79, 0x37 }
757     },
758     { "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+-",
759       { 0xf7, 0x1c, 0x85, 0x83, 0x90, 0x2a, 0xfb, 0x87,
760         0x9e, 0xdf, 0xe6, 0x10, 0xf8, 0x2c, 0x0d, 0x47,
761         0x86, 0xa3, 0xa5, 0x34, 0x50, 0x44, 0x86, 0xb5 }
762     },
763     { "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+-ABCDEFGHIJKLMNOPQRSTUVWXYZabcde
764       { 0xc5, 0x40, 0x34, 0xe5, 0xb4, 0x3e, 0xb8, 0x00,
765         0x58, 0x48, 0xa7, 0xe0, 0xae, 0x6a, 0xac, 0x76,
766         0xe4, 0xff, 0x59, 0x0a, 0xe7, 0x15, 0xfd, 0x25 }
767     },
768     };
769
770     int i;
771     unsigned char tmp[24];
772     hash_state md;
773
774     for (i = 0; i < (int)(sizeof(tests) / sizeof(tests[0])); i++) {
775         tiger_init(&md);
776         tiger_process(&md, (unsigned char *)tests[i].msg, (unsigned long)strlen(tests[i].msg));
777         tiger_done(&md, tmp);
778         if (XMEMCMP(tmp, tests[i].hash, 24) != 0) {
779             return CRYPT_FAIL_TESTVECTOR;
780         }
781     }
782     return CRYPT_OK;
783 #endif
784 }

```

Here is the call graph for this function:

## 5.69.4 Variable Documentation

### 5.69.4.1 `const ulong64 table[4 * 256]` `[static]`

Definition at line 44 of file tiger.c.

### 5.69.4.2 `const struct ltc\_hash\_descriptor tiger_desc`

**Initial value:**

```

{
    "tiger",
    1,
    24,
    64,

```

```
{ 1, 3, 6, 1, 4, 1, 11591, 12, 2,  },
9,

&tiger_init,
&tiger_process,
&tiger_done,
&tiger_test,
NULL
}
```

Definition at line 21 of file tiger.c.

Referenced by yarrow\_start().

## 5.70 hashes/whirl/whirl.c File Reference

### 5.70.1 Detailed Description

WHIRLPOOL (using their new sbbox) hash function by Tom St Denis.

Definition in file [whirl.c](#).

```
#include "tomcrypt.h"
```

```
#include "whirltab.c"
```

Include dependency graph for whirl.c:

### Defines

- #define [GB](#)(a, i, j) ((a[(i) & 7] >> (8 \* (j))) & 255)
- #define [theta\\_pi\\_gamma](#)(a, i)

### Functions

- static int [whirlpool\\_compress](#) ([hash\\_state](#) \*md, unsigned char \*buf)
- int [whirlpool\\_init](#) ([hash\\_state](#) \*md)  
*Initialize the hash state.*
- int [whirlpool\\_done](#) ([hash\\_state](#) \*md, unsigned char \*out)  
*Terminate the hash to get the digest.*
- int [whirlpool\\_test](#) (void)  
*Self-test the hash.*

### Variables

- const struct [ltc\\_hash\\_descriptor](#) [whirlpool\\_desc](#)

### 5.70.2 Define Documentation

#### 5.70.2.1 #define GB(a, i, j) ((a[(i) & 7] >> (8 \* (j))) & 255)

Definition at line 43 of file whirl.c.

#### 5.70.2.2 #define theta\_pi\_gamma(a, i)

#### Value:

```
SB0 (GB (a, i-0, 7)) ^ \
  SB1 (GB (a, i-1, 6)) ^ \
  SB2 (GB (a, i-2, 5)) ^ \
  SB3 (GB (a, i-3, 4)) ^ \
  SB4 (GB (a, i-4, 3)) ^ \
  SB5 (GB (a, i-5, 2)) ^ \
```

```

SB6(GB(a, i-6, 1)) ^
SB7(GB(a, i-7, 0))

```

Definition at line 46 of file whirl.c.

### 5.70.3 Function Documentation

#### 5.70.3.1 static int whirlpool\_compress ([hash\\_state](#) \* *md*, unsigned char \* *buf*) [static]

Definition at line 59 of file whirl.c.

References K.

Referenced by whirlpool\_done().

```

61 {
62     ulong64 K[2][8], T[3][8];
63     int x, y;
64
65     /* load the block/state */
66     for (x = 0; x < 8; x++) {
67         K[0][x] = md->whirlpool.state[x];
68
69         LOAD64H(T[0][x], buf + (8 * x));
70         T[2][x] = T[0][x];
71         T[0][x] ^= K[0][x];
72     }
73
74     /* do rounds 1..10 */
75     for (x = 0; x < 10; x += 2) {
76         /* odd round */
77         /* apply main transform to K[0] into K[1] */
78         for (y = 0; y < 8; y++) {
79             K[1][y] = theta_pi_gamma(K[0], y);
80         }
81         /* xor the constant */
82         K[1][0] ^= cont[x];
83
84         /* apply main transform to T[0] into T[1] */
85         for (y = 0; y < 8; y++) {
86             T[1][y] = theta_pi_gamma(T[0], y) ^ K[1][y];
87         }
88
89         /* even round */
90         /* apply main transform to K[1] into K[0] */
91         for (y = 0; y < 8; y++) {
92             K[0][y] = theta_pi_gamma(K[1], y);
93         }
94         /* xor the constant */
95         K[0][0] ^= cont[x+1];
96
97         /* apply main transform to T[1] into T[0] */
98         for (y = 0; y < 8; y++) {
99             T[0][y] = theta_pi_gamma(T[1], y) ^ K[0][y];
100        }
101    }
102
103    /* store state */
104    for (x = 0; x < 8; x++) {
105        md->whirlpool.state[x] ^= T[0][x] ^ T[2][x];
106    }
107
108    return CRYPT_OK;
109 }

```

**5.70.3.2** `int whirlpool_done (hash_state * md, unsigned char * out)`

Terminate the hash to get the digest.

**Parameters:**

- md* The hash state
- out* [out] The destination of the hash (64 bytes)

**Returns:**

CRYPT\_OK if successful

Definition at line 150 of file whirl.c.

References CRYPT\_INVALID\_ARG, LTC\_ARGCHK, and whirlpool\_compress().

Referenced by whirlpool\_test().

```

151 {
152     int i;
153
154     LTC_ARGCHK(md != NULL);
155     LTC_ARGCHK(out != NULL);
156
157     if (md->whirlpool.curlen >= sizeof(md->whirlpool.buf)) {
158         return CRYPT_INVALID_ARG;
159     }
160
161     /* increase the length of the message */
162     md->whirlpool.length += md->whirlpool.curlen * 8;
163
164     /* append the '1' bit */
165     md->whirlpool.buf[md->whirlpool.curlen++] = (unsigned char)0x80;
166
167     /* if the length is currently above 32 bytes we append zeros
168      * then compress. Then we can fall back to padding zeros and length
169      * encoding like normal.
170      */
171     if (md->whirlpool.curlen > 32) {
172         while (md->whirlpool.curlen < 64) {
173             md->whirlpool.buf[md->whirlpool.curlen++] = (unsigned char)0;
174         }
175         whirlpool_compress(md, md->whirlpool.buf);
176         md->whirlpool.curlen = 0;
177     }
178
179     /* pad upto 56 bytes of zeroes (should be 32 but we only support 64-bit lengths) */
180     while (md->whirlpool.curlen < 56) {
181         md->whirlpool.buf[md->whirlpool.curlen++] = (unsigned char)0;
182     }
183
184     /* store length */
185     STORE64H(md->whirlpool.length, md->whirlpool.buf+56);
186     whirlpool_compress(md, md->whirlpool.buf);
187
188     /* copy output */
189     for (i = 0; i < 8; i++) {
190         STORE64H(md->whirlpool.state[i], out+(8*i));
191     }
192 #ifdef LTC_CLEAN_STACK
193     zeromem(md, sizeof(*md));
194 #endif
195     return CRYPT_OK;
196 }

```

Here is the call graph for this function:

### 5.70.3.3 int whirlpool\_init (hash\_state \* md)

Initialize the hash state.

#### Parameters:

*md* The hash state you wish to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 128 of file whirl.c.

References CRYPT\_OK, LTC\_ARGCHK, and zeromem().

Referenced by whirlpool\_test().

```

129 {
130     LTC_ARGCHK(md != NULL);
131     zeromem(&md->whirlpool, sizeof(md->whirlpool));
132     return CRYPT_OK;
133 }
```

Here is the call graph for this function:

### 5.70.3.4 int whirlpool\_test (void)

Self-test the hash.

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-tests have been disabled

Definition at line 202 of file whirl.c.

References CRYPT\_FAIL\_TESTVECTOR, CRYPT\_NOP, len, whirlpool\_done(), whirlpool\_init(), and XMCMCMP.

```

203 {
204     #ifndef LTC_TEST
205         return CRYPT_NOP;
206     #else
207         static const struct {
208             int len;
209             unsigned char msg[128], hash[64];
210         } tests[] = {
211
212             /* NULL Message */
213             {
214                 0,
215                 { 0x00 },
216                 { 0x19, 0xFA, 0x61, 0xD7, 0x55, 0x22, 0xA4, 0x66, 0x9B, 0x44, 0xE3, 0x9C, 0x1D, 0x2E, 0x17, 0x26,
217                   0xC5, 0x30, 0x23, 0x21, 0x30, 0xD4, 0x07, 0xF8, 0x9A, 0xFE, 0xE0, 0x96, 0x49, 0x97, 0xF7, 0xA7,
218                   0x3E, 0x83, 0xBE, 0x69, 0x8B, 0x28, 0x8F, 0xEB, 0xCF, 0x88, 0xE3, 0xE0, 0x3C, 0x4F, 0x07, 0x57,
219                   0xEA, 0x89, 0x64, 0xE5, 0x9B, 0x63, 0xD9, 0x37, 0x08, 0xB1, 0x38, 0xCC, 0x42, 0xA6, 0x6E, 0xB3 },
220             },
221
222             /* 448-bits of 0 bits */
223             {
224
225
```



```

226     56,
227     { 0x00 },
228     { 0x0B, 0x3F, 0x53, 0x78, 0xEB, 0xED, 0x2B, 0xF4, 0xD7, 0xBE, 0x3C, 0xFD, 0x81, 0x8C, 0x1B, 0x03,
229       0xB6, 0xBB, 0x03, 0xD3, 0x46, 0x94, 0x8B, 0x04, 0xF4, 0xF4, 0x0C, 0x72, 0x6F, 0x07, 0x58, 0x70,
230       0x2A, 0x0F, 0x1E, 0x22, 0x58, 0x80, 0xE3, 0x8D, 0xD5, 0xF6, 0xED, 0x6D, 0xE9, 0xB1, 0xE9, 0x61,
231       0xE4, 0x9F, 0xC1, 0x31, 0x8D, 0x7C, 0xB7, 0x48, 0x22, 0xF3, 0xD0, 0xE2, 0xE9, 0xA7, 0xE7, 0xB0 }
232 },
233
234     /* 520-bits of 0 bits */
235 {
236     65,
237     { 0x00 },
238     { 0x85, 0xE1, 0x24, 0xC4, 0x41, 0x5B, 0xCF, 0x43, 0x19, 0x54, 0x3E, 0x3A, 0x63, 0xFF, 0x57, 0x1D,
239       0x09, 0x35, 0x4C, 0xEE, 0xBE, 0xE1, 0xE3, 0x25, 0x30, 0x8C, 0x90, 0x69, 0xF4, 0x3E, 0x2A, 0xE4,
240       0xD0, 0xE5, 0x1D, 0x4E, 0xB1, 0xE8, 0x64, 0x28, 0x70, 0x19, 0x4E, 0x95, 0x30, 0xD8, 0xD8, 0xAF,
241       0x65, 0x89, 0xD1, 0xBF, 0x69, 0x49, 0xDD, 0xF9, 0x0A, 0x7F, 0x12, 0x08, 0x62, 0x37, 0x95, 0xB9 }
242 },
243
244     /* 512-bits, leading set */
245 {
246     64,
247     { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
248       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
249       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
250       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
251     { 0x10, 0x3E, 0x00, 0x55, 0xA9, 0xB0, 0x90, 0xE1, 0x1C, 0x8F, 0xDD, 0xEB, 0xBA, 0x06, 0xC0, 0x5A,
252       0xCE, 0x8B, 0x64, 0xB8, 0x96, 0x12, 0x8F, 0x6E, 0xED, 0x30, 0x71, 0xFC, 0xF3, 0xDC, 0x16, 0x94,
253       0x67, 0x78, 0xE0, 0x72, 0x23, 0x23, 0x3F, 0xD1, 0x80, 0xFC, 0x40, 0xCC, 0xDB, 0x84, 0x30, 0xA6,
254       0x40, 0xE3, 0x76, 0x34, 0x27, 0x1E, 0x65, 0x5C, 0xA1, 0x67, 0x4E, 0xBF, 0xF5, 0x07, 0xF8, 0xCB }
255 },
256
257     /* 512-bits, leading set of second byte */
258 {
259     64,
260     { 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
261       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
262       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
263       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
264     { 0x35, 0x7B, 0x42, 0xEA, 0x79, 0xBC, 0x97, 0x86, 0x97, 0x5A, 0x3C, 0x44, 0x70, 0xAA, 0xB2, 0x3E,
265       0x62, 0x29, 0x79, 0x7B, 0xAD, 0xBD, 0x54, 0x36, 0x5B, 0x54, 0x96, 0xE5, 0x5D, 0x9D, 0xD7, 0x9F,
266       0xE9, 0x62, 0x4F, 0xB4, 0x22, 0x66, 0x93, 0x0A, 0x62, 0x8E, 0xD4, 0xDB, 0x08, 0xF9, 0xDD, 0x35,
267       0xEF, 0x1B, 0xE1, 0x04, 0x53, 0xFC, 0x18, 0xF4, 0x2C, 0x7F, 0x5E, 0x1F, 0x9B, 0xAE, 0x55, 0xE0 }
268 },
269
270     /* 512-bits, leading set of last byte */
271 {
272     64,
273     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
274       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
275       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
276       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x80 },
277     { 0x8B, 0x39, 0x04, 0xDD, 0x19, 0x81, 0x41, 0x26, 0xFD, 0x02, 0x74, 0xAB, 0x49, 0xC5, 0x97, 0xF6,
278       0xD7, 0x75, 0x33, 0x52, 0xA2, 0xDD, 0x91, 0xFD, 0x8F, 0x9F, 0x54, 0x05, 0x4C, 0x54, 0xBF, 0x0F,
279       0x06, 0xDB, 0x4F, 0xF7, 0x08, 0xA3, 0xA2, 0x8B, 0xC3, 0x7A, 0x92, 0x1E, 0xEE, 0x11, 0xED, 0x7B,
280       0x6A, 0x53, 0x79, 0x32, 0xCC, 0x5E, 0x94, 0xEE, 0x1E, 0xA6, 0x57, 0x60, 0x7E, 0x36, 0xC9, 0xF7 }
281 },
282
283 };
284
285 int i;
286 unsigned char tmp[64];
287 hash_state md;
288
289 for (i = 0; i < (int)(sizeof(tests)/sizeof(tests[0])); i++) {
290     whirlpool_init(&md);
291     whirlpool_process(&md, (unsigned char *)tests[i].msg, tests[i].len);
292     whirlpool_done(&md, tmp);

```

```
293     if (XMEMCMP(tmp, tests[i].hash, 64) != 0) {
294 #if 0
295         printf("\nFailed test %d\n", i);
296         for (i = 0; i < 64; ) {
297             printf("%02x ", tmp[i]);
298             if (!(++i & 15)) printf("\n");
299         }
300 #endif
301         return CRYPT_FAIL_TESTVECTOR;
302     }
303 }
304 return CRYPT_OK;
305 #endif
306 }
```

Here is the call graph for this function:

## 5.70.4 Variable Documentation

### 5.70.4.1 const struct [ltc\\_hash\\_descriptor](#) whirlpool\_desc

**Initial value:**

```
{
    "whirlpool",
    11,
    64,
    64,

    { 1, 0, 10118, 3, 0, 55 },
    6,

    &whirlpool_init,
    &whirlpool_process,
    &whirlpool_done,
    &whirlpool_test,
    NULL
}
```

Definition at line 21 of file whirl.c.

Referenced by yarrow\_start().

## 5.71 hashes/whirl/whirltab.c File Reference

### 5.71.1 Detailed Description

WHIRLPOOL tables, Tom St Denis.

Definition in file [whirltab.c](#).

This graph shows which files directly or indirectly include this file:

#### Defines

- #define [SB0](#)(x) [sbox0](#)[x]
- #define [SB1](#)(x) [sbox1](#)[x]
- #define [SB2](#)(x) [sbox2](#)[x]
- #define [SB3](#)(x) [sbox3](#)[x]
- #define [SB4](#)(x) [sbox4](#)[x]
- #define [SB5](#)(x) [sbox5](#)[x]
- #define [SB6](#)(x) [sbox6](#)[x]
- #define [SB7](#)(x) [sbox7](#)[x]

#### Variables

- static const [ulong64](#) [sbox0](#) []
- static const [ulong64](#) [sbox1](#) []
- static const [ulong64](#) [sbox2](#) []
- static const [ulong64](#) [sbox3](#) []
- static const [ulong64](#) [sbox4](#) []
- static const [ulong64](#) [sbox5](#) []
- static const [ulong64](#) [sbox6](#) []
- static const [ulong64](#) [sbox7](#) []
- static const [ulong64](#) [cont](#) []

### 5.71.2 Define Documentation

#### 5.71.2.1 #define [SB0](#)(x) [sbox0](#)[x]

Definition at line 85 of file [whirltab.c](#).

#### 5.71.2.2 #define [SB1](#)(x) [sbox1](#)[x]

Definition at line 86 of file [whirltab.c](#).

#### 5.71.2.3 #define [SB2](#)(x) [sbox2](#)[x]

Definition at line 87 of file [whirltab.c](#).

**5.71.2.4 #define SB3(x) [sbox3](#)[x]**

Definition at line 88 of file whirltab.c.

**5.71.2.5 #define SB4(x) [sbox4](#)[x]**

Definition at line 89 of file whirltab.c.

**5.71.2.6 #define SB5(x) [sbox5](#)[x]**

Definition at line 90 of file whirltab.c.

**5.71.2.7 #define SB6(x) [sbox6](#)[x]**

Definition at line 91 of file whirltab.c.

**5.71.2.8 #define SB7(x) [sbox7](#)[x]**

Definition at line 92 of file whirltab.c.

**5.71.3 Variable Documentation****5.71.3.1 const [ulong64](#) [cont](#)[] [static]**

**Initial value:**

```
{
  CONST64 (0x1823c6e887b8014f) ,
  CONST64 (0x36a6d2f5796f9152) ,
  CONST64 (0x60bc9b8ea30c7b35) ,
  CONST64 (0x1de0d7c22e4bfe57) ,
  CONST64 (0x157737e59ff04ada) ,
  CONST64 (0x58c9290ab1a06b85) ,
  CONST64 (0xbd5d10f4cb3e0567) ,
  CONST64 (0xe427418ba77d95d8) ,
  CONST64 (0xfbee7c66dd17479e) ,
  CONST64 (0xca2dbf07ad5a8333) ,
  CONST64 (0x6302aa71c81949d9) ,
}
```

Definition at line 566 of file whirltab.c.

**5.71.3.2 const [ulong64](#) [sbox0](#)[] [static]**

Definition at line 5 of file whirltab.c.

**5.71.3.3 const [ulong64](#) [sbox1](#)[] [static]**

Definition at line 95 of file whirltab.c.

**5.71.3.4** `const ulong64 sbox2[]` `[static]`

Definition at line 162 of file whirltab.c.

**5.71.3.5** `const ulong64 sbox3[]` `[static]`

Definition at line 229 of file whirltab.c.

**5.71.3.6** `const ulong64 sbox4[]` `[static]`

Definition at line 296 of file whirltab.c.

**5.71.3.7** `const ulong64 sbox5[]` `[static]`

Definition at line 363 of file whirltab.c.

**5.71.3.8** `const ulong64 sbox6[]` `[static]`

Definition at line 430 of file whirltab.c.

**5.71.3.9** `const ulong64 sbox7[]` `[static]`

Definition at line 497 of file whirltab.c.

## 5.72 headers/tomcrypt.h File Reference

```
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>
#include <limits.h>
#include <tomcrypt_custom.h>
#include <tomcrypt_cfg.h>
#include <tomcrypt_macros.h>
#include <tomcrypt_cipher.h>
#include <tomcrypt_hash.h>
#include <tomcrypt_mac.h>
#include <tomcrypt_prng.h>
#include <tomcrypt_pk.h>
#include <tomcrypt_math.h>
#include <tomcrypt_misc.h>
#include <tomcrypt_argchk.h>
#include <tomcrypt_pkcs.h>
```

Include dependency graph for tomlcrypt.h:

This graph shows which files directly or indirectly include this file:

### Defines

- #define [CRYPT](#) 0x0115
- #define [SCRYPT](#) "1.15"
- #define [MAXBLOCKSIZE](#) 128
- #define [TAB\\_SIZE](#) 32

### Enumerations

- enum {  
    [CRYPT\\_OK](#) = 0,  
    [CRYPT\\_ERROR](#),  
    [CRYPT\\_NOP](#),  
    [CRYPT\\_INVALID\\_KEYSIZE](#),  
    [CRYPT\\_INVALID\\_ROUNDS](#),  
    [CRYPT\\_FAIL\\_TESTVECTOR](#),

```
CRYPT_BUFFER_OVERFLOW,  
CRYPT_INVALID_PACKET,  
CRYPT_INVALID_PRNGSIZE,  
CRYPT_ERROR_READPRNG,  
CRYPT_INVALID_CIPHER,  
CRYPT_INVALID_HASH,  
CRYPT_INVALID_PRNG,  
CRYPT_MEM,  
CRYPT_PK_TYPE_MISMATCH,  
CRYPT_PK_NOT_PRIVATE,  
CRYPT_INVALID_ARG,  
CRYPT_FILE_NOTFOUND,  
CRYPT_PK_INVALID_TYPE,  
CRYPT_PK_INVALID_SYSTEM,  
CRYPT_PK_DUP,  
CRYPT_PK_NOT_FOUND,  
CRYPT_PK_INVALID_SIZE,  
CRYPT_INVALID_PRIME_SIZE,  
CRYPT_PK_INVALID_PADDING }
```

## 5.72.1 Define Documentation

### 5.72.1.1 #define CRYPT 0x0115

Definition at line 19 of file tomlcrypt.h.

### 5.72.1.2 #define MAXBLOCKSIZE 128

Definition at line 23 of file tomlcrypt.h.

Referenced by chc\_compress(), chc\_init(), dsa\_decrypt\_key(), dsa\_encrypt\_key(), eax\_done(), eax\_init(), eax\_test(), ecc\_decrypt\_key(), ecc\_encrypt\_key(), f8\_encrypt(), f8\_start(), find\_hash\_any(), fortuna\_reseed(), fortuna\_start(), hmac\_test(), ocb\_decrypt(), ocb\_done\_decrypt(), ocb\_encrypt(), ocb\_test(), pkcs\_5\_alg1(), pkcs\_5\_alg2(), pmac\_process(), pmac\_test(), and s\_ocr\_done().

### 5.72.1.3 #define SCRYPT "1.15"

Definition at line 20 of file tomlcrypt.h.

### 5.72.1.4 #define TAB\_SIZE 32

Definition at line 26 of file tomlcrypt.h.

Referenced by cipher\_is\_valid(), find\_cipher(), find\_cipher\_any(), find\_cipher\_id(), find\_hash(), find\_hash\_any(), find\_hash\_id(), find\_hash\_oid(), find\_prng(), hash\_is\_valid(), prng\_is\_valid(), register\_cipher(), register\_hash(), register\_prng(), unregister\_cipher(), unregister\_hash(), and unregister\_prng().

## 5.72.2 Enumeration Type Documentation

### 5.72.2.1 anonymous enum

Enumerator:

***CRYPT\_OK***  
***CRYPT\_ERROR***  
***CRYPT\_NOP***  
***CRYPT\_INVALID\_KEYSIZE***  
***CRYPT\_INVALID\_ROUNDS***  
***CRYPT\_FAIL\_TESTVECTOR***  
***CRYPT\_BUFFER\_OVERFLOW***  
***CRYPT\_INVALID\_PACKET***  
***CRYPT\_INVALID\_PRNGSIZE***  
***CRYPT\_ERROR\_READPRNG***  
***CRYPT\_INVALID\_CIPHER***  
***CRYPT\_INVALID\_HASH***  
***CRYPT\_INVALID\_PRNG***  
***CRYPT\_MEM***  
***CRYPT\_PK\_TYPE\_MISMATCH***  
***CRYPT\_PK\_NOT\_PRIVATE***  
***CRYPT\_INVALID\_ARG***  
***CRYPT\_FILE\_NOTFOUND***  
***CRYPT\_PK\_INVALID\_TYPE***  
***CRYPT\_PK\_INVALID\_SYSTEM***  
***CRYPT\_PK\_DUP***  
***CRYPT\_PK\_NOT\_FOUND***  
***CRYPT\_PK\_INVALID\_SIZE***  
***CRYPT\_INVALID\_PRIME\_SIZE***  
***CRYPT\_PK\_INVALID\_PADDING***

Definition at line 29 of file tomcrypt.h.

```

29     {
30     CRYPT_OK=0,           /* Result OK */
31     CRYPT_ERROR,         /* Generic Error */
32     CRYPT_NOP,           /* Not a failure but no operation was performed */
33
34     CRYPT_INVALID_KEYSIZE, /* Invalid key size given */
35     CRYPT_INVALID_ROUNDS,  /* Invalid number of rounds */
36     CRYPT_FAIL_TESTVECTOR, /* Algorithm failed test vectors */
37
38     CRYPT_BUFFER_OVERFLOW, /* Not enough space for output */
39     CRYPT_INVALID_PACKET,  /* Invalid input packet given */
40
41     CRYPT_INVALID_PRNGSIZE, /* Invalid number of bits for a PRNG */
42     CRYPT_ERROR_READPRNG,   /* Could not read enough from PRNG */
43
44     CRYPT_INVALID_CIPHER,   /* Invalid cipher specified */

```



```
45  CRYPT_INVALID_HASH,      /* Invalid hash specified */
46  CRYPT_INVALID_PRNG,      /* Invalid PRNG specified */
47
48  CRYPT_MEM,                /* Out of memory */
49
50  CRYPT_PK_TYPE_MISMATCH,   /* Not equivalent types of PK keys */
51  CRYPT_PK_NOT_PRIVATE,     /* Requires a private PK key */
52
53  CRYPT_INVALID_ARG,        /* Generic invalid argument */
54  CRYPT_FILE_NOTFOUND,      /* File Not Found */
55
56  CRYPT_PK_INVALID_TYPE,    /* Invalid type of PK key */
57  CRYPT_PK_INVALID_SYSTEM,  /* Invalid PK system specified */
58  CRYPT_PK_DUP,             /* Duplicate key already in key ring */
59  CRYPT_PK_NOT_FOUND,       /* Key not found in keyring */
60  CRYPT_PK_INVALID_SIZE,    /* Invalid size input for PK parameters */
61
62  CRYPT_INVALID_PRIME_SIZE, /* Invalid size of prime requested */
63  CRYPT_PK_INVALID_PADDING /* Invalid padding on input */
64 };
```

## 5.73 headers/tomcrypt\_argchk.h File Reference

```
#include <signal.h>
```

Include dependency graph for tomlcrypt\_argchk.h:

This graph shows which files directly or indirectly include this file:

### Defines

- #define [LTC\\_ARGCHK\(x\)](#) if (![\(x\)](#)) { crypt\_argchk(#x, \_\_FILE\_\_, \_\_LINE\_\_); }
- #define [LTC\\_ARGCHKVD\(x\)](#) LTC\_ARGCHK(x)

### Functions

- void [crypt\\_argchk](#) (char \*v, char \*s, int d)

#### 5.73.1 Define Documentation

##### 5.73.1.1 #define LTC\_ARGCHK(x) if (![\(x\)](#)) { crypt\_argchk(#x, \_\_FILE\_\_, \_\_LINE\_\_); }

Definition at line 9 of file tomlcrypt\_argchk.h.

Referenced by anubis\_ecb\_decrypt(), anubis\_ecb\_encrypt(), anubis\_keysize(), anubis\_setup(), base64\_decode(), base64\_encode(), blowfish\_ecb\_decrypt(), blowfish\_ecb\_encrypt(), blowfish\_keysize(), blowfish\_setup(), cast5\_ecb\_decrypt(), cast5\_ecb\_encrypt(), cast5\_keysize(), cast5\_setup(), cbc\_decrypt(), cbc\_done(), cbc\_encrypt(), cbc\_getiv(), cbc\_setiv(), cbc\_start(), ccm\_memory(), cfb\_decrypt(), cfb\_done(), cfb\_encrypt(), cfb\_getiv(), cfb\_setiv(), cfb\_start(), chc\_init(), ctr\_decrypt(), ctr\_done(), ctr\_encrypt(), ctr\_getiv(), ctr\_setiv(), ctr\_start(), der\_decode\_bit\_string(), der\_decode\_boolean(), der\_decode\_choice(), der\_decode\_ia5\_string(), der\_decode\_integer(), der\_decode\_object\_identifier(), der\_decode\_octet\_string(), der\_decode\_printable\_string(), der\_decode\_sequence\_ex(), der\_decode\_sequence\_flexi(), der\_decode\_sequence\_multi(), der\_decode\_short\_integer(), der\_decode\_utctime(), der\_encode\_bit\_string(), der\_encode\_boolean(), der\_encode\_ia5\_string(), der\_encode\_integer(), der\_encode\_object\_identifier(), der\_encode\_octet\_string(), der\_encode\_printable\_string(), der\_encode\_sequence\_ex(), der\_encode\_sequence\_multi(), der\_encode\_short\_integer(), der\_encode\_utctime(), der\_length\_bit\_string(), der\_length\_boolean(), der\_length\_ia5\_string(), der\_length\_integer(), der\_length\_object\_identifier(), der\_length\_octet\_string(), der\_length\_printable\_string(), der\_length\_sequence(), der\_length\_short\_integer(), der\_length\_utctime(), des3\_ecb\_decrypt(), des3\_ecb\_encrypt(), des3\_keysize(), des3\_setup(), des\_ecb\_decrypt(), des\_ecb\_encrypt(), des\_keysize(), des\_setup(), dsa\_decrypt\_key(), dsa\_encrypt\_key(), dsa\_export(), dsa\_import(), dsa\_make\_key(), dsa\_shared\_secret(), dsa\_sign\_hash(), dsa\_sign\_hash\_raw(), dsa\_verify\_hash\_raw(), dsa\_verify\_key(), eax\_addheader(), eax\_decrypt(), eax\_decrypt\_verify\_memory(), eax\_done(), eax\_encrypt(), eax\_encrypt\_authenticate\_memory(), eax\_init(), ECB\_DEC(), ecb\_decrypt(), ecb\_done(), ECB\_ENC(), ecb\_encrypt(), ECB\_KS(), ecb\_start(), ecc\_ansi\_x963\_export(), ecc\_ansi\_x963\_import(), ecc\_decrypt\_key(), ecc\_encrypt\_key(), ecc\_export(), ecc\_get\_size(), ecc\_import(), ecc\_make\_key(), ecc\_shared\_secret(), ecc\_sign\_hash(), ecc\_verify\_hash(), f8\_decrypt(), f8\_done(), f8\_encrypt(), f8\_getiv(), f8\_setiv(), f8\_start(), f9\_done(), f9\_file(), f9\_init(), f9\_memory\_multi(), f9\_process(), find\_cipher(), find\_cipher\_any(), find\_hash(), find\_hash\_any(), find\_hash\_oid(), find\_prng(), fortuna\_add\_entropy(), fortuna\_done(), fortuna\_export(), fortuna\_import(), fortuna\_read(), fortuna\_start(), gcm\_add\_aad(), gcm\_add\_iv(), gcm\_done(), gcm\_init(), gcm\_process(), gcm\_reset(), hash\_file(), hash\_filehandle(), hash\_memory(), hash\_memory\_multi(), hmac\_done(), hmac\_file(), hmac\_init(), hmac\_memory(), hmac\_memory\_multi(), hmac\_process(), kasumi\_ecb\_decrypt(), kasumi\_ecb\_encrypt(), kasumi\_keysize(), kasumi\_setup(), khazad\_ecb\_decrypt(),

khazad\_ecb\_encrypt(), khazad\_keysize(), khazad\_setup(), kseed\_keysize(), lrw\_decrypt(), lrw\_done(), lrw\_encrypt(), lrw\_getiv(), lrw\_process(), lrw\_setiv(), lrw\_start(), ltc\_ecc\_map(), ltc\_ecc\_mulmod(), ltc\_ecc\_projective\_add\_point(), ltc\_ecc\_projective\_dbl\_point(), md2\_done(), md2\_init(), md2\_process(), md4\_done(), md4\_init(), md5\_done(), md5\_init(), noekeon\_ecb\_decrypt(), noekeon\_ecb\_encrypt(), noekeon\_keysize(), noekeon\_setup(), ocb\_decrypt(), ocb\_decrypt\_verify\_memory(), ocb\_done\_decrypt(), ocb\_done\_encrypt(), ocb\_encrypt(), ocb\_encrypt\_authenticate\_memory(), ocb\_init(), ofb\_decrypt(), ofb\_done(), ofb\_encrypt(), ofb\_getiv(), ofb\_setiv(), ofb\_start(), omac\_done(), omac\_file(), omac\_init(), omac\_memory(), omac\_memory\_multi(), omac\_process(), pelican\_done(), pelican\_init(), pelican\_process(), pkcs\_1\_mgf1(), pkcs\_1\_oaep\_decode(), pkcs\_1\_oaep\_encode(), pkcs\_1\_pss\_decode(), pkcs\_1\_pss\_encode(), pkcs\_5\_alg1(), pkcs\_5\_alg2(), pmac\_done(), pmac\_file(), pmac\_init(), pmac\_memory(), pmac\_memory\_multi(), pmac\_process(), rand\_prime(), rc2\_ecb\_decrypt(), rc2\_ecb\_encrypt(), rc2\_keysize(), rc2\_setup(), rc4\_add\_entropy(), rc4\_done(), rc4\_export(), rc4\_import(), rc4\_read(), rc4\_ready(), rc4\_start(), rc5\_ecb\_decrypt(), rc5\_ecb\_encrypt(), rc5\_keysize(), rc5\_setup(), rc6\_ecb\_decrypt(), rc6\_ecb\_encrypt(), rc6\_keysize(), rc6\_setup(), register\_cipher(), register\_hash(), register\_prng(), rmd128\_done(), rmd128\_init(), rmd160\_done(), rmd160\_init(), rmd256\_done(), rmd256\_init(), rmd320\_done(), rmd320\_init(), rng\_get\_bytes(), rng\_make\_prng(), rsa\_decrypt\_key\_ex(), rsa\_encrypt\_key\_ex(), rsa\_export(), rsa\_exptmod(), rsa\_import(), rsa\_make\_key(), rsa\_sign\_hash\_ex(), rsa\_verify\_hash\_ex(), s\_ocrb\_done(), safer\_128\_keysize(), safer\_64\_keysize(), safer\_ecb\_decrypt(), safer\_ecb\_encrypt(), safer\_k128\_setup(), safer\_k64\_setup(), safer\_sk128\_setup(), safer\_sk64\_setup(), saferp\_ecb\_decrypt(), saferp\_ecb\_encrypt(), saferp\_keysize(), saferp\_setup(), SETUP(), sha1\_done(), sha1\_init(), sha224\_done(), sha224\_init(), sha256\_done(), sha256\_init(), sha384\_done(), sha384\_init(), sha512\_done(), sha512\_init(), skipjack\_ecb\_decrypt(), skipjack\_ecb\_encrypt(), skipjack\_keysize(), skipjack\_setup(), sober128\_add\_entropy(), sober128\_done(), sober128\_export(), sober128\_import(), sober128\_read(), sober128\_start(), sprng\_export(), sprng\_read(), tiger\_done(), tiger\_init(), twofish\_ecb\_decrypt(), twofish\_ecb\_encrypt(), twofish\_keysize(), twofish\_setup(), unregister\_cipher(), unregister\_hash(), unregister\_prng(), whirlpool\_done(), whirlpool\_init(), xcbc\_done(), xcbc\_file(), xcbc\_init(), xcbc\_memory\_multi(), xcbc\_process(), xtea\_ecb\_decrypt(), xtea\_ecb\_encrypt(), xtea\_keysize(), xtea\_setup(), yarrow\_add\_entropy(), yarrow\_done(), yarrow\_export(), yarrow\_import(), yarrow\_read(), yarrow\_ready(), and yarrow\_start().

### 5.73.1.2 #define LTC\_ARGCHKVD(x) LTC\_ARGCHK(x)

Definition at line 10 of file tomlcrypt\_argchk.h.

Referenced by dsa\_free(), ecc\_free(), ecc\_sizes(), rsa\_free(), and zeromem().

## 5.73.2 Function Documentation

### 5.73.2.1 void crypt\_argchk (char \* v, char \* s, int d)

Definition at line 20 of file crypt\_argchk.c.

```

21 {
22     fprintf(stderr, "LTC_ARGCHK '%s' failure on line %d of file %s\n",
23               v, d, s);
24     (void)raise(SIGABRT);
25 }
```

## 5.74 headers/tomcrypt\_cfg.h File Reference

This graph shows which files directly or indirectly include this file:

### Defines

- #define [ARGTYPE 0](#)
- #define [ENDIAN\\_NEUTRAL](#)

### Functions

- LTC\_EXPORT void \*LTC\_CALL [XMALLOC](#) (size\_t n)
- LTC\_EXPORT void \*LTC\_CALL [XREALLOC](#) (void \*p, size\_t n)
- LTC\_EXPORT void \*LTC\_CALL [XCALLOC](#) (size\_t n, size\_t s)
- LTC\_EXPORT void LTC\_CALL [XFREE](#) (void \*p)
- LTC\_EXPORT void LTC\_CALL [XQSORT](#) (void \*base, size\_t nmemb, size\_t size, int(\*compar)(const void \*, const void \*))
- LTC\_EXPORT clock\_t LTC\_CALL [XCLOCK](#) (void)
- LTC\_EXPORT void \*LTC\_CALL [XMEMCPY](#) (void \*dest, const void \*src, size\_t n)
- LTC\_EXPORT int LTC\_CALL [XMEMCMP](#) (const void \*s1, const void \*s2, size\_t n)
- LTC\_EXPORT void \*LTC\_CALL [XMEMSET](#) (void \*s, int c, size\_t n)

### 5.74.1 Define Documentation

#### 5.74.1.1 #define ARGTYPE 0

Definition at line 46 of file tomcrypt\_cfg.h.

#### 5.74.1.2 #define ENDIAN\_NEUTRAL

Definition at line 126 of file tomcrypt\_cfg.h.

## 5.74.2 Function Documentation

- 5.74.2.1 LTC\_EXPORT void\* LTC\_CALL XCALLOC (size\_t *n*, size\_t *s*)
- 5.74.2.2 LTC\_EXPORT clock\_t LTC\_CALL XCLOCK (void)
- 5.74.2.3 LTC\_EXPORT void LTC\_CALL XFREE (void \* *p*)
- 5.74.2.4 LTC\_EXPORT void\* LTC\_CALL XMALLOC (size\_t *n*)
- 5.74.2.5 LTC\_EXPORT int LTC\_CALL XMEMCMP (const void \* *s1*, const void \* *s2*, size\_t *n*)
- 5.74.2.6 LTC\_EXPORT void\* LTC\_CALL XMEMCPY (void \* *dest*, const void \* *src*, size\_t *n*)
- 5.74.2.7 LTC\_EXPORT void\* LTC\_CALL XMEMSET (void \* *s*, int *c*, size\_t *n*)
- 5.74.2.8 LTC\_EXPORT void LTC\_CALL XQSORT (void \* *base*, size\_t *nmemb*, size\_t *size*, int(\*)(const void \*, const void \*) *compar*)
- 5.74.2.9 LTC\_EXPORT void\* LTC\_CALL XREALLOC (void \* *p*, size\_t *n*)

## 5.75 headers/tomcrypt\_cipher.h File Reference

This graph shows which files directly or indirectly include this file:

### Data Structures

- union [Symmetric\\_key](#)
- struct [ltc\\_cipher\\_descriptor](#)  
*cipher descriptor table, last entry has "name == NULL" to mark the end of table*

### Typedefs

- typedef [Symmetric\\_key](#) symmetric\_key

### Functions

- int [find\\_cipher](#) (const char \*name)  
*Find a registered cipher by name.*
- int [find\\_cipher\\_any](#) (const char \*name, int blocklen, int keylen)  
*Find a cipher flexibly.*
- int [find\\_cipher\\_id](#) (unsigned char ID)  
*Find a cipher by ID number.*
- int [register\\_cipher](#) (const struct [ltc\\_cipher\\_descriptor](#) \*cipher)  
*Register a cipher with the descriptor table.*
- int [unregister\\_cipher](#) (const struct [ltc\\_cipher\\_descriptor](#) \*cipher)  
*Unregister a cipher from the descriptor table.*
- int [cipher\\_is\\_valid](#) (int idx)

### Variables

- [ltc\\_cipher\\_descriptor](#) cipher\_descriptor [ ]  
*cipher descriptor table, last entry has "name == NULL" to mark the end of table*

#### 5.75.1 Typedef Documentation

##### 5.75.1.1 typedef union [Symmetric\\_key](#) symmetric\_key

#### 5.75.2 Function Documentation

##### 5.75.2.1 int cipher\_is\_valid (int idx)

Definition at line 23 of file crypt\_cipher\_is\_valid.c.

References cipher\_descriptor, CRYPT\_INVALID\_CIPHER, CRYPT\_OK, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, ltc\_cipher\_descriptor::name, and TAB\_SIZE.

Referenced by cbc\_decrypt(), cbc\_done(), cbc\_encrypt(), cbc\_start(), ccm\_memory(), cfb\_decrypt(), cfb\_done(), cfb\_encrypt(), cfb\_setiv(), cfb\_start(), chc\_init(), chc\_register(), ctr\_done(), ctr\_encrypt(), ctr\_setiv(), ctr\_start(), eax\_init(), ecb\_decrypt(), ecb\_done(), ecb\_encrypt(), ecb\_start(), f8\_done(), f8\_encrypt(), f8\_setiv(), f8\_start(), f9\_done(), f9\_init(), f9\_memory(), f9\_process(), gcm\_add\_aad(), gcm\_add\_iv(), gcm\_done(), gcm\_init(), gcm\_memory(), gcm\_process(), lrw\_decrypt(), lrw\_done(), lrw\_encrypt(), lrw\_setiv(), lrw\_start(), ocb\_decrypt(), ocb\_encrypt(), ocb\_init(), ofb\_done(), ofb\_encrypt(), ofb\_setiv(), ofb\_start(), omac\_done(), omac\_init(), omac\_memory(), omac\_process(), pmac\_done(), pmac\_init(), pmac\_process(), s\_ocrb\_done(), xcbc\_done(), xcbc\_init(), xcbc\_memory(), xcbc\_process(), yarrow\_ready(), and yarrow\_start().

```

24 {
25     LTC_MUTEX_LOCK(&ltc_cipher_mutex);
26     if (idx < 0 || idx >= TAB_SIZE || cipher_descriptor[idx].name == NULL) {
27         LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
28         return CRYPT_INVALID_CIPHER;
29     }
30     LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
31     return CRYPT_OK;
32 }
```

### 5.75.2.2 int find\_cipher (const char \* name)

Find a registered cipher by name.

#### Parameters:

**name** The name of the cipher to look for

#### Returns:

>= 0 if found, -1 if not present

Definition at line 23 of file crypt\_find\_cipher.c.

References cipher\_descriptor, LTC\_ARGCHK, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, and TAB\_SIZE.

Referenced by ccm\_test(), ctr\_test(), eax\_test(), f8\_test\_mode(), f9\_test(), find\_cipher\_any(), gcm\_test(), lrw\_test(), ocb\_test(), omac\_test(), pmac\_test(), and xcbc\_test().

```

24 {
25     int x;
26     LTC_ARGCHK(name != NULL);
27     LTC_MUTEX_LOCK(&ltc_cipher_mutex);
28     for (x = 0; x < TAB_SIZE; x++) {
29         if (cipher_descriptor[x].name != NULL && !strcmp(cipher_descriptor[x].name, name)) {
30             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
31             return x;
32         }
33     }
34     LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
35     return -1;
36 }
```

### 5.75.2.3 int find\_cipher\_any (const char \* *name*, int *blocklen*, int *keylen*)

Find a cipher flexibly.

First by name then if not present by block and key size

#### Parameters:

***name*** The name of the cipher desired

***blocklen*** The minimum length of the block cipher desired (octets)

***keylen*** The minimum length of the key size desired (octets)

#### Returns:

>= 0 if found, -1 if not present

Definition at line 25 of file crypt\_find\_cipher\_any.c.

References cipher\_descriptor, find\_cipher(), LTC\_ARGCHK, LTC\_MUTEX\_LOCK, and TAB\_SIZE.

```

26 {
27     int x;
28
29     LTC_ARGCHK(name != NULL);
30
31     x = find_cipher(name);
32     if (x != -1) return x;
33
34     LTC_MUTEX_LOCK(&ltc_cipher_mutex);
35     for (x = 0; x < TAB_SIZE; x++) {
36         if (cipher_descriptor[x].name == NULL) {
37             continue;
38         }
39         if (blocklen <= (int)cipher_descriptor[x].block_length && keylen <= (int)cipher_descriptor[x].key_length) {
40             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
41             return x;
42         }
43     }
44     LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
45     return -1;
46 }
```

Here is the call graph for this function:

### 5.75.2.4 int find\_cipher\_id (unsigned char *ID*)

Find a cipher by ID number.

#### Parameters:

***ID*** The ID (not same as index) of the cipher to find

#### Returns:

>= 0 if found, -1 if not present

Definition at line 23 of file crypt\_find\_cipher\_id.c.

References cipher\_descriptor, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, ltc\_cipher\_descriptor::name, and TAB\_SIZE.



```

24 {
25     int x;
26     LTC_MUTEX_LOCK(&ltc_cipher_mutex);
27     for (x = 0; x < TAB_SIZE; x++) {
28         if (cipher_descriptor[x].ID == ID) {
29             x = (cipher_descriptor[x].name == NULL) ? -1 : x;
30             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
31             return x;
32         }
33     }
34     LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
35     return -1;
36 }

```

#### 5.75.2.5 int register\_cipher (const struct ltc\_cipher\_descriptor \* cipher)

Register a cipher with the descriptor table.

##### Parameters:

*cipher* The cipher you wish to register

##### Returns:

value  $\geq 0$  if successfully added (or already present), -1 if unsuccessful

Definition at line 23 of file crypt\_register\_cipher.c.

References cipher\_descriptor, ltc\_cipher\_descriptor::ID, LTC\_ARGCHK, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, ltc\_prng\_descriptor::name, and TAB\_SIZE.

Referenced by crypt\_fsa(), and yarrow\_start().

```

24 {
25     int x;
26
27     LTC_ARGCHK(cipher != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_cipher_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (cipher_descriptor[x].name != NULL && cipher_descriptor[x].ID == cipher->ID) {
33             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
34             return x;
35         }
36     }
37
38     /* find a blank spot */
39     for (x = 0; x < TAB_SIZE; x++) {
40         if (cipher_descriptor[x].name == NULL) {
41             XMEMCPY(&cipher_descriptor[x], cipher, sizeof(struct ltc_cipher_descriptor));
42             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
43             return x;
44         }
45     }
46
47     /* no spot */
48     LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
49     return -1;
50 }

```

### 5.75.2.6 `int unregister_cipher (const struct ltc_cipher_descriptor * cipher)`

Unregister a cipher from the descriptor table.

#### Parameters:

*cipher* The cipher descriptor to remove

#### Returns:

CRYPT\_OK on success

Definition at line 23 of file `crypt_unregister_cipher.c`.

References `cipher_descriptor`, `CRYPT_OK`, `ltc_cipher_descriptor::ID`, `LTC_ARGCHK`, `LTC_MUTEX_LOCK`, `LTC_MUTEX_UNLOCK`, `ltc_prng_descriptor::name`, `TAB_SIZE`, and `XMEMCMP`.

```

24 {
25     int x;
26
27     LTC_ARGCHK(cipher != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_cipher_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (XMEMCMP(&cipher_descriptor[x], cipher, sizeof(struct ltc_cipher_descriptor)) == 0) {
33             cipher_descriptor[x].name = NULL;
34             cipher_descriptor[x].ID = 255;
35             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
36             return CRYPT_OK;
37         }
38     }
39     LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
40     return CRYPT_ERROR;
41 }
```

## 5.75.3 Variable Documentation

### 5.75.3.1 `struct ltc_cipher_descriptor cipher_descriptor[]`

cipher descriptor table, last entry has "name == NULL" to mark the end of table

Referenced by `cbc_decrypt()`, `cbc_done()`, `cbc_encrypt()`, `cbc_start()`, `ccm_memory()`, `ccm_test()`, `cfb_decrypt()`, `cfb_done()`, `cfb_encrypt()`, `cfb_setiv()`, `cfb_start()`, `chc_compress()`, `chc_init()`, `chc_register()`, `cipher_is_valid()`, `ctr_done()`, `ctr_encrypt()`, `ctr_setiv()`, `ctr_start()`, `eax_init()`, `ecb_decrypt()`, `ecb_done()`, `ecb_encrypt()`, `ecb_start()`, `f8_done()`, `f8_encrypt()`, `f8_setiv()`, `f8_start()`, `f9_done()`, `f9_init()`, `f9_memory()`, `f9_process()`, `find_cipher()`, `find_cipher_any()`, `find_cipher_id()`, `gcm_init()`, `gcm_memory()`, `lrw_decrypt()`, `lrw_done()`, `lrw_encrypt()`, `lrw_setiv()`, `lrw_start()`, `ocb_decrypt()`, `ocb_encrypt()`, `ocb_init()`, `ofb_done()`, `ofb_encrypt()`, `ofb_setiv()`, `ofb_start()`, `omac_done()`, `omac_init()`, `omac_memory()`, `omac_process()`, `pmac_init()`, `pmac_process()`, `register_cipher()`, `s_ocb_done()`, `unregister_cipher()`, `xcbc_done()`, `xcbc_init()`, `xcbc_memory()`, `xcbc_process()`, `yarrow_ready()`, and `yarrow_test()`.

## 5.76 headers/tomcrypt\_custom.h File Reference

This graph shows which files directly or indirectly include this file:

### Defines

- #define [XMALLOC](#) malloc
- #define [XREALLOC](#) realloc
- #define [XCALLOC](#) calloc
- #define [XFREE](#) free
- #define [XMEMSET](#) memset
- #define [XMEMCPY](#) memcpy
- #define [XMEMCMP](#) memcmp
- #define [XCLOCK](#) clock
- #define [XCLOCKS\\_PER\\_SEC](#) CLOCKS\_PER\_SEC
- #define [XQSORT](#) qsort
- #define [LTC\\_TEST](#)
- #define [BLOWFISH](#)
- #define [RC2](#)
- #define [RC5](#)
- #define [RC6](#)
- #define [SAFERP](#)
- #define [RIJNDAEL](#)
- #define [XTEA](#)
- #define [TWOFISH](#)
- #define [TWOFISH\\_TABLES](#)
- #define [DES](#)
- #define [CAST5](#)
- #define [NOEKEON](#)
- #define [SKIPJACK](#)
- #define [SAFER](#)
- #define [KHAZAD](#)
- #define [ANUBIS](#)
- #define [ANUBIS\\_TWEAK](#)
- #define [KSEED](#)
- #define [LTC\\_KASUMI](#)
- #define [LTC\\_CFB\\_MODE](#)
- #define [LTC\\_OFB\\_MODE](#)
- #define [LTC\\_ECB\\_MODE](#)
- #define [LTC\\_CBC\\_MODE](#)
- #define [LTC\\_CTR\\_MODE](#)
- #define [LTC\\_F8\\_MODE](#)
- #define [LTC\\_LRW\\_MODE](#)
- #define [LRW\\_TABLES](#)
- #define [CHC\\_HASH](#)
- #define [WHIRLPOOL](#)
- #define [SHA512](#)
- #define [SHA384](#)
- #define [SHA256](#)

- #define [SHA224](#)
- #define [TIGER](#)
- #define [SHA1](#)
- #define [MD5](#)
- #define [MD4](#)
- #define [MD2](#)
- #define [RIPEMD128](#)
- #define [RIPEMD160](#)
- #define [RIPEMD256](#)
- #define [RIPEMD320](#)
- #define [LTC\\_HMAC](#)
- #define [LTC\\_OMAC](#)
- #define [LTC\\_PMAC](#)
- #define [LTC\\_XCBC](#)
- #define [LTC\\_F9\\_MODE](#)
- #define [PELICAN](#)
- #define [EAX\\_MODE](#)
- #define [OCB\\_MODE](#)
- #define [CCM\\_MODE](#)
- #define [GCM\\_MODE](#)
- #define [GCM\\_TABLES](#)
- #define [BASE64](#)
- #define [YARROW](#)
- #define [YARROW\\_AES](#) 0
- #define [SPRNG](#)
- #define [RC4](#)
- #define [FORTUNA](#)
- #define [FORTUNA\\_WD](#) 10
- #define [FORTUNA\\_POOLS](#) 32
- #define [SOBER128](#)
- #define [DEVRANDOM](#)
- #define [TRY\\_URANDOM\\_FIRST](#)
- #define [MRSA](#)
- #define [MDSA](#)
- #define [MECC](#)
- #define [PKCS\\_1](#)
- #define [PKCS\\_5](#)
- #define [LTC\\_DER](#)
- #define [ECC112](#)
- #define [ECC128](#)
- #define [ECC160](#)
- #define [ECC192](#)
- #define [ECC224](#)
- #define [ECC256](#)
- #define [ECC384](#)
- #define [ECC521](#)
- #define [MPI](#)
- #define [PKCS\\_1](#)
- #define [LTC\\_MUTEX\\_GLOBAL](#)(x)
- #define [LTC\\_MUTEX\\_PROTO](#)(x)

- #define [LTC\\_MUTEX\\_TYPE](#)(x)
- #define [LTC\\_MUTEX\\_INIT](#)(x)
- #define [LTC\\_MUTEX\\_LOCK](#)(x)
- #define [LTC\\_MUTEX\\_UNLOCK](#)(x)

## 5.76.1 Define Documentation

### 5.76.1.1 #define ANUBIS

Definition at line 125 of file tomlcrypt\_custom.h.

### 5.76.1.2 #define ANUBIS\_TWEAK

Definition at line 126 of file tomlcrypt\_custom.h.

### 5.76.1.3 #define BASE64

Definition at line 215 of file tomlcrypt\_custom.h.

### 5.76.1.4 #define BLOWFISH

Definition at line 101 of file tomlcrypt\_custom.h.

### 5.76.1.5 #define CAST5

Definition at line 120 of file tomlcrypt\_custom.h.

### 5.76.1.6 #define CCM\_MODE

Definition at line 199 of file tomlcrypt\_custom.h.

### 5.76.1.7 #define CHC\_HASH

Definition at line 159 of file tomlcrypt\_custom.h.

### 5.76.1.8 #define DES

Definition at line 119 of file tomlcrypt\_custom.h.

### 5.76.1.9 #define DEVRANDOM

Definition at line 247 of file tomlcrypt\_custom.h.

### 5.76.1.10 #define EAX\_MODE

Definition at line 193 of file tomlcrypt\_custom.h.

**5.76.1.11 #define ECC112**

Definition at line 307 of file tomcrypt\_custom.h.

**5.76.1.12 #define ECC128**

Definition at line 308 of file tomcrypt\_custom.h.

**5.76.1.13 #define ECC160**

Definition at line 309 of file tomcrypt\_custom.h.

**5.76.1.14 #define ECC192**

Definition at line 310 of file tomcrypt\_custom.h.

**5.76.1.15 #define ECC224**

Definition at line 311 of file tomcrypt\_custom.h.

**5.76.1.16 #define ECC256**

Definition at line 312 of file tomcrypt\_custom.h.

**5.76.1.17 #define ECC384**

Definition at line 313 of file tomcrypt\_custom.h.

**5.76.1.18 #define ECC521**

Definition at line 314 of file tomcrypt\_custom.h.

**5.76.1.19 #define FORTUNA**

Definition at line 237 of file tomcrypt\_custom.h.

**5.76.1.20 #define FORTUNA\_POOLS 32**

Definition at line 241 of file tomcrypt\_custom.h.

Referenced by fortuna\_add\_entropy(), fortuna\_done(), fortuna\_export(), fortuna\_import(), fortuna\_reseed(), and fortuna\_start().

**5.76.1.21 #define FORTUNA\_WD 10**

Definition at line 239 of file tomcrypt\_custom.h.

Referenced by fortuna\_read().

**5.76.1.22 #define GCM\_MODE**

Definition at line 200 of file tomlcrypt\_custom.h.

**5.76.1.23 #define GCM\_TABLES**

Definition at line 204 of file tomlcrypt\_custom.h.

**5.76.1.24 #define KHAZAD**

Definition at line 124 of file tomlcrypt\_custom.h.

**5.76.1.25 #define KSEED**

Definition at line 127 of file tomlcrypt\_custom.h.

**5.76.1.26 #define LRW\_TABLES**

Definition at line 151 of file tomlcrypt\_custom.h.

**5.76.1.27 #define LTC\_CBC\_MODE**

Definition at line 139 of file tomlcrypt\_custom.h.

**5.76.1.28 #define LTC\_CFB\_MODE**

Definition at line 136 of file tomlcrypt\_custom.h.

**5.76.1.29 #define LTC\_CTR\_MODE**

Definition at line 140 of file tomlcrypt\_custom.h.

**5.76.1.30 #define LTC\_DER**

Definition at line 298 of file tomlcrypt\_custom.h.

**5.76.1.31 #define LTC\_ECB\_MODE**

Definition at line 138 of file tomlcrypt\_custom.h.

**5.76.1.32 #define LTC\_F8\_MODE**

Definition at line 143 of file tomlcrypt\_custom.h.

**5.76.1.33 #define LTC\_F9\_MODE**

Definition at line 184 of file tomcrypt\_custom.h.

**5.76.1.34 #define LTC\_HMAC**

Definition at line 180 of file tomcrypt\_custom.h.

**5.76.1.35 #define LTC\_KASUMI**

Definition at line 128 of file tomcrypt\_custom.h.

**5.76.1.36 #define LTC\_LRW\_MODE**

Definition at line 146 of file tomcrypt\_custom.h.

**5.76.1.37 #define LTC\_MUTEX\_GLOBAL(x)**

Definition at line 350 of file tomcrypt\_custom.h.

**5.76.1.38 #define LTC\_MUTEX\_INIT(x)**

Definition at line 353 of file tomcrypt\_custom.h.

Referenced by yarrow\_start().

**5.76.1.39 #define LTC\_MUTEX\_LOCK(x)**

Definition at line 354 of file tomcrypt\_custom.h.

Referenced by cipher\_is\_valid(), find\_cipher(), find\_cipher\_any(), find\_cipher\_id(), find\_hash(), find\_hash\_any(), find\_hash\_id(), find\_hash\_oid(), find\_prng(), fortuna\_add\_entropy(), fortuna\_done(), fortuna\_export(), fortuna\_read(), hash\_is\_valid(), prng\_is\_valid(), register\_cipher(), register\_hash(), register\_prng(), unregister\_cipher(), unregister\_hash(), unregister\_prng(), yarrow\_add\_entropy(), yarrow\_done(), yarrow\_export(), yarrow\_import(), yarrow\_read(), and yarrow\_ready().

**5.76.1.40 #define LTC\_MUTEX\_PROTO(x)**

Definition at line 351 of file tomcrypt\_custom.h.

**5.76.1.41 #define LTC\_MUTEX\_TYPE(x)**

Definition at line 352 of file tomcrypt\_custom.h.

**5.76.1.42 #define LTC\_MUTEX\_UNLOCK(x)**

Definition at line 355 of file tomcrypt\_custom.h.



Referenced by `cipher_is_valid()`, `find_cipher()`, `find_cipher_id()`, `find_hash()`, `find_hash_id()`, `find_hash_oid()`, `find_prng()`, `fortuna_add_entropy()`, `fortuna_done()`, `fortuna_export()`, `fortuna_read()`, `hash_is_valid()`, `prng_is_valid()`, `register_cipher()`, `register_hash()`, `register_prng()`, `unregister_cipher()`, `unregister_hash()`, `unregister_prng()`, `yarrow_add_entropy()`, `yarrow_done()`, `yarrow_export()`, `yarrow_import()`, `yarrow_read()`, and `yarrow_ready()`.

#### 5.76.1.43 `#define LTC_OFB_MODE`

Definition at line 137 of file `tomcrypt_custom.h`.

#### 5.76.1.44 `#define LTC_OMAC`

Definition at line 181 of file `tomcrypt_custom.h`.

#### 5.76.1.45 `#define LTC_PMAC`

Definition at line 182 of file `tomcrypt_custom.h`.

#### 5.76.1.46 `#define LTC_TEST`

Definition at line 80 of file `tomcrypt_custom.h`.

#### 5.76.1.47 `#define LTC_XCBC`

Definition at line 183 of file `tomcrypt_custom.h`.

#### 5.76.1.48 `#define MD2`

Definition at line 169 of file `tomcrypt_custom.h`.

#### 5.76.1.49 `#define MD4`

Definition at line 168 of file `tomcrypt_custom.h`.

#### 5.76.1.50 `#define MD5`

Definition at line 167 of file `tomcrypt_custom.h`.

#### 5.76.1.51 `#define MDSA`

Definition at line 274 of file `tomcrypt_custom.h`.

#### 5.76.1.52 `#define MECC`

Definition at line 277 of file `tomcrypt_custom.h`.

**5.76.1.53 #define MPI**

Definition at line 320 of file tomcrypt\_custom.h.

**5.76.1.54 #define MRSA**

Definition at line 268 of file tomcrypt\_custom.h.

**5.76.1.55 #define NOEKEON**

Definition at line 121 of file tomcrypt\_custom.h.

**5.76.1.56 #define OCB\_MODE**

Definition at line 198 of file tomcrypt\_custom.h.

**5.76.1.57 #define PELICAN**

Definition at line 185 of file tomcrypt\_custom.h.

**5.76.1.58 #define PKCS\_1**

Definition at line 324 of file tomcrypt\_custom.h.

**5.76.1.59 #define PKCS\_1**

Definition at line 324 of file tomcrypt\_custom.h.

**5.76.1.60 #define PKCS\_5**

Definition at line 295 of file tomcrypt\_custom.h.

**5.76.1.61 #define RC2**

Definition at line 102 of file tomcrypt\_custom.h.

**5.76.1.62 #define RC4**

Definition at line 234 of file tomcrypt\_custom.h.

**5.76.1.63 #define RC5**

Definition at line 103 of file tomcrypt\_custom.h.

**5.76.1.64 #define RC6**

Definition at line 104 of file tomlcrypt\_custom.h.

**5.76.1.65 #define RIJNDAEL**

Definition at line 106 of file tomlcrypt\_custom.h.

**5.76.1.66 #define RIPEMD128**

Definition at line 170 of file tomlcrypt\_custom.h.

**5.76.1.67 #define RIPEMD160**

Definition at line 171 of file tomlcrypt\_custom.h.

**5.76.1.68 #define RIPEMD256**

Definition at line 172 of file tomlcrypt\_custom.h.

**5.76.1.69 #define RIPEMD320**

Definition at line 173 of file tomlcrypt\_custom.h.

**5.76.1.70 #define SAFER**

Definition at line 123 of file tomlcrypt\_custom.h.

**5.76.1.71 #define SAFERP**

Definition at line 105 of file tomlcrypt\_custom.h.

**5.76.1.72 #define SHA1**

Definition at line 166 of file tomlcrypt\_custom.h.

**5.76.1.73 #define SHA224**

Definition at line 164 of file tomlcrypt\_custom.h.

**5.76.1.74 #define SHA256**

Definition at line 163 of file tomlcrypt\_custom.h.

**5.76.1.75 #define SHA384**

Definition at line 162 of file tomcrypt\_custom.h.

**5.76.1.76 #define SHA512**

Definition at line 161 of file tomcrypt\_custom.h.

**5.76.1.77 #define SKIPJACK**

Definition at line 122 of file tomcrypt\_custom.h.

**5.76.1.78 #define SOBER128**

Definition at line 244 of file tomcrypt\_custom.h.

**5.76.1.79 #define SPRNG**

Definition at line 231 of file tomcrypt\_custom.h.

**5.76.1.80 #define TIGER**

Definition at line 165 of file tomcrypt\_custom.h.

**5.76.1.81 #define TRY\_URANDOM\_FIRST**

Definition at line 249 of file tomcrypt\_custom.h.

**5.76.1.82 #define TWOFISH**

Definition at line 110 of file tomcrypt\_custom.h.

**5.76.1.83 #define TWOFISH\_TABLES**

Definition at line 112 of file tomcrypt\_custom.h.

**5.76.1.84 #define WHIRLPOOL**

Definition at line 160 of file tomcrypt\_custom.h.

**5.76.1.85 #define XCALLOC calloc**

Definition at line 12 of file tomcrypt\_custom.h.

Referenced by der\_decode\_sequence\_flexi(), der\_encode\_set(), f9\_memory(), rand\_prime(), rsa\_import(), xcbc\_init(), and xcbc\_memory().

**5.76.1.86 #define XCLOCK clock**

Definition at line 29 of file tomlcrypt\_custom.h.

**5.76.1.87 #define XCLOCKS\_PER\_SEC CLOCKS\_PER\_SEC**

Definition at line 32 of file tomlcrypt\_custom.h.

**5.76.1.88 #define XFREE free**

Definition at line 15 of file tomlcrypt\_custom.h.

Referenced by ccm\_memory(), chc\_compress(), chc\_init(), der\_decode\_sequence\_flexi(), der\_sequence\_free(), dsa\_decrypt\_key(), dsa\_encrypt\_key(), dsa\_sign\_hash\_raw(), eax\_decrypt\_verify\_memory(), eax\_done(), eax\_encrypt\_authenticate\_memory(), eax\_init(), ecc\_decrypt\_key(), ecc\_encrypt\_key(), f9\_memory(), gcm\_memory(), hash\_memory(), hmac\_done(), hmac\_init(), hmac\_memory(), ltc\_ecc\_del\_point(), ltc\_ecc\_new\_point(), ocb\_decrypt\_verify\_memory(), ocb\_done\_decrypt(), ocb\_encrypt\_authenticate\_memory(), omac\_memory(), pelican\_memory(), pkcs\_1\_mgf1(), pkcs\_1\_oaep\_decode(), pkcs\_1\_oaep\_encode(), pkcs\_1\_pss\_decode(), pkcs\_1\_pss\_encode(), pkcs\_5\_alg1(), pkcs\_5\_alg2(), pmac\_memory(), rand\_prime(), rsa\_decrypt\_key\_ex(), rsa\_sign\_hash\_ex(), rsa\_verify\_hash\_ex(), s\_ocr\_done(), and xcbr\_memory().

**5.76.1.89 #define XMALLOC malloc**

Definition at line 6 of file tomlcrypt\_custom.h.

Referenced by ccm\_memory(), chc\_compress(), chc\_init(), dsa\_decrypt\_key(), dsa\_encrypt\_key(), dsa\_make\_key(), dsa\_sign\_hash\_raw(), eax\_decrypt\_verify\_memory(), eax\_done(), eax\_encrypt\_authenticate\_memory(), eax\_init(), ecc\_decrypt\_key(), ecc\_encrypt\_key(), f9\_memory\_multi(), fortuna\_export(), gcm\_memory(), hash\_memory(), hash\_memory\_multi(), hmac\_done(), hmac\_init(), hmac\_memory(), hmac\_memory\_multi(), ltc\_ecc\_new\_point(), ocb\_decrypt\_verify\_memory(), ocb\_done\_decrypt(), ocb\_encrypt\_authenticate\_memory(), omac\_memory(), omac\_memory\_multi(), pelican\_memory(), pkcs\_1\_mgf1(), pkcs\_1\_oaep\_decode(), pkcs\_1\_oaep\_encode(), pkcs\_1\_pss\_decode(), pkcs\_1\_pss\_encode(), pkcs\_5\_alg1(), pkcs\_5\_alg2(), pmac\_memory(), pmac\_memory\_multi(), rsa\_decrypt\_key\_ex(), rsa\_sign\_hash\_ex(), rsa\_verify\_hash\_ex(), s\_ocr\_done(), and xcbr\_memory\_multi().

**5.76.1.90 #define XMEMCMP memcmp**

Definition at line 25 of file tomlcrypt\_custom.h.

Referenced by anubis\_test(), ccm\_test(), eax\_decrypt\_verify\_memory(), f8\_test\_mode(), find\_hash\_oid(), khazad\_test(), kseed\_test(), md2\_test(), md4\_test(), md5\_test(), ocb\_done\_decrypt(), qsort\_helper(), register\_hash(), register\_prng(), rmd128\_test(), rmd160\_test(), rmd256\_test(), rmd320\_test(), rsa\_verify\_hash\_ex(), safer\_k64\_test(), safer\_sk128\_test(), safer\_sk64\_test(), sha1\_test(), sha224\_test(), sha256\_test(), sha384\_test(), sha512\_test(), tiger\_test(), unregister\_cipher(), unregister\_hash(), unregister\_prng(), whirlpool\_test(), and xtea\_test().

**5.76.1.91 #define XMEMCPY memcpy**

Definition at line 22 of file tomlcrypt\_custom.h.

Referenced by `cast5_setup()`, `cbc_getiv()`, `cbc_setiv()`, `cfb_getiv()`, `chc_compress()`, `crypt_fsa()`, `ctr_getiv()`, `ctr_setiv()`, `ecc_ansi_x963_export()`, `f8_getiv()`, `fortuna_export()`, `fortuna_read()`, `gcm_add_aad()`, `gcm_gf_mult()`, `hmac_init()`, `lrw_getiv()`, `lrw_process()`, `lrw_setiv()`, `lrw_start()`, `md2_done()`, `md2_process()`, `ofb_getiv()`, `pkcs_1_oaep_decode()`, `pkcs_1_oaep_encode()`, `pkcs_1_pss_decode()`, `pkcs_1_pss_encode()`, `pkcs_5_alg2()`, `rc4_ready()`, `s_ocb_done()`, `sha224_done()`, and `sha384_done()`.

#### **5.76.1.92 #define XMEMSET memset**

Definition at line 19 of file `tomcrypt_custom.h`.

Referenced by `pkcs_1_oaep_encode()`, and `pkcs_1_pss_encode()`.

#### **5.76.1.93 #define XQSORT qsort**

Definition at line 36 of file `tomcrypt_custom.h`.

#### **5.76.1.94 #define XREALLOC realloc**

Definition at line 9 of file `tomcrypt_custom.h`.

Referenced by `der_decode_sequence_flexi()`.

#### **5.76.1.95 #define XTEA**

Definition at line 107 of file `tomcrypt_custom.h`.

#### **5.76.1.96 #define YARROW**

Definition at line 221 of file `tomcrypt_custom.h`.

#### **5.76.1.97 #define YARROW\_AES 0**

Definition at line 224 of file `tomcrypt_custom.h`.

## 5.77 headers/tomcrypt\_hash.h File Reference

This graph shows which files directly or indirectly include this file:

### Data Structures

- union [Hash\\_state](#)
- struct [ltc\\_hash\\_descriptor](#)  
*hash descriptor*

### Defines

- #define [HASH\\_PROCESS](#)(func\_name, compress\_name, state\_var, block\_size)

### Typedefs

- typedef [Hash\\_state](#) [hash\\_state](#)

### Functions

- int [find\\_hash](#) (const char \*name)  
*Find a registered hash by name.*
- int [find\\_hash\\_id](#) (unsigned char ID)  
*Find a hash by ID number.*
- int [find\\_hash\\_oid](#) (const unsigned long \*ID, unsigned long IDlen)
- int [find\\_hash\\_any](#) (const char \*name, int digestlen)  
*Find a hash flexibly.*
- int [register\\_hash](#) (const struct [ltc\\_hash\\_descriptor](#) \*hash)  
*Register a hash with the descriptor table.*
- int [unregister\\_hash](#) (const struct [ltc\\_hash\\_descriptor](#) \*hash)  
*Unregister a hash from the descriptor table.*
- int [hash\\_is\\_valid](#) (int idx)
- int [hash\\_memory](#) (int hash, const unsigned char \*in, unsigned long inlen, unsigned char \*out, unsigned long \*outlen)  
*Hash a block of memory and store the digest.*
- int [hash\\_memory\\_multi](#) (int hash, unsigned char \*out, unsigned long \*outlen, const unsigned char \*in, unsigned long inlen,...)  
*Hash multiple (non-adjacent) blocks of memory at once.*
- int [hash\\_filehandle](#) (int hash, FILE \*in, unsigned char \*out, unsigned long \*outlen)  
*Hash data from an open file handle.*
- int [hash\\_file](#) (int hash, const char \*fname, unsigned char \*out, unsigned long \*outlen)

## Variables

- [ltc\\_hash\\_descriptor](#) [hash\\_descriptor](#) []  
*hash descriptor*

### 5.77.1 Define Documentation

#### 5.77.1.1 #define HASH\_PROCESS(func\_name, compress\_name, state\_var, block\_size)

Definition at line 341 of file tomcrypt\_hash.h.

### 5.77.2 Typedef Documentation

#### 5.77.2.1 typedef union [Hash\\_state](#) [hash\\_state](#)

### 5.77.3 Function Documentation

#### 5.77.3.1 int find\_hash (const char \* *name*)

Find a registered hash by name.

##### Parameters:

*name* The name of the hash to look for

##### Returns:

>= 0 if found, -1 if not present

Definition at line 23 of file crypt\_find\_hash.c.

References [hash\\_descriptor](#), [LTC\\_ARGCHK](#), [LTC\\_MUTEX\\_LOCK](#), [LTC\\_MUTEX\\_UNLOCK](#), and [TAB\\_SIZE](#).

Referenced by [chc\\_register\(\)](#), [find\\_hash\\_any\(\)](#), and [hmac\\_test\(\)](#).

```

24 {
25     int x;
26     LTC_ARGCHK(name != NULL);
27     LTC_MUTEX_LOCK(&ltc_hash_mutex);
28     for (x = 0; x < TAB_SIZE; x++) {
29         if (hash_descriptor[x].name != NULL && strcmp(hash_descriptor[x].name, name) == 0) {
30             LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
31             return x;
32         }
33     }
34     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
35     return -1;
36 }
```

#### 5.77.3.2 int find\_hash\_any (const char \* *name*, int *digestlen*)

Find a hash flexibly.

First by name then if not present by digest size



**Parameters:**

- name** The name of the hash desired
- digestlen** The minimum length of the digest size (octets)

**Returns:**

$\geq 0$  if found, -1 if not present

Definition at line 23 of file crypt\_find\_hash\_any.c.

References find\_hash(), hash\_descriptor, LTC\_ARGCHK, LTC\_MUTEX\_LOCK, MAXBLOCKSIZE, and TAB\_SIZE.

```

24 {
25     int x, y, z;
26     LTC_ARGCHK(name != NULL);
27
28     x = find_hash(name);
29     if (x != -1) return x;
30
31     LTC_MUTEX_LOCK(&ltc_hash_mutex);
32     y = MAXBLOCKSIZE+1;
33     z = -1;
34     for (x = 0; x < TAB_SIZE; x++) {
35         if (hash_descriptor[x].name == NULL) {
36             continue;
37         }
38         if ((int)hash_descriptor[x].hashsize >= digestlen && (int)hash_descriptor[x].hashsize < y) {
39             z = x;
40             y = hash_descriptor[x].hashsize;
41         }
42     }
43     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
44     return z;
45 }
```

Here is the call graph for this function:

**5.77.3.3 int find\_hash\_id (unsigned char ID)**

Find a hash by ID number.

**Parameters:**

- ID** The ID (not same as index) of the hash to find

**Returns:**

$\geq 0$  if found, -1 if not present

Definition at line 23 of file crypt\_find\_hash\_id.c.

References hash\_descriptor, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, ltc\_cipher\_descriptor::name, and TAB\_SIZE.

```

24 {
25     int x;
26     LTC_MUTEX_LOCK(&ltc_hash_mutex);
27     for (x = 0; x < TAB_SIZE; x++) {
28         if (hash_descriptor[x].ID == ID) {
29             x = (hash_descriptor[x].name == NULL) ? -1 : x;

```

```

30         LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
31         return x;
32     }
33 }
34 LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
35 return -1;
36 }

```

#### 5.77.3.4 int find\_hash\_oid (const unsigned long \*ID, unsigned long IDlen)

Definition at line 18 of file crypt\_find\_hash\_oid.c.

References hash\_descriptor, LTC\_ARGCHK, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, ltc\_cipher\_descriptor::name, TAB\_SIZE, and XMEMCMP.

Referenced by dsa\_decrypt\_key(), and ecc\_decrypt\_key().

```

19 {
20     int x;
21     LTC_ARGCHK(ID != NULL);
22     LTC_MUTEX_LOCK(&ltc_hash_mutex);
23     for (x = 0; x < TAB_SIZE; x++) {
24         if (hash_descriptor[x].name != NULL && hash_descriptor[x].OIDlen == IDlen && !XMEMCMP(hash_desc
25             LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
26             return x;
27         }
28     }
29     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
30     return -1;
31 }

```

#### 5.77.3.5 int hash\_file (int hash, const char \*fname, unsigned char \*out, unsigned long \*outlen)

##### Parameters:

- hash** The index of the hash desired
- fname** The name of the file you wish to hash
- out** [out] The destination of the digest
- outlen** [in/out] The max size and resulting size of the message digest

##### Returns:

- CRYPT\_OK if successful

Definition at line 25 of file hash\_file.c.

References CRYPT\_ERROR, CRYPT\_FILE\_NOTFOUND, CRYPT\_NOP, CRYPT\_OK, hash\_filehandle(), hash\_is\_valid(), in, and LTC\_ARGCHK.

```

26 {
27 #ifdef LTC_NO_FILE
28     return CRYPT_NOP;
29 #else
30     FILE *in;
31     int err;
32     LTC_ARGCHK(fname != NULL);
33     LTC_ARGCHK(out != NULL);
34     LTC_ARGCHK(outlen != NULL);
35

```

```

36     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
37         return err;
38     }
39
40     in = fopen(fname, "rb");
41     if (in == NULL) {
42         return CRYPT_FILE_NOTFOUND;
43     }
44
45     err = hash_filehandle(hash, in, out, outlen);
46     if (fclose(in) != 0) {
47         return CRYPT_ERROR;
48     }
49
50     return err;
51 #endif
52 }

```

Here is the call graph for this function:

### 5.77.3.6 int hash\_filehandle (int *hash*, FILE \* *in*, unsigned char \* *out*, unsigned long \* *outlen*)

Hash data from an open file handle.

#### Parameters:

- hash*** The index of the hash you want to use
- in*** The FILE\* handle of the file you want to hash
- out*** [out] The destination of the digest
- outlen*** [in/out] The max size and resulting size of the digest

#### Returns:

CRYPT\_OK if successful

Definition at line 26 of file hash\_filehandle.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_NOP, CRYPT\_OK, ltc\_hash\_descriptor::done, hash\_descriptor, hash\_is\_valid(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, and zeromem().

Referenced by hash\_file().

```

27 {
28 #ifdef LTC_NO_FILE
29     return CRYPT_NOP;
30 #else
31     hash_state md;
32     unsigned char buf[512];
33     size_t x;
34     int err;
35
36     LTC_ARGCHK(out != NULL);
37     LTC_ARGCHK(outlen != NULL);
38     LTC_ARGCHK(in != NULL);
39
40     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
41         return err;
42     }
43
44     if (*outlen < hash_descriptor[hash].hashsize) {
45         *outlen = hash_descriptor[hash].hashsize;
46         return CRYPT_BUFFER_OVERFLOW;

```

```

47     }
48     if ((err = hash_descriptor[hash].init(&md)) != CRYPT_OK) {
49         return err;
50     }
51
52     *outlen = hash_descriptor[hash].hashsize;
53     do {
54         x = fread(buf, 1, sizeof(buf), in);
55         if ((err = hash_descriptor[hash].process(&md, buf, x)) != CRYPT_OK) {
56             return err;
57         }
58     } while (x == sizeof(buf));
59     err = hash_descriptor[hash].done(&md, out);
60
61 #ifdef LTC_CLEAN_STACK
62     zeromem(buf, sizeof(buf));
63 #endif
64     return err;
65 #endif
66 }

```

Here is the call graph for this function:

### 5.77.3.7 int hash\_is\_valid (int idx)

Definition at line 23 of file crypt\_hash\_is\_valid.c.

References CRYPT\_INVALID\_HASH, CRYPT\_OK, hash\_descriptor, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, ltc\_hash\_descriptor::name, and TAB\_SIZE.

Referenced by chc\_register(), dsa\_decrypt\_key(), dsa\_encrypt\_key(), ecc\_decrypt\_key(), ecc\_encrypt\_key(), hash\_file(), hash\_filehandle(), hash\_memory(), hash\_memory\_multi(), hmac\_done(), hmac\_file(), hmac\_init(), hmac\_memory(), hmac\_process(), pkcs\_1\_mgf1(), pkcs\_1\_oaep\_decode(), pkcs\_1\_oaep\_encode(), pkcs\_1\_pss\_decode(), pkcs\_1\_pss\_encode(), pkcs\_5\_alg1(), pkcs\_5\_alg2(), rsa\_decrypt\_key\_ex(), rsa\_encrypt\_key\_ex(), rsa\_sign\_hash\_ex(), rsa\_verify\_hash\_ex(), yarrow\_add\_entropy(), yarrow\_ready(), and yarrow\_start().

```

24 {
25     LTC_MUTEX_LOCK(&ltc_hash_mutex);
26     if (idx < 0 || idx >= TAB_SIZE || hash_descriptor[idx].name == NULL) {
27         LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
28         return CRYPT_INVALID_HASH;
29     }
30     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
31     return CRYPT_OK;
32 }

```

### 5.77.3.8 int hash\_memory (int hash, const unsigned char \*in, unsigned long inlen, unsigned char \*out, unsigned long \*outlen)

Hash a block of memory and store the digest.

#### Parameters:

- hash** The index of the hash you wish to use
- in** The data you wish to hash
- inlen** The length of the data to hash (octets)
- out** [out] Where to store the digest

*outlen* [in/out] Max size and resulting size of the digest

#### Returns:

CRYPT\_OK if successful

Definition at line 27 of file hash\_memory.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_MEM, CRYPT\_OK, ltc\_hash\_descriptor::done, hash\_descriptor, hash\_is\_valid(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, XFREE, XMALLOC, and zeromem().

Referenced by dsa\_decrypt\_key(), dsa\_encrypt\_key(), ecc\_decrypt\_key(), ecc\_encrypt\_key(), hmac\_init(), pkcs\_1\_oaep\_encode(), and pkcs\_5\_alg1().

```

28 {
29     hash_state *md;
30     int err;
31
32     LTC_ARGCHK(in      != NULL);
33     LTC_ARGCHK(out     != NULL);
34     LTC_ARGCHK(outlen  != NULL);
35
36     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
37         return err;
38     }
39
40     if (*outlen < hash_descriptor[hash].hashsize) {
41         *outlen = hash_descriptor[hash].hashsize;
42         return CRYPT_BUFFER_OVERFLOW;
43     }
44
45     md = XMALLOC(sizeof(hash_state));
46     if (md == NULL) {
47         return CRYPT_MEM;
48     }
49
50     if ((err = hash_descriptor[hash].init(md)) != CRYPT_OK) {
51         goto LBL_ERR;
52     }
53     if ((err = hash_descriptor[hash].process(md, in, inlen)) != CRYPT_OK) {
54         goto LBL_ERR;
55     }
56     err = hash_descriptor[hash].done(md, out);
57     *outlen = hash_descriptor[hash].hashsize;
58 LBL_ERR:
59 #ifdef LTC_CLEAN_STACK
60     zeromem(md, sizeof(hash_state));
61 #endif
62     XFREE(md);
63
64     return err;
65 }
```

Here is the call graph for this function:

#### 5.77.3.9 int hash\_memory\_multi (int *hash*, unsigned char \* *out*, unsigned long \* *outlen*, const unsigned char \* *in*, unsigned long *inlen*, ...)

Hash multiple (non-adjacent) blocks of memory at once.

#### Parameters:

*hash* The index of the hash you wish to use

**out** [out] Where to store the digest

**outlen** [in/out] Max size and resulting size of the digest

**in** The data you wish to hash

**inlen** The length of the data to hash (octets)

... tuples of (data,len) pairs to hash, terminated with a (NULL,x) (x=don't care)

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file hash\_memory\_multi.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_MEM, CRYPT\_OK, hash\_descriptor, hash\_is\_valid(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, and XMALLOC.

```

30 {
31     hash_state      *md;
32     int             err;
33     va_list         args;
34     const unsigned char *curptr;
35     unsigned long    curlen;
36
37     LTC_ARGCHK(in     != NULL);
38     LTC_ARGCHK(out     != NULL);
39     LTC_ARGCHK(outlen != NULL);
40
41     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
42         return err;
43     }
44
45     if (*outlen < hash_descriptor[hash].hashsize) {
46         *outlen = hash_descriptor[hash].hashsize;
47         return CRYPT_BUFFER_OVERFLOW;
48     }
49
50     md = XMALLOC(sizeof(hash_state));
51     if (md == NULL) {
52         return CRYPT_MEM;
53     }
54
55     if ((err = hash_descriptor[hash].init(md)) != CRYPT_OK) {
56         goto LBL_ERR;
57     }
58
59     va_start(args, inlen);
60     curptr = in;
61     curlen = inlen;
62     for (;;) {
63         /* process buf */
64         if ((err = hash_descriptor[hash].process(md, curptr, curlen)) != CRYPT_OK) {
65             goto LBL_ERR;
66         }
67         /* step to next */
68         curptr = va_arg(args, const unsigned char*);
69         if (curptr == NULL) {
70             break;
71         }
72         curlen = va_arg(args, unsigned long);
73     }
74     err = hash_descriptor[hash].done(md, out);
75     *outlen = hash_descriptor[hash].hashsize;
76 LBL_ERR:
77 #ifdef LTC_CLEAN_STACK
78     zeromem(md, sizeof(hash_state));

```

```

79 #endif
80     XFREE(md);
81     va_end(args);
82     return err;
83 }

```

Here is the call graph for this function:

### 5.77.3.10 int register\_hash (const struct ltc\_hash\_descriptor \* hash)

Register a hash with the descriptor table.

#### Parameters:

**hash** The hash you wish to register

#### Returns:

value  $\geq 0$  if successfully added (or already present), -1 if unsuccessful

Definition at line 23 of file crypt\_register\_hash.c.

References hash\_descriptor, LTC\_ARGCHK, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, TAB\_SIZE, and XMEMCMP.

Referenced by crypt\_fsa(), and yarrow\_start().

```

24 {
25     int x;
26
27     LTC_ARGCHK(hash != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_hash_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (XMEMCMP(&hash_descriptor[x], hash, sizeof(struct ltc_hash_descriptor)) == 0) {
33             LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
34             return x;
35         }
36     }
37
38     /* find a blank spot */
39     for (x = 0; x < TAB_SIZE; x++) {
40         if (hash_descriptor[x].name == NULL) {
41             XMEMCPY(&hash_descriptor[x], hash, sizeof(struct ltc_hash_descriptor));
42             LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
43             return x;
44         }
45     }
46
47     /* no spot */
48     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
49     return -1;
50 }

```

### 5.77.3.11 int unregister\_hash (const struct ltc\_hash\_descriptor \* hash)

Unregister a hash from the descriptor table.

#### Parameters:

**hash** The hash descriptor to remove

**Returns:**

CRYPT\_OK on success

Definition at line 23 of file crypt\_unregister\_hash.c.

References CRYPT\_OK, hash\_descriptor, LTC\_ARGCHK, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, ltc\_prng\_descriptor::name, TAB\_SIZE, and XMEMCMP.

```
24 {
25     int x;
26
27     LTC_ARGCHK(hash != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_hash_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (XMEMCMP(&hash_descriptor[x], hash, sizeof(struct ltc_hash_descriptor)) == 0) {
33             hash_descriptor[x].name = NULL;
34             LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
35             return CRYPT_OK;
36         }
37     }
38     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
39     return CRYPT_ERROR;
40 }
```

## 5.77.4 Variable Documentation

### 5.77.4.1 struct [ltc\\_hash\\_descriptor](#) hash\_descriptor[]

hash descriptor

Referenced by chc\_register(), dsa\_encrypt\_key(), ecc\_encrypt\_key(), find\_hash(), find\_hash\_any(), find\_hash\_id(), find\_hash\_oid(), hash\_filehandle(), hash\_is\_valid(), hash\_memory(), hash\_memory\_multi(), hmac\_done(), hmac\_init(), hmac\_memory(), hmac\_process(), pkcs\_1\_mgf1(), pkcs\_1\_oaep\_decode(), pkcs\_1\_oaep\_encode(), pkcs\_1\_pss\_decode(), pkcs\_1\_pss\_encode(), pkcs\_5\_alg1(), register\_hash(), rsa\_sign\_hash\_ex(), rsa\_verify\_hash\_ex(), unregister\_hash(), yarrow\_add\_entropy(), yarrow\_ready(), and yarrow\_test().



## 5.78 headers/tomcrypt\_mac.h File Reference

This graph shows which files directly or indirectly include this file:

## 5.79 headers/tomcrypt\_macros.h File Reference

This graph shows which files directly or indirectly include this file:

### Defines

- #define [CONST64](#)(n) n ## ULL
- #define [BSWAP](#)(x)
- #define [ROL](#)(x, y) ( (((unsigned long)(x)<<(unsigned long)((y)&31)) | (((unsigned long)(x)&0xFFFFFFFFFUL)>>(unsigned long)(32-((y)&31)))) & 0xFFFFFFFFFUL)
- #define [ROR](#)(x, y) ( (((((unsigned long)(x)&0xFFFFFFFFFUL)>>(unsigned long)((y)&31)) | ((unsigned long)(x)<<(unsigned long)(32-((y)&31)))) & 0xFFFFFFFFFUL)
- #define [ROLc](#)(x, y) ( (((unsigned long)(x)<<(unsigned long)((y)&31)) | (((unsigned long)(x)&0xFFFFFFFFFUL)>>(unsigned long)(32-((y)&31)))) & 0xFFFFFFFFFUL)
- #define [RORc](#)(x, y) ( (((((unsigned long)(x)&0xFFFFFFFFFUL)>>(unsigned long)((y)&31)) | ((unsigned long)(x)<<(unsigned long)(32-((y)&31)))) & 0xFFFFFFFFFUL)
- #define [ROL64](#)(x, y)
- #define [ROR64](#)(x, y)
- #define [ROL64c](#)(x, y)
- #define [ROR64c](#)(x, y)
- #define [MAX](#)(x, y) ( ((x)>(y))?(x):(y) )
- #define [MIN](#)(x, y) ( ((x)<(y))?(x):(y) )
- #define [byte](#)(x, n) (((x) >> (8 \* (n))) & 255)

### Typedefs

- typedef unsigned long long [ulong64](#)
- typedef unsigned long [ulong32](#)

### 5.79.1 Define Documentation

#### 5.79.1.1 #define BSWAP(x)

**Value:**

```
( ((x>>24)&0x000000FFUL) | ((x<<24)&0xFF000000UL) | \
  ((x>>8)&0x0000FF00UL) | ((x<<8)&0x00FF0000UL) )
```

Definition at line 230 of file tomcrypt\_macros.h.

Referenced by rc5\_setup(), and rc6\_setup().

#### 5.79.1.2 #define byte(x, n) (((x) >> (8 \* (n))) & 255)

Definition at line 419 of file tomcrypt\_macros.h.

Referenced by desfunc(), ECB\_DEC(), ECB\_ENC(), FI(), FII(), FIII(), four\_rounds(), setup\_mix(), and tiger\_round().

**5.79.1.3 #define CONST64(n) n ## ULL**

Definition at line 6 of file tomlcrypt\_macros.h.

Referenced by gcm\_add\_aad(), gcm\_done(), gcm\_process(), key\_schedule(), sha384\_init(), sha512\_done(), sha512\_init(), and tiger\_init().

**5.79.1.4 #define MAX(x, y) ( ((x)>(y))?(x):(y) )**

Definition at line 408 of file tomlcrypt\_macros.h.

**5.79.1.5 #define MIN(x, y) ( ((x)<(y))?(x):(y) )**

Definition at line 412 of file tomlcrypt\_macros.h.

Referenced by dsa\_decrypt\_key(), ecc\_decrypt\_key(), md2\_process(), and qsort\_helper().

**5.79.1.6 #define ROL(x, y) ( (((unsigned long)(x)<<(unsigned long)((y)&31)) | (((unsigned long)(x)&0xFFFFFFFFFUL)>>(unsigned long)(32-((y)&31)))) & 0xFFFFFFFFFUL)**

Definition at line 335 of file tomlcrypt\_macros.h.

Referenced by FI(), FII(), FIII(), and rc5\_ecb\_encrypt().

**5.79.1.7 #define ROL64(x, y)**

**Value:**

```
( ((x)<<((ulong64)(y)&63)) | \
  ((x)&CONST64(0xFFFFFFFFFFFFFFFF))>>((ulong64)64-((y)&63))) & CONST64(0xFFFFFFFFFFFFFFFF)
```

Definition at line 389 of file tomlcrypt\_macros.h.

**5.79.1.8 #define ROL64c(x, y)**

**Value:**

```
( ((x)<<((ulong64)(y)&63)) | \
  ((x)&CONST64(0xFFFFFFFFFFFFFFFF))>>((ulong64)64-((y)&63))) & CONST64(0xFFFFFFFFFFFFFFFF)
```

Definition at line 397 of file tomlcrypt\_macros.h.

**5.79.1.9 #define ROLc(x, y) ( (((unsigned long)(x)<<(unsigned long)((y)&31)) | (((unsigned long)(x)&0xFFFFFFFFFUL)>>(unsigned long)(32-((y)&31)))) & 0xFFFFFFFFFUL)**

Definition at line 337 of file tomlcrypt\_macros.h.

Referenced by desfunc(), twofish\_ecb\_decrypt(), and twofish\_ecb\_encrypt().

**5.79.1.10** `#define ROR(x, y) ( (((unsigned long)(x)&0xFFFFFFFFUL)>>(unsigned long)((y)&31)) | ((unsigned long)(x)<<((unsigned long)(32-((y)&31)))) & 0xFFFFFFFFUL)`

Definition at line 336 of file tomcrypt\_macros.h.

Referenced by rc5\_ecb\_decrypt().

**5.79.1.11** `#define ROR64(x, y)`

**Value:**

```
( (((x)&CONST64(0xFFFFFFFFFFFFFFFF))>>((ulong64)(y)&CONST64(63))) | \
  ((x)<<((ulong64)(64-((y)&CONST64(63))))) & CONST64(0xFFFFFFFFFFFFFFFF))
```

Definition at line 393 of file tomcrypt\_macros.h.

**5.79.1.12** `#define ROR64c(x, y)`

**Value:**

```
( (((x)&CONST64(0xFFFFFFFFFFFFFFFF))>>((ulong64)(y)&CONST64(63))) | \
  ((x)<<((ulong64)(64-((y)&CONST64(63))))) & CONST64(0xFFFFFFFFFFFFFFFF))
```

Definition at line 401 of file tomcrypt\_macros.h.

**5.79.1.13** `#define RORc(x, y) ( (((unsigned long)(x)&0xFFFFFFFFUL)>>(unsigned long)((y)&31)) | ((unsigned long)(x)<<((unsigned long)(32-((y)&31)))) & 0xFFFFFFFFUL)`

Definition at line 338 of file tomcrypt\_macros.h.

Referenced by desfunc(), twofish\_ecb\_decrypt(), and twofish\_ecb\_encrypt().

## 5.79.2 Typedef Documentation

**5.79.2.1** `typedef unsigned long ulong32`

Definition at line 16 of file tomcrypt\_macros.h.

**5.79.2.2** `typedef unsigned long long ulong64`

Definition at line 7 of file tomcrypt\_macros.h.

## 5.80 headers/tomcrypt\_math.h File Reference

This graph shows which files directly or indirectly include this file:

### Data Structures

- struct [ltc\\_math\\_descriptor](#)  
*math descriptor*

### Defines

- #define [LTC\\_MP\\_LT](#) -1  
*math functions*
- #define [LTC\\_MP\\_EQ](#) 0
- #define [LTC\\_MP\\_GT](#) 1
- #define [LTC\\_MP\\_NO](#) 0
- #define [LTC\\_MP\\_YES](#) 1

### Typedefs

- typedef void [ecc\\_point](#)
- typedef void [rsa\\_key](#)

### Functions

- int [ltc\\_init\\_multi](#) (void \*\*a,...)
- void [ltc\\_deinit\\_multi](#) (void \*a,...)

### Variables

- [ltc\\_math\\_descriptor](#) [ltc\\_mp](#)

#### 5.80.1 Define Documentation

##### 5.80.1.1 #define LTC\_MP\_EQ 0

Definition at line 4 of file `tomcrypt_math.h`.

Referenced by `dsa_sign_hash_raw()`, `dsa_verify_hash_raw()`, `dsa_verify_key()`, `is_point()`, and `ltc_ecc_projective_add_point()`.

##### 5.80.1.2 #define LTC\_MP\_GT 1

Definition at line 5 of file `tomcrypt_math.h`.

Referenced by `der_encode_integer()`, `dsa_sign_hash_raw()`, and `dsa_verify_key()`.

### 5.80.1.3 `#define LTC_MP_LT -1`

math functions

Definition at line 3 of file tomcrypt\_math.h.

Referenced by `der_encode_integer()`, `der_length_integer()`, `dsa_verify_hash_raw()`, `dsa_verify_key()`, `ecc_verify_hash()`, `is_point()`, `ltc_ecc_projective_add_point()`, `ltc_ecc_projective_dbl_point()`, and `rsa_exptmod()`.

### 5.80.1.4 `#define LTC_MP_NO 0`

Definition at line 7 of file tomcrypt\_math.h.

### 5.80.1.5 `#define LTC_MP_YES 1`

Definition at line 8 of file tomcrypt\_math.h.

Referenced by `der_encode_integer()`, `der_length_integer()`, `dsa_sign_hash_raw()`, `dsa_verify_hash_raw()`, and `dsa_verify_key()`.

## 5.80.2 Typedef Documentation

### 5.80.2.1 `typedef void ecc_point`

Definition at line 11 of file tomcrypt\_math.h.

### 5.80.2.2 `typedef void rsa_key`

Definition at line 15 of file tomcrypt\_math.h.

## 5.80.3 Function Documentation

### 5.80.3.1 `void ltc_deinit_multi (void *a, ...)`

Definition at line 44 of file multi.c.

```
45 {  
46     void      *cur = a;  
47     va_list   args;  
48  
49     va_start(args, a);  
50     while (cur != NULL) {  
51         mp_clear(cur);  
52         cur = va_arg(args, void *);  
53     }  
54     va_end(args);  
55 }
```

### 5.80.3.2 `int ltc_init_multi (void **a, ...)`

Definition at line 16 of file multi.c.

References CRYPT\_MEM, and CRYPT\_OK.

```
17 {
18     void    **cur = a;
19     int      np   = 0;
20     va_list  args;
21
22     va_start(args, a);
23     while (cur != NULL) {
24         if (mp_init(cur) != CRYPT_OK) {
25             /* failed */
26             va_list clean_list;
27
28             va_start(clean_list, a);
29             cur = a;
30             while (np-- > 0) {
31                 mp_clear(*cur);
32                 cur = va_arg(clean_list, void**);
33             }
34             va_end(clean_list);
35             return CRYPT_MEM;
36         }
37         ++np;
38         cur = va_arg(args, void**);
39     }
40     va_end(args);
41     return CRYPT_OK;
42 }
```

## 5.80.4 Variable Documentation

### 5.80.4.1 [ltc\\_math\\_descriptor ltc\\_mp](#)

Definition at line 13 of file crypt\_ltc\_mp\_descriptor.c.

Referenced by [crypt\\_fsa\(\)](#), [dsa\\_import\(\)](#), [dsa\\_make\\_key\(\)](#), [ecc\\_import\(\)](#), [ecc\\_make\\_key\(\)](#), [ecc\\_shared\\_secret\(\)](#), [ecc\\_verify\\_hash\(\)](#), [rsa\\_decrypt\\_key\\_ex\(\)](#), [rsa\\_encrypt\\_key\\_ex\(\)](#), [rsa\\_import\(\)](#), [rsa\\_make\\_key\(\)](#), [rsa\\_sign\\_hash\\_ex\(\)](#), and [rsa\\_verify\\_hash\\_ex\(\)](#).

## 5.81 headers/tomcrypt\_misc.h File Reference

This graph shows which files directly or indirectly include this file:

### Functions

- void [zeromem](#) (void \*dst, size\_t len)  
*Zero a block of memory.*
- void [burn\\_stack](#) (unsigned long len)  
*Burn some stack memory.*
- const char \* [error\\_to\\_string](#) (int err)  
*Convert an LTC error code to ASCII.*
- int [crypt\\_fsa](#) (void \*mp,...)

### Variables

- const char \* [crypt\\_build\\_settings](#)

### 5.81.1 Function Documentation

#### 5.81.1.1 void burn\_stack (unsigned long len)

Burn some stack memory.

##### Parameters:

*len* amount of stack to burn in bytes

Definition at line 22 of file burn\_stack.c.

References [burn\\_stack\(\)](#), and [zeromem\(\)](#).

Referenced by [burn\\_stack\(\)](#).

```

23 {
24     unsigned char buf[32];
25     zeromem(buf, sizeof(buf));
26     if (len > (unsigned long)sizeof(buf))
27         burn_stack(len - sizeof(buf));
28 }
```

Here is the call graph for this function:

#### 5.81.1.2 int crypt\_fsa (void \*mp, ...)

Definition at line 20 of file crypt\_fsa.c.

References [CRYPT\\_OK](#), [ltc\\_mp](#), [register\\_cipher\(\)](#), [register\\_hash\(\)](#), [register\\_prng\(\)](#), and [XMEMCPY](#).



```

21 {
22     int      err;
23     va_list  args;
24     void     *p;
25
26     va_start(args, mp);
27     if (mp != NULL) {
28         XMEMCPY(&ltc_mp, mp, sizeof(ltc_mp));
29     }
30
31     while ((p = va_arg(args, void*)) != NULL) {
32         if ((err = register_cipher(p)) != CRYPT_OK) {
33             va_end(args);
34             return err;
35         }
36     }
37
38     while ((p = va_arg(args, void*)) != NULL) {
39         if ((err = register_hash(p)) != CRYPT_OK) {
40             va_end(args);
41             return err;
42         }
43     }
44
45     while ((p = va_arg(args, void*)) != NULL) {
46         if ((err = register_prng(p)) != CRYPT_OK) {
47             va_end(args);
48             return err;
49         }
50     }
51
52     va_end(args);
53     return CRYPT_OK;
54 }

```

Here is the call graph for this function:

### 5.81.1.3 const char\* error\_to\_string (int err)

Convert an LTC error code to ASCII.

#### Parameters:

*err* The error code

#### Returns:

A pointer to the ASCII NUL terminated string for the error or "Invalid error code." if the err code was not valid.

Definition at line 62 of file error\_to\_string.c.

References `err_2_str`.

Referenced by `hmac_test()`.

```

63 {
64     if (err < 0 || err >= (int)(sizeof(err_2_str)/sizeof(err_2_str[0]))) {
65         return "Invalid error code.";
66     } else {
67         return err_2_str[err];
68     }
69 }

```

#### 5.81.1.4 void zeromem (void \* out, size\_t outlen)

Zero a block of memory.

##### Parameters:

**out** The destination of the area to zero

**outlen** The length of the area to zero (octets)

Definition at line 23 of file zeromem.c.

References LTC\_ARGCHKVD.

Referenced by burn\_stack(), cast5\_setup(), chc\_init(), dsa\_shared\_secret(), dsa\_sign\_hash\_raw(), eax\_decrypt\_verify\_memory(), eax\_encrypt\_authenticate\_memory(), eax\_init(), ECB\_TEST(), ecc\_ansi\_x963\_export(), ecc\_shared\_secret(), f8\_encrypt(), f8\_start(), f9\_file(), fortuna\_read(), gcm\_add\_aad(), gcm\_gf\_mult(), gcm\_init(), gcm\_reset(), hash\_filehandle(), hash\_memory(), hmac\_file(), hmac\_init(), hmac\_memory(), lrw\_start(), md2\_done(), md2\_init(), noekeon\_test(), ocb\_decrypt\_verify\_memory(), ocb\_done\_decrypt(), ocb\_encrypt\_authenticate\_memory(), omac\_done(), omac\_file(), omac\_init(), omac\_memory(), pelican\_init(), pkcs\_1\_i2osp(), pkcs\_1\_pss\_encode(), pkcs\_5\_alg2(), pmac\_file(), pmac\_memory(), rc2\_test(), rc4\_read(), rng\_make\_prng(), rsa\_exptmod(), rsa\_verify\_hash\_ex(), sha224\_done(), sha384\_done(), sober128\_read(), tiger\_done(), whirlpool\_init(), xcbc\_file(), yarrow\_read(), and yarrow\_start().

```
24 {  
25     unsigned char *mem = out;  
26     LTC_ARGCHKVD(out != NULL);  
27     while (outlen-- > 0) {  
28         *mem++ = 0;  
29     }  
30 }
```

## 5.81.2 Variable Documentation

### 5.81.2.1 const char\* crypt\_build\_settings

Definition at line 18 of file crypt.c.

## 5.82 headers/tomcrypt\_pk.h File Reference

This graph shows which files directly or indirectly include this file:

### Enumerations

- enum {  
    [PK\\_PUBLIC](#) = 0,  
    [PK\\_PRIVATE](#) = 1 }

### Functions

- int [rand\\_prime](#) (void \*N, long [len](#), [prng\\_state](#) \*prng, int wprng)

### 5.82.1 Enumeration Type Documentation

#### 5.82.1.1 anonymous enum

Enumerator:

***PK\_PUBLIC***

***PK\_PRIVATE***

Definition at line 3 of file tomcrypt\_pk.h.

```
3      {  
4      PK_PUBLIC=0,  
5      PK_PRIVATE=1  
6  };
```

### 5.82.2 Function Documentation

#### 5.82.2.1 int rand\_prime (void \*N, long len, [prng\\_state](#) \*prng, int wprng)

Definition at line 20 of file rand\_prime.c.

References [CRYPT\\_ERROR\\_READPRNG](#), [CRYPT\\_INVALID\\_PRIME\\_SIZE](#), [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), [prng\\_descriptor](#), [prng\\_is\\_valid\(\)](#), [USE\\_BBS](#), [XCALLOC](#), and [XFREE](#).

Referenced by [dsa\\_make\\_key\(\)](#), and [rsa\\_make\\_key\(\)](#).

```
21 {  
22     int                err, res, type;  
23     unsigned char *buf;  
24  
25     LTC_ARGCHK(N != NULL);  
26  
27     /* get type */  
28     if (len < 0) {  
29         type = USE_BBS;  
30         len = -len;  
31     } else {  
32         type = 0;  
33     }
```

```
34
35  /* allow sizes between 2 and 512 bytes for a prime size */
36  if (len < 2 || len > 512) {
37      return CRYPT_INVALID_PRIME_SIZE;
38  }
39
40  /* valid PRNG? Better be! */
41  if ((err = prng_is_valid(wprng)) != CRYPT_OK) {
42      return err;
43  }
44
45  /* allocate buffer to work with */
46  buf = XCALLOC(1, len);
47  if (buf == NULL) {
48      return CRYPT_MEM;
49  }
50
51  do {
52      /* generate value */
53      if (prng_descriptor[wprng].read(buf, len, prng) != (unsigned long)len) {
54          XFREE(buf);
55          return CRYPT_ERROR_READPRNG;
56      }
57
58      /* munge bits */
59      buf[0] |= 0x80 | 0x40;
60      buf[len-1] |= 0x01 | ((type & USE_BBS) ? 0x02 : 0x00);
61
62      /* load value */
63      if ((err = mp_read_unsigned_bin(N, buf, len)) != CRYPT_OK) {
64          XFREE(buf);
65          return err;
66      }
67
68      /* test */
69      if ((err = mp_prime_is_prime(N, 8, &res)) != CRYPT_OK) {
70          XFREE(buf);
71          return err;
72      }
73  } while (res == LTC_MP_NO);
74
75  #ifdef LTC_CLEAN_STACK
76      zeromem(buf, len);
77  #endif
78
79  XFREE(buf);
80  return CRYPT_OK;
81 }
```

Here is the call graph for this function:

## 5.83 headers/tomcrypt\_pkcs.h File Reference

This graph shows which files directly or indirectly include this file:

## 5.84 headers/tomcrypt\_prng.h File Reference

This graph shows which files directly or indirectly include this file:

### Data Structures

- union [Prng\\_state](#)
- struct [ltc\\_prng\\_descriptor](#)  
*PRNG descriptor.*

### Typedefs

- typedef [Prng\\_state](#) [prng\\_state](#)

### Functions

- int [find\\_prng](#) (const char \*name)  
*Find a registered PRNG by name.*
- int [register\\_prng](#) (const struct [ltc\\_prng\\_descriptor](#) \*prng)  
*Register a PRNG with the descriptor table.*
- int [unregister\\_prng](#) (const struct [ltc\\_prng\\_descriptor](#) \*prng)  
*Unregister a PRNG from the descriptor table.*
- int [prng\\_is\\_valid](#) (int idx)
- unsigned long [rng\\_get\\_bytes](#) (unsigned char \*out, unsigned long outlen, void(\*callback)(void))  
*Read the system RNG.*
- int [rng\\_make\\_prng](#) (int bits, int wprng, [prng\\_state](#) \*prng, void(\*callback)(void))  
*Create a PRNG from a RNG.*

### Variables

- [ltc\\_prng\\_descriptor](#) [prng\\_descriptor](#) []  
*PRNG descriptor.*

#### 5.84.1 Typedef Documentation

##### 5.84.1.1 typedef union [Prng\\_state](#) [prng\\_state](#)

#### 5.84.2 Function Documentation

##### 5.84.2.1 int [find\\_prng](#) (const char \* *name*)

Find a registered PRNG by name.

**Parameters:**

*name* The name of the PRNG to look for

**Returns:**

$\geq 0$  if found, -1 if not present

Definition at line 23 of file crypt\_find\_prng.c.

References LTC\_ARGCHK, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, prng\_descriptor, and TAB\_SIZE.

```

24 {
25     int x;
26     LTC_ARGCHK(name != NULL);
27     LTC_MUTEX_LOCK(&ltc_prng_mutex);
28     for (x = 0; x < TAB_SIZE; x++) {
29         if ((prng_descriptor[x].name != NULL) && strcmp(prng_descriptor[x].name, name) == 0) {
30             LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
31             return x;
32         }
33     }
34     LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
35     return -1;
36 }

```

**5.84.2.2 int prng\_is\_valid(int idx)**

Definition at line 23 of file crypt\_prng\_is\_valid.c.

References CRYPT\_INVALID\_PRNG, CRYPT\_OK, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, ltc\_prng\_descriptor::name, prng\_descriptor, and TAB\_SIZE.

Referenced by dsa\_encrypt\_key(), dsa\_make\_key(), dsa\_sign\_hash\_raw(), ecc\_encrypt\_key(), ecc\_make\_key(), ecc\_sign\_hash(), pkcs\_1\_oaep\_encode(), pkcs\_1\_pss\_encode(), pkcs\_1\_v1\_5\_encode(), rand\_prime(), rng\_make\_prng(), rsa\_encrypt\_key\_ex(), rsa\_make\_key(), and rsa\_sign\_hash\_ex().

```

24 {
25     LTC_MUTEX_LOCK(&ltc_prng_mutex);
26     if (idx < 0 || idx >= TAB_SIZE || prng_descriptor[idx].name == NULL) {
27         LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
28         return CRYPT_INVALID_PRNG;
29     }
30     LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
31     return CRYPT_OK;
32 }

```

**5.84.2.3 int register\_prng(const struct ltc\_prng\_descriptor \*prng)**

Register a PRNG with the descriptor table.

**Parameters:**

*prng* The PRNG you wish to register

**Returns:**

value  $\geq 0$  if successfully added (or already present), -1 if unsuccessful

Definition at line 23 of file `crypt_register_prng.c`.

References `LTC_ARGCHK`, `LTC_MUTEX_LOCK`, `LTC_MUTEX_UNLOCK`, `prng_descriptor`, `TAB_SIZE`, and `XMEMCMP`.

Referenced by `crypt_fsa()`.

```

24 {
25     int x;
26
27     LTC_ARGCHK(prng != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_prng_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (XMEMCMP(&prng_descriptor[x], prng, sizeof(struct ltc_prng_descriptor)) == 0) {
33             LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
34             return x;
35         }
36     }
37
38     /* find a blank spot */
39     for (x = 0; x < TAB_SIZE; x++) {
40         if (prng_descriptor[x].name == NULL) {
41             XMEMCPY(&prng_descriptor[x], prng, sizeof(struct ltc_prng_descriptor));
42             LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
43             return x;
44         }
45     }
46
47     /* no spot */
48     LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
49     return -1;
50 }

```

#### 5.84.2.4 unsigned long rng\_get\_bytes (unsigned char \* out, unsigned long outlen, void(\*) (void) callback)

Read the system RNG.

##### Parameters:

**out** Destination

**outlen** Length desired (octets)

**callback** Pointer to void function to act as "callback" when RNG is slow. This can be NULL

##### Returns:

Number of octets read

Definition at line 123 of file `rng_get_bytes.c`.

References `LTC_ARGCHK`, and `rng_nix()`.

Referenced by `rng_make_prng()`, and `sprng_read()`.

```

125 {
126     unsigned long x;
127
128     LTC_ARGCHK(out != NULL);
129
130     #if defined(DEVRANDOM)

```



```

131     x = rng_nix(out, outlen, callback);    if (x != 0) { return x; }
132 #endif
133 #ifdef WIN32
134     x = rng_win32(out, outlen, callback); if (x != 0) { return x; }
135 #endif
136 #ifdef ANSI_RNG
137     x = rng_ansi(out, outlen, callback); if (x != 0) { return x; }
138 #endif
139     return 0;
140 }

```

Here is the call graph for this function:

#### 5.84.2.5 int rng\_make\_prng (int bits, int wprng, prng\_state \* prng, void(\*) (void) callback)

Create a PRNG from a RNG.

##### Parameters:

- bits** Number of bits of entropy desired (64 ... 1024)
- wprng** Index of which PRNG to setup
- prng** [out] PRNG state to initialize
- callback** A pointer to a void function for when the RNG is slow, this can be NULL

##### Returns:

CRYPT\_OK if successful

Definition at line 26 of file rng\_make\_prng.c.

References CRYPT\_ERROR\_READPRNG, CRYPT\_INVALID\_PRNGSIZE, CRYPT\_OK, LTC\_ARGCHK, prng\_descriptor, prng\_is\_valid(), rng\_get\_bytes(), and zeromem().

```

28 {
29     unsigned char buf[256];
30     int err;
31
32     LTC_ARGCHK(prng != NULL);
33
34     /* check parameter */
35     if ((err = prng_is_valid(wprng)) != CRYPT_OK) {
36         return err;
37     }
38
39     if (bits < 64 || bits > 1024) {
40         return CRYPT_INVALID_PRNGSIZE;
41     }
42
43     if ((err = prng_descriptor[wprng].start(prng)) != CRYPT_OK) {
44         return err;
45     }
46
47     bits = ((bits/8)+((bits&7)!=0?1:0)) * 2;
48     if (rng_get_bytes(buf, (unsigned long)bits, callback) != (unsigned long)bits) {
49         return CRYPT_ERROR_READPRNG;
50     }
51
52     if ((err = prng_descriptor[wprng].add_entropy(buf, (unsigned long)bits, prng)) != CRYPT_OK) {
53         return err;
54     }
55
56     if ((err = prng_descriptor[wprng].ready(prng)) != CRYPT_OK) {

```

```

57     return err;
58 }
59
60 #ifdef LTC_CLEAN_STACK
61     zeromem(buf, sizeof(buf));
62 #endif
63 return CRYPT_OK;
64 }

```

Here is the call graph for this function:

#### 5.84.2.6 int unregister\_prng (const struct [ltc\\_prng\\_descriptor](#) \* *prng*)

Unregister a PRNG from the descriptor table.

##### Parameters:

*prng* The PRNG descriptor to remove

##### Returns:

CRYPT\_OK on success

Definition at line 23 of file `crypt_unregister_prng.c`.

References CRYPT\_OK, LTC\_ARGCHK, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, `ltc_prng_descriptor::name`, `prng_descriptor`, TAB\_SIZE, and XMEMCMP.

```

24 {
25     int x;
26
27     LTC_ARGCHK(prng != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_prng_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (XMEMCMP(&prng_descriptor[x], prng, sizeof(struct ltc_prng_descriptor)) != 0) {
33             prng_descriptor[x].name = NULL;
34             LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
35             return CRYPT_OK;
36         }
37     }
38     LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
39     return CRYPT_ERROR;
40 }

```

### 5.84.3 Variable Documentation

#### 5.84.3.1 struct [ltc\\_prng\\_descriptor](#) `prng_descriptor[]`

PRNG descriptor.

Referenced by `dsa_encrypt_key()`, `dsa_make_key()`, `dsa_sign_hash_raw()`, `find_prng()`, `pkcs1_oaep_encode()`, `pkcs1_pss_encode()`, `pkcs1_v1_5_encode()`, `prng_is_valid()`, `rand_prime()`, `register_prng()`, `rng_make_prng()`, and `unregister_prng()`.

## 5.85 mac/f9/f9\_done.c File Reference

### 5.85.1 Detailed Description

f9 Support, terminate the state

Definition in file [f9\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for f9\_done.c:

### Functions

- `int f9_done(f9_state *f9, unsigned char *out, unsigned long *outlen)`

*Terminate the f9-MAC state.*

### 5.85.2 Function Documentation

#### 5.85.2.1 `int f9_done(f9_state *f9, unsigned char *out, unsigned long *outlen)`

Terminate the f9-MAC state.

#### Parameters:

*f9* f9 state to terminate

*out* [out] Destination for the MAC tag

*outlen* [in/out] Destination size and final tag size Return CRYPT\_OK on success

Definition at line 26 of file f9\_done.c.

References `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `ltc_cipher_descriptor::ecb_encrypt`, and `LTC_ARGCHK`.

Referenced by `f9_file()`, and `f9_memory()`.

```
27 {
28     int err, x;
29     LTC_ARGCHK(f9 != NULL);
30     LTC_ARGCHK(out != NULL);
31
32     /* check structure */
33     if ((err = cipher_is_valid(f9->cipher)) != CRYPT_OK) {
34         return err;
35     }
36
37     if ((f9->blocksize > cipher_descriptor[f9->cipher].block_length) || (f9->blocksize < 0) ||
38         (f9->buflen > f9->blocksize) || (f9->buflen < 0)) {
39         return CRYPT_INVALID_ARG;
40     }
41
42     if (f9->buflen != 0) {
43         /* encrypt */
44         cipher_descriptor[f9->cipher].ecb_encrypt(f9->IV, f9->IV, &f9->key);
45         f9->buflen = 0;
46         for (x = 0; x < f9->blocksize; x++) {
47             f9->ACC[x] ^= f9->IV[x];
48         }
49     }
50 }
```

```
48     }
49 }
50
51 /* schedule modified key */
52 if ((err = cipher_descriptor[f9->cipher].setup(f9->akey, f9->keylen, 0, &f9->key)) != CRYPT_OK) {
53     return err;
54 }
55
56 /* encrypt the ACC */
57 cipher_descriptor[f9->cipher].ecb_encrypt(f9->ACC, f9->ACC, &f9->key);
58 cipher_descriptor[f9->cipher].done(&f9->key);
59
60 /* extract tag */
61 for (x = 0; x < f9->blocksize && (unsigned long)x < *outlen; x++) {
62     out[x] = f9->ACC[x];
63 }
64 *outlen = x;
65
66 #ifdef LTC_CLEAN_STACK
67     zeromem(f9, sizeof(*f9));
68 #endif
69     return CRYPT_OK;
70 }
```

Here is the call graph for this function:

## 5.86 mac/f9/f9\_file.c File Reference

### 5.86.1 Detailed Description

f9 support, process a file, Tom St Denis

Definition in file [f9\\_file.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for f9\_file.c:

### Functions

- [int f9\\_file](#) (int cipher, const unsigned char \*key, unsigned long keylen, const char \*filename, unsigned char \*out, unsigned long \*outlen)

*f9 a file*

### 5.86.2 Function Documentation

#### 5.86.2.1 int f9\_file (int cipher, const unsigned char \*key, unsigned long keylen, const char \*filename, unsigned char \*out, unsigned long \*outlen)

f9 a file

#### Parameters:

***cipher*** The index of the cipher desired

***key*** The secret key

***keylen*** The length of the secret key (octets)

***filename*** The name of the file you wish to f9

***out*** [out] Where the authentication tag is to be stored

***outlen*** [in/out] The max size and resulting size of the authentication tag

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if file support has been disabled

Definition at line 30 of file f9\_file.c.

References CRYPT\_FILE\_NOTFOUND, CRYPT\_NOP, CRYPT\_OK, f9\_done(), f9\_init(), f9\_process(), in, LTC\_ARGCHK, and zeromem().

```
34 {
35 #ifdef LTC_NO_FILE
36     return CRYPT_NOP;
37 #else
38     int err, x;
39     f9_state f9;
40     FILE *in;
41     unsigned char buf[512];
42
43     LTC_ARGCHK(key != NULL);
44     LTC_ARGCHK(filename != NULL);
45     LTC_ARGCHK(out != NULL);
```

```
46 LTC_ARGCHK(outlen != NULL);
47
48 in = fopen(filename, "rb");
49 if (in == NULL) {
50     return CRYPT_FILE_NOTFOUND;
51 }
52
53 if ((err = f9_init(&f9, cipher, key, keylen)) != CRYPT_OK) {
54     fclose(in);
55     return err;
56 }
57
58 do {
59     x = fread(buf, 1, sizeof(buf), in);
60     if ((err = f9_process(&f9, buf, x)) != CRYPT_OK) {
61         fclose(in);
62         return err;
63     }
64 } while (x == sizeof(buf));
65 fclose(in);
66
67 if ((err = f9_done(&f9, out, outlen)) != CRYPT_OK) {
68     return err;
69 }
70
71 #ifdef LTC_CLEAN_STACK
72     zeromem(buf, sizeof(buf));
73 #endif
74
75     return CRYPT_OK;
76 #endif
77 }
```

Here is the call graph for this function:

## 5.87 mac/f9/f9\_init.c File Reference

### 5.87.1 Detailed Description

F9 Support, start an F9 state.

Definition in file [f9\\_init.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for f9\_init.c:

### Functions

- [int f9\\_init](#) (f9\_state \*f9, int cipher, const unsigned char \*key, unsigned long keylen)  
*Initialize F9-MAC state.*

### 5.87.2 Function Documentation

#### 5.87.2.1 int f9\_init (f9\_state \*f9, int cipher, const unsigned char \*key, unsigned long keylen)

Initialize F9-MAC state.

##### Parameters:

**f9** [out] f9 state to initialize

**cipher** Index of cipher to use

**key** [in] Secret key

**keylen** Length of secret key in octets Return CRYPT\_OK on success

Definition at line 27 of file f9\_init.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_prng\\_descriptor::done](#), and [LTC\\_ARGCHK](#).

Referenced by [f9\\_file\(\)](#), [f9\\_memory\(\)](#), and [f9\\_memory\\_multi\(\)](#).

```
28 {
29     int                x, err;
30
31     LTC_ARGCHK(f9      != NULL);
32     LTC_ARGCHK(key     != NULL);
33
34     /* schedule the key */
35     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
36         return err;
37     }
38
39 #ifdef LTC_FAST
40     if (cipher_descriptor[cipher].block_length % sizeof(LTC_FAST_TYPE)) {
41         return CRYPT_INVALID_ARG;
42     }
43 #endif
44
45     if ((err = cipher_descriptor[cipher].setup(key, keylen, 0, &f9->key)) != CRYPT_OK) {
46         goto done;
47     }
```

```
48
49  /* make the second key */
50  for (x = 0; (unsigned)x < keylen; x++) {
51      f9->akey[x] = key[x] ^ 0xAA;
52  }
53
54  /* setup struct */
55  zeromem(f9->IV, cipher_descriptor[cipher].block_length);
56  zeromem(f9->ACC, cipher_descriptor[cipher].block_length);
57  f9->blocksize = cipher_descriptor[cipher].block_length;
58  f9->cipher     = cipher;
59  f9->buflen     = 0;
60  f9->keylen     = keylen;
61 done:
62     return err;
63 }
```

Here is the call graph for this function:



## 5.88 mac/f9/f9\_memory.c File Reference

```
#include "tomcrypt.h"
```

Include dependency graph for f9\_memory.c:

### Functions

- `int f9_memory` (int *cipher*, const unsigned char \**key*, unsigned long *keylen*, const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*f9-MAC a block of memory*

### 5.88.1 Function Documentation

#### 5.88.1.1 int f9\_memory (int *cipher*, const unsigned char \* *key*, unsigned long *keylen*, const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

*f9-MAC a block of memory*

#### Parameters:

*cipher* Index of cipher to use

*key* [in] Secret key

*keylen* Length of key in octets

*in* [in] Message to MAC

*inlen* Length of input in octets

*out* [out] Destination for the MAC tag

*outlen* [in/out] Output size and final tag size Return CRYPT\_OK on success.

Definition at line 30 of file f9\_memory.c.

References cipher\_descriptor, cipher\_is\_valid(), CRYPT\_MEM, CRYPT\_OK, ltc\_prng\_descriptor::done, f9\_done(), f9\_init(), ltc\_cipher\_descriptor::f9\_memory, f9\_process(), XCALLOC, and XFREE.

Referenced by f9\_test().

```
34 {
35     f9_state *f9;
36     int      err;
37
38     /* is the cipher valid? */
39     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
40         return err;
41     }
42
43     /* Use accelerator if found */
44     if (cipher_descriptor[cipher].f9_memory != NULL) {
45         return cipher_descriptor[cipher].f9_memory(key, keylen, in, inlen, out, outlen);
46     }
47
48     f9 = XCALLOC(1, sizeof(*f9));
49     if (f9 == NULL) {
50         return CRYPT_MEM;
51     }
52 }
```

```
53     if ((err = f9_init(f9, cipher, key, keylen)) != CRYPT_OK) {
54         goto done;
55     }
56
57     if ((err = f9_process(f9, in, inlen)) != CRYPT_OK) {
58         goto done;
59     }
60
61     err = f9_done(f9, out, outlen);
62 done:
63     XFREE(f9);
64     return err;
65 }
```

Here is the call graph for this function:

## 5.89 mac/f9/f9\_memory\_multi.c File Reference

### 5.89.1 Detailed Description

f9 support, process multiple blocks of memory, Tom St Denis

Definition in file [f9\\_memory\\_multi.c](#).

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for f9\_memory\_multi.c:

### Functions

- int [f9\\_memory\\_multi](#) (int cipher, const unsigned char \*key, unsigned long keylen, unsigned char \*out, unsigned long \*outlen, const unsigned char \*in, unsigned long inlen,...)  
*f9 multiple blocks of memory*

### 5.89.2 Function Documentation

**5.89.2.1** int [f9\\_memory\\_multi](#) (int *cipher*, const unsigned char \* *key*, unsigned long *keylen*, unsigned char \* *out*, unsigned long \* *outlen*, const unsigned char \* *in*, unsigned long *inlen*, ...)

f9 multiple blocks of memory

#### Parameters:

***cipher*** The index of the desired cipher

***key*** The secret key

***keylen*** The length of the secret key (octets)

***out*** [out] The destination of the authentication tag

***outlen*** [in/out] The max size and resulting size of the authentication tag (octets)

***in*** The data to send through f9

***inlen*** The length of the data to send through f9 (octets)

... tuples of (data,len) pairs to f9, terminated with a (NULL,x) (x=don't care)

#### Returns:

CRYPT\_OK if successful

Definition at line 33 of file f9\_memory\_multi.c.

References CRYPT\_MEM, CRYPT\_OK, f9\_init(), f9\_process(), LTC\_ARGCHK, and XMALLOC.

```
37 {
38     int                err;
39     f9_state            *f9;
40     va_list            args;
41     const unsigned char *curptr;
42     unsigned long       curlen;
```

```
43
44     LTC_ARGCHK(key    != NULL);
45     LTC_ARGCHK(in     != NULL);
46     LTC_ARGCHK(out    != NULL);
47     LTC_ARGCHK(outlen != NULL);
48
49     /* allocate ram for f9 state */
50     f9 = XMALLOC(sizeof(f9_state));
51     if (f9 == NULL) {
52         return CRYPT_MEM;
53     }
54
55     /* f9 process the message */
56     if ((err = f9_init(f9, cipher, key, keylen)) != CRYPT_OK) {
57         goto LBL_ERR;
58     }
59     va_start(args, inlen);
60     curptr = in;
61     curlen = inlen;
62     for (;;) {
63         /* process buf */
64         if ((err = f9_process(f9, curptr, curlen)) != CRYPT_OK) {
65             goto LBL_ERR;
66         }
67         /* step to next */
68         curptr = va_arg(args, const unsigned char*);
69         if (curptr == NULL) {
70             break;
71         }
72         curlen = va_arg(args, unsigned long);
73     }
74     if ((err = f9_done(f9, out, outlen)) != CRYPT_OK) {
75         goto LBL_ERR;
76     }
77 LBL_ERR:
78 #ifdef LTC_CLEAN_STACK
79     zeromem(f9, sizeof(f9_state));
80 #endif
81     XFREE(f9);
82     va_end(args);
83     return err;
84 }
```

Here is the call graph for this function:

## 5.90 mac/f9/f9\_process.c File Reference

### 5.90.1 Detailed Description

f9 Support, terminate the state

Definition in file [f9\\_process.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for f9\_process.c:

### Functions

- [int f9\\_process](#) (f9\_state \*f9, const unsigned char \*in, unsigned long inlen)  
*Process data through f9-MAC.*

### 5.90.2 Function Documentation

#### 5.90.2.1 int f9\_process (f9\_state \*f9, const unsigned char \*in, unsigned long inlen)

Process data through f9-MAC.

#### Parameters:

*f9* The f9-MAC state

*in* Input data to process

*inlen* Length of input in octets Return CRYPT\_OK on success

Definition at line 26 of file f9\_process.c.

References [ltc\\_cipher\\_descriptor::block\\_length](#), [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), and [LTC\\_ARGCHK](#).

Referenced by [f9\\_file\(\)](#), [f9\\_memory\(\)](#), and [f9\\_memory\\_multi\(\)](#).

```
27 {
28     int err, x;
29
30     LTC_ARGCHK(f9 != NULL);
31     LTC_ARGCHK(in != NULL);
32
33     /* check structure */
34     if ((err = cipher_is_valid(f9->cipher)) != CRYPT_OK) {
35         return err;
36     }
37
38     if ((f9->blocksize > cipher_descriptor[f9->cipher].block_length) || (f9->blocksize < 0) ||
39         (f9->buflen > f9->blocksize) || (f9->buflen < 0)) {
40         return CRYPT_INVALID_ARG;
41     }
42
43 #ifdef LTC_FAST
44     if (f9->buflen == 0) {
45         while (inlen >= (unsigned long)f9->blocksize) {
46             for (x = 0; x < f9->blocksize; x += sizeof(LTC_FAST_TYPE)) {
47                 *((LTC_FAST_TYPE*)&(f9->IV[x])) ^= *((LTC_FAST_TYPE*)&(in[x]));
```

```
48         }
49         cipher_descriptor[f9->cipher].ecb_encrypt(f9->IV, f9->IV, &f9->key);
50         for (x = 0; x < f9->blocksize; x += sizeof(LTC_FAST_TYPE)) {
51             *((LTC_FAST_TYPE*)&(f9->ACC[x])) ^= *((LTC_FAST_TYPE*)&(f9->IV[x]));
52         }
53         in += f9->blocksize;
54         inlen -= f9->blocksize;
55     }
56 }
57 #endif
58
59 while (inlen) {
60     if (f9->buflen == f9->blocksize) {
61         cipher_descriptor[f9->cipher].ecb_encrypt(f9->IV, f9->IV, &f9->key);
62         for (x = 0; x < f9->blocksize; x++) {
63             f9->ACC[x] ^= f9->IV[x];
64         }
65         f9->buflen = 0;
66     }
67     f9->IV[f9->buflen++] ^= *in++;
68     --inlen;
69 }
70 return CRYPT_OK;
71 }
```

Here is the call graph for this function:

## 5.91 mac/f9/f9\_test.c File Reference

### 5.91.1 Detailed Description

f9 Support, terminate the state

Definition in file [f9\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for f9\_test.c:

### Functions

- [int f9\\_test](#) (void)  
*Test f9-MAC mode Return CRYPT\_OK on succes.*

### 5.91.2 Function Documentation

#### 5.91.2.1 int f9\_test (void)

Test f9-MAC mode Return CRYPT\_OK on succes.

Definition at line 23 of file f9\_test.c.

References CRYPT\_NOP, CRYPT\_OK, f9\_memory(), find\_cipher(), and K.

```
24 {
25 #ifdef LTC_NO_TEST
26     return CRYPT_NOP;
27 #else
28     static const struct {
29         int msglen;
30         unsigned char K[16], M[128], T[4];
31     } tests[] = {
32 {
33     20,
34     { 0x2B, 0xD6, 0x45, 0x9F, 0x82, 0xC5, 0xB3, 0x00, 0x95, 0x2C, 0x49, 0x10, 0x48, 0x81, 0xFF, 0x48 },
35     { 0x38, 0xA6, 0xF0, 0x56, 0xB8, 0xAE, 0xFD, 0xA9, 0x33, 0x32, 0x34, 0x62, 0x63, 0x39, 0x38, 0x61, 0x00,
36     { 0x46, 0xE0, 0x0D, 0x4B }
37 },
38
39 {
40     105,
41     { 0x83, 0xFD, 0x23, 0xA2, 0x44, 0xA7, 0x4C, 0xF3, 0x58, 0xDA, 0x30, 0x19, 0xF1, 0x72, 0x26, 0x35 },
42     { 0x36, 0xAF, 0x61, 0x44, 0x4F, 0x30, 0x2A, 0xD2,
43     0x35, 0xC6, 0x87, 0x16, 0x63, 0x3C, 0x66, 0xFB, 0x75, 0x0C, 0x26, 0x68, 0x65, 0xD5, 0x3C, 0x11, 0x00,
44     0x47, 0x90, 0x28, 0x37, 0xF5, 0xAE, 0x96, 0xD5, 0xA0, 0x5B, 0xC8, 0xD6, 0x1C, 0xA8, 0xDB, 0xEF, 0x00,
45     0x93, 0x04, 0xC3, 0x82, 0xBE, 0x53, 0xA5, 0xAF, 0x05, 0x55, 0x61, 0x76, 0xF6, 0xEA, 0xA2, 0xEF, 0x00,
46     0x40|0x80 },
47     { 0x95, 0xAE, 0x41, 0xBA },
48 }
49 };
50 unsigned char T[16];
51 unsigned long taglen;
52 int err, x, idx;
53
54 /* find kasumi */
55 if ((idx = find_cipher("kasumi")) == -1) {
56     return CRYPT_NOP;
```

```
57  }
58
59  for (x = 0; x < (int) (sizeof(tests)/sizeof(tests[0])); x++) {
60      taglen = 4;
61      if ((err = f9_memory(idx, tests[x].K, 16, tests[x].M, tests[x].msglen, T, &taglen)) != CRYPT_OK) {
62          return err;
63      }
64      if (taglen != 4 || XMEMCMP(T, tests[x].T, 4)) {
65          return CRYPT_FAIL_TESTVECTOR;
66      }
67  }
68
69  return CRYPT_OK;
70 #endif
71 }
```

Here is the call graph for this function:



## 5.92 mac/hmac/hmac\_done.c File Reference

### 5.92.1 Detailed Description

HMAC support, terminate stream, Tom St Denis/Dobes Vandermeer.

Definition in file [hmac\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [hmac\\_done.c](#):

### Defines

- `#define HMAC_BLOCKSIZE hash\_descriptor\[hash\].blocksize`

### Functions

- `int hmac\_done (hmac_state *hmac, unsigned char *out, unsigned long *outlen)`  
*Terminate an HMAC session.*

### 5.92.2 Define Documentation

#### 5.92.2.1 `#define HMAC_BLOCKSIZE hash\_descriptor\[hash\].blocksize`

Definition at line 20 of file [hmac\\_done.c](#).

Referenced by [hmac\\_done\(\)](#), and [hmac\\_init\(\)](#).

### 5.92.3 Function Documentation

#### 5.92.3.1 `int hmac\_done (hmac_state * hmac, unsigned char * out, unsigned long * outlen)`

Terminate an HMAC session.

#### Parameters:

*hmac* The HMAC state

*out* [out] The destination of the HMAC authentication tag

*outlen* [in/out] The max size and resulting size of the HMAC authentication tag

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file [hmac\\_done.c](#).

References [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [hash\\_descriptor](#), [hash\\_is\\_valid\(\)](#), [ltc\\_hash\\_descriptor::hashsize](#), [HMAC\\_BLOCKSIZE](#), [LTC\\_ARGCHK](#), [XFREE](#), and [XMALLOC](#).

Referenced by [hmac\\_file\(\)](#), [hmac\\_memory\(\)](#), and [pkcs\\_5\\_alg2\(\)](#).

```
30 {
31     unsigned char *buf, *isha;
32     unsigned long hashsize, i;
33     int hash, err;
34
35     LTC_ARGCHK(hmac != NULL);
36     LTC_ARGCHK(out != NULL);
37
38     /* test hash */
39     hash = hmac->hash;
40     if((err = hash_is_valid(hash)) != CRYPT_OK) {
41         return err;
42     }
43
44     /* get the hash message digest size */
45     hashsize = hash_descriptor[hash].hashsize;
46
47     /* allocate buffers */
48     buf = XMALLOC(HMAC_BLOCKSIZE);
49     isha = XMALLOC(hashsize);
50     if (buf == NULL || isha == NULL) {
51         if (buf != NULL) {
52             XFREE(buf);
53         }
54         if (isha != NULL) {
55             XFREE(isha);
56         }
57         return CRYPT_MEM;
58     }
59
60     /* Get the hash of the first HMAC vector plus the data */
61     if ((err = hash_descriptor[hash].done(&hmac->md, isha)) != CRYPT_OK) {
62         goto LBL_ERR;
63     }
64
65     /* Create the second HMAC vector vector for step (3) */
66     for(i=0; i < HMAC_BLOCKSIZE; i++) {
67         buf[i] = hmac->key[i] ^ 0x5C;
68     }
69
70     /* Now calculate the "outer" hash for step (5), (6), and (7) */
71     if ((err = hash_descriptor[hash].init(&hmac->md)) != CRYPT_OK) {
72         goto LBL_ERR;
73     }
74     if ((err = hash_descriptor[hash].process(&hmac->md, buf, HMAC_BLOCKSIZE)) != CRYPT_OK) {
75         goto LBL_ERR;
76     }
77     if ((err = hash_descriptor[hash].process(&hmac->md, isha, hashsize)) != CRYPT_OK) {
78         goto LBL_ERR;
79     }
80     if ((err = hash_descriptor[hash].done(&hmac->md, buf)) != CRYPT_OK) {
81         goto LBL_ERR;
82     }
83
84     /* copy to output */
85     for (i = 0; i < hashsize && i < *outlen; i++) {
86         out[i] = buf[i];
87     }
88     *outlen = i;
89
90     err = CRYPT_OK;
91 LBL_ERR:
92     XFREE(hmac->key);
93 #ifdef LTC_CLEAN_STACK
94     zeromem(isha, hashsize);
95     zeromem(buf, hashsize);
96     zeromem(hmac, sizeof(*hmac));
```

```
97 #endif
98
99     XFREE(isha);
100     XFREE(buf);
101
102     return err;
103 }
```

Here is the call graph for this function:

## 5.93 mac/hmac/hmac\_file.c File Reference

### 5.93.1 Detailed Description

HMAC support, process a file, Tom St Denis/Dobes Vandermeer.

Definition in file [hmac\\_file.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for hmac\_file.c:

### Functions

- [int hmac\\_file](#) (int hash, const char \*fname, const unsigned char \*key, unsigned long keylen, unsigned char \*out, unsigned long \*outlen)

*HMAC a file.*

### 5.93.2 Function Documentation

#### 5.93.2.1 int hmac\_file (int hash, const char \*fname, const unsigned char \*key, unsigned long keylen, unsigned char \*out, unsigned long \*outlen)

HMAC a file.

#### Parameters:

**hash** The index of the hash you wish to use

**fname** The name of the file you wish to HMAC

**key** The secret key

**keylen** The length of the secret key

**out** [out] The HMAC authentication tag

**outlen** [in/out] The max size and resulting size of the authentication tag

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if file support has been disabled

Definition at line 30 of file hmac\_file.c.

References CRYPT\_ERROR, CRYPT\_FILE\_NOTFOUND, CRYPT\_NOP, CRYPT\_OK, hash\_is\_valid(), hmac\_done(), hmac\_init(), hmac\_process(), in, LTC\_ARGCHK, and zeromem().

```
33 {
34 #ifdef LTC_NO_FILE
35     return CRYPT_NOP;
36 #else
37     hmac_state hmac;
38     FILE *in;
39     unsigned char buf[512];
40     size_t x;
41     int err;
42
43     LTC_ARGCHK(fname != NULL);
44     LTC_ARGCHK(key != NULL);
```

```
45     LTC_ARGCHK(out      != NULL);
46     LTC_ARGCHK(outlen != NULL);
47
48     if((err = hash_is_valid(hash)) != CRYPT_OK) {
49         return err;
50     }
51
52     if ((err = hmac_init(&hmac, hash, key, keylen)) != CRYPT_OK) {
53         return err;
54     }
55
56     in = fopen(fname, "rb");
57     if (in == NULL) {
58         return CRYPT_FILE_NOTFOUND;
59     }
60
61     /* process the file contents */
62     do {
63         x = fread(buf, 1, sizeof(buf), in);
64         if ((err = hmac_process(&hmac, buf, (unsigned long)x)) != CRYPT_OK) {
65             /* we don't trap this error since we're already returning an error! */
66             fclose(in);
67             return err;
68         }
69     } while (x == sizeof(buf));
70
71     if (fclose(in) != 0) {
72         return CRYPT_ERROR;
73     }
74
75     /* get final hmac */
76     if ((err = hmac_done(&hmac, out, outlen)) != CRYPT_OK) {
77         return err;
78     }
79
80 #ifdef LTC_CLEAN_STACK
81     /* clear memory */
82     zeromem(buf, sizeof(buf));
83 #endif
84     return CRYPT_OK;
85 #endif
86 }
```

Here is the call graph for this function:

## 5.94 mac/hmac/hmac\_init.c File Reference

### 5.94.1 Detailed Description

HMAC support, initialize state, Tom St Denis/Dobes Vandermeer.

Definition in file [hmac\\_init.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for hmac\_init.c:

### Defines

- #define [HMAC\\_BLOCKSIZE](#) [hash\\_descriptor\[hash\].blocksize](#)

### Functions

- int [hmac\\_init](#) (hmac\_state \*hmac, int hash, const unsigned char \*key, unsigned long keylen)  
*Initialize an HMAC context.*

### 5.94.2 Define Documentation

#### 5.94.2.1 #define HMAC\_BLOCKSIZE [hash\\_descriptor\[hash\].blocksize](#)

Definition at line 20 of file hmac\_init.c.

### 5.94.3 Function Documentation

#### 5.94.3.1 int hmac\_init (hmac\_state \* *hmac*, int *hash*, const unsigned char \* *key*, unsigned long *keylen*)

Initialize an HMAC context.

#### Parameters:

- hmac* The HMAC state
- hash* The index of the hash you want to use
- key* The secret key
- keylen* The length of the secret key (octets)

#### Returns:

- CRYPT\_OK if successful

Definition at line 30 of file hmac\_init.c.

References [CRYPT\\_INVALID\\_KEYSIZE](#), [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [hash\\_descriptor](#), [hash\\_is\\_valid\(\)](#), [hash\\_memory\(\)](#), [ltc\\_hash\\_descriptor::hashsize](#), [HMAC\\_BLOCKSIZE](#), [LTC\\_ARGCHK](#), [XFREE](#), [XMALLOC](#), [XMEMCPY](#), and [zeromem\(\)](#).

Referenced by [hmac\\_file\(\)](#), [hmac\\_memory\(\)](#), [hmac\\_memory\\_multi\(\)](#), and [pkcs\\_5\\_alg2\(\)](#).

```

31 {
32     unsigned char *buf;
33     unsigned long hashsize;
34     unsigned long i, z;
35     int err;
36
37     LTC_ARGCHK(hmac != NULL);
38     LTC_ARGCHK(key != NULL);
39
40     /* valid hash? */
41     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
42         return err;
43     }
44     hmac->hash = hash;
45     hashsize = hash_descriptor[hash].hashsize;
46
47     /* valid key length? */
48     if (keylen == 0) {
49         return CRYPT_INVALID_KEYSIZE;
50     }
51
52     /* allocate ram for buf */
53     buf = XMALLOC(HMAC_BLOCKSIZE);
54     if (buf == NULL) {
55         return CRYPT_MEM;
56     }
57
58     /* allocate memory for key */
59     hmac->key = XMALLOC(HMAC_BLOCKSIZE);
60     if (hmac->key == NULL) {
61         XFREE(buf);
62         return CRYPT_MEM;
63     }
64
65     /* (1) make sure we have a large enough key */
66     if(keylen > HMAC_BLOCKSIZE) {
67         z = HMAC_BLOCKSIZE;
68         if ((err = hash_memory(hash, key, keylen, hmac->key, &z)) != CRYPT_OK) {
69             goto LBL_ERR;
70         }
71         if(hashsize < HMAC_BLOCKSIZE) {
72             zeromem((hmac->key) + hashsize, (size_t)(HMAC_BLOCKSIZE - hashsize));
73         }
74         keylen = hashsize;
75     } else {
76         XMEMCPY(hmac->key, key, (size_t)keylen);
77         if(keylen < HMAC_BLOCKSIZE) {
78             zeromem((hmac->key) + keylen, (size_t)(HMAC_BLOCKSIZE - keylen));
79         }
80     }
81
82     /* Create the initial vector for step (3) */
83     for(i=0; i < HMAC_BLOCKSIZE; i++) {
84         buf[i] = hmac->key[i] ^ 0x36;
85     }
86
87     /* Pre-pend that to the hash data */
88     if ((err = hash_descriptor[hash].init(&hmac->md)) != CRYPT_OK) {
89         goto LBL_ERR;
90     }
91
92     if ((err = hash_descriptor[hash].process(&hmac->md, buf, HMAC_BLOCKSIZE)) != CRYPT_OK) {
93         goto LBL_ERR;
94     }
95     goto done;
96 LBL_ERR:
97     /* free the key since we failed */

```

```
98     XFREE(hmac->key);
99 done:
100 #ifdef LTC_CLEAN_STACK
101     zeromem(buf, HMAC_BLOCKSIZE);
102 #endif
103
104     XFREE(buf);
105     return err;
106 }
```

Here is the call graph for this function:



## 5.95 mac/hmac/hmac\_memory.c File Reference

### 5.95.1 Detailed Description

HMAC support, process a block of memory, Tom St Denis/Dobes Vandermeer.

Definition in file [hmac\\_memory.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for hmac\_memory.c:

### Functions

- [int hmac\\_memory](#) (int *hash*, const unsigned char \**key*, unsigned long *keylen*, const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*HMAC a block of memory to produce the authentication tag.*

### 5.95.2 Function Documentation

#### 5.95.2.1 int hmac\_memory (int *hash*, const unsigned char \* *key*, unsigned long *keylen*, const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

HMAC a block of memory to produce the authentication tag.

#### Parameters:

*hash* The index of the hash to use

*key* The secret key

*keylen* The length of the secret key (octets)

*in* The data to HMAC

*inlen* The length of the data to HMAC (octets)

*out* [out] Destination of the authentication tag

*outlen* [in/out] Max size and resulting size of authentication tag

#### Returns:

CRYPT\_OK if successful

Definition at line 31 of file hmac\_memory.c.

References CRYPT\_MEM, CRYPT\_OK, hash\_descriptor, hash\_is\_valid(), ltc\_hash\_descriptor::hmac\_block, hmac\_done(), hmac\_init(), hmac\_process(), LTC\_ARGCHK, XFREE, XMALLOC, and zeromem().

Referenced by hmac\_test(), and pkcs\_5\_alg2().

```
35 {
36     hmac_state *hmac;
37     int         err;
38
39     LTC_ARGCHK(key    != NULL);
40     LTC_ARGCHK(in     != NULL);
41     LTC_ARGCHK(out     != NULL);
```

```
42     LTC_ARGCHK(outlen != NULL);
43
44     /* make sure hash descriptor is valid */
45     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
46         return err;
47     }
48
49     /* is there a descriptor? */
50     if (hash_descriptor[hash].hmac_block != NULL) {
51         return hash_descriptor[hash].hmac_block(key, keylen, in, inlen, out, outlen);
52     }
53
54     /* nope, so call the hmac functions */
55     /* allocate ram for hmac state */
56     hmac = XMALLOC(sizeof(hmac_state));
57     if (hmac == NULL) {
58         return CRYPT_MEM;
59     }
60
61     if ((err = hmac_init(hmac, hash, key, keylen)) != CRYPT_OK) {
62         goto LBL_ERR;
63     }
64
65     if ((err = hmac_process(hmac, in, inlen)) != CRYPT_OK) {
66         goto LBL_ERR;
67     }
68
69     if ((err = hmac_done(hmac, out, outlen)) != CRYPT_OK) {
70         goto LBL_ERR;
71     }
72
73     err = CRYPT_OK;
74 LBL_ERR:
75 #ifdef LTC_CLEAN_STACK
76     zeromem(hmac, sizeof(hmac_state));
77 #endif
78
79     XFREE(hmac);
80     return err;
81 }
```

Here is the call graph for this function:

## 5.96 mac/hmac/hmac\_memory\_multi.c File Reference

### 5.96.1 Detailed Description

HMAC support, process multiple blocks of memory, Tom St Denis/Dobes Vandermeer.

Definition in file [hmac\\_memory\\_multi.c](#).

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for hmac\_memory\_multi.c:

### Functions

- [int hmac\\_memory\\_multi](#) (int *hash*, const unsigned char \**key*, unsigned long *keylen*, unsigned char \**out*, unsigned long \**outlen*, const unsigned char \**in*, unsigned long *inlen*,...)

*HMAC multiple blocks of memory to produce the authentication tag.*

### 5.96.2 Function Documentation

**5.96.2.1** `int hmac_memory_multi (int hash, const unsigned char * key, unsigned long keylen, unsigned char * out, unsigned long * outlen, const unsigned char * in, unsigned long inlen, ...)`

HMAC multiple blocks of memory to produce the authentication tag.

#### Parameters:

***hash*** The index of the hash to use

***key*** The secret key

***keylen*** The length of the secret key (octets)

***out*** [out] Destination of the authentication tag

***outlen*** [in/out] Max size and resulting size of authentication tag

***in*** The data to HMAC

***inlen*** The length of the data to HMAC (octets)

... tuples of (data,len) pairs to HMAC, terminated with a (NULL,x) (x=don't care)

#### Returns:

CRYPT\_OK if successful

Definition at line 33 of file hmac\_memory\_multi.c.

References CRYPT\_MEM, CRYPT\_OK, hmac\_init(), hmac\_process(), LTC\_ARGCHK, and XMALLOC.

```
38 {
39     hmac_state      *hmac;
40     int             err;
41     va_list         args;
42     const unsigned char *curptr;
43     unsigned long    curlen;
```

```
44
45     LTC_ARGCHK(key      != NULL);
46     LTC_ARGCHK(in       != NULL);
47     LTC_ARGCHK(out       != NULL);
48     LTC_ARGCHK(outlen    != NULL);
49
50     /* allocate ram for hmac state */
51     hmac = XMALLOC(sizeof(hmac_state));
52     if (hmac == NULL) {
53         return CRYPT_MEM;
54     }
55
56     if ((err = hmac_init(hmac, hash, key, keylen)) != CRYPT_OK) {
57         goto LBL_ERR;
58     }
59
60     va_start(args, inlen);
61     curptr = in;
62     curlen = inlen;
63     for (;;) {
64         /* process buf */
65         if ((err = hmac_process(hmac, curptr, curlen)) != CRYPT_OK) {
66             goto LBL_ERR;
67         }
68         /* step to next */
69         curptr = va_arg(args, const unsigned char*);
70         if (curptr == NULL) {
71             break;
72         }
73         curlen = va_arg(args, unsigned long);
74     }
75     if ((err = hmac_done(hmac, out, outlen)) != CRYPT_OK) {
76         goto LBL_ERR;
77     }
78 LBL_ERR:
79 #ifdef LTC_CLEAN_STACK
80     zeromem(hmac, sizeof(hmac_state));
81 #endif
82     XFREE(hmac);
83     va_end(args);
84     return err;
85 }
```

Here is the call graph for this function:

## 5.97 mac/hmac/hmac\_process.c File Reference

### 5.97.1 Detailed Description

HMAC support, process data, Tom St Denis/Dobes Vandermeer.

Definition in file [hmac\\_process.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for hmac\_process.c:

### Functions

- [int hmac\\_process](#) (hmac\_state \*hmac, const unsigned char \*in, unsigned long inlen)  
*Process data through HMAC.*

### 5.97.2 Function Documentation

#### 5.97.2.1 int hmac\_process (hmac\_state \*hmac, const unsigned char \*in, unsigned long inlen)

Process data through HMAC.

#### Parameters:

**hmac** The hmac state

**in** The data to send through HMAC

**inlen** The length of the data to HMAC (octets)

#### Returns:

CRYPT\_OK if successful

Definition at line 27 of file hmac\_process.c.

References [CRYPT\\_OK](#), [hash\\_descriptor](#), [hash\\_is\\_valid\(\)](#), [LTC\\_ARGCHK](#), and [ltc\\_hash\\_descriptor::process](#).

Referenced by [hmac\\_file\(\)](#), [hmac\\_memory\(\)](#), [hmac\\_memory\\_multi\(\)](#), and [pkcs\\_5\\_alg2\(\)](#).

```
28 {
29     int err;
30     LTC_ARGCHK(hmac != NULL);
31     LTC_ARGCHK(in != NULL);
32     if ((err = hash_is_valid(hmac->hash)) != CRYPT_OK) {
33         return err;
34     }
35     return hash_descriptor[hmac->hash].process(&hmac->md, in, inlen);
36 }
```

Here is the call graph for this function:

## 5.98 mac/hmac/hmac\_test.c File Reference

### 5.98.1 Detailed Description

HMAC support, self-test, Tom St Denis/Dobes Vandermeer.

Definition in file [hmac\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for hmac\_test.c:

### Defines

- #define [HMAC\\_BLOCKSIZE](#) [hash\\_descriptor\[hash\].blocksize](#)

### Functions

- int [hmac\\_test](#) (void)  
*HMAC self-test.*

### 5.98.2 Define Documentation

#### 5.98.2.1 #define HMAC\_BLOCKSIZE [hash\\_descriptor\[hash\].blocksize](#)

Definition at line 20 of file hmac\_test.c.

### 5.98.3 Function Documentation

#### 5.98.3.1 int hmac\_test (void)

HMAC self-test.

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if tests have been disabled.

Definition at line 37 of file hmac\_test.c.

References [CRYPT\\_NOP](#), [CRYPT\\_OK](#), [error\\_to\\_string\(\)](#), [find\\_hash\(\)](#), [hmac\\_memory\(\)](#), and [MAXBLOCKSIZE](#).

```
38 {
39     #ifndef LTC_TEST
40         return CRYPT_NOP;
41     #else
42         unsigned char digest[MAXBLOCKSIZE];
43         int i;
44
45         static const struct hmac_test_case {
46             int num;
47             char *algo;
48             unsigned char key[128];
49             unsigned long keylen;
```

```

50     unsigned char data[128];
51     unsigned long datalen;
52     unsigned char digest[MAXBLOCKSIZE];
53 } cases[] = {
54     /*
55     3. Test Cases for HMAC-SHA-1
56
57     test_case =      1
58     key =            0x0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c
59     key_len =        20
60     data =            "Hi Ther      20
61     digest =          0x4c1a03424b55e07fe7f27be1d58bb9324a9a5a04
62     digest-96 =       0x4c1a03424b55e07fe7f27be1
63     */
64     { 5, "sha1",
65         {0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
66          0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
67          0x0c, 0x0c, 0x0c, 0x0c}, 20,
68         "Test With Truncation", 20,
69         {0x4c, 0x1a, 0x03, 0x42, 0x4b, 0x55, 0xe0, 0x7f, 0xe7, 0xf2,
70          0x7b, 0xe1, 0xd5, 0x8b, 0xb9, 0x32, 0x4a, 0x9a, 0x5a, 0x04} },
71
72     /*
73     test_case =      6
74     key =            0xaa repeated 80 times
75     key_len =        80
76     data =            "Test Using Larger Than Block-Size Key - Hash Key First"
77     data_len =       54
78     digest =          0xaa4ae5e15272d00e95705637ce8a3b55ed402112
79     */
80     { 6, "sha1",
81         {0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
82          0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
83          0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
84          0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
85          0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
86          0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
87          0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
88          0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
89          0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
90          0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa}, 80,
91         "Test Using Larger Than Block-Size Key - Hash Key First", 54,
92         {0xaa, 0x4a, 0xe5, 0xe1, 0x52, 0x72, 0xd0, 0x0e,
93          0x95, 0x70, 0x56, 0x37, 0xce, 0x8a, 0x3b, 0x55,
94          0xed, 0x40, 0x21, 0x12} },
95
96     /*
97     test_case =      7
98     key =            0xaa repeated 80 times
99     key_len =        80
100    data =            "Test Using Larger Than Block-Size Key and Larger
101                      Than One Block-Size Data"
102    data_len =        73
103    digest =          0xe8e99d0f45237d786d6bbaa7965c7808bbffa91
104    */
105    { 7, "sha1",
106        {0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
107         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
108         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
109         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
110         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
111         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
112         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
113         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
114         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
115         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa}, 80,
116        "Test Using Larger Than Block-Size Key and Larger Than One Block-Size Data", 73,

```

```

117         {0xe8, 0xe9, 0x9d, 0x0f, 0x45, 0x23, 0x7d, 0x78, 0x6d,
118         0x6b, 0xba, 0xa7, 0x96, 0x5c, 0x78, 0x08, 0xbb, 0xff, 0x1a, 0x91} },
119
120     /*
121     2. Test Cases for HMAC-MD5
122
123     test_case =      1
124     key =            0x0b 0b 0b 0b
125                     0b 0b 0b 0b
126                     0b 0b 0b 0b
127                     0b 0b 0b 0b
128     key_len =        16
129     data =            "Hi There"
130     data_len =        8
131     digest =          0x92 94 72 7a
132                     36 38 bb 1c
133                     13 f4 8e f8
134                     15 8b fc 9d
135
136     { 1, "md5",
137       {0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b,
138        0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b, 0x0b}, 16,
139       "Hi There", 8,
140       {0x92, 0x94, 0x72, 0x7a, 0x36, 0x38, 0xbb, 0x1c,
141        0x13, 0xf4, 0x8e, 0xf8, 0x15, 0x8b, 0xfc, 0x9d} },
142
143     /*
144     test_case =      2
145     key =            "Jefe"
146     key_len =         4
147     data =            "what do ya want for nothing?"
148     data_len =        28
149     digest =          0x750c783e6ab0b503eaa86e310a5db738
150
151     { 2, "md5",
152       "Jefe", 4,
153       "what do ya want for nothing?", 28,
154       {0x75, 0x0c, 0x78, 0x3e, 0x6a, 0xb0, 0xb5, 0x03,
155        0xea, 0xa8, 0x6e, 0x31, 0x0a, 0x5d, 0xb7, 0x38} },
156
157     /*
158     test_case =      3
159     key =            0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
160     key_len =         16
161     data =            0xdd repeated 50 times
162     data_len =        50
163     digest =          0x56be34521d144c88dbb8c733f0e8b3f6
164
165     { 3, "md5",
166       {0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
167        0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa}, 16,
168       {0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd,
169        0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd,
170        0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd,
171        0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd, 0xdd}, 50,
172       {0x56, 0xbe, 0x34, 0x52, 0x1d, 0x14, 0x4c, 0x88,
173        0xdb, 0xb8, 0xc7, 0x33, 0xf0, 0xe8, 0xb3, 0xf6} },
174
175     /*
176     test_case =      4
177     key = 0x0102030405060708090a0b0c0d0e0f10111213141516171819
178     key_len =         25
179     data =            0xcd repeated 50 times
180     data_len =        50
181     digest =          0x697eaf0aca3a3aea3a75164746ffaa79
182
183     { 4, "md5",

```



```

184         {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
185          0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, 0x14,
186          0x15, 0x16, 0x17, 0x18, 0x19}, 25,
187         {0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd,
188          0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd,
189          0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd,
190          0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd,
191          0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd, 0xcd}, 50,
192         {0x69, 0x7e, 0xaf, 0x0a, 0xca, 0x3a, 0x3a, 0xea,
193          0x3a, 0x75, 0x16, 0x47, 0x46, 0xff, 0xaa, 0x79} },
194
195
196     /*
197
198     test_case =      5
199     key =            0x0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c
200     key_len =        16
201     data =           "Test With Truncation"
202     data_len =       20
203     digest =         0x56461ef2342edc00f9bab995690efd4c
204     digest-96        0x56461ef2342edc00f9bab995
205     */
206     { 5, "md5",
207       {0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
208        0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c}, 16,
209       "Test With Truncation", 20,
210       {0x56, 0x46, 0x1e, 0xf2, 0x34, 0x2e, 0xdc, 0x00,
211        0xf9, 0xba, 0xb9, 0x95, 0x69, 0x0e, 0xfd, 0x4c} },
212
213     /*
214
215     test_case =      6
216     key =            0xaa repeated 80 times
217     key_len =        80
218     data =           "Test Using Larger Than Block-Size Key - Hash
219 Key First"
220     data_len =       54
221     digest =         0x6b1ab7fe4bd7bf8f0b62e6ce61b9d0cd
222     */
223     { 6, "md5",
224       {0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
225        0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
226        0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
227        0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
228        0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
229
230        0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
231        0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
232        0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
233        0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
234        0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa}, 80,
235       "Test Using Larger Than Block-Size Key - Hash Key First", 54,
236       {0x6b, 0x1a, 0xb7, 0xfe, 0x4b, 0xd7, 0xbf, 0x8f,
237        0x0b, 0x62, 0xe6, 0xce, 0x61, 0xb9, 0xd0, 0xcd} },
238
239     /*
240
241     test_case =      7
242     key =            0xaa repeated 80 times
243     key_len =        80
244     data =           "Test Using Larger Than Block-Size Key and Larger
245 Than One Block-Size Data"
246     data_len =       73
247     digest =         0x6f630fad67cda0ee1fb1f562db3aa53e
248     */
249     { 7, "md5",
250       {0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,

```

```

251         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
252         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
253         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
254         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
255         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
256         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
257         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
258         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa,
259         0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa}, 80,
260         "Test Using Larger Than Block-Size Key and Larger Than One Block-Size Data", 73,
261         {0x6f, 0x63, 0x0f, 0xad, 0x67, 0xcd, 0xa0, 0xee,
262          0x1f, 0xb1, 0xf5, 0x62, 0xdb, 0x3a, 0xa5, 0x3e} }
263     };
264
265     unsigned long outlen;
266     int err;
267     int tested=0, failed=0;
268     for(i=0; i < (int)(sizeof(cases) / sizeof(cases[0])); i++) {
269         int hash = find_hash(cases[i].algo);
270         if (hash == -1) continue;
271         ++tested;
272         outlen = sizeof(digest);
273         if((err = hmac_memory(hash, cases[i].key, cases[i].keylen, cases[i].data, cases[i].datalen, digest, &outlen)) != 0)
274             #if 0
275                 printf("HMAC-%s test #%d, %s\n", cases[i].algo, cases[i].num, error_to_string(err));
276             #endif
277         return err;
278     }
279
280     if(XMEMCMP(digest, cases[i].digest, (size_t)hash_descriptor[hash].hashsize) != 0) {
281         failed++;
282         #if 0
283             unsigned int j;
284             printf("\nHMAC-%s test #%d:\n", cases[i].algo, cases[i].num);
285             printf("Result: 0x");
286             for(j=0; j < hash_descriptor[hash].hashsize; j++) {
287                 printf("%2x ", digest[j]);
288             }
289             printf("\nCorrect: 0x");
290             for(j=0; j < hash_descriptor[hash].hashsize; j++) {
291                 printf("%2x ", cases[i].digest[j]);
292             }
293             printf("\n");
294             return CRYPT_ERROR;
295         #endif
296     } else {
297         /* printf("HMAC-%s test #%d: Passed\n", cases[i].algo, cases[i].num); */
298     }
299 }
300
301 if (failed != 0) {
302     return CRYPT_FAIL_TESTVECTOR;
303 } else if (tested == 0) {
304     return CRYPT_NOP;
305 } else {
306     return CRYPT_OK;
307 }
308 #endif
309 }

```

Here is the call graph for this function:

## 5.99 mac/omac/omac\_done.c File Reference

### 5.99.1 Detailed Description

OMAC1 support, terminate a stream, Tom St Denis.

Definition in file [omac\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [omac\\_done.c](#):

### Functions

- `int omac\_done (omac_state *omac, unsigned char *out, unsigned long *outlen)`

*Terminate an OMAC stream.*

### 5.99.2 Function Documentation

#### 5.99.2.1 `int omac\_done (omac_state *omac, unsigned char *out, unsigned long *outlen)`

Terminate an OMAC stream.

#### Parameters:

*omac* The OMAC state

*out* [out] Destination for the authentication tag

*outlen* [in/out] The max size and resulting size of the authentication tag

#### Returns:

CRYPT\_OK if successful

Definition at line 27 of file [omac\\_done.c](#).

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::done](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), [LTC\\_ARGCHK](#), and [zeromem\(\)](#).

Referenced by [eax\\_done\(\)](#), [eax\\_init\(\)](#), [omac\\_file\(\)](#), and [omac\\_memory\(\)](#).

```
28 {
29     int      err, mode;
30     unsigned x;
31
32     LTC_ARGCHK(omac != NULL);
33     LTC_ARGCHK(out != NULL);
34     LTC_ARGCHK(outlen != NULL);
35     if ((err = cipher_is_valid(omac->cipher_idx)) != CRYPT_OK) {
36         return err;
37     }
38
39     if ((omac->buflen > (int)sizeof(omac->block)) || (omac->buflen < 0) ||
40         (omac->blklen > (int)sizeof(omac->block)) || (omac->buflen > omac->blklen)) {
41         return CRYPT_INVALID_ARG;
42     }
43
44     /* figure out mode */
```

```
45     if (omac->buflen != omac->blklen) {
46         /* add the 0x80 byte */
47         omac->block[omac->buflen++] = 0x80;
48
49         /* pad with 0x00 */
50         while (omac->buflen < omac->blklen) {
51             omac->block[omac->buflen++] = 0x00;
52         }
53         mode = 1;
54     } else {
55         mode = 0;
56     }
57
58     /* now xor prev + Lu[mode] */
59     for (x = 0; x < (unsigned)omac->blklen; x++) {
60         omac->block[x] ^= omac->prev[x] ^ omac->Lu[mode][x];
61     }
62
63     /* encrypt it */
64     if ((err = cipher_descriptor[omac->cipher_idx].ecb_encrypt(omac->block, omac->block, &omac->key)) != 0)
65         return err;
66     }
67     cipher_descriptor[omac->cipher_idx].done(&omac->key);
68
69     /* output it */
70     for (x = 0; x < (unsigned)omac->blklen && x < *outlen; x++) {
71         out[x] = omac->block[x];
72     }
73     *outlen = x;
74
75 #ifdef LTC_CLEAN_STACK
76     zeromem(omac, sizeof(*omac));
77 #endif
78     return CRYPT_OK;
79 }
```

Here is the call graph for this function:

## 5.100 mac/omac/omac\_file.c File Reference

### 5.100.1 Detailed Description

OMAC1 support, process a file, Tom St Denis.

Definition in file [omac\\_file.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for omac\_file.c:

### Functions

- [int omac\\_file](#) (int cipher, const unsigned char \*key, unsigned long keylen, const char \*filename, unsigned char \*out, unsigned long \*outlen)  
*OMAC a file.*

### 5.100.2 Function Documentation

#### 5.100.2.1 int omac\_file (int cipher, const unsigned char \*key, unsigned long keylen, const char \*filename, unsigned char \*out, unsigned long \*outlen)

OMAC a file.

#### Parameters:

- cipher* The index of the cipher desired
- key* The secret key
- keylen* The length of the secret key (octets)
- filename* The name of the file you wish to OMAC
- out* [out] Where the authentication tag is to be stored
- outlen* [in/out] The max size and resulting size of the authentication tag

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if file support has been disabled

Definition at line 30 of file omac\_file.c.

References CRYPT\_FILE\_NOTFOUND, CRYPT\_NOP, CRYPT\_OK, in, LTC\_ARGCHK, omac\_done(), omac\_init(), omac\_process(), and zeromem().

```
34 {
35 #ifdef LTC_NO_FILE
36     return CRYPT_NOP;
37 #else
38     int err, x;
39     omac_state omac;
40     FILE *in;
41     unsigned char buf[512];
42
43     LTC_ARGCHK(key != NULL);
44     LTC_ARGCHK(filename != NULL);
45     LTC_ARGCHK(out != NULL);
```

```
46 LTC_ARGCHK(outlen != NULL);
47
48 in = fopen(filename, "rb");
49 if (in == NULL) {
50     return CRYPT_FILE_NOTFOUND;
51 }
52
53 if ((err = omac_init(&omac, cipher, key, keylen)) != CRYPT_OK) {
54     fclose(in);
55     return err;
56 }
57
58 do {
59     x = fread(buf, 1, sizeof(buf), in);
60     if ((err = omac_process(&omac, buf, x)) != CRYPT_OK) {
61         fclose(in);
62         return err;
63     }
64 } while (x == sizeof(buf));
65 fclose(in);
66
67 if ((err = omac_done(&omac, out, outlen)) != CRYPT_OK) {
68     return err;
69 }
70
71 #ifdef LTC_CLEAN_STACK
72     zeromem(buf, sizeof(buf));
73 #endif
74
75     return CRYPT_OK;
76 #endif
77 }
```

Here is the call graph for this function:

## 5.101 mac/omac/omac\_init.c File Reference

### 5.101.1 Detailed Description

OMAC1 support, initialize state, by Tom St Denis.

Definition in file [omac\\_init.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `omac_init.c`:

### Functions

- `int omac_init` (`omac_state *omac`, `int cipher`, `const unsigned char *key`, `unsigned long keylen`)  
*Initialize an OMAC state.*

### 5.101.2 Function Documentation

#### 5.101.2.1 `int omac_init` (`omac_state *omac`, `int cipher`, `const unsigned char *key`, `unsigned long keylen`)

Initialize an OMAC state.

#### Parameters:

- omac* The OMAC state to initialize
- cipher* The index of the desired cipher
- key* The secret key
- keylen* The length of the secret key (octets)

#### Returns:

- CRYPT\_OK if successful

Definition at line 29 of file `omac_init.c`.

References `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `ecb_encrypt()`, `len`, `LTC_ARGCHK`, `mask`, and `zeromem()`.

Referenced by `eax_init()`, `omac_file()`, `omac_memory()`, and `omac_memory_multi()`.

```
30 {
31     int err, x, y, mask, msb, len;
32
33     LTC_ARGCHK(omac != NULL);
34     LTC_ARGCHK(key != NULL);
35
36     /* schedule the key */
37     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
38         return err;
39     }
40
41 #ifdef LTC_FAST
42     if (cipher_descriptor[cipher].block_length % sizeof(LTC_FAST_TYPE)) {
43         return CRYPT_INVALID_ARG;
44     }
45 }
```

```

45 #endif
46
47 /* now setup the system */
48 switch (cipher_descriptor[cipher].block_length) {
49     case 8: mask = 0x1B;
50             len = 8;
51             break;
52     case 16: mask = 0x87;
53             len = 16;
54             break;
55     default: return CRYPT_INVALID_ARG;
56 }
57
58 if ((err = cipher_descriptor[cipher].setup(key, keylen, 0, &omac->key)) != CRYPT_OK) {
59     return err;
60 }
61
62 /* ok now we need Lu and Lu^2 [calc one from the other] */
63
64 /* first calc L which is Ek(0) */
65 zeromem(omac->Lu[0], cipher_descriptor[cipher].block_length);
66 if ((err = cipher_descriptor[cipher].ecb_encrypt(omac->Lu[0], omac->Lu[0], &omac->key)) != CRYPT_OK)
67     return err;
68 }
69
70 /* now do the mults, whoopy! */
71 for (x = 0; x < 2; x++) {
72     /* if msb(L * u^(x+1)) = 0 then just shift, otherwise shift and xor constant mask */
73     msb = omac->Lu[x][0] >> 7;
74
75     /* shift left */
76     for (y = 0; y < (len - 1); y++) {
77         omac->Lu[x][y] = ((omac->Lu[x][y] << 1) | (omac->Lu[x][y+1] >> 7)) & 255;
78     }
79     omac->Lu[x][len - 1] = ((omac->Lu[x][len - 1] << 1) ^ (msb ? mask : 0)) & 255;
80
81     /* copy up as require */
82     if (x == 0) {
83         XMEMCPY(omac->Lu[1], omac->Lu[0], sizeof(omac->Lu[0]));
84     }
85 }
86
87 /* setup state */
88 omac->cipher_idx = cipher;
89 omac->buflen = 0;
90 omac->blklen = len;
91 zeromem(omac->prev, sizeof(omac->prev));
92 zeromem(omac->block, sizeof(omac->block));
93
94 return CRYPT_OK;
95 }

```

Here is the call graph for this function:



## 5.102 mac/omac/omac\_memory.c File Reference

### 5.102.1 Detailed Description

OMAC1 support, process a block of memory, Tom St Denis.

Definition in file [omac\\_memory.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [omac\\_memory.c](#):

### Functions

- [int omac\\_memory](#) (int *cipher*, const unsigned char \**key*, unsigned long *keylen*, const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)  
*OMAC a block of memory.*

### 5.102.2 Function Documentation

**5.102.2.1** [int omac\\_memory](#) (int *cipher*, const unsigned char \* *key*, unsigned long *keylen*, const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

OMAC a block of memory.

#### Parameters:

*cipher* The index of the desired cipher

*key* The secret key

*keylen* The length of the secret key (octets)

*in* The data to send through OMAC

*inlen* The length of the data to send through OMAC (octets)

*out* [out] The destination of the authentication tag

*outlen* [in/out] The max size and resulting size of the authentication tag (octets)

#### Returns:

CRYPT\_OK if successful

Definition at line 31 of file [omac\\_memory.c](#).

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), [omac\\_done\(\)](#), [omac\\_init\(\)](#), [ltc\\_cipher\\_descriptor::omac\\_memory](#), [omac\\_process\(\)](#), [XFREE](#), [XMALLOC](#), and [zeromem\(\)](#).

Referenced by [omac\\_test\(\)](#).

```
35 {
36     int err;
37     omac_state *omac;
38
39     LTC_ARGCHK(key    != NULL);
40     LTC_ARGCHK(in     != NULL);
41     LTC_ARGCHK(out    != NULL);
```

```
42 LTC_ARGCHK(outlen != NULL);
43
44 /* is the cipher valid? */
45 if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
46     return err;
47 }
48
49 /* Use accelerator if found */
50 if (cipher_descriptor[cipher].omac_memory != NULL) {
51     return cipher_descriptor[cipher].omac_memory(key, keylen, in, inlen, out, outlen);
52 }
53
54 /* allocate ram for omac state */
55 omac = XMALLOC(sizeof(omac_state));
56 if (omac == NULL) {
57     return CRYPT_MEM;
58 }
59
60 /* omac process the message */
61 if ((err = omac_init(omac, cipher, key, keylen)) != CRYPT_OK) {
62     goto LBL_ERR;
63 }
64 if ((err = omac_process(omac, in, inlen)) != CRYPT_OK) {
65     goto LBL_ERR;
66 }
67 if ((err = omac_done(omac, out, outlen)) != CRYPT_OK) {
68     goto LBL_ERR;
69 }
70
71 err = CRYPT_OK;
72 LBL_ERR:
73 #ifdef LTC_CLEAN_STACK
74     zeromem(omac, sizeof(omac_state));
75 #endif
76
77 XFREE(omac);
78 return err;
79 }
```

Here is the call graph for this function:

## 5.103 mac/omac/omac\_memory\_multi.c File Reference

### 5.103.1 Detailed Description

OMAC1 support, process multiple blocks of memory, Tom St Denis.

Definition in file [omac\\_memory\\_multi.c](#).

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for `omac_memory_multi.c`:

### Functions

- `int` [omac\\_memory\\_multi](#) (`int` cipher, `const unsigned char *`key, `unsigned long` keylen, `unsigned char *`out, `unsigned long *`outlen, `const unsigned char *`in, `unsigned long` inlen,...)

*OMAC multiple blocks of memory.*

### 5.103.2 Function Documentation

**5.103.2.1** `int` `omac_memory_multi` (`int` cipher, `const unsigned char *`key, `unsigned long` keylen, `unsigned char *`out, `unsigned long *`outlen, `const unsigned char *`in, `unsigned long` inlen, ...)

OMAC multiple blocks of memory.

#### Parameters:

**cipher** The index of the desired cipher

**key** The secret key

**keylen** The length of the secret key (octets)

**out** [out] The destination of the authentication tag

**outlen** [in/out] The max size and resulting size of the authentication tag (octets)

**in** The data to send through OMAC

**inlen** The length of the data to send through OMAC (octets)

... tuples of (data,len) pairs to OMAC, terminated with a (NULL,x) (x=don't care)

#### Returns:

CRYPT\_OK if successful

Definition at line 33 of file `omac_memory_multi.c`.

References CRYPT\_MEM, CRYPT\_OK, LTC\_ARGCHK, `omac_init()`, `omac_process()`, and XMALLOC.

```
37 {
38     int                err;
39     omac_state         *omac;
40     va_list            args;
41     const unsigned char *curptr;
42     unsigned long       curlen;
```

```
43
44     LTC_ARGCHK(key    != NULL);
45     LTC_ARGCHK(in     != NULL);
46     LTC_ARGCHK(out    != NULL);
47     LTC_ARGCHK(outlen != NULL);
48
49     /* allocate ram for omac state */
50     omac = XMALLOC(sizeof(omac_state));
51     if (omac == NULL) {
52         return CRYPT_MEM;
53     }
54
55     /* omac process the message */
56     if ((err = omac_init(omac, cipher, key, keylen)) != CRYPT_OK) {
57         goto LBL_ERR;
58     }
59     va_start(args, inlen);
60     curptr = in;
61     curlen = inlen;
62     for (;;) {
63         /* process buf */
64         if ((err = omac_process(omac, curptr, curlen)) != CRYPT_OK) {
65             goto LBL_ERR;
66         }
67         /* step to next */
68         curptr = va_arg(args, const unsigned char*);
69         if (curptr == NULL) {
70             break;
71         }
72         curlen = va_arg(args, unsigned long);
73     }
74     if ((err = omac_done(omac, out, outlen)) != CRYPT_OK) {
75         goto LBL_ERR;
76     }
77 LBL_ERR:
78 #ifdef LTC_CLEAN_STACK
79     zeromem(omac, sizeof(omac_state));
80 #endif
81     XFREE(omac);
82     va_end(args);
83     return err;
84 }
```

Here is the call graph for this function:

## 5.104 mac/omac/omac\_process.c File Reference

### 5.104.1 Detailed Description

OMAC1 support, process data, Tom St Denis.

Definition in file [omac\\_process.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [omac\\_process.c](#):

### Functions

- [int omac\\_process](#) (omac\_state \*omac, const unsigned char \*in, unsigned long inlen)  
*Process data through OMAC.*

### 5.104.2 Function Documentation

#### 5.104.2.1 int omac\_process (omac\_state \*omac, const unsigned char \*in, unsigned long inlen)

Process data through OMAC.

#### Parameters:

- omac** The OMAC state
- in** The input data to send through OMAC
- inlen** The length of the input (octets)

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file [omac\\_process.c](#).

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), and [LTC\\_ARGCHK](#).

Referenced by [eax\\_addheader\(\)](#), [eax\\_decrypt\(\)](#), [eax\\_encrypt\(\)](#), [eax\\_init\(\)](#), [omac\\_file\(\)](#), [omac\\_memory\(\)](#), and [omac\\_memory\\_multi\(\)](#).

```
29 {
30     unsigned long n, x;
31     int          err;
32
33     LTC_ARGCHK(omac != NULL);
34     LTC_ARGCHK(in  != NULL);
35     if ((err = cipher_is_valid(omac->cipher_idx)) != CRYPT_OK) {
36         return err;
37     }
38
39     if ((omac->buflen > (int)sizeof(omac->block)) || (omac->buflen < 0) ||
40         (omac->blklen > (int)sizeof(omac->block)) || (omac->buflen > omac->blklen)) {
41         return CRYPT_INVALID_ARG;
42     }
43
44 #ifdef LTC_FAST
```

```
45     if (omac->buflen == 0 && inlen > 16) {
46         int y;
47         for (x = 0; x < (inlen - 16); x += 16) {
48             for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
49                 *((LTC_FAST_TYPE*)&omac->prev[y]) ^= *((LTC_FAST_TYPE*)&in[y]);
50             }
51             in += 16;
52             if ((err = cipher_descriptor[omac->cipher_idx].ecb_encrypt(omac->prev, omac->prev, &omac->key)) != 0)
53                 return err;
54         }
55     }
56     inlen -= x;
57 }
58 #endif
59
60 while (inlen != 0) {
61     /* ok if the block is full we xor in prev, encrypt and replace prev */
62     if (omac->buflen == omac->blklen) {
63         for (x = 0; x < (unsigned long)omac->blklen; x++) {
64             omac->block[x] ^= omac->prev[x];
65         }
66         if ((err = cipher_descriptor[omac->cipher_idx].ecb_encrypt(omac->block, omac->prev, &omac->key)) != 0)
67             return err;
68     }
69     omac->buflen = 0;
70 }
71
72 /* add bytes */
73 n = MIN(inlen, (unsigned long)(omac->blklen - omac->buflen));
74 XMEMCPY(omac->block + omac->buflen, in, n);
75 omac->buflen += n;
76 inlen      -= n;
77 in         += n;
78 }
79
80 return CRYPT_OK;
81 }
```

Here is the call graph for this function:

## 5.105 mac/omac/omac\_test.c File Reference

### 5.105.1 Detailed Description

OMAC1 support, self-test, by Tom St Denis.

Definition in file [omac\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for omac\_test.c:

### Functions

- [int omac\\_test](#) (void)  
*Test the OMAC setup.*

### 5.105.2 Function Documentation

#### 5.105.2.1 int omac\_test (void)

Test the OMAC setup.

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if tests have been disabled

Definition at line 24 of file omac\_test.c.

References CRYPT\_NOP, CRYPT\_OK, find\_cipher(), len, and omac\_memory().

```
25 {
26 #if !defined(LTC_TEST)
27     return CRYPT_NOP;
28 #else
29     static const struct {
30         int keylen, msglen;
31         unsigned char key[16], msg[64], tag[16];
32     } tests[] = {
33     { 16, 0,
34       { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
35         0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
36       { 0x00 },
37       { 0xbb, 0x1d, 0x69, 0x29, 0xe9, 0x59, 0x37, 0x28,
38         0x7f, 0xa3, 0x7d, 0x12, 0x9b, 0x75, 0x67, 0x46 }
39     },
40     { 16, 16,
41       { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
42         0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
43       { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
44         0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a },
45       { 0x07, 0x0a, 0x16, 0xb4, 0x6b, 0x4d, 0x41, 0x44,
46         0xf7, 0x9b, 0xdd, 0x9d, 0xd0, 0x4a, 0x28, 0x7c }
47     },
48     { 16, 40,
49       { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
50         0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
51       { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
52         0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
```

```

53         0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
54         0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
55         0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11 },
56     { 0xdf, 0xa6, 0x67, 0x47, 0xde, 0x9a, 0xe6, 0x30,
57       0x30, 0xca, 0x32, 0x61, 0x14, 0x97, 0xc8, 0x27 }
58 },
59 { 16, 64,
60   { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
61     0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c },
62   { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
63     0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
64     0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
65     0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
66     0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11,
67     0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
68     0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17,
69     0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10 },
70   { 0x51, 0xf0, 0xbe, 0xbf, 0x7e, 0x3b, 0x9d, 0x92,
71     0xfc, 0x49, 0x74, 0x17, 0x79, 0x36, 0x3c, 0xfe }
72 },
73
74 };
75 unsigned char out[16];
76 int x, err, idx;
77 unsigned long len;
78
79
80 /* AES can be under rijndael or aes... try to find it */
81 if ((idx = find_cipher("aes")) == -1) {
82     if ((idx = find_cipher("rijndael")) == -1) {
83         return CRYPT_NOP;
84     }
85 }
86
87 for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
88     len = sizeof(out);
89     if ((err = omac_memory(idx, tests[x].key, tests[x].keylen, tests[x].msg, tests[x].msglen, out, &len)) != 0)
90         return err;
91 }
92
93 if (XMEMCMP(out, tests[x].tag, 16) != 0) {
94 #if 0
95     int y;
96     printf("\n\nTag: ");
97     for (y = 0; y < 16; y++) printf("%02x", out[y]); printf("\n\n");
98 #endif
99     return CRYPT_FAIL_TESTVECTOR;
100 }
101 }
102 return CRYPT_OK;
103 #endif
104 }

```

Here is the call graph for this function:



## 5.106 mac/pelican/pelican.c File Reference

### 5.106.1 Detailed Description

Pelican MAC, initialize state, by Tom St Denis.

Definition in file [pelican.c](#).

```
#include "tomcrypt.h"
#include "../ciphers/aes/aes_tab.c"
```

Include dependency graph for pelican.c:

### Defines

- #define [ENCRYPT\\_ONLY](#)
- #define [PELI\\_TAB](#)

### Functions

- int [pelican\\_init](#) (pelican\_state \*pelmac, const unsigned char \*key, unsigned long keylen)  
*Initialize a Pelican state.*
- static void [four\\_rounds](#) (pelican\_state \*pelmac)
- int [pelican\\_process](#) (pelican\_state \*pelmac, const unsigned char \*in, unsigned long inlen)  
*Process a block of text through Pelican.*
- int [pelican\\_done](#) (pelican\_state \*pelmac, unsigned char \*out)  
*Terminate Pelican MAC.*

### 5.106.2 Define Documentation

#### 5.106.2.1 #define ENCRYPT\_ONLY

Definition at line 20 of file pelican.c.

#### 5.106.2.2 #define PELI\_TAB

Definition at line 21 of file pelican.c.

### 5.106.3 Function Documentation

#### 5.106.3.1 static void four\_rounds (pelican\_state \*pelmac) [static]

Definition at line 55 of file pelican.c.

References byte, t1, t2, t3, Te0, Te1, Te2, and Te3.

Referenced by pelican\_done(), and pelican\_process().

```

56 {
57     ulong32 s0, s1, s2, s3, t0, t1, t2, t3;
58     int r;
59
60     LOAD32H(s0, pelmac->state      );
61     LOAD32H(s1, pelmac->state + 4);
62     LOAD32H(s2, pelmac->state + 8);
63     LOAD32H(s3, pelmac->state + 12);
64     for (r = 0; r < 4; r++) {
65         t0 =
66             Te0(byte(s0, 3)) ^
67             Te1(byte(s1, 2)) ^
68             Te2(byte(s2, 1)) ^
69             Te3(byte(s3, 0));
70         t1 =
71             Te0(byte(s1, 3)) ^
72             Te1(byte(s2, 2)) ^
73             Te2(byte(s3, 1)) ^
74             Te3(byte(s0, 0));
75         t2 =
76             Te0(byte(s2, 3)) ^
77             Te1(byte(s3, 2)) ^
78             Te2(byte(s0, 1)) ^
79             Te3(byte(s1, 0));
80         t3 =
81             Te0(byte(s3, 3)) ^
82             Te1(byte(s0, 2)) ^
83             Te2(byte(s1, 1)) ^
84             Te3(byte(s2, 0));
85         s0 = t0; s1 = t1; s2 = t2; s3 = t3;
86     }
87     STORE32H(s0, pelmac->state      );
88     STORE32H(s1, pelmac->state + 4);
89     STORE32H(s2, pelmac->state + 8);
90     STORE32H(s3, pelmac->state + 12);
91 }

```

### 5.106.3.2 int pelican\_done (pelican\_state \* pelmac, unsigned char \* out)

Terminate Pelican MAC.

#### Parameters:

*pelmac* The Pelican MAC state

*out* [out] The TAG

#### Returns:

CRYPT\_OK on success

Definition at line 141 of file pelican.c.

References CRYPT\_INVALID\_ARG, CRYPT\_OK, four\_rounds(), and LTC\_ARGCHK.

Referenced by pelican\_memory().

```

142 {
143     LTC_ARGCHK(pelmac != NULL);
144     LTC_ARGCHK(out != NULL);
145
146     /* check range */
147     if (pelmac->buflen < 0 || pelmac->buflen > 16) {
148         return CRYPT_INVALID_ARG;

```

```

149     }
150
151     if (pelmac->buflen == 16) {
152         four_rounds(pelmac);
153         pelmac->buflen = 0;
154     }
155     pelmac->state[pelmac->buflen++] ^= 0x80;
156     aes_ecb_encrypt(pelmac->state, out, &pelmac->K);
157     aes_done(&pelmac->K);
158     return CRYPT_OK;
159 }

```

Here is the call graph for this function:

### 5.106.3.3 int pelican\_init (pelican\_state \* *pelmac*, const unsigned char \* *key*, unsigned long *keylen*)

Initialize a Pelican state.

#### Parameters:

- pelmac* The Pelican state to initialize
- key* The secret key
- keylen* The length of the secret key (octets)

#### Returns:

CRYPT\_OK if successful

Definition at line 31 of file pelican.c.

References CRYPT\_INVALID\_ARG, CRYPT\_OK, LTC\_ARGCHK, and zeromem().

Referenced by pelican\_memory(), and pelican\_test().

```

32 {
33     int err;
34
35     LTC_ARGCHK(pelmac != NULL);
36     LTC_ARGCHK(key != NULL);
37
38     #ifdef LTC_FAST
39     if (16 % sizeof(LTC_FAST_TYPE)) {
40         return CRYPT_INVALID_ARG;
41     }
42     #endif
43
44     if ((err = aes_setup(key, keylen, 0, &pelmac->K)) != CRYPT_OK) {
45         return err;
46     }
47
48     zeromem(pelmac->state, 16);
49     aes_ecb_encrypt(pelmac->state, pelmac->state, &pelmac->K);
50     pelmac->buflen = 0;
51
52     return CRYPT_OK;
53 }

```

Here is the call graph for this function:

#### 5.106.3.4 int pelican\_process (pelican\_state \* *pelmac*, const unsigned char \* *in*, unsigned long *inlen*)

Process a block of text through Pelican.

##### Parameters:

*pelmac* The Pelican MAC state

*in* The input

*inlen* The length input (octets)

##### Returns:

CRYPT\_OK on success

Definition at line 100 of file pelican.c.

References CRYPT\_INVALID\_ARG, CRYPT\_OK, four\_rounds(), and LTC\_ARGCHK.

Referenced by pelican\_memory().

```

101 {
102     LTC_ARGCHK(pelmac != NULL);
103     LTC_ARGCHK(in != NULL);
104     /* check range */
105     if (pelmac->buflen < 0 || pelmac->buflen > 15) {
106         return CRYPT_INVALID_ARG;
107     }
108     #ifdef LTC_FAST
109     if (pelmac->buflen == 0) {
110         while (inlen & ~15) {
111             int x;
112             for (x = 0; x < 16; x += sizeof(LTC_FAST_TYPE)) {
113                 *((LTC_FAST_TYPE*)((unsigned char *)pelmac->state + x)) ^= *((LTC_FAST_TYPE*)((unsigned char *)in + x));
114             }
115             four_rounds(pelmac);
116             in += 16;
117             inlen -= 16;
118         }
119     }
120     #endif
121     while (inlen-- > 0) {
122         pelmac->state[pelmac->buflen++] ^= *in++;
123         if (pelmac->buflen == 16) {
124             four_rounds(pelmac);
125             pelmac->buflen = 0;
126         }
127     }
128     return CRYPT_OK;
129 }

```

Here is the call graph for this function:

## 5.107 mac/pelican/pelican\_memory.c File Reference

### 5.107.1 Detailed Description

Pelican MAC, MAC a block of memory, by Tom St Denis.

Definition in file [pelican\\_memory.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pelican\_memory.c:

### Functions

- [int pelican\\_memory](#) (const unsigned char \*key, unsigned long keylen, const unsigned char \*in, unsigned long inlen, unsigned char \*out)

*Pelican block of memory.*

### 5.107.2 Function Documentation

#### 5.107.2.1 int pelican\_memory (const unsigned char \*key, unsigned long keylen, const unsigned char \*in, unsigned long inlen, unsigned char \*out)

Pelican block of memory.

#### Parameters:

**key** The key for the MAC

**keylen** The length of the key (octets)

**in** The input to MAC

**inlen** The length of the input (octets)

**out** [out] The output TAG

#### Returns:

CRYPT\_OK on success

Definition at line 29 of file pelican\_memory.c.

References CRYPT\_MEM, CRYPT\_OK, pelican\_done(), pelican\_init(), pelican\_process(), XFREE, and XMALLOC.

```
32 {
33     pelican_state *pel;
34     int err;
35
36     pel = XMALLOC(sizeof(*pel));
37     if (pel == NULL) {
38         return CRYPT_MEM;
39     }
40
41     if ((err = pelican_init(pel, key, keylen)) != CRYPT_OK) {
42         XFREE(pel);
43         return err;
44     }
```

```
45     if ((err = pelican_process(pel, in ,inlen)) != CRYPT_OK) {
46         XFREE(pel);
47         return err;
48     }
49     err = pelican_done(pel, out);
50     XFREE(pel);
51     return err;
52 }
```

Here is the call graph for this function:

## 5.108 mac/pelican/pelican\_test.c File Reference

### 5.108.1 Detailed Description

Pelican MAC, test, by Tom St Denis.

Definition in file [pelican\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pelican\_test.c:

### Functions

- [int pelican\\_test](#) (void)

### 5.108.2 Function Documentation

#### 5.108.2.1 int pelican\_test (void)

Definition at line 20 of file pelican\_test.c.

References `CRYPT_NOP`, `CRYPT_OK`, `K`, and `pelican_init()`.

```
21 {
22 #ifndef LTC_TEST
23     return CRYPT_NOP;
24 #else
25     static const struct {
26         unsigned char K[32], MSG[64], T[16];
27         int keylen, ptlen;
28     } tests[] = {
29 /* K=16, M=0 */
30 {
31     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
32       0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F },
33     { 0 },
34     { 0xeb, 0x58, 0x37, 0x15, 0xf8, 0x34, 0xde, 0xe5,
35       0xa4, 0xd1, 0x6e, 0xe4, 0xb9, 0xd7, 0x76, 0x0e, },
36     16, 0
37 },
38
39 /* K=16, M=3 */
40 {
41     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
42       0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F },
43     { 0x00, 0x01, 0x02 },
44     { 0x1c, 0x97, 0x40, 0x60, 0x6c, 0x58, 0x17, 0x2d,
45       0x03, 0x94, 0x19, 0x70, 0x81, 0xc4, 0x38, 0x54, },
46     16, 3
47 },
48
49 /* K=16, M=16 */
50 {
51     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
52       0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F },
53     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
54       0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F },
55     { 0x03, 0xcc, 0x46, 0xb8, 0xac, 0xa7, 0x9c, 0x36,
56       0x1e, 0x8c, 0x6e, 0xa6, 0x7b, 0x89, 0x32, 0x49, },
57     16, 16
58 }
```

```

58 },
59
60 /* K=16, M=32 */
61 {
62     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
63       0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F },
64     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
65       0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
66       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
67       0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F },
68     { 0x89, 0xcc, 0x36, 0x58, 0x1b, 0xdd, 0x4d, 0xb5,
69       0x78, 0xbb, 0xac, 0xf0, 0xff, 0x8b, 0x08, 0x15, },
70     16, 32
71 },
72
73 /* K=16, M=35 */
74 {
75     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
76       0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F },
77     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
78       0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
79       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
80       0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
81       0x20, 0x21, 0x23 },
82     { 0x4a, 0x7d, 0x45, 0x4d, 0xcd, 0xb5, 0xda, 0x8d,
83       0x48, 0x78, 0x16, 0x48, 0x5d, 0x45, 0x95, 0x99, },
84     16, 35
85 },
86 };
87     int x, err;
88     unsigned char out[16];
89     pelican_state pel;
90
91     for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
92         if ((err = pelican_init(&pel, tests[x].K, tests[x].keylen)) != CRYPT_OK) {
93             return err;
94         }
95         if ((err = pelican_process(&pel, tests[x].MSG, tests[x].ptlen)) != CRYPT_OK) {
96             return err;
97         }
98         if ((err = pelican_done(&pel, out)) != CRYPT_OK) {
99             return err;
100         }
101
102         if (XMEMCMP(out, tests[x].T, 16)) {
103 #if 0
104             int y;
105             printf("\nFailed test %d\n", x);
106             printf("{ "; for (y = 0; y < 16; ) { printf("0x%02x, ", out[y]); if (!(++y & 7)) printf("\n"); }
107 #endif
108             return CRYPT_FAIL_TESTVECTOR;
109         }
110     }
111     return CRYPT_OK;
112 #endif
113 }

```

Here is the call graph for this function:



## 5.109 mac/pmac/pmac\_done.c File Reference

### 5.109.1 Detailed Description

PMAC implementation, terminate a session, by Tom St Denis.

Definition in file [pmac\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pmac\_done.c:

### Functions

- [int pmac\\_done](#) (pmac\_state \*state, unsigned char \*out, unsigned long \*outlen)

### 5.109.2 Function Documentation

#### 5.109.2.1 int pmac\_done (pmac\_state \*state, unsigned char \*out, unsigned long \*outlen)

Definition at line 20 of file pmac\_done.c.

References [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), and [LTC\\_ARGCHK](#).

Referenced by [pmac\\_file\(\)](#), and [pmac\\_memory\(\)](#).

```

21 {
22     int err, x;
23
24     LTC_ARGCHK(state != NULL);
25     LTC_ARGCHK(out != NULL);
26     if ((err = cipher_is_valid(state->cipher_idx)) != CRYPT_OK) {
27         return err;
28     }
29
30     if ((state->buflen > (int)sizeof(state->block)) || (state->buflen < 0) ||
31         (state->block_len > (int)sizeof(state->block)) || (state->buflen > state->block_len)) {
32         return CRYPT_INVALID_ARG;
33     }
34
35
36     /* handle padding.  If multiple xor in L/x */
37
38     if (state->buflen == state->block_len) {
39         /* xor Lr against the checksum */
40         for (x = 0; x < state->block_len; x++) {
41             state->checksum[x] ^= state->block[x] ^ state->Lr[x];
42         }
43     } else {
44         /* otherwise xor message bytes then the 0x80 byte */
45         for (x = 0; x < state->buflen; x++) {
46             state->checksum[x] ^= state->block[x];
47         }
48         state->checksum[x] ^= 0x80;
49     }
50
51     /* encrypt it */
52     if ((err = cipher_descriptor[state->cipher_idx].ecb_encrypt(state->checksum, state->checksum, &state->key)) != CRYPT_OK) {
53         return err;
54     }
55     cipher_descriptor[state->cipher_idx].done(&state->key);

```

```
56
57  /* store it */
58  for (x = 0; x < state->block_len && x < (int)*outlen; x++) {
59      out[x] = state->checksum[x];
60  }
61  *outlen = x;
62
63  #ifdef LTC_CLEAN_STACK
64      zeromem(state, sizeof(*state));
65  #endif
66  return CRYPT_OK;
67 }
```

Here is the call graph for this function:

## 5.110 mac/pmac/pmac\_file.c File Reference

### 5.110.1 Detailed Description

PMAC implementation, process a file, by Tom St Denis.

Definition in file [pmac\\_file.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pmac\_file.c:

### Functions

- [int pmac\\_file](#) (int *cipher*, const unsigned char \**key*, unsigned long *keylen*, const char \**filename*, unsigned char \**out*, unsigned long \**outlen*)  
*PMAC a file.*

### 5.110.2 Function Documentation

**5.110.2.1** `int pmac_file (int cipher, const unsigned char * key, unsigned long keylen, const char * filename, unsigned char * out, unsigned long * outlen)`

PMAC a file.

#### Parameters:

- cipher* The index of the cipher desired
- key* The secret key
- keylen* The length of the secret key (octets)
- filename* The name of the file to send through PMAC
- out* [out] Destination for the authentication tag
- outlen* [in/out] Max size and resulting size of the authentication tag

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if file support has been disabled

Definition at line 30 of file pmac\_file.c.

References CRYPT\_FILE\_NOTFOUND, CRYPT\_NOP, CRYPT\_OK, in, LTC\_ARGCHK, pmac\_done(), pmac\_init(), pmac\_process(), and zeromem().

```
34 {
35 #ifdef LTC_NO_FILE
36     return CRYPT_NOP;
37 #else
38     int err, x;
39     pmac_state pmac;
40     FILE *in;
41     unsigned char buf[512];
42
43
44     LTC_ARGCHK(key      != NULL);
45     LTC_ARGCHK(filename != NULL);
```

```
46 LTC_ARGCHK(out      != NULL);
47 LTC_ARGCHK(outlen   != NULL);
48
49 in = fopen(filename, "rb");
50 if (in == NULL) {
51     return CRYPT_FILE_NOTFOUND;
52 }
53
54 if ((err = pmac_init(&pmac, cipher, key, keylen)) != CRYPT_OK) {
55     fclose(in);
56     return err;
57 }
58
59 do {
60     x = fread(buf, 1, sizeof(buf), in);
61     if ((err = pmac_process(&pmac, buf, x)) != CRYPT_OK) {
62         fclose(in);
63         return err;
64     }
65 } while (x == sizeof(buf));
66 fclose(in);
67
68 if ((err = pmac_done(&pmac, out, outlen)) != CRYPT_OK) {
69     return err;
70 }
71
72 #ifdef LTC_CLEAN_STACK
73     zeromem(buf, sizeof(buf));
74 #endif
75
76     return CRYPT_OK;
77 #endif
78 }
```

Here is the call graph for this function:

## 5.111 mac/pmac/pmac\_init.c File Reference

### 5.111.1 Detailed Description

PMAC implementation, initialize state, by Tom St Denis.

Definition in file [pmac\\_init.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pmac\_init.c:

### Functions

- int [pmac\\_init](#) (pmac\_state \*pmac, int cipher, const unsigned char \*key, unsigned long keylen)  
*Initialize a PMAC state.*

### Variables

- struct {  
    int [len](#)  
    unsigned char [poly\\_div](#) [MAXBLOCKSIZE]  
    unsigned char [poly\\_mul](#) [MAXBLOCKSIZE]  
    int [code](#)  
    int [value](#)  
} [polys](#) []

### 5.111.2 Function Documentation

#### 5.111.2.1 int pmac\_init (pmac\_state \*pmac, int cipher, const unsigned char \*key, unsigned long keylen)

Initialize a PMAC state.

#### Parameters:

- pmac* The PMAC state to initialize  
*cipher* The index of the desired cipher  
*key* The secret key  
*keylen* The length of the secret key (octets)

#### Returns:

- CRYPT\_OK if successful

Definition at line 46 of file pmac\_init.c.

References [ltc\\_cipher\\_descriptor::block\\_length](#), [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_OK](#), [len](#), [LTC\\_ARGCHK](#), [poly](#), and [polys](#).

Referenced by [pmac\\_file\(\)](#), [pmac\\_memory\(\)](#), and [pmac\\_memory\\_multi\(\)](#).

```

47 {
48     int poly, x, y, m, err;
49     unsigned char *L;
50
51     LTC_ARGCHK(pmac != NULL);
52     LTC_ARGCHK(key != NULL);
53
54     /* valid cipher? */
55     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
56         return err;
57     }
58
59     /* determine which polys to use */
60     pmac->block_len = cipher_descriptor[cipher].block_length;
61     for (poly = 0; poly < (int)(sizeof(polys)/sizeof(polys[0])); poly++) {
62         if (polys[poly].len == pmac->block_len) {
63             break;
64         }
65     }
66     if (polys[poly].len != pmac->block_len) {
67         return CRYPT_INVALID_ARG;
68     }
69
70 #ifdef LTC_FAST
71     if (pmac->block_len % sizeof(LTC_FAST_TYPE)) {
72         return CRYPT_INVALID_ARG;
73     }
74 #endif
75
76
77     /* schedule the key */
78     if ((err = cipher_descriptor[cipher].setup(key, keylen, 0, &pmac->key)) != CRYPT_OK) {
79         return err;
80     }
81
82     /* allocate L */
83     L = XMALLOC(pmac->block_len);
84     if (L == NULL) {
85         return CRYPT_MEM;
86     }
87
88     /* find L = E[0] */
89     zeromem(L, pmac->block_len);
90     if ((err = cipher_descriptor[cipher].ecb_encrypt(L, L, &pmac->key)) != CRYPT_OK) {
91         goto error;
92     }
93
94     /* find Ls[i] = L << i for i == 0..31 */
95     XMEMCPY(pmac->Ls[0], L, pmac->block_len);
96     for (x = 1; x < 32; x++) {
97         m = pmac->Ls[x-1][0] >> 7;
98         for (y = 0; y < pmac->block_len-1; y++) {
99             pmac->Ls[x][y] = ((pmac->Ls[x-1][y] << 1) | (pmac->Ls[x-1][y+1] >> 7)) & 255;
100         }
101         pmac->Ls[x][pmac->block_len-1] = (pmac->Ls[x-1][pmac->block_len-1] << 1) & 255;
102
103         if (m == 1) {
104             for (y = 0; y < pmac->block_len; y++) {
105                 pmac->Ls[x][y] ^= polys[poly].poly_mul[y];
106             }
107         }
108     }
109
110     /* find Lr = L / x */
111     m = L[pmac->block_len-1] & 1;
112
113     /* shift right */

```

```

114     for (x = pmac->block_len - 1; x > 0; x--) {
115         pmac->Lr[x] = ((L[x] >> 1) | (L[x-1] << 7)) & 255;
116     }
117     pmac->Lr[0] = L[0] >> 1;
118
119     if (m == 1) {
120         for (x = 0; x < pmac->block_len; x++) {
121             pmac->Lr[x] ^= polys[poly].poly_div[x];
122         }
123     }
124
125     /* zero buffer, counters, etc... */
126     pmac->block_index = 1;
127     pmac->cipher_idx = cipher;
128     pmac->buflen = 0;
129     zeromem(pmac->block, sizeof(pmac->block));
130     zeromem(pmac->Li, sizeof(pmac->Li));
131     zeromem(pmac->checksum, sizeof(pmac->checksum));
132     err = CRYPT_OK;
133 error:
134 #ifdef LTC_CLEAN_STACK
135     zeromem(L, pmac->block_len);
136 #endif
137
138     XFREE(L);
139
140     return err;
141 }

```

Here is the call graph for this function:

### 5.111.3 Variable Documentation

#### 5.111.3.1 `int len`

Definition at line 21 of file pmac\_init.c.

#### 5.111.3.2 `unsigned char poly_div[MAXBLOCKSIZE]`

Definition at line 22 of file pmac\_init.c.

#### 5.111.3.3 `unsigned char poly_mul[MAXBLOCKSIZE]`

Definition at line 22 of file pmac\_init.c.

#### 5.111.3.4 `const { ... } polys[]` [static]

## 5.112 mac/pmac/pmac\_memory.c File Reference

### 5.112.1 Detailed Description

PMAC implementation, process a block of memory, by Tom St Denis.

Definition in file [pmac\\_memory.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pmac\_memory.c:

### Functions

- [int pmac\\_memory](#) (int cipher, const unsigned char \*key, unsigned long keylen, const unsigned char \*in, unsigned long inlen, unsigned char \*out, unsigned long \*outlen)

*PMAC a block of memory.*

### 5.112.2 Function Documentation

**5.112.2.1** `int pmac_memory (int cipher, const unsigned char *key, unsigned long keylen, const unsigned char *in, unsigned long inlen, unsigned char *out, unsigned long *outlen)`

PMAC a block of memory.

#### Parameters:

*cipher* The index of the cipher desired

*key* The secret key

*keylen* The length of the secret key (octets)

*in* The data you wish to send through PMAC

*inlen* The length of data you wish to send through PMAC (octets)

*out* [out] Destination for the authentication tag

*outlen* [in/out] The max size and resulting size of the authentication tag

#### Returns:

CRYPT\_OK if successful

Definition at line 31 of file pmac\_memory.c.

References CRYPT\_MEM, CRYPT\_OK, LTC\_ARGCHK, pmac\_done(), pmac\_init(), pmac\_process(), XFREE, XMALLOC, and zeromem().

Referenced by pmac\_test().

```
35 {
36     int err;
37     pmac_state *pmac;
38
39     LTC_ARGCHK(key != NULL);
40     LTC_ARGCHK(in != NULL);
41     LTC_ARGCHK(out != NULL);
42     LTC_ARGCHK(outlen != NULL);
```



```
43
44  /* allocate ram for pmac state */
45  pmac = XMALLOC(sizeof(pmac_state));
46  if (pmac == NULL) {
47      return CRYPT_MEM;
48  }
49
50  if ((err = pmac_init(pmac, cipher, key, keylen)) != CRYPT_OK) {
51      goto LBL_ERR;
52  }
53  if ((err = pmac_process(pmac, in, inlen)) != CRYPT_OK) {
54      goto LBL_ERR;
55  }
56  if ((err = pmac_done(pmac, out, outlen)) != CRYPT_OK) {
57      goto LBL_ERR;
58  }
59
60  err = CRYPT_OK;
61 LBL_ERR:
62 #ifdef LTC_CLEAN_STACK
63     zeromem(pmac, sizeof(pmac_state));
64 #endif
65
66     XFREE(pmac);
67     return err;
68 }
```

Here is the call graph for this function:

## 5.113 mac/pmac/pmac\_memory\_multi.c File Reference

### 5.113.1 Detailed Description

PMAC implementation, process multiple blocks of memory, by Tom St Denis.

Definition in file [pmac\\_memory\\_multi.c](#).

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for pmac\_memory\_multi.c:

### Functions

- int [pmac\\_memory\\_multi](#) (int cipher, const unsigned char \*key, unsigned long keylen, unsigned char \*out, unsigned long \*outlen, const unsigned char \*in, unsigned long inlen,...)

*PMAC multiple blocks of memory.*

### 5.113.2 Function Documentation

**5.113.2.1** int [pmac\\_memory\\_multi](#) (int *cipher*, const unsigned char \* *key*, unsigned long *keylen*, unsigned char \* *out*, unsigned long \* *outlen*, const unsigned char \* *in*, unsigned long *inlen*, ...)

PMAC multiple blocks of memory.

#### Parameters:

***cipher*** The index of the cipher desired

***key*** The secret key

***keylen*** The length of the secret key (octets)

***out*** [out] Destination for the authentication tag

***outlen*** [in/out] The max size and resulting size of the authentication tag

***in*** The data you wish to send through PMAC

***inlen*** The length of data you wish to send through PMAC (octets)

... tuples of (data,len) pairs to PMAC, terminated with a (NULL,x) (x=don't care)

#### Returns:

CRYPT\_OK if successful

Definition at line 33 of file pmac\_memory\_multi.c.

References CRYPT\_MEM, CRYPT\_OK, LTC\_ARGCHK, pmac\_init(), pmac\_process(), and XMALLOC.

```
37 {
38     int                err;
39     pmac_state          *pmac;
40     va_list             args;
41     const unsigned char *curptr;
42     unsigned long        curlen;
```

```
43
44     LTC_ARGCHK(key    != NULL);
45     LTC_ARGCHK(in     != NULL);
46     LTC_ARGCHK(out    != NULL);
47     LTC_ARGCHK(outlen != NULL);
48
49     /* allocate ram for pmac state */
50     pmac = XMALLOC(sizeof(pmac_state));
51     if (pmac == NULL) {
52         return CRYPT_MEM;
53     }
54
55     if ((err = pmac_init(pmac, cipher, key, keylen)) != CRYPT_OK) {
56         goto LBL_ERR;
57     }
58     va_start(args, inlen);
59     curptr = in;
60     curlen = inlen;
61     for (;;) {
62         /* process buf */
63         if ((err = pmac_process(pmac, curptr, curlen)) != CRYPT_OK) {
64             goto LBL_ERR;
65         }
66         /* step to next */
67         curptr = va_arg(args, const unsigned char*);
68         if (curptr == NULL) {
69             break;
70         }
71         curlen = va_arg(args, unsigned long);
72     }
73     if ((err = pmac_done(pmac, out, outlen)) != CRYPT_OK) {
74         goto LBL_ERR;
75     }
76 LBL_ERR:
77 #ifdef LTC_CLEAN_STACK
78     zeromem(pmac, sizeof(pmac_state));
79 #endif
80     XFREE(pmac);
81     va_end(args);
82     return err;
83 }
```

Here is the call graph for this function:

## 5.114 mac/pmac/pmac\_ntz.c File Reference

### 5.114.1 Detailed Description

PMAC implementation, internal function, by Tom St Denis.

Definition in file [pmac\\_ntz.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pmac\_ntz.c:

### Functions

- int [pmac\\_ntz](#) (unsigned long x)  
*Internal PMAC function.*

### 5.114.2 Function Documentation

#### 5.114.2.1 int pmac\_ntz (unsigned long x)

Internal PMAC function.

Definition at line 23 of file pmac\_ntz.c.

References [c](#).

Referenced by [pmac\\_shift\\_xor\(\)](#).

```
24 {  
25     int c;  
26     x &= 0xFFFFFFFFFUL;  
27     c = 0;  
28     while ((x & 1) == 0) {  
29         ++c;  
30         x >>= 1;  
31     }  
32     return c;  
33 }
```

## 5.115 mac/pmac/pmac\_process.c File Reference

### 5.115.1 Detailed Description

PMAC implementation, process data, by Tom St Denis.

Definition in file [pmac\\_process.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [pmac\\_process.c](#):

### Functions

- [int pmac\\_process](#) (pmac\_state \*pmac, const unsigned char \*in, unsigned long inlen)  
*Process data in a PMAC stream.*

### 5.115.2 Function Documentation

#### 5.115.2.1 int pmac\_process (pmac\_state \*pmac, const unsigned char \*in, unsigned long inlen)

Process data in a PMAC stream.

#### Parameters:

- pmac* The PMAC state
- in* The data to send through PMAC
- inlen* The length of the data to send through PMAC

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file [pmac\\_process.c](#).

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), [LTC\\_ARGCHK](#), [MAXBLOCKSIZE](#), and [pmac\\_shift\\_xor\(\)](#).

Referenced by [pmac\\_file\(\)](#), [pmac\\_memory\(\)](#), and [pmac\\_memory\\_multi\(\)](#).

```
29 {
30     int err, n;
31     unsigned long x;
32     unsigned char Z[MAXBLOCKSIZE];
33
34     LTC_ARGCHK(pmac != NULL);
35     LTC_ARGCHK(in != NULL);
36     if ((err = cipher_is_valid(pmac->cipher_idx)) != CRYPT_OK) {
37         return err;
38     }
39
40     if ((pmac->buflen > (int)sizeof(pmac->block)) || (pmac->buflen < 0) ||
41         (pmac->block_len > (int)sizeof(pmac->block)) || (pmac->buflen > pmac->block_len)) {
42         return CRYPT_INVALID_ARG;
43     }
44
45 #ifdef LTC_FAST
```

```

46     if (pmac->buflen == 0 && inlen > 16) {
47         unsigned long y;
48         for (x = 0; x < (inlen - 16); x += 16) {
49             pmac_shift_xor(pmac);
50             for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
51                 *((LTC_FAST_TYPE*) (&Z[y])) = *((LTC_FAST_TYPE*) (&in[y])) ^ *((LTC_FAST_TYPE*) (&pmac->Li[y]));
52             }
53             if ((err = cipher_descriptor[pmac->cipher_idx].ecb_encrypt(Z, Z, &pmac->key)) != CRYPT_OK) {
54                 return err;
55             }
56             for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
57                 *((LTC_FAST_TYPE*) (&pmac->checksum[y])) ^= *((LTC_FAST_TYPE*) (&Z[y]));
58             }
59             in += 16;
60         }
61         inlen -= x;
62     }
63 #endif
64
65     while (inlen != 0) {
66         /* ok if the block is full we xor in prev, encrypt and replace prev */
67         if (pmac->buflen == pmac->block_len) {
68             pmac_shift_xor(pmac);
69             for (x = 0; x < (unsigned long)pmac->block_len; x++) {
70                 Z[x] = pmac->Li[x] ^ pmac->block[x];
71             }
72             if ((err = cipher_descriptor[pmac->cipher_idx].ecb_encrypt(Z, Z, &pmac->key)) != CRYPT_OK) {
73                 return err;
74             }
75             for (x = 0; x < (unsigned long)pmac->block_len; x++) {
76                 pmac->checksum[x] ^= Z[x];
77             }
78             pmac->buflen = 0;
79         }
80
81         /* add bytes */
82         n = MIN(inlen, (unsigned long)(pmac->block_len - pmac->buflen));
83         XMEMCPY(pmac->block + pmac->buflen, in, n);
84         pmac->buflen += n;
85         inlen -= n;
86         in += n;
87     }
88
89 #ifdef LTC_CLEAN_STACK
90     zeromem(Z, sizeof(Z));
91 #endif
92
93     return CRYPT_OK;
94 }

```

Here is the call graph for this function:

## 5.116 mac/pmac/pmac\_shift\_xor.c File Reference

### 5.116.1 Detailed Description

PMAC implementation, internal function, by Tom St Denis.

Definition in file [pmac\\_shift\\_xor.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `pmac_shift_xor.c`:

### Functions

- void [pmac\\_shift\\_xor](#) (pmac\_state \*pmac)  
*Internal function.*

### 5.116.2 Function Documentation

#### 5.116.2.1 void `pmac_shift_xor` (pmac\_state \* *pmac*)

Internal function.

Performs the state update (adding correct multiple)

#### Parameters:

*pmac* The PMAC state.

Definition at line 24 of file `pmac_shift_xor.c`.

References `pmac_ntz()`.

Referenced by `pmac_process()`.

```
25 {
26     int x, y;
27     y = pmac_ntz(pmac->block_index++);
28 #ifdef LTC_FAST
29     for (x = 0; x < pmac->block_len; x += sizeof(LTC_FAST_TYPE)) {
30         *((LTC_FAST_TYPE*)((unsigned char *)pmac->Li + x)) ^=
31         *((LTC_FAST_TYPE*)((unsigned char *)pmac->Ls[y] + x));
32     }
33 #else
34     for (x = 0; x < pmac->block_len; x++) {
35         pmac->Li[x] ^= pmac->Ls[y][x];
36     }
37 #endif
38 }
```

Here is the call graph for this function:

## 5.117 mac/pmac/pmac\_test.c File Reference

### 5.117.1 Detailed Description

PMAC implementation, self-test, by Tom St Denis.

Definition in file [pmac\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pmac\_test.c:

### Functions

- [int pmac\\_test](#) (void)  
*Test the OMAC implementation.*

### 5.117.2 Function Documentation

#### 5.117.2.1 int pmac\_test (void)

Test the OMAC implementation.

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if testing has been disabled

Definition at line 25 of file pmac\_test.c.

References CRYPT\_NOP, CRYPT\_OK, find\_cipher(), len, MAXBLOCKSIZE, and pmac\_memory().

```
26 {
27 #if !defined(LTC_TEST)
28     return CRYPT_NOP;
29 #else
30     static const struct {
31         int msglen;
32         unsigned char key[16], msg[34], tag[16];
33     } tests[] = {
34
35         /* PMAC-AES-128-0B */
36     {
37         0,
38         /* key */
39         { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
40           0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
41         /* msg */
42         { 0x00 },
43         /* tag */
44         { 0x43, 0x99, 0x57, 0x2c, 0xd6, 0xea, 0x53, 0x41,
45           0xb8, 0xd3, 0x58, 0x76, 0xa7, 0x09, 0x8a, 0xf7 },
46     },
47
48         /* PMAC-AES-128-3B */
49     {
50         3,
51         /* key */
52         { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
53           0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
```



```

54  /* msg */
55  { 0x00, 0x01, 0x02 },
56  /* tag */
57  { 0x25, 0x6b, 0xa5, 0x19, 0x3c, 0x1b, 0x99, 0x1b,
58    0x4d, 0xf0, 0xc5, 0x1f, 0x38, 0x8a, 0x9e, 0x27 }
59 },
60
61  /* PMAC-AES-128-16B */
62 {
63     16,
64     /* key */
65     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
66       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
67     /* msg */
68     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
69       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
70     /* tag */
71     { 0xeb, 0xbd, 0x82, 0x2f, 0xa4, 0x58, 0xda, 0xf6,
72       0xdf, 0xda, 0xd7, 0xc2, 0x7d, 0xa7, 0x63, 0x38 }
73 },
74
75  /* PMAC-AES-128-20B */
76 {
77     20,
78     /* key */
79     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
80       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
81     /* msg */
82     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
83       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
84       0x10, 0x11, 0x12, 0x13 },
85     /* tag */
86     { 0x04, 0x12, 0xca, 0x15, 0x0b, 0xbf, 0x79, 0x05,
87       0x8d, 0x8c, 0x75, 0xa5, 0x8c, 0x99, 0x3f, 0x55 }
88 },
89
90  /* PMAC-AES-128-32B */
91 {
92     32,
93     /* key */
94     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
95       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
96     /* msg */
97     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
98       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
99       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
100     0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f },
101     /* tag */
102     { 0xe9, 0x7a, 0xc0, 0x4e, 0x9e, 0x5e, 0x33, 0x99,
103       0xce, 0x53, 0x55, 0xcd, 0x74, 0x07, 0xbc, 0x75 }
104 },
105
106  /* PMAC-AES-128-34B */
107 {
108     34,
109     /* key */
110     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
111       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
112     /* msg */
113     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
114       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
115       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
116       0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
117       0x20, 0x21 },
118     /* tag */
119     { 0x5c, 0xba, 0x7d, 0x5e, 0xb2, 0x4f, 0x7c, 0x86,
120       0xcc, 0xc5, 0x46, 0x04, 0xe5, 0x3d, 0x55, 0x12 }

```

```
121 }
122
123 };
124 int err, x, idx;
125 unsigned long len;
126 unsigned char outtag[MAXBLOCKSIZE];
127
128 /* AES can be under rijndael or aes... try to find it */
129 if ((idx = find_cipher("aes")) == -1) {
130     if ((idx = find_cipher("rijndael")) == -1) {
131         return CRYPT_NOP;
132     }
133 }
134
135 for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
136     len = sizeof(outtag);
137     if ((err = pmac_memory(idx, tests[x].key, 16, tests[x].msg, tests[x].msglen, outtag, &len)) != 0)
138         return err;
139 }
140
141 if (XMEMCMP(outtag, tests[x].tag, len)) {
142 #if 0
143     unsigned long y;
144     printf("\nTAG:\n");
145     for (y = 0; y < len; ) {
146         printf("0x%02x", outtag[y]);
147         if (y < len-1) printf(", ");
148         if (!(++y % 8)) printf("\n");
149     }
150 #endif
151     return CRYPT_FAIL_TESTVECTOR;
152 }
153 }
154 return CRYPT_OK;
155 #endif /* LTC_TEST */
156 }
```

Here is the call graph for this function:

## 5.118 mac/xcbc/xcbc\_done.c File Reference

### 5.118.1 Detailed Description

XCBC Support, terminate the state.

Definition in file [xcbc\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for xcbc\_done.c:

### Functions

- `int xcbc_done(xcbc_state *xcbc, unsigned char *out, unsigned long *outlen)`

*Terminate the XCBC-MAC state.*

### 5.118.2 Function Documentation

#### 5.118.2.1 `int xcbc_done(xcbc_state *xcbc, unsigned char *out, unsigned long *outlen)`

Terminate the XCBC-MAC state.

#### Parameters:

***xcbc*** XCBC state to terminate

***out*** [out] Destination for the MAC tag

***outlen*** [in/out] Destination size and final tag size Return CRYPT\_OK on success

Definition at line 26 of file xcbc\_done.c.

References `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, and `LTC_ARGCHK`.

Referenced by `xcbc_file()`, and `xcbc_memory()`.

```
27 {
28     int err, x;
29     LTC_ARGCHK(xcbc != NULL);
30     LTC_ARGCHK(out != NULL);
31
32     /* check structure */
33     if ((err = cipher_is_valid(xcbc->cipher)) != CRYPT_OK) {
34         return err;
35     }
36
37     if ((xcbc->blocksize > cipher_descriptor[xcbc->cipher].block_length) || (xcbc->blocksize < 0) ||
38         (xcbc->buflen > xcbc->blocksize) || (xcbc->buflen < 0)) {
39         return CRYPT_INVALID_ARG;
40     }
41
42     /* which key do we use? */
43     if (xcbc->buflen == xcbc->blocksize) {
44         /* k2 */
45         for (x = 0; x < xcbc->blocksize; x++) {
46             xcbc->IV[x] ^= xcbc->K[1][x];
47         }
48     }
49 }
```

```
48     } else {
49         xcbc->IV[xcbc->buflen] ^= 0x80;
50         /* k3 */
51         for (x = 0; x < xcbc->blocksize; x++) {
52             xcbc->IV[x] ^= xcbc->K[2][x];
53         }
54     }
55
56     /* encrypt */
57     cipher_descriptor[xcbc->cipher].ecb_encrypt(xcbc->IV, xcbc->IV, &xcbc->key);
58     cipher_descriptor[xcbc->cipher].done(&xcbc->key);
59
60     /* extract tag */
61     for (x = 0; x < xcbc->blocksize && (unsigned long)x < *outlen; x++) {
62         out[x] = xcbc->IV[x];
63     }
64     *outlen = x;
65
66 #ifdef LTC_CLEAN_STACK
67     zeromem(xcbc, sizeof(*xcbc));
68 #endif
69     return CRYPT_OK;
70 }
```

Here is the call graph for this function:

## 5.119 mac/xcbc/xcbc\_file.c File Reference

### 5.119.1 Detailed Description

XCBC support, process a file, Tom St Denis.

Definition in file [xcbc\\_file.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for xcbc\_file.c:

### Functions

- [int xcbc\\_file](#) (int cipher, const unsigned char \*key, unsigned long keylen, const char \*filename, unsigned char \*out, unsigned long \*outlen)  
*XCBC a file.*

### 5.119.2 Function Documentation

#### 5.119.2.1 int xcbc\_file (int cipher, const unsigned char \* key, unsigned long keylen, const char \* filename, unsigned char \* out, unsigned long \* outlen)

XCBC a file.

#### Parameters:

- cipher* The index of the cipher desired
- key* The secret key
- keylen* The length of the secret key (octets)
- filename* The name of the file you wish to XCBC
- out* [out] Where the authentication tag is to be stored
- outlen* [in/out] The max size and resulting size of the authentication tag

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if file support has been disabled

Definition at line 30 of file xcbc\_file.c.

References [CRYPT\\_FILE\\_NOTFOUND](#), [CRYPT\\_NOP](#), [CRYPT\\_OK](#), [in](#), [LTC\\_ARGCHK](#), [xcbc\\_done\(\)](#), [xcbc\\_init\(\)](#), [xcbc\\_process\(\)](#), and [zeromem\(\)](#).

```
34 {
35 #ifdef LTC_NO_FILE
36     return CRYPT_NOP;
37 #else
38     int err, x;
39     xcbc_state xcbc;
40     FILE *in;
41     unsigned char buf[512];
42
43     LTC_ARGCHK(key != NULL);
44     LTC_ARGCHK(filename != NULL);
45     LTC_ARGCHK(out != NULL);
```

```
46 LTC_ARGCHK(outlen != NULL);
47
48 in = fopen(filename, "rb");
49 if (in == NULL) {
50     return CRYPT_FILE_NOTFOUND;
51 }
52
53 if ((err = xcbc_init(&xcbc, cipher, key, keylen)) != CRYPT_OK) {
54     fclose(in);
55     return err;
56 }
57
58 do {
59     x = fread(buf, 1, sizeof(buf), in);
60     if ((err = xcbc_process(&xcbc, buf, x)) != CRYPT_OK) {
61         fclose(in);
62         return err;
63     }
64 } while (x == sizeof(buf));
65 fclose(in);
66
67 if ((err = xcbc_done(&xcbc, out, outlen)) != CRYPT_OK) {
68     return err;
69 }
70
71 #ifdef LTC_CLEAN_STACK
72     zeromem(buf, sizeof(buf));
73 #endif
74
75     return CRYPT_OK;
76 #endif
77 }
```

Here is the call graph for this function:

## 5.120 mac/xcbc/xcbc\_init.c File Reference

### 5.120.1 Detailed Description

XCBC Support, start an XCBC state.

Definition in file [xcbc\\_init.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for xcbc\_init.c:

### Functions

- `int xcbc_init(xcbc_state *xcbc, int cipher, const unsigned char *key, unsigned long keylen)`  
*Initialize XCBC-MAC state.*

### 5.120.2 Function Documentation

#### 5.120.2.1 `int xcbc_init(xcbc_state *xcbc, int cipher, const unsigned char *key, unsigned long keylen)`

Initialize XCBC-MAC state.

##### Parameters:

- xcbc*** [out] XCBC state to initialize
- cipher*** Index of cipher to use
- key*** [in] Secret key
- keylen*** Length of secret key in octets Return CRYPT\_OK on success

Definition at line 27 of file xcbc\_init.c.

References `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_MEM`, `CRYPT_OK`, `LTC_ARGCHK`, and `XCALLOC`.

Referenced by `xcbc_file()`, `xcbc_memory()`, and `xcbc_memory_multi()`.

```
28 {
29     int          x, y, err;
30     symmetric_key *skey;
31
32     LTC_ARGCHK(xcbc != NULL);
33     LTC_ARGCHK(key != NULL);
34
35     /* schedule the key */
36     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
37         return err;
38     }
39
40     #ifdef LTC_FAST
41     if (cipher_descriptor[cipher].block_length % sizeof(LTC_FAST_TYPE)) {
42         return CRYPT_INVALID_ARG;
43     }
44 #endif
45 }
```

```
46  /* schedule the user key */
47  skey = XCALLOC(1, sizeof(*skey));
48  if (skey == NULL) {
49      return CRYPT_MEM;
50  }
51
52  if ((err = cipher_descriptor[cipher].setup(key, keylen, 0, skey)) != CRYPT_OK) {
53      goto done;
54  }
55
56  /* make the three keys */
57  for (y = 0; y < 3; y++) {
58      for (x = 0; x < cipher_descriptor[cipher].block_length; x++) {
59          xcbc->K[y][x] = y + 1;
60      }
61      cipher_descriptor[cipher].ecb_encrypt(xcbc->K[y], xcbc->K[y], skey);
62  }
63
64  /* setup K1 */
65  err = cipher_descriptor[cipher].setup(xcbc->K[0], cipher_descriptor[cipher].block_length, 0, &xcbc->K[0]);
66
67  /* setup struct */
68  zeromem(xcbc->IV, cipher_descriptor[cipher].block_length);
69  xcbc->blocksize = cipher_descriptor[cipher].block_length;
70  xcbc->cipher     = cipher;
71  xcbc->buflen     = 0;
72 done:
73  cipher_descriptor[cipher].done(skey);
74 #ifdef LTC_CLEAN_STACK
75  zeromem(skey, sizeof(*skey));
76 #endif
77  XFREE(skey);
78  return err;
79 }
```

Here is the call graph for this function:



## 5.121 mac/xcbc/xcbc\_memory.c File Reference

```
#include "tomcrypt.h"
```

Include dependency graph for xcbc\_memory.c:

### Functions

- `int xcbc_memory` (int *cipher*, const unsigned char \**key*, unsigned long *keylen*, const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*XCBC-MAC a block of memory.*

### 5.121.1 Function Documentation

**5.121.1.1** `int xcbc_memory` (int *cipher*, const unsigned char \* *key*, unsigned long *keylen*, const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

XCBC-MAC a block of memory.

#### Parameters:

*cipher* Index of cipher to use

*key* [in] Secret key

*keylen* Length of key in octets

*in* [in] Message to MAC

*inlen* Length of input in octets

*out* [out] Destination for the MAC tag

*outlen* [in/out] Output size and final tag size Return CRYPT\_OK on success.

Definition at line 30 of file xcbc\_memory.c.

References cipher\_descriptor, cipher\_is\_valid(), CRYPT\_MEM, CRYPT\_OK, XCALLOC, xcbc\_done(), xcbc\_init(), ltc\_cipher\_descriptor::xcbc\_memory, xcbc\_process(), and XFREE.

Referenced by xcbc\_test().

```
34 {
35     xcbc_state *xcbc;
36     int      err;
37
38     /* is the cipher valid? */
39     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
40         return err;
41     }
42
43     /* Use accelerator if found */
44     if (cipher_descriptor[cipher].xcbc_memory != NULL) {
45         return cipher_descriptor[cipher].xcbc_memory(key, keylen, in, inlen, out, outlen);
46     }
47
48     xcbc = XCALLOC(1, sizeof(*xcbc));
49     if (xcbc == NULL) {
50         return CRYPT_MEM;
51     }
52 }
```

```
53     if ((err = xcbc_init(xcbc, cipher, key, keylen)) != CRYPT_OK) {
54         goto done;
55     }
56
57     if ((err = xcbc_process(xcbc, in, inlen)) != CRYPT_OK) {
58         goto done;
59     }
60
61     err = xcbc_done(xcbc, out, outlen);
62 done:
63     XFREE(xcbc);
64     return err;
65 }
```

Here is the call graph for this function:

## 5.122 mac/xcbc/xcbc\_memory\_multi.c File Reference

### 5.122.1 Detailed Description

XCBC support, process multiple blocks of memory, Tom St Denis.

Definition in file [xcbc\\_memory\\_multi.c](#).

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for xcbc\_memory\_multi.c:

### Functions

- [int xcbc\\_memory\\_multi](#) (int *cipher*, const unsigned char \**key*, unsigned long *keylen*, unsigned char \**out*, unsigned long \**outlen*, const unsigned char \**in*, unsigned long *inlen*,...)

*XCBC multiple blocks of memory.*

### 5.122.2 Function Documentation

**5.122.2.1** `int xcbc_memory_multi (int cipher, const unsigned char * key, unsigned long keylen, unsigned char * out, unsigned long * outlen, const unsigned char * in, unsigned long inlen, ...)`

XCBC multiple blocks of memory.

#### Parameters:

***cipher*** The index of the desired cipher

***key*** The secret key

***keylen*** The length of the secret key (octets)

***out*** [out] The destination of the authentication tag

***outlen*** [in/out] The max size and resulting size of the authentication tag (octets)

***in*** The data to send through XCBC

***inlen*** The length of the data to send through XCBC (octets)

... tuples of (data,len) pairs to XCBC, terminated with a (NULL,x) (x=don't care)

#### Returns:

CRYPT\_OK if successful

Definition at line 33 of file xcbc\_memory\_multi.c.

References CRYPT\_MEM, CRYPT\_OK, LTC\_ARGCHK, xcbc\_init(), xcbc\_process(), and XMALLOC.

```
37 {
38     int                err;
39     xcbc_state          *xcbc;
40     va_list             args;
41     const unsigned char *curptr;
42     unsigned long        curlen;
```

```
43
44     LTC_ARGCHK(key    != NULL);
45     LTC_ARGCHK(in     != NULL);
46     LTC_ARGCHK(out    != NULL);
47     LTC_ARGCHK(outlen != NULL);
48
49     /* allocate ram for xcbc state */
50     xcbc = XMALLOC(sizeof(xcbc_state));
51     if (xcbc == NULL) {
52         return CRYPT_MEM;
53     }
54
55     /* xcbc process the message */
56     if ((err = xcbc_init(xcbc, cipher, key, keylen)) != CRYPT_OK) {
57         goto LBL_ERR;
58     }
59     va_start(args, inlen);
60     curptr = in;
61     curlen = inlen;
62     for (;;) {
63         /* process buf */
64         if ((err = xcbc_process(xcbc, curptr, curlen)) != CRYPT_OK) {
65             goto LBL_ERR;
66         }
67         /* step to next */
68         curptr = va_arg(args, const unsigned char*);
69         if (curptr == NULL) {
70             break;
71         }
72         curlen = va_arg(args, unsigned long);
73     }
74     if ((err = xcbc_done(xcbc, out, outlen)) != CRYPT_OK) {
75         goto LBL_ERR;
76     }
77 LBL_ERR:
78 #ifdef LTC_CLEAN_STACK
79     zeromem(xcbc, sizeof(xcbc_state));
80 #endif
81     XFREE(xcbc);
82     va_end(args);
83     return err;
84 }
```

Here is the call graph for this function:

## 5.123 mac/xcbc/xcbc\_process.c File Reference

### 5.123.1 Detailed Description

XCBC Support, terminate the state.

Definition in file [xcbc\\_process.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for xcbc\_process.c:

### Functions

- [int xcbc\\_process](#) (xcbc\_state \*xcbc, const unsigned char \*[in](#), unsigned long inlen)  
*Process data through XCBC-MAC.*

### 5.123.2 Function Documentation

#### 5.123.2.1 int xcbc\_process (xcbc\_state \*xcbc, const unsigned char \*in, unsigned long inlen)

Process data through XCBC-MAC.

#### Parameters:

**xcbc** The XCBC-MAC state

**in** Input data to process

**inlen** Length of input in octets Return CRYPT\_OK on success

Definition at line 26 of file xcbc\_process.c.

References [ltc\\_cipher\\_descriptor::block\\_length](#), [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), and [LTC\\_ARGCHK](#).

Referenced by [xcbc\\_file\(\)](#), [xcbc\\_memory\(\)](#), and [xcbc\\_memory\\_multi\(\)](#).

```
27 {
28     int err;
29 #ifdef LTC_FAST
30     int x;
31 #endif
32
33     LTC_ARGCHK(xcbc != NULL);
34     LTC_ARGCHK(in != NULL);
35
36     /* check structure */
37     if ((err = cipher_is_valid(xcbc->cipher)) != CRYPT_OK) {
38         return err;
39     }
40
41     if ((xcbc->blocksize > cipher_descriptor[xcbc->cipher].block_length) || (xcbc->blocksize < 0) ||
42         (xcbc->buflen > xcbc->blocksize) || (xcbc->buflen < 0)) {
43         return CRYPT_INVALID_ARG;
44     }
45
46 #ifdef LTC_FAST
47     if (xcbc->buflen == 0) {
```

```
48     while (inlen > (unsigned long)xcbc->blocksize) {
49         for (x = 0; x < xcbc->blocksize; x += sizeof(LTC_FAST_TYPE)) {
50             *((LTC_FAST_TYPE*)&(xcbc->IV[x])) ^= *((LTC_FAST_TYPE*)&(in[x]));
51         }
52         cipher_descriptor[xcbc->cipher].ecb_encrypt(xcbc->IV, xcbc->IV, &xcbc->key);
53         in += xcbc->blocksize;
54         inlen -= xcbc->blocksize;
55     }
56 }
57 #endif
58
59 while (inlen) {
60     if (xcbc->buflen == xcbc->blocksize) {
61         cipher_descriptor[xcbc->cipher].ecb_encrypt(xcbc->IV, xcbc->IV, &xcbc->key);
62         xcbc->buflen = 0;
63     }
64     xcbc->IV[xcbc->buflen++] ^= *in++;
65     --inlen;
66 }
67 return CRYPT_OK;
68 }
```

Here is the call graph for this function:

## 5.124 mac/xcbc/xcbc\_test.c File Reference

### 5.124.1 Detailed Description

XCBC Support, terminate the state.

Definition in file [xcbc\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for xcbc\_test.c:

### Functions

- [int xcbc\\_test](#) (void)  
*Test XCBC-MAC mode Return CRYPT\_OK on succes.*

### 5.124.2 Function Documentation

#### 5.124.2.1 int xcbc\_test (void)

Test XCBC-MAC mode Return CRYPT\_OK on succes.

Definition at line 23 of file xcbc\_test.c.

References [CRYPT\\_NOP](#), [CRYPT\\_OK](#), [find\\_cipher\(\)](#), [K](#), and [xcbc\\_memory\(\)](#).

```

24 {
25 #ifdef LTC_NO_TEST
26     return CRYPT_NOP;
27 #else
28     static const struct {
29         int msglen;
30         unsigned char K[16], M[34], T[16];
31     } tests[] = {
32 {
33     0,
34     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
35       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
36
37     { 0 },
38
39     { 0x75, 0xf0, 0x25, 0x1d, 0x52, 0x8a, 0xc0, 0x1c,
40       0x45, 0x73, 0xdf, 0xd5, 0x84, 0xd7, 0x9f, 0x29 }
41 },
42
43 {
44     3,
45     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
46       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
47
48     { 0x00, 0x01, 0x02 },
49
50     { 0x5b, 0x37, 0x65, 0x80, 0xae, 0x2f, 0x19, 0xaf,
51       0xe7, 0x21, 0x9c, 0xee, 0xf1, 0x72, 0x75, 0x6f }
52 },
53
54 {
55     16,
56     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,

```

```

57     0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
58
59     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
60       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
61
62     { 0xd2, 0xa2, 0x46, 0xfa, 0x34, 0x9b, 0x68, 0xa7,
63       0x99, 0x98, 0xa4, 0x39, 0x4f, 0xf7, 0xa2, 0x63 }
64 },
65
66 {
67     32,
68     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
69       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
70
71     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
72       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
73       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
74       0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f },
75
76     { 0xf5, 0x4f, 0x0e, 0xc8, 0xd2, 0xb9, 0xf3, 0xd3,
77       0x68, 0x07, 0x73, 0x4b, 0xd5, 0x28, 0x3f, 0xd4 }
78 },
79
80 {
81     34,
82     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
83       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f },
84
85     { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
86       0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
87       0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
88       0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
89       0x20, 0x21 },
90
91     { 0xbe, 0xcb, 0xb3, 0xbc, 0xcd, 0xb5, 0x18, 0xa3,
92       0x06, 0x77, 0xd5, 0x48, 0x1f, 0xb6, 0xb4, 0xd8 }
93 },
94
95
96
97 };
98 unsigned char T[16];
99 unsigned long taglen;
100 int err, x, idx;
101
102 /* AES can be under rijndael or aes... try to find it */
103 if ((idx = find_cipher("aes")) == -1) {
104     if ((idx = find_cipher("rijndael")) == -1) {
105         return CRYPT_NOP;
106     }
107 }
108
109 for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
110     taglen = 16;
111     if ((err = xcbc_memory(idx, tests[x].K, 16, tests[x].M, tests[x].msglen, T, &taglen)) != CRYPT_OK)
112         return err;
113 }
114 if (taglen != 16 || XMEMCMP(T, tests[x].T, 16)) {
115     return CRYPT_FAIL_TESTVECTOR;
116 }
117 }
118
119 return CRYPT_OK;
120 #endif
121 }

```

Here is the call graph for this function:



## 5.125 math/fp/ltc\_ecc\_fp\_mulmod.c File Reference

### 5.125.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ltc\\_ecc\\_fp\\_mulmod.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ltc\_ecc\_fp\_mulmod.c:

## 5.126 math/gmp\_desc.c File Reference

```
#include "tomcrypt.h"
```

Include dependency graph for gmp\_desc.c:

### Defines

- #define [DESC\\_DEF\\_ONLY](#)

### 5.126.1 Define Documentation

#### 5.126.1.1 #define DESC\_DEF\_ONLY

Definition at line 12 of file gmp\_desc.c.

## 5.127 math/ltn\_desc.c File Reference

```
#include "tomcrypt.h"
```

Include dependency graph for ltn\_desc.c:

### Defines

- #define [DESC\\_DEF\\_ONLY](#)

#### 5.127.1 Define Documentation

##### 5.127.1.1 #define DESC\_DEF\_ONLY

Definition at line 12 of file ltn\_desc.c.

## 5.128 math/multi.c File Reference

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for multi.c:

### Functions

- int [ltc\\_init\\_multi](#) (void \*\*a,...)
- void [ltc\\_deinit\\_multi](#) (void \*a,...)

### 5.128.1 Function Documentation

#### 5.128.1.1 void [ltc\\_deinit\\_multi](#) (void \*a,...)

Definition at line 44 of file multi.c.

```
45 {  
46     void      *cur = a;  
47     va_list   args;  
48  
49     va_start(args, a);  
50     while (cur != NULL) {  
51         mp_clear(cur);  
52         cur = va_arg(args, void *);  
53     }  
54     va_end(args);  
55 }
```

#### 5.128.1.2 int [ltc\\_init\\_multi](#) (void \*\*a,...)

Definition at line 16 of file multi.c.

References [CRYPT\\_MEM](#), and [CRYPT\\_OK](#).

```
17 {  
18     void      **cur = a;  
19     int       np   = 0;  
20     va_list   args;  
21  
22     va_start(args, a);  
23     while (cur != NULL) {  
24         if (mp_init(cur) != CRYPT_OK) {  
25             /* failed */  
26             va_list clean_list;  
27  
28             va_start(clean_list, a);  
29             cur = a;  
30             while (np--) {  
31                 mp_clear(*cur);  
32                 cur = va_arg(clean_list, void**);  
33             }  
34             va_end(clean_list);  
35             return CRYPT_MEM;  
36         }  
37         ++np;
```

---

```
38     cur = va_arg(args, void**);
39 }
40 va_end(args);
41 return CRYPT_OK;
42 }
```

## 5.129 math/rand\_prime.c File Reference

### 5.129.1 Detailed Description

Generate a random prime, Tom St Denis.

Definition in file [rand\\_prime.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for rand\_prime.c:

### Defines

- #define [USE\\_BBS](#) 1

### Functions

- int [rand\\_prime](#) (void \*N, long [len](#), [prng\\_state](#) \*prng, int wprng)

### 5.129.2 Define Documentation

#### 5.129.2.1 #define USE\_BBS 1

Definition at line 18 of file rand\_prime.c.

Referenced by rand\_prime().

### 5.129.3 Function Documentation

#### 5.129.3.1 int rand\_prime (void \*N, long len, prng\_state \*prng, int wprng)

Definition at line 20 of file rand\_prime.c.

References [CRYPT\\_ERROR\\_READPRNG](#), [CRYPT\\_INVALID\\_PRIME\\_SIZE](#), [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), [prng\\_descriptor](#), [prng\\_is\\_valid\(\)](#), [USE\\_BBS](#), [XCALLOC](#), and [XFREE](#).

Referenced by [dsa\\_make\\_key\(\)](#), and [rsa\\_make\\_key\(\)](#).

```
21 {
22     int                err, res, type;
23     unsigned char *buf;
24
25     LTC_ARGCHK(N != NULL);
26
27     /* get type */
28     if (len < 0) {
29         type = USE_BBS;
30         len = -len;
31     } else {
32         type = 0;
33     }
34
35     /* allow sizes between 2 and 512 bytes for a prime size */
36     if (len < 2 || len > 512) {
37         return CRYPT_INVALID_PRIME_SIZE;
38     }
39 }
```

```
38     }
39
40     /* valid PRNG? Better be! */
41     if ((err = prng_is_valid(wprng)) != CRYPT_OK) {
42         return err;
43     }
44
45     /* allocate buffer to work with */
46     buf = XCALLOC(1, len);
47     if (buf == NULL) {
48         return CRYPT_MEM;
49     }
50
51     do {
52         /* generate value */
53         if (prng_descriptor[wprng].read(buf, len, prng) != (unsigned long)len) {
54             XFREE(buf);
55             return CRYPT_ERROR_READPRNG;
56         }
57
58         /* munge bits */
59         buf[0] |= 0x80 | 0x40;
60         buf[len-1] |= 0x01 | ((type & USE_BBS) ? 0x02 : 0x00);
61
62         /* load value */
63         if ((err = mp_read_unsigned_bin(N, buf, len)) != CRYPT_OK) {
64             XFREE(buf);
65             return err;
66         }
67
68         /* test */
69         if ((err = mp_prime_is_prime(N, 8, &res)) != CRYPT_OK) {
70             XFREE(buf);
71             return err;
72         }
73     } while (res == LTC_MP_NO);
74
75 #ifdef LTC_CLEAN_STACK
76     zeromem(buf, len);
77 #endif
78
79     XFREE(buf);
80     return CRYPT_OK;
81 }
```

Here is the call graph for this function:

## 5.130 math/tfm\_desc.c File Reference

```
#include "tomcrypt.h"
```

Include dependency graph for tfm\_desc.c:

### Defines

- #define [DESC\\_DEF\\_ONLY](#)

### 5.130.1 Define Documentation

#### 5.130.1.1 #define DESC\_DEF\_ONLY

Definition at line 12 of file tfm\_desc.c.



## 5.131 misc/base64/base64\_decode.c File Reference

### 5.131.1 Detailed Description

Compliant base64 code donated by Wayne Scott ([wscott@bitmover.com](mailto:wscott@bitmover.com)).

Definition in file [base64\\_decode.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for base64\_decode.c:

### Functions

- int [base64\\_decode](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*base64 decode a block of memory*

### Variables

- static const unsigned char [map](#) [256]

### 5.131.2 Function Documentation

#### 5.131.2.1 int base64\_decode (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

base64 decode a block of memory

#### Parameters:

*in* The base64 data to decode

*inlen* The length of the base64 data

*out* [out] The destination of the binary decoded data

*outlen* [in/out] The max size and resulting size of the decoded data

#### Returns:

CRYPT\_OK if successful

Definition at line 53 of file base64\_decode.c.

References [c](#), [CRYPT\\_INVALID\\_PACKET](#), [LTC\\_ARGCHK](#), and [map](#).

```
55 {
56     unsigned long t, x, y, z;
57     unsigned char c;
58     int          g;
59
60     LTC_ARGCHK(in      != NULL);
61     LTC_ARGCHK(out     != NULL);
62     LTC_ARGCHK(outlen != NULL);
63
64     g = 3;
```

```

65     for (x = y = z = t = 0; x < inlen; x++) {
66         c = map[in[x]&0xFF];
67         if (c == 255) continue;
68         /* the final = symbols are read and used to trim the remaining bytes */
69         if (c == 254) {
70             c = 0;
71             /* prevent g < 0 which would potentially allow an overflow later */
72             if (--g < 0) {
73                 return CRYPT_INVALID_PACKET;
74             }
75         } else if (g != 3) {
76             /* we only allow = to be at the end */
77             return CRYPT_INVALID_PACKET;
78         }
79
80         t = (t<<6)|c;
81
82         if (++y == 4) {
83             if (z + g > *outlen) {
84                 return CRYPT_BUFFER_OVERFLOW;
85             }
86             out[z++] = (unsigned char) ((t>>16)&255);
87             if (g > 1) out[z++] = (unsigned char) ((t>>8)&255);
88             if (g > 2) out[z++] = (unsigned char) (t&255);
89             y = t = 0;
90         }
91     }
92     if (y != 0) {
93         return CRYPT_INVALID_PACKET;
94     }
95     *outlen = z;
96     return CRYPT_OK;
97 }

```

### 5.131.3 Variable Documentation

#### 5.131.3.1 `const unsigned char map[256]` [static]

**Initial value:**

```

{
255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255, 255, 255, 255, 62, 255, 255, 255, 63,
52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 255, 255,
255, 254, 255, 255, 255, 0, 1, 2, 3, 4, 5, 6,
7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 255, 255, 255, 255, 255,
255, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
49, 50, 51, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
255, 255, 255, 255 }

```

Definition at line 21 of file base64\_decode.c.

Referenced by base64\_decode().

## 5.132 misc/base64/base64\_encode.c File Reference

### 5.132.1 Detailed Description

Compliant base64 encoder donated by Wayne Scott ([wscott@bitmover.com](mailto:wscott@bitmover.com)).

Definition in file [base64\\_encode.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for base64\_encode.c:

### Functions

- int [base64\\_encode](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*base64 Encode a buffer (NUL terminated)*

### Variables

- static const char \* [codes](#)

### 5.132.2 Function Documentation

#### 5.132.2.1 int base64\_encode (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

base64 Encode a buffer (NUL terminated)

#### Parameters:

- in* The input buffer to encode
- inlen* The length of the input buffer
- out* [out] The destination of the base64 encoded data
- outlen* [in/out] The max size and resulting size

#### Returns:

CRYPT\_OK if successful

Definition at line 32 of file base64\_encode.c.

References [codes](#), [CRYPT\\_BUFFER\\_OVERFLOW](#), and [LTC\\_ARGCHK](#).

```
34 {
35     unsigned long i, len2, leven;
36     unsigned char *p;
37
38     LTC_ARGCHK(in != NULL);
39     LTC_ARGCHK(out != NULL);
40     LTC_ARGCHK(outlen != NULL);
41
42     /* valid output size ? */
43     len2 = 4 * ((inlen + 2) / 3);
```

```
44     if (*outlen < len2 + 1) {
45         *outlen = len2 + 1;
46         return CRYPT_BUFFER_OVERFLOW;
47     }
48     p = out;
49     leven = 3*(inlen / 3);
50     for (i = 0; i < leven; i += 3) {
51         *p++ = codes[(in[0] >> 2) & 0x3F];
52         *p++ = codes[((in[0] & 3) << 4) + (in[1] >> 4) & 0x3F];
53         *p++ = codes[((in[1] & 0xf) << 2) + (in[2] >> 6) & 0x3F];
54         *p++ = codes[in[2] & 0x3F];
55         in += 3;
56     }
57     /* Pad it if necessary... */
58     if (i < inlen) {
59         unsigned a = in[0];
60         unsigned b = (i+1 < inlen) ? in[1] : 0;
61
62         *p++ = codes[(a >> 2) & 0x3F];
63         *p++ = codes[((a & 3) << 4) + (b >> 4) & 0x3F];
64         *p++ = (i+1 < inlen) ? codes[((b & 0xf) << 2) & 0x3F] : '=';
65         *p++ = '=';
66     }
67
68     /* append a NULL byte */
69     *p = '\0';
70
71     /* return ok */
72     *outlen = p - out;
73     return CRYPT_OK;
74 }
```

### 5.132.3 Variable Documentation

#### 5.132.3.1 `const char* codes` [static]

##### Initial value:

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

Definition at line 21 of file base64\_encode.c.

Referenced by base64\_encode().

## 5.133 misc/burn\_stack.c File Reference

### 5.133.1 Detailed Description

Burn stack, Tom St Denis.

Definition in file [burn\\_stack.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for burn\_stack.c:

### Functions

- void [burn\\_stack](#) (unsigned long *len*)  
*Burn some stack memory.*

### 5.133.2 Function Documentation

#### 5.133.2.1 void burn\_stack (unsigned long *len*)

Burn some stack memory.

#### Parameters:

*len* amount of stack to burn in bytes

Definition at line 22 of file burn\_stack.c.

References [burn\\_stack\(\)](#), and [zeromem\(\)](#).

Referenced by [burn\\_stack\(\)](#).

```
23 {  
24     unsigned char buf[32];  
25     zeromem(buf, sizeof(buf));  
26     if (len > (unsigned long)sizeof(buf))  
27         burn_stack(len - sizeof(buf));  
28 }
```

Here is the call graph for this function:

## 5.134 misc/crypt/crypt.c File Reference

### 5.134.1 Detailed Description

Build strings, Tom St Denis.

Definition in file [crypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt.c:

### Variables

- const char \* [crypt\\_build\\_settings](#)

### 5.134.2 Variable Documentation

#### 5.134.2.1 const char\* [crypt\\_build\\_settings](#)

Definition at line 18 of file crypt.c.

## 5.135 misc/crypt/crypt\_argchk.c File Reference

### 5.135.1 Detailed Description

Perform argument checking, Tom St Denis.

Definition in file [crypt\\_argchk.c](#).

```
#include "tomcrypt.h"
```

```
#include <signal.h>
```

Include dependency graph for crypt\_argchk.c:

### Functions

- void [crypt\\_argchk](#) (char \*v, char \*s, int d)

### 5.135.2 Function Documentation

#### 5.135.2.1 void [crypt\\_argchk](#) (char \* v, char \* s, int d)

Definition at line 20 of file crypt\_argchk.c.

```
21 {  
22     fprintf(stderr, "LTC_ARGCHK '%s' failure on line %d of file %s\n",  
23             v, d, s);  
24     (void)raise(SIGABRT);  
25 }
```



## 5.136 misc/crypt/crypt\_cipher\_descriptor.c File Reference

### 5.136.1 Detailed Description

Stores the cipher descriptor table, Tom St Denis.

Definition in file [crypt\\_cipher\\_descriptor.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_cipher\_descriptor.c:

### Variables

- [ltc\\_cipher\\_descriptor](#) [cipher\\_descriptor](#) [TAB\_SIZE]

### 5.136.2 Variable Documentation

#### 5.136.2.1 struct [ltc\\_cipher\\_descriptor](#) [cipher\\_descriptor](#)[TAB\_SIZE]

**Initial value:**

```
{  
{ NULL, 0, 0, 0, 0, 0, 0, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,  
}
```

Definition at line 18 of file crypt\_cipher\_descriptor.c.

## 5.137 misc/crypt/crypt\_cipher\_is\_valid.c File Reference

### 5.137.1 Detailed Description

Determine if cipher is valid, Tom St Denis.

Definition in file [crypt\\_cipher\\_is\\_valid.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_cipher\_is\_valid.c:

### Functions

- [int cipher\\_is\\_valid](#) (int idx)

### 5.137.2 Function Documentation

#### 5.137.2.1 int cipher\_is\_valid (int idx)

Definition at line 23 of file crypt\_cipher\_is\_valid.c.

References [cipher\\_descriptor](#), [CRYPT\\_INVALID\\_CIPHER](#), [CRYPT\\_OK](#), [LTC\\_MUTEX\\_LOCK](#), [LTC\\_MUTEX\\_UNLOCK](#), [ltc\\_cipher\\_descriptor::name](#), and [TAB\\_SIZE](#).

Referenced by [cbc\\_decrypt\(\)](#), [cbc\\_done\(\)](#), [cbc\\_encrypt\(\)](#), [cbc\\_start\(\)](#), [ccm\\_memory\(\)](#), [cfb\\_decrypt\(\)](#), [cfb\\_done\(\)](#), [cfb\\_encrypt\(\)](#), [cfb\\_setiv\(\)](#), [cfb\\_start\(\)](#), [chc\\_init\(\)](#), [chc\\_register\(\)](#), [ctr\\_done\(\)](#), [ctr\\_encrypt\(\)](#), [ctr\\_setiv\(\)](#), [ctr\\_start\(\)](#), [eax\\_init\(\)](#), [ecb\\_decrypt\(\)](#), [ecb\\_done\(\)](#), [ecb\\_encrypt\(\)](#), [ecb\\_start\(\)](#), [f8\\_done\(\)](#), [f8\\_encrypt\(\)](#), [f8\\_setiv\(\)](#), [f8\\_start\(\)](#), [f9\\_done\(\)](#), [f9\\_init\(\)](#), [f9\\_memory\(\)](#), [f9\\_process\(\)](#), [gcm\\_add\\_aad\(\)](#), [gcm\\_add\\_iv\(\)](#), [gcm\\_done\(\)](#), [gcm\\_init\(\)](#), [gcm\\_memory\(\)](#), [gcm\\_process\(\)](#), [lrw\\_decrypt\(\)](#), [lrw\\_done\(\)](#), [lrw\\_encrypt\(\)](#), [lrw\\_setiv\(\)](#), [lrw\\_start\(\)](#), [ocb\\_decrypt\(\)](#), [ocb\\_encrypt\(\)](#), [ocb\\_init\(\)](#), [ofb\\_done\(\)](#), [ofb\\_encrypt\(\)](#), [ofb\\_setiv\(\)](#), [ofb\\_start\(\)](#), [omac\\_done\(\)](#), [omac\\_init\(\)](#), [omac\\_memory\(\)](#), [omac\\_process\(\)](#), [pmac\\_done\(\)](#), [pmac\\_init\(\)](#), [pmac\\_process\(\)](#), [s\\_ocb\\_done\(\)](#), [xcbc\\_done\(\)](#), [xcbc\\_init\(\)](#), [xcbc\\_memory\(\)](#), [xcbc\\_process\(\)](#), [yarrow\\_ready\(\)](#), and [yarrow\\_start\(\)](#).

```
24 {
25     LTC_MUTEX_LOCK(&lttc_cipher_mutex);
26     if (idx < 0 || idx >= TAB_SIZE || cipher_descriptor[idx].name == NULL) {
27         LTC_MUTEX_UNLOCK(&lttc_cipher_mutex);
28         return CRYPT_INVALID_CIPHER;
29     }
30     LTC_MUTEX_UNLOCK(&lttc_cipher_mutex);
31     return CRYPT_OK;
32 }
```

## 5.138 misc/crypt/crypt\_find\_cipher.c File Reference

### 5.138.1 Detailed Description

Find a cipher in the descriptor tables, Tom St Denis.

Definition in file [crypt\\_find\\_cipher.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_find\_cipher.c:

### Functions

- int [find\\_cipher](#) (const char \*name)  
*Find a registered cipher by name.*

### 5.138.2 Function Documentation

#### 5.138.2.1 int find\_cipher (const char \* name)

Find a registered cipher by name.

##### Parameters:

**name** The name of the cipher to look for

##### Returns:

>= 0 if found, -1 if not present

Definition at line 23 of file crypt\_find\_cipher.c.

References [cipher\\_descriptor](#), [LTC\\_ARGCHK](#), [LTC\\_MUTEX\\_LOCK](#), [LTC\\_MUTEX\\_UNLOCK](#), and [TAB\\_SIZE](#).

Referenced by [ccm\\_test\(\)](#), [ctr\\_test\(\)](#), [eax\\_test\(\)](#), [f8\\_test\\_mode\(\)](#), [f9\\_test\(\)](#), [find\\_cipher\\_any\(\)](#), [gcm\\_test\(\)](#), [lrw\\_test\(\)](#), [ocb\\_test\(\)](#), [omac\\_test\(\)](#), [pmac\\_test\(\)](#), and [xcbc\\_test\(\)](#).

```
24 {
25     int x;
26     LTC_ARGCHK(name != NULL);
27     LTC_MUTEX_LOCK(&ltc_cipher_mutex);
28     for (x = 0; x < TAB_SIZE; x++) {
29         if (cipher_descriptor[x].name != NULL && !strcmp(cipher_descriptor[x].name, name)) {
30             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
31             return x;
32         }
33     }
34     LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
35     return -1;
36 }
```

## 5.139 misc/crypt/crypt\_find\_cipher\_any.c File Reference

### 5.139.1 Detailed Description

Find a cipher in the descriptor tables, Tom St Denis.

Definition in file [crypt\\_find\\_cipher\\_any.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_find\_cipher\_any.c:

### Functions

- [int find\\_cipher\\_any](#) (const char \*name, int blocklen, int keylen)  
*Find a cipher flexibly.*

### 5.139.2 Function Documentation

#### 5.139.2.1 int find\_cipher\_any (const char \* name, int blocklen, int keylen)

Find a cipher flexibly.

First by name then if not present by block and key size

#### Parameters:

- name** The name of the cipher desired
- blocklen** The minimum length of the block cipher desired (octets)
- keylen** The minimum length of the key size desired (octets)

#### Returns:

>= 0 if found, -1 if not present

Definition at line 25 of file crypt\_find\_cipher\_any.c.

References [cipher\\_descriptor](#), [find\\_cipher\(\)](#), [LTC\\_ARGCHK](#), [LTC\\_MUTEX\\_LOCK](#), and [TAB\\_SIZE](#).

```

26 {
27     int x;
28
29     LTC_ARGCHK(name != NULL);
30
31     x = find_cipher(name);
32     if (x != -1) return x;
33
34     LTC_MUTEX_LOCK(&ltc_cipher_mutex);
35     for (x = 0; x < TAB_SIZE; x++) {
36         if (cipher_descriptor[x].name == NULL) {
37             continue;
38         }
39         if (blocklen <= (int)cipher_descriptor[x].block_length && keylen <= (int)cipher_descriptor[x].ma
40             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
41             return x;
42         }
43     }
44     LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);

```

```
45     return -1;
46 }
```

Here is the call graph for this function:

## 5.140 misc/crypt/crypt\_find\_cipher\_id.c File Reference

### 5.140.1 Detailed Description

Find cipher by ID, Tom St Denis.

Definition in file [crypt\\_find\\_cipher\\_id.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `crypt_find_cipher_id.c`:

### Functions

- `int find_cipher_id` (unsigned char ID)

*Find a cipher by ID number.*

### 5.140.2 Function Documentation

#### 5.140.2.1 `int find_cipher_id` (unsigned char ID)

Find a cipher by ID number.

#### Parameters:

**ID** The ID (not same as index) of the cipher to find

#### Returns:

$\geq 0$  if found, -1 if not present

Definition at line 23 of file `crypt_find_cipher_id.c`.

References `cipher_descriptor`, `LTC_MUTEX_LOCK`, `LTC_MUTEX_UNLOCK`, `ltc_cipher_descriptor::name`, and `TAB_SIZE`.

```
24 {
25     int x;
26     LTC_MUTEX_LOCK(&ltc_cipher_mutex);
27     for (x = 0; x < TAB_SIZE; x++) {
28         if (cipher_descriptor[x].ID == ID) {
29             x = (cipher_descriptor[x].name == NULL) ? -1 : x;
30             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
31             return x;
32         }
33     }
34     LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
35     return -1;
36 }
```

## 5.141 misc/crypt/crypt\_find\_hash.c File Reference

### 5.141.1 Detailed Description

Find a hash, Tom St Denis.

Definition in file [crypt\\_find\\_hash.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_find\_hash.c:

### Functions

- int [find\\_hash](#) (const char \*name)  
*Find a registered hash by name.*

### 5.141.2 Function Documentation

#### 5.141.2.1 int find\_hash (const char \* name)

Find a registered hash by name.

##### Parameters:

**name** The name of the hash to look for

##### Returns:

>= 0 if found, -1 if not present

Definition at line 23 of file crypt\_find\_hash.c.

References [hash\\_descriptor](#), [LTC\\_ARGCHK](#), [LTC\\_MUTEX\\_LOCK](#), [LTC\\_MUTEX\\_UNLOCK](#), and [TAB\\_SIZE](#).

Referenced by [chc\\_register\(\)](#), [find\\_hash\\_any\(\)](#), and [hmac\\_test\(\)](#).

```
24 {
25     int x;
26     LTC_ARGCHK(name != NULL);
27     LTC_MUTEX_LOCK(&ltc_hash_mutex);
28     for (x = 0; x < TAB_SIZE; x++) {
29         if (hash_descriptor[x].name != NULL && strcmp(hash_descriptor[x].name, name) == 0) {
30             LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
31             return x;
32         }
33     }
34     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
35     return -1;
36 }
```

## 5.142 misc/crypt/crypt\_find\_hash\_any.c File Reference

### 5.142.1 Detailed Description

Find a hash, Tom St Denis.

Definition in file [crypt\\_find\\_hash\\_any.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [crypt\\_find\\_hash\\_any.c](#):

### Functions

- [int find\\_hash\\_any](#) (const char \*name, int digestlen)

*Find a hash flexibly.*

### 5.142.2 Function Documentation

#### 5.142.2.1 int find\_hash\_any (const char \* name, int digestlen)

Find a hash flexibly.

First by name then if not present by digest size

#### Parameters:

**name** The name of the hash desired

**digestlen** The minimum length of the digest size (octets)

#### Returns:

>= 0 if found, -1 if not present

Definition at line 23 of file [crypt\\_find\\_hash\\_any.c](#).

References [find\\_hash\(\)](#), [hash\\_descriptor](#), [LTC\\_ARGCHK](#), [LTC\\_MUTEX\\_LOCK](#), [MAXBLOCKSIZE](#), and [TAB\\_SIZE](#).

```

24 {
25     int x, y, z;
26     LTC_ARGCHK(name != NULL);
27
28     x = find_hash(name);
29     if (x != -1) return x;
30
31     LTC_MUTEX_LOCK(&ltc_hash_mutex);
32     y = MAXBLOCKSIZE+1;
33     z = -1;
34     for (x = 0; x < TAB_SIZE; x++) {
35         if (hash_descriptor[x].name == NULL) {
36             continue;
37         }
38         if ((int)hash_descriptor[x].hashsize >= digestlen && (int)hash_descriptor[x].hashsize < y) {
39             z = x;
40             y = hash_descriptor[x].hashsize;
41         }
42     }

```



```
43     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);  
44     return z;  
45 }
```

Here is the call graph for this function:

## 5.143 misc/crypt/crypt\_find\_hash\_id.c File Reference

### 5.143.1 Detailed Description

Find hash by ID, Tom St Denis.

Definition in file [crypt\\_find\\_hash\\_id.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_find\_hash\_id.c:

### Functions

- int [find\\_hash\\_id](#) (unsigned char ID)

*Find a hash by ID number.*

### 5.143.2 Function Documentation

#### 5.143.2.1 int find\_hash\_id (unsigned char ID)

Find a hash by ID number.

#### Parameters:

**ID** The ID (not same as index) of the hash to find

#### Returns:

>= 0 if found, -1 if not present

Definition at line 23 of file crypt\_find\_hash\_id.c.

References [hash\\_descriptor](#), [LTC\\_MUTEX\\_LOCK](#), [LTC\\_MUTEX\\_UNLOCK](#), [ltc\\_cipher\\_descriptor::name](#), and [TAB\\_SIZE](#).

```
24 {
25     int x;
26     LTC_MUTEX_LOCK(&ltc_hash_mutex);
27     for (x = 0; x < TAB_SIZE; x++) {
28         if (hash_descriptor[x].ID == ID) {
29             x = (hash_descriptor[x].name == NULL) ? -1 : x;
30             LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
31             return x;
32         }
33     }
34     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
35     return -1;
36 }
```

## 5.144 misc/crypt/crypt\_find\_hash\_oid.c File Reference

### 5.144.1 Detailed Description

Find a hash, Tom St Denis.

Definition in file [crypt\\_find\\_hash\\_oid.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_find\_hash\_oid.c:

### Functions

- int [find\\_hash\\_oid](#) (const unsigned long \*ID, unsigned long IDlen)

### 5.144.2 Function Documentation

#### 5.144.2.1 int find\_hash\_oid (const unsigned long \*ID, unsigned long IDlen)

Definition at line 18 of file crypt\_find\_hash\_oid.c.

References [hash\\_descriptor](#), [LTC\\_ARGCHK](#), [LTC\\_MUTEX\\_LOCK](#), [LTC\\_MUTEX\\_UNLOCK](#), [ltc\\_cipher\\_descriptor::name](#), [TAB\\_SIZE](#), and [XMEMCMP](#).

Referenced by [dsa\\_decrypt\\_key\(\)](#), and [ecc\\_decrypt\\_key\(\)](#).

```
19 {
20     int x;
21     LTC_ARGCHK(ID != NULL);
22     LTC_MUTEX_LOCK(&ltc_hash_mutex);
23     for (x = 0; x < TAB_SIZE; x++) {
24         if (hash_descriptor[x].name != NULL && hash_descriptor[x].OIDlen == IDlen && !XMEMCMP(hash_desc
25             LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
26             return x;
27         }
28     }
29     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
30     return -1;
31 }
```

## 5.145 misc/crypt/crypt\_find\_prng.c File Reference

### 5.145.1 Detailed Description

Find a PRNG, Tom St Denis.

Definition in file [crypt\\_find\\_prng.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_find\_prng.c:

### Functions

- int [find\\_prng](#) (const char \*name)  
*Find a registered PRNG by name.*

### 5.145.2 Function Documentation

#### 5.145.2.1 int find\_prng (const char \* name)

Find a registered PRNG by name.

#### Parameters:

**name** The name of the PRNG to look for

#### Returns:

>= 0 if found, -1 if not present

Definition at line 23 of file crypt\_find\_prng.c.

References [LTC\\_ARGCHK](#), [LTC\\_MUTEX\\_LOCK](#), [LTC\\_MUTEX\\_UNLOCK](#), [prng\\_descriptor](#), and [TAB\\_SIZE](#).

```
24 {
25     int x;
26     LTC_ARGCHK(name != NULL);
27     LTC_MUTEX_LOCK(&ltc_prng_mutex);
28     for (x = 0; x < TAB_SIZE; x++) {
29         if ((prng_descriptor[x].name != NULL) && strcmp(prng_descriptor[x].name, name) == 0) {
30             LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
31             return x;
32         }
33     }
34     LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
35     return -1;
36 }
```

## 5.146 misc/crypt/crypt\_fsa.c File Reference

### 5.146.1 Detailed Description

LibTomCrypt FULL SPEED AHEAD!, Tom St Denis.

Definition in file [crypt\\_fsa.c](#).

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for crypt\_fsa.c:

### Functions

- int [crypt\\_fsa](#) (void \*mp,...)

### 5.146.2 Function Documentation

#### 5.146.2.1 int crypt\_fsa (void \* mp, ...)

Definition at line 20 of file crypt\_fsa.c.

References CRYPT\_OK, ltc\_mp, register\_cipher(), register\_hash(), register\_prng(), and XMEMCPY.

```
21 {
22     int      err;
23     va_list  args;
24     void     *p;
25
26     va_start(args, mp);
27     if (mp != NULL) {
28         XMEMCPY(&ltc_mp, mp, sizeof(ltc_mp));
29     }
30
31     while ((p = va_arg(args, void*)) != NULL) {
32         if ((err = register_cipher(p)) != CRYPT_OK) {
33             va_end(args);
34             return err;
35         }
36     }
37
38     while ((p = va_arg(args, void*)) != NULL) {
39         if ((err = register_hash(p)) != CRYPT_OK) {
40             va_end(args);
41             return err;
42         }
43     }
44
45     while ((p = va_arg(args, void*)) != NULL) {
46         if ((err = register_prng(p)) != CRYPT_OK) {
47             va_end(args);
48             return err;
49         }
50     }
51
52     va_end(args);
53     return CRYPT_OK;
54 }
```

Here is the call graph for this function:

## 5.147 misc/crypt/crypt\_hash\_descriptor.c File Reference

### 5.147.1 Detailed Description

Stores the hash descriptor table, Tom St Denis.

Definition in file [crypt\\_hash\\_descriptor.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_hash\_descriptor.c:

### Variables

- [ltc\\_hash\\_descriptor](#) [hash\\_descriptor](#) [TAB\_SIZE]

### 5.147.2 Variable Documentation

#### 5.147.2.1 struct [ltc\\_hash\\_descriptor](#) [hash\\_descriptor](#)[TAB\_SIZE]

**Initial value:**

```
{  
{ NULL, 0, 0, 0, { 0 }, 0, NULL, NULL, NULL, NULL, NULL }  
}
```

Definition at line 18 of file crypt\_hash\_descriptor.c.

## 5.148 misc/crypt/crypt\_hash\_is\_valid.c File Reference

### 5.148.1 Detailed Description

Determine if hash is valid, Tom St Denis.

Definition in file [crypt\\_hash\\_is\\_valid.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_hash\_is\_valid.c:

### Functions

- int [hash\\_is\\_valid](#) (int idx)

### 5.148.2 Function Documentation

#### 5.148.2.1 int hash\_is\_valid (int *idx*)

Definition at line 23 of file crypt\_hash\_is\_valid.c.

References `CRYPT_INVALID_HASH`, `CRYPT_OK`, `hash_descriptor`, `LTC_MUTEX_LOCK`, `LTC_MUTEX_UNLOCK`, `ltc_hash_descriptor::name`, and `TAB_SIZE`.

Referenced by `chc_register()`, `dsa_decrypt_key()`, `dsa_encrypt_key()`, `ecc_decrypt_key()`, `ecc_encrypt_key()`, `hash_file()`, `hash_filehandle()`, `hash_memory()`, `hash_memory_multi()`, `hmac_done()`, `hmac_file()`, `hmac_init()`, `hmac_memory()`, `hmac_process()`, `pkcs_1_mgf1()`, `pkcs_1_oaep_decode()`, `pkcs_1_oaep_encode()`, `pkcs_1_pss_decode()`, `pkcs_1_pss_encode()`, `pkcs_5_alg1()`, `pkcs_5_alg2()`, `rsa_decrypt_key_ex()`, `rsa_encrypt_key_ex()`, `rsa_sign_hash_ex()`, `rsa_verify_hash_ex()`, `yarrow_add_entropy()`, `yarrow_ready()`, and `yarrow_start()`.

```
24 {
25     LTC_MUTEX_LOCK(&ltc_hash_mutex);
26     if (idx < 0 || idx >= TAB_SIZE || hash_descriptor[idx].name == NULL) {
27         LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
28         return CRYPT_INVALID_HASH;
29     }
30     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
31     return CRYPT_OK;
32 }
```



## 5.149 misc/crypt/crypt\_ltc\_mp\_descriptor.c File Reference

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_ltc\_mp\_descriptor.c:

### Variables

- [ltc\\_math\\_descriptor ltc\\_mp](#)

### 5.149.1 Variable Documentation

#### 5.149.1.1 [ltc\\_math\\_descriptor ltc\\_mp](#)

Definition at line 13 of file crypt\_ltc\_mp\_descriptor.c.

Referenced by crypt\_fsa(), dsa\_import(), dsa\_make\_key(), ecc\_import(), ecc\_make\_key(), ecc\_shared\_secret(), ecc\_verify\_hash(), rsa\_decrypt\_key\_ex(), rsa\_encrypt\_key\_ex(), rsa\_import(), rsa\_make\_key(), rsa\_sign\_hash\_ex(), and rsa\_verify\_hash\_ex().

## 5.150 misc/crypt/crypt\_prng\_descriptor.c File Reference

### 5.150.1 Detailed Description

Stores the PRNG descriptors, Tom St Denis.

Definition in file [crypt\\_prng\\_descriptor.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_prng\_descriptor.c:

### Variables

- [ltc\\_prng\\_descriptor](#) [prng\\_descriptor](#) [TAB\_SIZE]

### 5.150.2 Variable Documentation

#### 5.150.2.1 struct [ltc\\_prng\\_descriptor](#) [prng\\_descriptor](#)[TAB\_SIZE]

**Initial value:**

```
{  
{ NULL, 0, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL }  
}
```

Definition at line 17 of file crypt\_prng\_descriptor.c.

## 5.151 misc/crypt/crypt\_prng\_is\_valid.c File Reference

### 5.151.1 Detailed Description

Determine if PRNG is valid, Tom St Denis.

Definition in file [crypt\\_prng\\_is\\_valid.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_prng\_is\_valid.c:

### Functions

- [int prng\\_is\\_valid](#) (int idx)

### 5.151.2 Function Documentation

#### 5.151.2.1 int prng\_is\_valid (int idx)

Definition at line 23 of file crypt\_prng\_is\_valid.c.

References `CRYPT_INVALID_PRNG`, `CRYPT_OK`, `LTC_MUTEX_LOCK`, `LTC_MUTEX_UNLOCK`, `ltc_prng_descriptor::name`, `prng_descriptor`, and `TAB_SIZE`.

Referenced by `dsa_encrypt_key()`, `dsa_make_key()`, `dsa_sign_hash_raw()`, `ecc_encrypt_key()`, `ecc_make_key()`, `ecc_sign_hash()`, `pkcs_1_oaep_encode()`, `pkcs_1_pss_encode()`, `pkcs_1_v1_5_encode()`, `rand_prime()`, `rng_make_prng()`, `rsa_encrypt_key_ex()`, `rsa_make_key()`, and `rsa_sign_hash_ex()`.

```
24 {  
25     LTC_MUTEX_LOCK(&ltc_prng_mutex);  
26     if (idx < 0 || idx >= TAB_SIZE || prng_descriptor[idx].name == NULL) {  
27         LTC_MUTEX_UNLOCK(&ltc_prng_mutex);  
28         return CRYPT_INVALID_PRNG;  
29     }  
30     LTC_MUTEX_UNLOCK(&ltc_prng_mutex);  
31     return CRYPT_OK;  
32 }
```

## 5.152 misc/crypt/crypt\_register\_cipher.c File Reference

### 5.152.1 Detailed Description

Register a cipher, Tom St Denis.

Definition in file [crypt\\_register\\_cipher.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_register\_cipher.c:

### Functions

- [int register\\_cipher](#) (const struct [ltc\\_cipher\\_descriptor](#) \*cipher)  
*Register a cipher with the descriptor table.*

### 5.152.2 Function Documentation

#### 5.152.2.1 int register\_cipher (const struct [ltc\\_cipher\\_descriptor](#) \* cipher)

Register a cipher with the descriptor table.

#### Parameters:

***cipher*** The cipher you wish to register

#### Returns:

value  $\geq 0$  if successfully added (or already present), -1 if unsuccessful

Definition at line 23 of file crypt\_register\_cipher.c.

References [cipher\\_descriptor](#), [ltc\\_cipher\\_descriptor::ID](#), [LTC\\_ARGCHK](#), [LTC\\_MUTEX\\_LOCK](#), [LTC\\_MUTEX\\_UNLOCK](#), [ltc\\_prng\\_descriptor::name](#), and [TAB\\_SIZE](#).

Referenced by [crypt\\_fsa\(\)](#), and [yarrow\\_start\(\)](#).

```

24 {
25     int x;
26
27     LTC_ARGCHK(cipher != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_cipher_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (cipher_descriptor[x].name != NULL && cipher_descriptor[x].ID == cipher->ID) {
33             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
34             return x;
35         }
36     }
37
38     /* find a blank spot */
39     for (x = 0; x < TAB_SIZE; x++) {
40         if (cipher_descriptor[x].name == NULL) {
41             XMEMCOPY(&cipher_descriptor[x], cipher, sizeof(struct ltc_cipher_descriptor));
42             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
43             return x;
44         }
45     }

```

---

```
45     }
46
47     /* no spot */
48     LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
49     return -1;
50 }
```

## 5.153 misc/crypt/crypt\_register\_hash.c File Reference

### 5.153.1 Detailed Description

Register a HASH, Tom St Denis.

Definition in file [crypt\\_register\\_hash.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_register\_hash.c:

### Functions

- [int register\\_hash](#) (const struct [ltc\\_hash\\_descriptor](#) \*hash)  
*Register a hash with the descriptor table.*

### 5.153.2 Function Documentation

#### 5.153.2.1 int register\_hash (const struct [ltc\\_hash\\_descriptor](#) \* hash)

Register a hash with the descriptor table.

#### Parameters:

*hash* The hash you wish to register

#### Returns:

value  $\geq 0$  if successfully added (or already present), -1 if unsuccessful

Definition at line 23 of file crypt\_register\_hash.c.

References [hash\\_descriptor](#), [LTC\\_ARGCHK](#), [LTC\\_MUTEX\\_LOCK](#), [LTC\\_MUTEX\\_UNLOCK](#), [TAB\\_SIZE](#), and [XMEMCMP](#).

Referenced by [crypt\\_fsa\(\)](#), and [yarrow\\_start\(\)](#).

```
24 {
25     int x;
26
27     LTC_ARGCHK(hash != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_hash_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (XMEMCMP(&hash_descriptor[x], hash, sizeof(struct ltc_hash_descriptor)) == 0) {
33             LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
34             return x;
35         }
36     }
37
38     /* find a blank spot */
39     for (x = 0; x < TAB_SIZE; x++) {
40         if (hash_descriptor[x].name == NULL) {
41             XMEMCPY(&hash_descriptor[x], hash, sizeof(struct ltc_hash_descriptor));
42             LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
43             return x;
44         }
45     }
```

```
45     }
46
47     /* no spot */
48     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
49     return -1;
50 }
```

## 5.154 misc/crypt/crypt\_register\_prng.c File Reference

### 5.154.1 Detailed Description

Register a PRNG, Tom St Denis.

Definition in file [crypt\\_register\\_prng.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [crypt\\_register\\_prng.c](#):

### Functions

- [int register\\_prng](#) (const struct [ltc\\_prng\\_descriptor](#) \*prng)  
*Register a PRNG with the descriptor table.*

### 5.154.2 Function Documentation

#### 5.154.2.1 int register\_prng (const struct [ltc\\_prng\\_descriptor](#) \*prng)

Register a PRNG with the descriptor table.

#### Parameters:

*prng* The PRNG you wish to register

#### Returns:

value  $\geq 0$  if successfully added (or already present), -1 if unsuccessful

Definition at line 23 of file [crypt\\_register\\_prng.c](#).

References [LTC\\_ARGCHK](#), [LTC\\_MUTEX\\_LOCK](#), [LTC\\_MUTEX\\_UNLOCK](#), [prng\\_descriptor](#), [TAB\\_SIZE](#), and [XMEMCMP](#).

Referenced by [crypt\\_fsa\(\)](#).

```
24 {
25     int x;
26
27     LTC_ARGCHK(prng != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_prng_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (XMEMCMP(&prng_descriptor[x], prng, sizeof(struct ltc_prng_descriptor)) == 0) {
33             LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
34             return x;
35         }
36     }
37
38     /* find a blank spot */
39     for (x = 0; x < TAB_SIZE; x++) {
40         if (prng_descriptor[x].name == NULL) {
41             XMEMCPY(&prng_descriptor[x], prng, sizeof(struct ltc_prng_descriptor));
42             LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
43             return x;
44         }
45     }
```



```
45     }
46
47     /* no spot */
48     LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
49     return -1;
50 }
```

## 5.155 misc/crypt/crypt\_unregister\_cipher.c File Reference

### 5.155.1 Detailed Description

Unregister a cipher, Tom St Denis.

Definition in file [crypt\\_unregister\\_cipher.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `crypt_unregister_cipher.c`:

### Functions

- `int unregister_cipher (const struct ltc\_cipher\_descriptor *cipher)`  
*Unregister a cipher from the descriptor table.*

### 5.155.2 Function Documentation

#### 5.155.2.1 `int unregister_cipher (const struct ltc\_cipher\_descriptor * cipher)`

Unregister a cipher from the descriptor table.

#### Parameters:

***cipher*** The cipher descriptor to remove

#### Returns:

CRYPT\_OK on success

Definition at line 23 of file `crypt_unregister_cipher.c`.

References `cipher_descriptor`, `CRYPT_OK`, `ltc_cipher_descriptor::ID`, `LTC_ARGCHK`, `LTC_MUTEX_LOCK`, `LTC_MUTEX_UNLOCK`, `ltc_prng_descriptor::name`, `TAB_SIZE`, and `XMEMCMP`.

```
24 {
25     int x;
26
27     LTC_ARGCHK(cipher != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_cipher_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (XMEMCMP(&cipher_descriptor[x], cipher, sizeof(struct ltc_cipher_descriptor)) == 0) {
33             cipher_descriptor[x].name = NULL;
34             cipher_descriptor[x].ID = 255;
35             LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
36             return CRYPT_OK;
37         }
38     }
39     LTC_MUTEX_UNLOCK(&ltc_cipher_mutex);
40     return CRYPT_ERROR;
41 }
```

## 5.156 misc/crypt/crypt\_unregister\_hash.c File Reference

### 5.156.1 Detailed Description

Unregister a hash, Tom St Denis.

Definition in file [crypt\\_unregister\\_hash.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for crypt\_unregister\_hash.c:

### Functions

- int [unregister\\_hash](#) (const struct [ltc\\_hash\\_descriptor](#) \*hash)

*Unregister a hash from the descriptor table.*

### 5.156.2 Function Documentation

#### 5.156.2.1 int unregister\_hash (const struct [ltc\\_hash\\_descriptor](#) \* hash)

Unregister a hash from the descriptor table.

#### Parameters:

*hash* The hash descriptor to remove

#### Returns:

CRYPT\_OK on success

Definition at line 23 of file crypt\_unregister\_hash.c.

References [CRYPT\\_OK](#), [hash\\_descriptor](#), [LTC\\_ARGCHK](#), [LTC\\_MUTEX\\_LOCK](#), [LTC\\_MUTEX\\_UNLOCK](#), [ltc\\_prng\\_descriptor::name](#), [TAB\\_SIZE](#), and [XMEMCMP](#).

```
24 {
25     int x;
26
27     LTC_ARGCHK(hash != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_hash_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (XMEMCMP(&hash_descriptor[x], hash, sizeof(struct ltc_hash_descriptor)) == 0) {
33             hash_descriptor[x].name = NULL;
34             LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
35             return CRYPT_OK;
36         }
37     }
38     LTC_MUTEX_UNLOCK(&ltc_hash_mutex);
39     return CRYPT_ERROR;
40 }
```

## 5.157 misc/crypt/crypt\_unregister\_prng.c File Reference

### 5.157.1 Detailed Description

Unregister a PRNG, Tom St Denis.

Definition in file [crypt\\_unregister\\_prng.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `crypt_unregister_prng.c`:

### Functions

- `int unregister_prng (const struct ltc\_prng\_descriptor *prng)`  
*Unregister a PRNG from the descriptor table.*

### 5.157.2 Function Documentation

#### 5.157.2.1 `int unregister_prng (const struct ltc\_prng\_descriptor *prng)`

Unregister a PRNG from the descriptor table.

#### Parameters:

*prng* The PRNG descriptor to remove

#### Returns:

CRYPT\_OK on success

Definition at line 23 of file `crypt_unregister_prng.c`.

References `CRYPT_OK`, `LTC_ARGCHK`, `LTC_MUTEX_LOCK`, `LTC_MUTEX_UNLOCK`, `ltc_prng_descriptor::name`, `prng_descriptor`, `TAB_SIZE`, and `XMEMCMP`.

```
24 {
25     int x;
26
27     LTC_ARGCHK(prng != NULL);
28
29     /* is it already registered? */
30     LTC_MUTEX_LOCK(&ltc_prng_mutex);
31     for (x = 0; x < TAB_SIZE; x++) {
32         if (XMEMCMP(&prng_descriptor[x], prng, sizeof(struct ltc_prng_descriptor)) != 0) {
33             prng_descriptor[x].name = NULL;
34             LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
35             return CRYPT_OK;
36         }
37     }
38     LTC_MUTEX_UNLOCK(&ltc_prng_mutex);
39     return CRYPT_ERROR;
40 }
```

## 5.158 misc/error\_to\_string.c File Reference

### 5.158.1 Detailed Description

Convert error codes to ASCII strings, Tom St Denis.

Definition in file [error\\_to\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for error\_to\_string.c:

### Functions

- `const char * error\_to\_string (int err)`  
*Convert an LTC error code to ASCII.*

### Variables

- `static const char * err\_2\_str []`

### 5.158.2 Function Documentation

#### 5.158.2.1 `const char* error_to_string (int err)`

Convert an LTC error code to ASCII.

##### Parameters:

*err* The error code

##### Returns:

A pointer to the ASCII NUL terminated string for the error or "Invalid error code." if the err code was not valid.

Definition at line 62 of file error\_to\_string.c.

References [err\\_2\\_str](#).

Referenced by [hmac\\_test\(\)](#).

```
63 {  
64     if (err < 0 || err >= (int) (sizeof(err_2_str)/sizeof(err_2_str[0]))) {  
65         return "Invalid error code.";  
66     } else {  
67         return err_2_str[err];  
68     }  
69 }
```

### 5.158.3 Variable Documentation

#### 5.158.3.1 `const char* err\_2\_str[]` [static]

Definition at line 19 of file error\_to\_string.c.

Referenced by [error\\_to\\_string\(\)](#).

## 5.159 misc/pkcs5/pkcs\_5\_1.c File Reference

### 5.159.1 Detailed Description

PKCS #5, Algorithm #1, Tom St Denis.

Definition in file [pkcs\\_5\\_1.c](#).

```
#include <tomcrypt.h>
```

Include dependency graph for pkcs\_5\_1.c:

### Functions

- int [pkcs\\_5\\_alg1](#) (const unsigned char \*password, unsigned long password\_len, const unsigned char \*salt, int iteration\_count, int hash\_idx, unsigned char \*out, unsigned long \*outlen)

*Execute PKCS #5 v1.*

### 5.159.2 Function Documentation

**5.159.2.1 int pkcs\_5\_alg1** (const unsigned char \* *password*, unsigned long *password\_len*, const unsigned char \* *salt*, int *iteration\_count*, int *hash\_idx*, unsigned char \* *out*, unsigned long \* *outlen*)

Execute PKCS #5 v1.

#### Parameters:

*password* The password (or key)  
*password\_len* The length of the password (octet)  
*salt* The salt (or nonce) which is 8 octets long  
*iteration\_count* The PKCS #5 v1 iteration count  
*hash\_idx* The index of the hash desired  
*out* [out] The destination for this algorithm  
*outlen* [in/out] The max size and resulting size of the algorithm output

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file pkcs\_5\_1.c.

References CRYPT\_MEM, CRYPT\_OK, ltc\_prng\_descriptor::done, hash\_descriptor, hash\_is\_valid(), hash\_memory(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, MAXBLOCKSIZE, XFREE, and XMALLOC.

```
33 {  
34     int err;  
35     unsigned long x;  
36     hash_state *md;  
37     unsigned char *buf;  
38  
39     LTC_ARGCHK(password != NULL);  
40     LTC_ARGCHK(salt != NULL);
```

```

41     LTC_ARGCHK(out      != NULL);
42     LTC_ARGCHK(outlen   != NULL);
43
44     /* test hash IDX */
45     if ((err = hash_is_valid(hash_idx)) != CRYPT_OK) {
46         return err;
47     }
48
49     /* allocate memory */
50     md = XMALLOC(sizeof(hash_state));
51     buf = XMALLOC(MAXBLOCKSIZE);
52     if (md == NULL || buf == NULL) {
53         if (md != NULL) {
54             XFREE(md);
55         }
56         if (buf != NULL) {
57             XFREE(buf);
58         }
59         return CRYPT_MEM;
60     }
61
62     /* hash initial password + salt */
63     if ((err = hash_descriptor[hash_idx].init(md)) != CRYPT_OK) {
64         goto LBL_ERR;
65     }
66     if ((err = hash_descriptor[hash_idx].process(md, password, password_len)) != CRYPT_OK) {
67         goto LBL_ERR;
68     }
69     if ((err = hash_descriptor[hash_idx].process(md, salt, 8)) != CRYPT_OK) {
70         goto LBL_ERR;
71     }
72     if ((err = hash_descriptor[hash_idx].done(md, buf)) != CRYPT_OK) {
73         goto LBL_ERR;
74     }
75
76     while (--iteration_count) {
77         /* code goes here. */
78         x = MAXBLOCKSIZE;
79         if ((err = hash_memory(hash_idx, buf, hash_descriptor[hash_idx].hashsize, buf, &x)) != CRYPT_OK)
80             goto LBL_ERR;
81     }
82 }
83
84 /* copy upto outlen bytes */
85 for (x = 0; x < hash_descriptor[hash_idx].hashsize && x < *outlen; x++) {
86     out[x] = buf[x];
87 }
88 *outlen = x;
89 err = CRYPT_OK;
90 LBL_ERR:
91 #ifdef LTC_CLEAN_STACK
92     zeromem(buf, MAXBLOCKSIZE);
93     zeromem(md, sizeof(hash_state));
94 #endif
95
96     XFREE(buf);
97     XFREE(md);
98
99     return err;
100 }

```

Here is the call graph for this function:

## 5.160 misc/pkcs5/pkcs\_5\_2.c File Reference

### 5.160.1 Detailed Description

PKCS #5, Algorithm #2, Tom St Denis.

Definition in file [pkcs\\_5\\_2.c](#).

```
#include <tomcrypt.h>
```

Include dependency graph for pkcs\_5\_2.c:

### Functions

- [int pkcs\\_5\\_alg2](#) (const unsigned char \*password, unsigned long password\_len, const unsigned char \*salt, unsigned long salt\_len, int iteration\_count, int hash\_idx, unsigned char \*out, unsigned long \*outlen)

*Execute PKCS #5 v2.*

### 5.160.2 Function Documentation

**5.160.2.1 int pkcs\_5\_alg2** (const unsigned char \* *password*, unsigned long *password\_len*, const unsigned char \* *salt*, unsigned long *salt\_len*, int *iteration\_count*, int *hash\_idx*, unsigned char \* *out*, unsigned long \* *outlen*)

Execute PKCS #5 v2.

#### Parameters:

***password*** The input password (or key)

***password\_len*** The length of the password (octets)

***salt*** The salt (or nonce)

***salt\_len*** The length of the salt (octets)

***iteration\_count*** # of iterations desired for PKCS #5 v2 [read specs for more]

***hash\_idx*** The index of the hash desired

***out*** [out] The destination for this algorithm

***outlen*** [in/out] The max size and resulting size of the algorithm output

#### Returns:

CRYPT\_OK if successful

Definition at line 31 of file pkcs\_5\_2.c.

References CRYPT\_MEM, CRYPT\_OK, hash\_is\_valid(), hmac\_done(), hmac\_init(), hmac\_memory(), hmac\_process(), LTC\_ARGCHK, MAXBLOCKSIZE, XFREE, XMALLOC, XMEMCPY, and zeromem().

```
35 {
36     int err, itts;
37     ulong32 blkno;
38     unsigned long stored, left, x, y;
39     unsigned char *buf[2];
```



```

40     hmac_state      *hmac;
41
42     LTC_ARGCHK(password != NULL);
43     LTC_ARGCHK(salt      != NULL);
44     LTC_ARGCHK(out       != NULL);
45     LTC_ARGCHK(outlen    != NULL);
46
47     /* test hash IDX */
48     if ((err = hash_is_valid(hash_idx)) != CRYPT_OK) {
49         return err;
50     }
51
52     buf[0] = XMALLOC(MAXBLOCKSIZE * 2);
53     hmac   = XMALLOC(sizeof(hmac_state));
54     if (hmac == NULL || buf[0] == NULL) {
55         if (hmac != NULL) {
56             XFREE(hmac);
57         }
58         if (buf[0] != NULL) {
59             XFREE(buf[0]);
60         }
61         return CRYPT_MEM;
62     }
63     /* buf[1] points to the second block of MAXBLOCKSIZE bytes */
64     buf[1] = buf[0] + MAXBLOCKSIZE;
65
66     left   = *outlen;
67     blkno  = 1;
68     stored = 0;
69     while (left != 0) {
70         /* process block number blkno */
71         zeromem(buf[0], MAXBLOCKSIZE*2);
72
73         /* store current block number and increment for next pass */
74         STORE32H(blkno, buf[1]);
75         ++blkno;
76
77         /* get PRF(P, S||int(blkno)) */
78         if ((err = hmac_init(hmac, hash_idx, password, password_len)) != CRYPT_OK) {
79             goto LBL_ERR;
80         }
81         if ((err = hmac_process(hmac, salt, salt_len)) != CRYPT_OK) {
82             goto LBL_ERR;
83         }
84         if ((err = hmac_process(hmac, buf[1], 4)) != CRYPT_OK) {
85             goto LBL_ERR;
86         }
87         x = MAXBLOCKSIZE;
88         if ((err = hmac_done(hmac, buf[0], &x)) != CRYPT_OK) {
89             goto LBL_ERR;
90         }
91
92         /* now compute repeated and XOR it in buf[1] */
93         XMEMCPY(buf[1], buf[0], x);
94         for (itts = 1; itt < iteration_count; ++itts) {
95             if ((err = hmac_memory(hash_idx, password, password_len, buf[0], x, buf[0], &x)) != CRYPT_OK) {
96                 goto LBL_ERR;
97             }
98             for (y = 0; y < x; y++) {
99                 buf[1][y] ^= buf[0][y];
100             }
101         }
102
103         /* now emit upto x bytes of buf[1] to output */
104         for (y = 0; y < x && left != 0; ++y) {
105             out[stored++] = buf[1][y];
106             --left;

```

```
107     }
108 }
109 *outlen = stored;
110
111 err = CRYPT_OK;
112 LBL_ERR:
113 #ifdef LTC_CLEAN_STACK
114     zeromem(buf[0], MAXBLOCKSIZE*2);
115     zeromem(hmac, sizeof(hmac_state));
116 #endif
117
118     XFREE(hmac);
119     XFREE(buf[0]);
120
121     return err;
122 }
```

Here is the call graph for this function:

## 5.161 misc/zeromem.c File Reference

### 5.161.1 Detailed Description

Zero a block of memory, Tom St Denis.

Definition in file [zeromem.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for zeromem.c:

### Functions

- void [zeromem](#) (void \*out, size\_t outlen)  
*Zero a block of memory.*

### 5.161.2 Function Documentation

#### 5.161.2.1 void zeromem (void \* out, size\_t outlen)

Zero a block of memory.

#### Parameters:

- out** The destination of the area to zero
- outlen** The length of the area to zero (octets)

Definition at line 23 of file zeromem.c.

References LTC\_ARGCHKVD.

Referenced by burn\_stack(), cast5\_setup(), chc\_init(), dsa\_shared\_secret(), dsa\_sign\_hash\_raw(), eax\_decrypt\_verify\_memory(), eax\_encrypt\_authenticate\_memory(), eax\_init(), ECB\_TEST(), ecc\_ansi\_x963\_export(), ecc\_shared\_secret(), f8\_encrypt(), f8\_start(), f9\_file(), fortuna\_read(), gcm\_add\_aad(), gcm\_gf\_mult(), gcm\_init(), gcm\_reset(), hash\_filehandle(), hash\_memory(), hmac\_file(), hmac\_init(), hmac\_memory(), lrw\_start(), md2\_done(), md2\_init(), noekeon\_test(), ocb\_decrypt\_verify\_memory(), ocb\_done\_decrypt(), ocb\_encrypt\_authenticate\_memory(), omac\_done(), omac\_file(), omac\_init(), omac\_memory(), pelican\_init(), pkcs\_1\_i2osp(), pkcs\_1\_pss\_encode(), pkcs\_5\_alg2(), pmac\_file(), pmac\_memory(), rc2\_test(), rc4\_read(), rng\_make\_prng(), rsa\_exptmod(), rsa\_verify\_hash\_ex(), sha224\_done(), sha384\_done(), sober128\_read(), tiger\_done(), whirlpool\_init(), xcbc\_file(), yarrow\_read(), and yarrow\_start().

```
24 {
25     unsigned char *mem = out;
26     LTC_ARGCHKVD(out != NULL);
27     while (outlen-- > 0) {
28         *mem++ = 0;
29     }
30 }
```

## 5.162 modes/cbc/cbc\_decrypt.c File Reference

### 5.162.1 Detailed Description

CBC implementation, encrypt block, Tom St Denis.

Definition in file [cbc\\_decrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cbc\_decrypt.c:

### Functions

- `int cbc_decrypt` (const unsigned char \*ct, unsigned char \*pt, unsigned long len, symmetric\_CBC \*cbc)  
*CBC decrypt.*

### 5.162.2 Function Documentation

#### 5.162.2.1 `int cbc_decrypt` (const unsigned char \*ct, unsigned char \*pt, unsigned long len, symmetric\_CBC \*cbc)

CBC decrypt.

#### Parameters:

- ct* Ciphertext  
*pt* [out] Plaintext  
*len* The number of bytes to process (must be multiple of block length)  
*cbc* CBC state

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file cbc\_decrypt.c.

References `ltc_cipher_descriptor::accel_cbc_decrypt`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `ltc_cipher_descriptor::ecb_decrypt`, and `LTC_ARGCHK`.

```
30 {
31     int x, err;
32     unsigned char tmp[16];
33 #ifdef LTC_FAST
34     LTC_FAST_TYPE tmpy;
35 #else
36     unsigned char tmpy;
37 #endif
38
39     LTC_ARGCHK(pt != NULL);
40     LTC_ARGCHK(ct != NULL);
41     LTC_ARGCHK(cbc != NULL);
42
43     if ((err = cipher_is_valid(cbc->cipher)) != CRYPT_OK) {
44         return err;
45     }
```

```

45     }
46
47     /* is blocklen valid? */
48     if (cbc->blocklen < 1 || cbc->blocklen > (int)sizeof(cbc->IV)) {
49         return CRYPT_INVALID_ARG;
50     }
51
52     if (len % cbc->blocklen) {
53         return CRYPT_INVALID_ARG;
54     }
55 #ifdef LTC_FAST
56     if (cbc->blocklen % sizeof(LTC_FAST_TYPE)) {
57         return CRYPT_INVALID_ARG;
58     }
59 #endif
60
61     if (cipher_descriptor[cbc->cipher].accel_cbc_decrypt != NULL) {
62         return cipher_descriptor[cbc->cipher].accel_cbc_decrypt(ct, pt, len / cbc->blocklen, cbc->IV, &cbc->key);
63     } else {
64         while (len) {
65             /* decrypt */
66             if ((err = cipher_descriptor[cbc->cipher].ecb_decrypt(ct, tmp, &cbc->key)) != CRYPT_OK) {
67                 return err;
68             }
69
70             /* xor IV against plaintext */
71             #if defined(LTC_FAST)
72             for (x = 0; x < cbc->blocklen; x += sizeof(LTC_FAST_TYPE)) {
73                 tmpy = *((LTC_FAST_TYPE*)((unsigned char *)cbc->IV + x)) ^ *((LTC_FAST_TYPE*)((unsigned char *)ct + x));
74                 *((LTC_FAST_TYPE*)((unsigned char *)cbc->IV + x)) = *((LTC_FAST_TYPE*)((unsigned char *)ct + x));
75                 *((LTC_FAST_TYPE*)((unsigned char *)pt + x)) = tmpy;
76             }
77             #else
78             for (x = 0; x < cbc->blocklen; x++) {
79                 tmpy = tmp[x] ^ cbc->IV[x];
80                 cbc->IV[x] = ct[x];
81                 pt[x] = tmpy;
82             }
83             #endif
84
85             ct += cbc->blocklen;
86             pt += cbc->blocklen;
87             len -= cbc->blocklen;
88         }
89     }
90     return CRYPT_OK;
91 }

```

Here is the call graph for this function:

## 5.163 modes/cbc/cbc\_done.c File Reference

### 5.163.1 Detailed Description

CBC implementation, finish chain, Tom St Denis.

Definition in file [cbc\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cbc\_done.c:

### Functions

- `int cbc_done (symmetric_CBC *cbc)`  
*Terminate the chain.*

### 5.163.2 Function Documentation

#### 5.163.2.1 `int cbc_done (symmetric_CBC *cbc)`

Terminate the chain.

#### Parameters:

*cbc* The CBC chain to terminate

#### Returns:

CRYPT\_OK on success

Definition at line 24 of file cbc\_done.c.

References `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_OK`, `ltc_cipher_descriptor::done`, and `LTC_ARGCHK`.

```
25 {
26     int err;
27     LTC_ARGCHK(cbc != NULL);
28
29     if ((err = cipher_is_valid(cbc->cipher)) != CRYPT_OK) {
30         return err;
31     }
32     cipher_descriptor[cbc->cipher].done(&cbc->key);
33     return CRYPT_OK;
34 }
```

Here is the call graph for this function:

## 5.164 modes/cbc/cbc\_encrypt.c File Reference

### 5.164.1 Detailed Description

CBC implementation, encrypt block, Tom St Denis.

Definition in file [cbc\\_encrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cbc\_encrypt.c:

### Functions

- `int cbc_encrypt` (const unsigned char \*pt, unsigned char \*ct, unsigned long len, symmetric\_CBC \*cbc)  
*CBC encrypt.*

### 5.164.2 Function Documentation

#### 5.164.2.1 `int cbc_encrypt` (const unsigned char \*pt, unsigned char \*ct, unsigned long len, symmetric\_CBC \*cbc)

CBC encrypt.

#### Parameters:

*pt* Plaintext

*ct* [out] Ciphertext

*len* The number of bytes to process (must be multiple of block length)

*cbc* CBC state

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file cbc\_encrypt.c.

References `ltc_cipher_descriptor::accel_cbc_encrypt`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `ltc_cipher_descriptor::ecb_encrypt`, and `LTC_ARGCHK`.

```
30 {
31     int x, err;
32
33     LTC_ARGCHK(pt != NULL);
34     LTC_ARGCHK(ct != NULL);
35     LTC_ARGCHK(cbc != NULL);
36
37     if ((err = cipher_is_valid(cbc->cipher)) != CRYPT_OK) {
38         return err;
39     }
40
41     /* is blocklen valid? */
42     if (cbc->blocklen < 1 || cbc->blocklen > (int)sizeof(cbc->IV)) {
43         return CRYPT_INVALID_ARG;
44     }
```

```

45
46     if (len % cbc->blocklen) {
47         return CRYPT_INVALID_ARG;
48     }
49 #ifdef LTC_FAST
50     if (cbc->blocklen % sizeof(LTC_FAST_TYPE)) {
51         return CRYPT_INVALID_ARG;
52     }
53 #endif
54
55     if (cipher_descriptor[cbc->cipher].accel_cbc_encrypt != NULL) {
56         return cipher_descriptor[cbc->cipher].accel_cbc_encrypt(pt, ct, len / cbc->blocklen, cbc->IV, &cbc->key);
57     } else {
58         while (len) {
59             /* xor IV against plaintext */
60             #if defined(LTC_FAST)
61                 for (x = 0; x < cbc->blocklen; x += sizeof(LTC_FAST_TYPE)) {
62                     *((LTC_FAST_TYPE*)((unsigned char *)cbc->IV + x)) ^= *((LTC_FAST_TYPE*)((unsigned char *)ct + x));
63                 }
64             #else
65                 for (x = 0; x < cbc->blocklen; x++) {
66                     cbc->IV[x] ^= pt[x];
67                 }
68             #endif
69
70             /* encrypt */
71             if ((err = cipher_descriptor[cbc->cipher].ecb_encrypt(cbc->IV, ct, &cbc->key)) != CRYPT_OK) {
72                 return err;
73             }
74
75             /* store IV [ciphertext] for a future block */
76             #if defined(LTC_FAST)
77                 for (x = 0; x < cbc->blocklen; x += sizeof(LTC_FAST_TYPE)) {
78                     *((LTC_FAST_TYPE*)((unsigned char *)cbc->IV + x)) = *((LTC_FAST_TYPE*)((unsigned char *)ct + x));
79                 }
80             #else
81                 for (x = 0; x < cbc->blocklen; x++) {
82                     cbc->IV[x] = ct[x];
83                 }
84             #endif
85
86             ct += cbc->blocklen;
87             pt += cbc->blocklen;
88             len -= cbc->blocklen;
89         }
90     }
91     return CRYPT_OK;
92 }

```

Here is the call graph for this function:



## 5.165 modes/cbc/cbc\_getiv.c File Reference

### 5.165.1 Detailed Description

CBC implementation, get IV, Tom St Denis.

Definition in file [cbc\\_getiv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cbc\_getiv.c:

### Functions

- int [cbc\\_getiv](#) (unsigned char \*IV, unsigned long \*len, symmetric\_CBC \*cbc)

*Get the current initial vector.*

### 5.165.2 Function Documentation

#### 5.165.2.1 int cbc\_getiv (unsigned char \*IV, unsigned long \*len, symmetric\_CBC \*cbc)

Get the current initial vector.

##### Parameters:

*IV* [out] The destination of the initial vector

*len* [in/out] The max size and resulting size of the initial vector

*cbc* The CBC state

##### Returns:

CRYPT\_OK if successful

Definition at line 27 of file cbc\_getiv.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_OK, LTC\_ARGCHK, and XMEMCPY.

```
28 {
29     LTC_ARGCHK(IV != NULL);
30     LTC_ARGCHK(len != NULL);
31     LTC_ARGCHK(cbc != NULL);
32     if ((unsigned long)cbc->blocklen > *len) {
33         *len = cbc->blocklen;
34         return CRYPT_BUFFER_OVERFLOW;
35     }
36     XMEMCPY(IV, cbc->IV, cbc->blocklen);
37     *len = cbc->blocklen;
38
39     return CRYPT_OK;
40 }
```

## 5.166 modes/cbc/cbc\_setiv.c File Reference

### 5.166.1 Detailed Description

CBC implementation, set IV, Tom St Denis.

Definition in file [cbc\\_setiv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cbc\_setiv.c:

### Functions

- int [cbc\\_setiv](#) (const unsigned char \*IV, unsigned long [len](#), symmetric\_CBC \*cbc)  
*Set an initial vector.*

### 5.166.2 Function Documentation

#### 5.166.2.1 int cbc\_setiv (const unsigned char \*IV, unsigned long len, symmetric\_CBC \*cbc)

Set an initial vector.

##### Parameters:

- IV* The initial vector  
*len* The length of the vector (in octets)  
*cbc* The CBC state

##### Returns:

CRYPT\_OK if successful

Definition at line 28 of file cbc\_setiv.c.

References [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), and [XMEMCPY](#).

```
29 {
30     LTC_ARGCHK(IV != NULL);
31     LTC_ARGCHK(cbc != NULL);
32     if (len != (unsigned long)cbc->blocklen) {
33         return CRYPT_INVALID_ARG;
34     }
35     XMEMCPY(cbc->IV, IV, len);
36     return CRYPT_OK;
37 }
```

## 5.167 modes/cbc/cbc\_start.c File Reference

### 5.167.1 Detailed Description

CBC implementation, start chain, Tom St Denis.

Definition in file [cbc\\_start.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cbc\_start.c:

### Functions

- [int cbc\\_start](#) (int cipher, const unsigned char \*IV, const unsigned char \*key, int keylen, int num\_rounds, symmetric\_CBC \*cbc)

*Initialize a CBC context.*

### 5.167.2 Function Documentation

#### 5.167.2.1 int cbc\_start (int cipher, const unsigned char \*IV, const unsigned char \*key, int keylen, int num\_rounds, symmetric\_CBC \*cbc)

Initialize a CBC context.

#### Parameters:

*cipher* The index of the cipher desired

*IV* The initial vector

*key* The secret key

*keylen* The length of the secret key (octets)

*num\_rounds* Number of rounds in the cipher desired (0 for default)

*cbc* The CBC state to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 30 of file cbc\_start.c.

References [ltc\\_cipher\\_descriptor::block\\_length](#), [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_OK](#), and [LTC\\_ARGCHK](#).

```
32 {
33     int x, err;
34
35     LTC_ARGCHK(IV != NULL);
36     LTC_ARGCHK(key != NULL);
37     LTC_ARGCHK(cbc != NULL);
38
39     /* bad param? */
40     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
41         return err;
42     }
43 }
```

```
44  /* setup cipher */
45  if ((err = cipher_descriptor[cipher].setup(key, keylen, num_rounds, &cbc->key)) != CRYPT_OK) {
46      return err;
47  }
48
49  /* copy IV */
50  cbc->blocklen = cipher_descriptor[cipher].block_length;
51  cbc->cipher    = cipher;
52  for (x = 0; x < cbc->blocklen; x++) {
53      cbc->IV[x] = IV[x];
54  }
55  return CRYPT_OK;
56 }
```

Here is the call graph for this function:

## 5.168 modes/cfb/cfb\_decrypt.c File Reference

### 5.168.1 Detailed Description

CFB implementation, decrypt data, Tom St Denis.

Definition in file [cfb\\_decrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cfb\_decrypt.c:

### Functions

- int [cfb\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, unsigned long [len](#), symmetric\_CFB \*cfb)  
*CFB decrypt.*

### 5.168.2 Function Documentation

#### 5.168.2.1 int cfb\_decrypt (const unsigned char \* ct, unsigned char \* pt, unsigned long len, symmetric\_CFB \* cfb)

CFB decrypt.

##### Parameters:

*ct* Ciphertext  
*pt* [out] Plaintext  
*len* Length of ciphertext (octets)  
*cfb* CFB state

##### Returns:

CRYPT\_OK if successful

Definition at line 28 of file cfb\_decrypt.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), and [LTC\\_ARGCHK](#).

```
29 {
30     int err;
31
32     LTC_ARGCHK(pt != NULL);
33     LTC_ARGCHK(ct != NULL);
34     LTC_ARGCHK(cfb != NULL);
35
36     if ((err = cipher_is_valid(cfb->cipher)) != CRYPT_OK) {
37         return err;
38     }
39
40     /* is blocklen/padlen valid? */
41     if (cfb->blocklen < 0 || cfb->blocklen > (int)sizeof(cfb->IV) ||
42         cfb->padlen < 0 || cfb->padlen > (int)sizeof(cfb->pad)) {
43         return CRYPT_INVALID_ARG;
44     }
45 }
```

```
44     }
45
46     while (len-- > 0) {
47         if (cfb->padlen == cfb->blocklen) {
48             if ((err = cipher_descriptor[cfb->cipher].ecb_encrypt(cfb->pad, cfb->IV, &cfb->key)) != CRYPT_OK)
49                 return err;
50             }
51             cfb->padlen = 0;
52         }
53         cfb->pad[cfb->padlen] = *ct;
54         *pt = *ct ^ cfb->IV[cfb->padlen];
55         ++pt;
56         ++ct;
57         ++cfb->padlen;
58     }
59     return CRYPT_OK;
60 }
```

Here is the call graph for this function:

## 5.169 modes/cfb/cfb\_done.c File Reference

### 5.169.1 Detailed Description

CFB implementation, finish chain, Tom St Denis.

Definition in file [cfb\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cfb\_done.c:

### Functions

- [int cfb\\_done](#) (symmetric\_CFB \*cfb)  
*Terminate the chain.*

### 5.169.2 Function Documentation

#### 5.169.2.1 int cfb\_done (symmetric\_CFB \*cfb)

Terminate the chain.

##### Parameters:

*cfb* The CFB chain to terminate

##### Returns:

CRYPT\_OK on success

Definition at line 24 of file cfb\_done.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::done](#), and [LTC\\_ARGCHK](#).

```
25 {
26     int err;
27     LTC_ARGCHK(cfb != NULL);
28
29     if ((err = cipher_is_valid(cfb->cipher)) != CRYPT_OK) {
30         return err;
31     }
32     cipher_descriptor[cfb->cipher].done(&cfb->key);
33     return CRYPT_OK;
34 }
```

Here is the call graph for this function:

## 5.170 modes/cfb/cfb\_encrypt.c File Reference

### 5.170.1 Detailed Description

CFB implementation, encrypt data, Tom St Denis.

Definition in file [cfb\\_encrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cfb\_encrypt.c:

### Functions

- int [cfb\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, unsigned long [len](#), symmetric\_CFB \*cfb)  
*CFB encrypt.*

### 5.170.2 Function Documentation

#### 5.170.2.1 int cfb\_encrypt (const unsigned char \*pt, unsigned char \*ct, unsigned long len, symmetric\_CFB \*cfb)

CFB encrypt.

#### Parameters:

*pt* Plaintext  
*ct* [out] Ciphertext  
*len* Length of plaintext (octets)  
*cfb* CFB state

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file cfb\_encrypt.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), and [LTC\\_ARGCHK](#).

```
29 {
30     int err;
31
32     LTC_ARGCHK(pt != NULL);
33     LTC_ARGCHK(ct != NULL);
34     LTC_ARGCHK(cfb != NULL);
35
36     if ((err = cipher_is_valid(cfb->cipher)) != CRYPT_OK) {
37         return err;
38     }
39
40     /* is blocklen/padlen valid? */
41     if (cfb->blocklen < 0 || cfb->blocklen > (int)sizeof(cfb->IV) ||
42         cfb->padlen < 0 || cfb->padlen > (int)sizeof(cfb->pad)) {
43         return CRYPT_INVALID_ARG;
44     }
45 }
```



```
44     }
45
46     while (len-- > 0) {
47         if (cfb->padlen == cfb->blocklen) {
48             if ((err = cipher_descriptor[cfb->cipher].ecb_encrypt(cfb->pad, cfb->IV, &cfb->key)) != CRYPT_OK)
49                 return err;
50             }
51             cfb->padlen = 0;
52         }
53         cfb->pad[cfb->padlen] = (*ct = *pt ^ cfb->IV[cfb->padlen]);
54         ++pt;
55         ++ct;
56         ++cfb->padlen;
57     }
58     return CRYPT_OK;
59 }
```

Here is the call graph for this function:

## 5.171 modes/cfb/cfb\_getiv.c File Reference

### 5.171.1 Detailed Description

CFB implementation, get IV, Tom St Denis.

Definition in file [cfb\\_getiv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cfb\_getiv.c:

### Functions

- int [cfb\\_getiv](#) (unsigned char \*IV, unsigned long \*len, symmetric\_CFB \*cfb)

*Get the current initial vector.*

### 5.171.2 Function Documentation

#### 5.171.2.1 int cfb\_getiv (unsigned char \* IV, unsigned long \* len, symmetric\_CFB \* cfb)

Get the current initial vector.

##### Parameters:

*IV* [out] The destination of the initial vector

*len* [in/out] The max size and resulting size of the initial vector

*cfb* The CFB state

##### Returns:

CRYPT\_OK if successful

Definition at line 27 of file cfb\_getiv.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_OK, LTC\_ARGCHK, and XMEMCPY.

```
28 {
29     LTC_ARGCHK(IV != NULL);
30     LTC_ARGCHK(len != NULL);
31     LTC_ARGCHK(cfb != NULL);
32     if ((unsigned long)cfb->blocklen > *len) {
33         *len = cfb->blocklen;
34         return CRYPT_BUFFER_OVERFLOW;
35     }
36     XMEMCPY(IV, cfb->IV, cfb->blocklen);
37     *len = cfb->blocklen;
38
39     return CRYPT_OK;
40 }
```

## 5.172 modes/cfb/cfb\_setiv.c File Reference

### 5.172.1 Detailed Description

CFB implementation, set IV, Tom St Denis.

Definition in file [cfb\\_setiv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cfb\_setiv.c:

### Functions

- [int cfb\\_setiv](#) (const unsigned char \*IV, unsigned long [len](#), symmetric\_CFB \*cfb)  
*Set an initial vector.*

### 5.172.2 Function Documentation

#### 5.172.2.1 int cfb\_setiv (const unsigned char \*IV, unsigned long len, symmetric\_CFB \*cfb)

Set an initial vector.

##### Parameters:

- IV* The initial vector
- len* The length of the vector (in octets)
- cfb* The CFB state

##### Returns:

CRYPT\_OK if successful

Definition at line 27 of file cfb\_setiv.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), and [LTC\\_ARGCHK](#).

```
28 {
29     int err;
30
31     LTC_ARGCHK(IV != NULL);
32     LTC_ARGCHK(cfb != NULL);
33
34     if ((err = cipher_is_valid(cfb->cipher)) != CRYPT_OK) {
35         return err;
36     }
37
38     if (len != (unsigned long)cfb->blocklen) {
39         return CRYPT_INVALID_ARG;
40     }
41
42     /* force next block */
43     cfb->padlen = 0;
44     return cipher_descriptor[cfb->cipher].ecb_encrypt(IV, cfb->IV, &cfb->key);
45 }
```

Here is the call graph for this function:

## 5.173 modes/cfb/cfb\_start.c File Reference

### 5.173.1 Detailed Description

CFB implementation, start chain, Tom St Denis.

Definition in file [cfb\\_start.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for cfb\_start.c:

### Functions

- [int cfb\\_start](#) (int cipher, const unsigned char \*IV, const unsigned char \*key, int keylen, int num\_rounds, symmetric\_CFB \*cfb)

*Initialize a CFB context.*

### 5.173.2 Function Documentation

#### 5.173.2.1 int cfb\_start (int cipher, const unsigned char \*IV, const unsigned char \*key, int keylen, int num\_rounds, symmetric\_CFB \*cfb)

Initialize a CFB context.

#### Parameters:

**cipher** The index of the cipher desired

**IV** The initial vector

**key** The secret key

**keylen** The length of the secret key (octets)

**num\_rounds** Number of rounds in the cipher desired (0 for default)

**cfb** The CFB state to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 31 of file cfb\_start.c.

References `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_OK`, and `LTC_ARGCHK`.

```
33 {
34     int x, err;
35
36     LTC_ARGCHK(IV != NULL);
37     LTC_ARGCHK(key != NULL);
38     LTC_ARGCHK(cfb != NULL);
39
40     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
41         return err;
42     }
43
44
```

```
45  /* copy data */
46  cfb->cipher = cipher;
47  cfb->blocklen = cipher_descriptor[cipher].block_length;
48  for (x = 0; x < cfb->blocklen; x++)
49      cfb->IV[x] = IV[x];
50
51  /* init the cipher */
52  if ((err = cipher_descriptor[cipher].setup(key, keylen, num_rounds, &cfb->key)) != CRYPT_OK) {
53      return err;
54  }
55
56  /* encrypt the IV */
57  cfb->padlen = 0;
58  return cipher_descriptor[cfb->cipher].ecb_encrypt(cfb->IV, cfb->IV, &cfb->key);
59 }
```

Here is the call graph for this function:

## 5.174 modes/ctr/ctr\_decrypt.c File Reference

### 5.174.1 Detailed Description

CTR implementation, decrypt data, Tom St Denis.

Definition in file [ctr\\_decrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ctr\_decrypt.c:

### Functions

- int [ctr\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, unsigned long [len](#), symmetric\_CTR \*ctr)  
*CTR decrypt.*

### 5.174.2 Function Documentation

#### 5.174.2.1 int ctr\_decrypt (const unsigned char \* ct, unsigned char \* pt, unsigned long len, symmetric\_CTR \* ctr)

CTR decrypt.

#### Parameters:

*ct* Ciphertext  
*pt* [out] Plaintext  
*len* Length of ciphertext (octets)  
*ctr* CTR state

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file ctr\_decrypt.c.

References [ctr\\_encrypt\(\)](#), and [LTC\\_ARGCHK](#).

Referenced by [eax\\_decrypt\(\)](#).

```
29 {  
30     LTC_ARGCHK(pt != NULL);  
31     LTC_ARGCHK(ct != NULL);  
32     LTC_ARGCHK(ctr != NULL);  
33  
34     return ctr_encrypt(ct, pt, len, ctr);  
35 }
```

Here is the call graph for this function:

## 5.175 modes/ctr/ctr\_done.c File Reference

### 5.175.1 Detailed Description

CTR implementation, finish chain, Tom St Denis.

Definition in file [ctr\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ctr\_done.c:

### Functions

- [int ctr\\_done](#) (symmetric\_CTR \*ctr)  
*Terminate the chain.*

### 5.175.2 Function Documentation

#### 5.175.2.1 int ctr\_done (symmetric\_CTR \* ctr)

Terminate the chain.

##### Parameters:

*ctr* The CTR chain to terminate

##### Returns:

CRYPT\_OK on success

Definition at line 24 of file ctr\_done.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::done](#), and [LTC\\_ARGCHK](#).

Referenced by [eax\\_done\(\)](#), and [yarrow\\_done\(\)](#).

```
25 {  
26     int err;  
27     LTC_ARGCHK(ctr != NULL);  
28  
29     if ((err = cipher_is_valid(ctr->cipher)) != CRYPT_OK) {  
30         return err;  
31     }  
32     cipher_descriptor[ctr->cipher].done(&ctr->key);  
33     return CRYPT_OK;  
34 }
```

Here is the call graph for this function:

## 5.176 modes/ctr/ctr\_encrypt.c File Reference

### 5.176.1 Detailed Description

CTR implementation, encrypt data, Tom St Denis.

Definition in file [ctr\\_encrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ctr\_encrypt.c:

### Functions

- int [ctr\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, unsigned long [len](#), symmetric\_CTR \*ctr)  
*CTR encrypt.*

### 5.176.2 Function Documentation

#### 5.176.2.1 int ctr\_encrypt (const unsigned char \*pt, unsigned char \*ct, unsigned long len, symmetric\_CTR \*ctr)

CTR encrypt.

#### Parameters:

*pt* Plaintext  
*ct* [out] Ciphertext  
*len* Length of plaintext (octets)  
*ctr* CTR state

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file ctr\_encrypt.c.

References [ltc\\_cipher\\_descriptor::accel\\_ctr\\_encrypt](#), [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), and [LTC\\_ARGCHK](#).

Referenced by [ctr\\_decrypt\(\)](#), [eax\\_encrypt\(\)](#), and [yarrow\\_read\(\)](#).

```
30 {
31     int x, err;
32
33     LTC_ARGCHK(pt != NULL);
34     LTC_ARGCHK(ct != NULL);
35     LTC_ARGCHK(ctr != NULL);
36
37     if ((err = cipher_is_valid(ctr->cipher)) != CRYPT_OK) {
38         return err;
39     }
40
41     /* is blocklen/padlen valid? */
42     if (ctr->blocklen < 1 || ctr->blocklen > (int)sizeof(ctr->ctr) ||
```



```

43     ctr->padlen < 0 || ctr->padlen > (int)sizeof(ctr->pad)) {
44     return CRYPT_INVALID_ARG;
45 }
46
47 #ifdef LTC_FAST
48     if (ctr->blocklen % sizeof(LTC_FAST_TYPE)) {
49         return CRYPT_INVALID_ARG;
50     }
51 #endif
52
53     /* handle acceleration only if pad is empty, accelerator is present and length is >= a block size */
54     if ((ctr->padlen == ctr->blocklen) && cipher_descriptor[ctr->cipher].accel_ctr_encrypt != NULL && (1
55         if ((err = cipher_descriptor[ctr->cipher].accel_ctr_encrypt(pt, ct, len/ctr->blocklen, ctr->ctr,
56             return err;
57         }
58         len %= ctr->blocklen;
59     }
60
61     while (len) {
62         /* is the pad empty? */
63         if (ctr->padlen == ctr->blocklen) {
64             /* increment counter */
65             if (ctr->mode == CTR_COUNTER_LITTLE_ENDIAN) {
66                 /* little-endian */
67                 for (x = 0; x < ctr->blocklen; x++) {
68                     ctr->ctr[x] = (ctr->ctr[x] + (unsigned char)1) & (unsigned char)255;
69                     if (ctr->ctr[x] != (unsigned char)0) {
70                         break;
71                     }
72                 }
73             } else {
74                 /* big-endian */
75                 for (x = ctr->blocklen-1; x >= 0; x--) {
76                     ctr->ctr[x] = (ctr->ctr[x] + (unsigned char)1) & (unsigned char)255;
77                     if (ctr->ctr[x] != (unsigned char)0) {
78                         break;
79                     }
80                 }
81             }
82
83             /* encrypt it */
84             if ((err = cipher_descriptor[ctr->cipher].ecb_encrypt(ctr->ctr, ctr->pad, &ctr->key)) != CRYPT
85                 return err;
86             }
87             ctr->padlen = 0;
88         }
89 #ifdef LTC_FAST
90         if (ctr->padlen == 0 && len >= (unsigned long)ctr->blocklen) {
91             for (x = 0; x < ctr->blocklen; x += sizeof(LTC_FAST_TYPE)) {
92                 *((LTC_FAST_TYPE*)((unsigned char *)ct + x)) = *((LTC_FAST_TYPE*)((unsigned char *)pt + x))
93                 *((LTC_FAST_TYPE*)((unsigned char *)ctr->pad + x)) = *((LTC_FAST_TYPE*)((unsigned char *)ctr->ctr + x))
94             }
95             pt += ctr->blocklen;
96             ct += ctr->blocklen;
97             len -= ctr->blocklen;
98             ctr->padlen = ctr->blocklen;
99             continue;
100         }
101 #endif
102         *ct++ = *pt++ ^ ctr->pad[ctr->padlen++];
103         --len;
104     }
105     return CRYPT_OK;
106 }

```

Here is the call graph for this function:

## 5.177 modes/ctr/ctr\_getiv.c File Reference

### 5.177.1 Detailed Description

CTR implementation, get IV, Tom St Denis.

Definition in file [ctr\\_getiv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ctr\_getiv.c:

### Functions

- int [ctr\\_getiv](#) (unsigned char \*IV, unsigned long \*len, symmetric\_CTR \*ctr)

*Get the current initial vector.*

### 5.177.2 Function Documentation

#### 5.177.2.1 int ctr\_getiv (unsigned char \*IV, unsigned long \*len, symmetric\_CTR \*ctr)

Get the current initial vector.

#### Parameters:

*IV* [out] The destination of the initial vector

*len* [in/out] The max size and resulting size of the initial vector

*ctr* The CTR state

#### Returns:

CRYPT\_OK if successful

Definition at line 27 of file ctr\_getiv.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_OK, LTC\_ARGCHK, and XMEMCPY.

```
28 {
29     LTC_ARGCHK(IV != NULL);
30     LTC_ARGCHK(len != NULL);
31     LTC_ARGCHK(ctr != NULL);
32     if ((unsigned long)ctr->blocklen > *len) {
33         *len = ctr->blocklen;
34         return CRYPT_BUFFER_OVERFLOW;
35     }
36     XMEMCPY(IV, ctr->ctr, ctr->blocklen);
37     *len = ctr->blocklen;
38
39     return CRYPT_OK;
40 }
```

## 5.178 modes/ctr/ctr\_setiv.c File Reference

### 5.178.1 Detailed Description

CTR implementation, set IV, Tom St Denis.

Definition in file [ctr\\_setiv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ctr\_setiv.c:

### Functions

- [int ctr\\_setiv](#) (const unsigned char \*IV, unsigned long [len](#), symmetric\_CTR \*ctr)  
*Set an initial vector.*

### 5.178.2 Function Documentation

#### 5.178.2.1 int ctr\_setiv (const unsigned char \*IV, unsigned long len, symmetric\_CTR \*ctr)

Set an initial vector.

##### Parameters:

- IV** The initial vector
- len** The length of the vector (in octets)
- ctr** The CTR state

##### Returns:

- CRYPT\_OK if successful

Definition at line 27 of file ctr\_setiv.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), [LTC\\_ARGCHK](#), and [XMEMCPY](#).

```
28 {
29     int err;
30
31     LTC_ARGCHK(IV != NULL);
32     LTC_ARGCHK(ctr != NULL);
33
34     /* bad param? */
35     if ((err = cipher_is_valid(ctr->cipher)) != CRYPT_OK) {
36         return err;
37     }
38
39     if (len != (unsigned long)ctr->blocklen) {
40         return CRYPT_INVALID_ARG;
41     }
42
43     /* set IV */
44     XMEMCPY(ctr->ctr, IV, len);
45
46     /* force next block */
47     ctr->padlen = 0;
```

```
48     return cipher_descriptor[ctr->cipher].ecb_encrypt (IV, ctr->pad, &ctr->key);  
49 }
```

Here is the call graph for this function:

## 5.179 modes/ctr/ctr\_start.c File Reference

### 5.179.1 Detailed Description

CTR implementation, start chain, Tom St Denis.

Definition in file [ctr\\_start.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ctr\_start.c:

### Functions

- `int ctr_start` (int cipher, const unsigned char \*IV, const unsigned char \*key, int keylen, int num\_rounds, int ctr\_mode, symmetric\_CTR \*ctr)

*Initialize a CTR context.*

### 5.179.2 Function Documentation

**5.179.2.1** `int ctr_start` (int cipher, const unsigned char \*IV, const unsigned char \*key, int keylen, int num\_rounds, int ctr\_mode, symmetric\_CTR \*ctr)

Initialize a CTR context.

#### Parameters:

*cipher* The index of the cipher desired

*IV* The initial vector

*key* The secret key

*keylen* The length of the secret key (octets)

*num\_rounds* Number of rounds in the cipher desired (0 for default)

*ctr\_mode* The counter mode (CTR\_COUNTER\_LITTLE\_ENDIAN or CTR\_COUNTER\_BIG\_ENDIAN)

*ctr* The CTR state to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 32 of file ctr\_start.c.

References `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_OK`, and `LTC_ARGCHK`.

Referenced by `ctr_test()`, `eax_init()`, and `yarrow_ready()`.

```
37 {
38     int x, err;
39
40     LTC_ARGCHK(IV != NULL);
41     LTC_ARGCHK(key != NULL);
42     LTC_ARGCHK(ctr != NULL);
43 }
```

```
44  /* bad param? */
45  if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
46      return err;
47  }
48
49  /* setup cipher */
50  if ((err = cipher_descriptor[cipher].setup(key, keylen, num_rounds, &ctr->key)) != CRYPT_OK) {
51      return err;
52  }
53
54  /* copy ctr */
55  ctr->blocklen = cipher_descriptor[cipher].block_length;
56  ctr->cipher    = cipher;
57  ctr->padlen    = 0;
58  ctr->mode      = ctr_mode & 1;
59  for (x = 0; x < ctr->blocklen; x++) {
60      ctr->ctr[x] = IV[x];
61  }
62
63  if (ctr_mode & LTC_CTR_RFC3686) {
64      /* increment the IV as per RFC 3686 */
65      if (ctr->mode == CTR_COUNTER_LITTLE_ENDIAN) {
66          /* little-endian */
67          for (x = 0; x < ctr->blocklen; x++) {
68              ctr->ctr[x] = (ctr->ctr[x] + (unsigned char)1) & (unsigned char)255;
69              if (ctr->ctr[x] != (unsigned char)0) {
70                  break;
71              }
72          }
73      } else {
74          /* big-endian */
75          for (x = ctr->blocklen-1; x >= 0; x--) {
76              ctr->ctr[x] = (ctr->ctr[x] + (unsigned char)1) & (unsigned char)255;
77              if (ctr->ctr[x] != (unsigned char)0) {
78                  break;
79              }
80          }
81      }
82  }
83
84  return cipher_descriptor[ctr->cipher].ecb_encrypt(ctr->ctr, ctr->pad, &ctr->key);
85 }
```

Here is the call graph for this function:

## 5.180 modes/ctr/ctr\_test.c File Reference

### 5.180.1 Detailed Description

CTR implementation, Tests again RFC 3686, Tom St Denis.

Definition in file [ctr\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ctr\_test.c:

### Functions

- [int ctr\\_test](#) (void)

### 5.180.2 Function Documentation

#### 5.180.2.1 int ctr\_test (void)

Definition at line 20 of file ctr\_test.c.

References CRYPT\_NOP, CRYPT\_OK, ctr\_start(), and find\_cipher().

```
21 {
22 #ifdef LTC_NO_TEST
23     return CRYPT_NOP;
24 #else
25     static const struct {
26         int keylen, msglen;
27         unsigned char key[32], IV[16], pt[64], ct[64];
28     } tests[] = {
29 /* 128-bit key, 16-byte pt */
30 {
31     16, 16,
32     {0xAE,0x68,0x52,0xF8,0x12,0x10,0x67,0xCC,0x4B,0xF7,0xA5,0x76,0x55,0x77,0xF3,0x9E },
33     {0x00,0x00,0x00,0x30,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 },
34     {0x53,0x69,0x6E,0x67,0x6C,0x65,0x20,0x62,0x6C,0x6F,0x63,0x6B,0x20,0x6D,0x73,0x67 },
35     {0xE4,0x09,0x5D,0x4F,0xB7,0xA7,0xB3,0x79,0x2D,0x61,0x75,0xA3,0x26,0x13,0x11,0xB8 },
36 },
37
38 /* 128-bit key, 36-byte pt */
39 {
40     16, 36,
41     {0x76,0x91,0xBE,0x03,0x5E,0x50,0x20,0xA8,0xAC,0x6E,0x61,0x85,0x29,0xF9,0xA0,0xDC },
42     {0x00,0xE0,0x01,0x7B,0x27,0x77,0x7F,0x3F,0x4A,0x17,0x86,0xF0,0x00,0x00,0x00,0x00 },
43     {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,
44     0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D,0x1E,0x1F,
45     0x20,0x21,0x22,0x23},
46     {0xC1,0xCF,0x48,0xA8,0x9F,0x2F,0xFD,0xD9,0xCF,0x46,0x52,0xE9,0xEF,0xDB,0x72,0xD7,
47     0x45,0x40,0xA4,0x2B,0xDE,0x6D,0x78,0x36,0xD5,0x9A,0x5C,0xEA,0xAE,0xF3,0x10,0x53,
48     0x25,0xB2,0x07,0x2F },
49 },
50 };
51 int idx, err, x;
52 unsigned char buf[64];
53 symmetric_CTR ctr;
54
55 /* AES can be under rijndael or aes... try to find it */
56 if ((idx = find_cipher("aes")) == -1) {
57     if ((idx = find_cipher("rijndael")) == -1) {
```

```
58         return CRYPT_NOP;
59     }
60 }
61
62 for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
63     if ((err = ctr_start(idx, tests[x].IV, tests[x].key, tests[x].keylen, 0, CTR_COUNTER_BIG_ENDIAN|L
64         return err;
65     }
66     if ((err = ctr_encrypt(tests[x].pt, buf, tests[x].msglen, &ctr)) != CRYPT_OK) {
67         return err;
68     }
69     ctr_done(&ctr);
70     if (XMEMCMP(buf, tests[x].ct, tests[x].msglen)) {
71         return CRYPT_FAIL_TESTVECTOR;
72     }
73 }
74 return CRYPT_OK;
75 #endif
76 }
```

Here is the call graph for this function:



## 5.181 modes/ecb/ecb\_decrypt.c File Reference

### 5.181.1 Detailed Description

ECB implementation, decrypt a block, Tom St Denis.

Definition in file [ecb\\_decrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecb\_decrypt.c:

### Functions

- `int ecb_decrypt` (const unsigned char \*ct, unsigned char \*pt, unsigned long len, symmetric\_ECB \*ecb)  
*ECB decrypt.*

### 5.181.2 Function Documentation

#### 5.181.2.1 `int ecb_decrypt` (const unsigned char \*ct, unsigned char \*pt, unsigned long len, symmetric\_ECB \*ecb)

ECB decrypt.

##### Parameters:

*ct* Ciphertext

*pt* [out] Plaintext

*len* The number of octets to process (must be multiple of the cipher block size)

*ecb* ECB state

##### Returns:

CRYPT\_OK if successful

Definition at line 28 of file ecb\_decrypt.c.

References `ltc_cipher_descriptor::accel_ecb_decrypt`, `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `ltc_cipher_descriptor::ecb_decrypt`, and `LTC_ARGCHK`.

```
29 {
30     int err;
31     LTC_ARGCHK(pt != NULL);
32     LTC_ARGCHK(ct != NULL);
33     LTC_ARGCHK(ecb != NULL);
34     if ((err = cipher_is_valid(ecb->cipher)) != CRYPT_OK) {
35         return err;
36     }
37     if (len % cipher_descriptor[ecb->cipher].block_length) {
38         return CRYPT_INVALID_ARG;
39     }
40
41     /* check for accel */
42     if (cipher_descriptor[ecb->cipher].accel_ecb_decrypt != NULL) {
```

```
43     return cipher_descriptor[ecb->cipher].accel_ecb_decrypt(ct, pt, len / cipher_descriptor[ecb->cipher].block_length, &ecb->key);
44 } else {
45     while (len) {
46         if ((err = cipher_descriptor[ecb->cipher].ecb_decrypt(ct, pt, &ecb->key)) != CRYPT_OK) {
47             return err;
48         }
49         pt += cipher_descriptor[ecb->cipher].block_length;
50         ct += cipher_descriptor[ecb->cipher].block_length;
51         len -= cipher_descriptor[ecb->cipher].block_length;
52     }
53 }
54 return CRYPT_OK;
55 }
```

Here is the call graph for this function:

## 5.182 modes/ecb/ecb\_done.c File Reference

### 5.182.1 Detailed Description

ECB implementation, finish chain, Tom St Denis.

Definition in file [ecb\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecb\_done.c:

### Functions

- `int ecb_done (symmetric_ECB *ecb)`  
*Terminate the chain.*

### 5.182.2 Function Documentation

#### 5.182.2.1 `int ecb_done (symmetric_ECB * ecb)`

Terminate the chain.

##### Parameters:

*ecb* The ECB chain to terminate

##### Returns:

CRYPT\_OK on success

Definition at line 24 of file ecb\_done.c.

References `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_OK`, `ltc_cipher_descriptor::done`, and `LTC_ARGCHK`.

```
25 {  
26     int err;  
27     LTC_ARGCHK(ecb != NULL);  
28  
29     if ((err = cipher_is_valid(ecb->cipher)) != CRYPT_OK) {  
30         return err;  
31     }  
32     cipher_descriptor[ecb->cipher].done(&ecb->key);  
33     return CRYPT_OK;  
34 }
```

Here is the call graph for this function:

## 5.183 modes/ecb/ecb\_encrypt.c File Reference

### 5.183.1 Detailed Description

ECB implementation, encrypt a block, Tom St Denis.

Definition in file [ecb\\_encrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecb\_encrypt.c:

### Functions

- `int ecb_encrypt` (const unsigned char \*pt, unsigned char \*ct, unsigned long len, symmetric\_ECB \*ecb)  
*ECB encrypt.*

### 5.183.2 Function Documentation

#### 5.183.2.1 `int ecb_encrypt` (const unsigned char \*pt, unsigned char \*ct, unsigned long len, symmetric\_ECB \*ecb)

ECB encrypt.

#### Parameters:

*pt* Plaintext

*ct* [out] Ciphertext

*len* The number of octets to process (must be multiple of the cipher block size)

*ecb* ECB state

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file ecb\_encrypt.c.

References `ltc_cipher_descriptor::accel_ecb_encrypt`, `ltc_cipher_descriptor::block_length`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `ltc_cipher_descriptor::ecb_encrypt`, and `LTC_ARGCHK`.

Referenced by `gcm_init()`, and `omac_init()`.

```
29 {
30     int err;
31     LTC_ARGCHK(pt != NULL);
32     LTC_ARGCHK(ct != NULL);
33     LTC_ARGCHK(ecb != NULL);
34     if ((err = cipher_is_valid(ecb->cipher)) != CRYPT_OK) {
35         return err;
36     }
37     if (len % cipher_descriptor[ecb->cipher].block_length) {
38         return CRYPT_INVALID_ARG;
39     }
40 }
```

```
41  /* check for accel */
42  if (cipher_descriptor[ecb->cipher].accel_ecb_encrypt != NULL) {
43      return cipher_descriptor[ecb->cipher].accel_ecb_encrypt(pt, ct, len / cipher_descriptor[ecb->cipher].block_length);
44  } else {
45      while (len) {
46          if ((err = cipher_descriptor[ecb->cipher].ecb_encrypt(pt, ct, &ecb->key)) != CRYPT_OK) {
47              return err;
48          }
49          pt += cipher_descriptor[ecb->cipher].block_length;
50          ct += cipher_descriptor[ecb->cipher].block_length;
51          len -= cipher_descriptor[ecb->cipher].block_length;
52      }
53  }
54  return CRYPT_OK;
55 }
```

Here is the call graph for this function:

## 5.184 modes/ecb/ecb\_start.c File Reference

### 5.184.1 Detailed Description

ECB implementation, start chain, Tom St Denis.

Definition in file [ecb\\_start.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecb\_start.c:

### Functions

- [int ecb\\_start](#) (int cipher, const unsigned char \*key, int keylen, int num\_rounds, symmetric\_ECB \*ecb)

*Initialize a ECB context.*

### 5.184.2 Function Documentation

#### 5.184.2.1 int ecb\_start (int *cipher*, const unsigned char \* *key*, int *keylen*, int *num\_rounds*, symmetric\_ECB \* *ecb*)

Initialize a ECB context.

#### Parameters:

***cipher*** The index of the cipher desired

***key*** The secret key

***keylen*** The length of the secret key (octets)

***num\_rounds*** Number of rounds in the cipher desired (0 for default)

***ecb*** The ECB state to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 30 of file ecb\_start.c.

References [ltc\\_cipher\\_descriptor::block\\_length](#), [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), and [ltc\\_cipher\\_descriptor::setup](#).

```
31 {
32     int err;
33     LTC_ARGCHK(key != NULL);
34     LTC_ARGCHK(ecb != NULL);
35
36     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
37         return err;
38     }
39     ecb->cipher = cipher;
40     ecb->blocklen = cipher_descriptor[cipher].block_length;
41     return cipher_descriptor[cipher].setup(key, keylen, num_rounds, &ecb->key);
42 }
```

Here is the call graph for this function:

## 5.185 modes/f8/f8\_decrypt.c File Reference

### 5.185.1 Detailed Description

F8 implementation, decrypt data, Tom St Denis.

Definition in file [f8\\_decrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for f8\_decrypt.c:

### Functions

- [int f8\\_decrypt](#) (const unsigned char \*ct, unsigned char \*pt, unsigned long len, symmetric\_F8 \*f8)  
*F8 decrypt.*

### 5.185.2 Function Documentation

#### 5.185.2.1 int f8\_decrypt (const unsigned char \* ct, unsigned char \* pt, unsigned long len, symmetric\_F8 \* f8)

F8 decrypt.

##### Parameters:

*ct* Ciphertext  
*pt* [out] Plaintext  
*len* Length of ciphertext (octets)  
*f8* F8 state

##### Returns:

CRYPT\_OK if successful

Definition at line 28 of file f8\_decrypt.c.

References [f8\\_encrypt\(\)](#), and [LTC\\_ARGCHK](#).

```
29 {  
30     LTC_ARGCHK(pt != NULL);  
31     LTC_ARGCHK(ct != NULL);  
32     LTC_ARGCHK(f8 != NULL);  
33     return f8_encrypt(ct, pt, len, f8);  
34 }
```

Here is the call graph for this function:

## 5.186 modes/f8/f8\_done.c File Reference

### 5.186.1 Detailed Description

F8 implementation, finish chain, Tom St Denis.

Definition in file [f8\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for f8\_done.c:

### Functions

- [int f8\\_done](#) (symmetric\_F8 \*f8)  
*Terminate the chain.*

### 5.186.2 Function Documentation

#### 5.186.2.1 int f8\_done (symmetric\_F8 \*f8)

Terminate the chain.

#### Parameters:

*f8* The F8 chain to terminate

#### Returns:

CRYPT\_OK on success

Definition at line 24 of file f8\_done.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::done](#), and [LTC\\_ARGCHK](#).

Referenced by [f8\\_test\\_mode\(\)](#).

```
25 {  
26     int err;  
27     LTC_ARGCHK(f8 != NULL);  
28  
29     if ((err = cipher_is_valid(f8->cipher)) != CRYPT_OK) {  
30         return err;  
31     }  
32     cipher_descriptor[f8->cipher].done(&f8->key);  
33     return CRYPT_OK;  
34 }
```

Here is the call graph for this function:



## 5.187 modes/f8/f8\_encrypt.c File Reference

### 5.187.1 Detailed Description

F8 implementation, encrypt data, Tom St Denis.

Definition in file [f8\\_encrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for f8\_encrypt.c:

### Functions

- [int f8\\_encrypt](#) (const unsigned char \*pt, unsigned char \*ct, unsigned long [len](#), symmetric\_F8 \*f8)  
*F8 encrypt.*

### 5.187.2 Function Documentation

#### 5.187.2.1 int f8\_encrypt (const unsigned char \*pt, unsigned char \*ct, unsigned long len, symmetric\_F8 \*f8)

F8 encrypt.

#### Parameters:

*pt* Plaintext  
*ct* [out] Ciphertext  
*len* Length of plaintext (octets)  
*f8* F8 state

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file f8\_encrypt.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), [LTC\\_ARGCHK](#), [MAXBLOCKSIZE](#), and [zeromem\(\)](#).

Referenced by [f8\\_decrypt\(\)](#), and [f8\\_test\\_mode\(\)](#).

```
29 {
30     int            err, x;
31     unsigned char buf[MAXBLOCKSIZE];
32     LTC_ARGCHK(pt != NULL);
33     LTC_ARGCHK(ct != NULL);
34     LTC_ARGCHK(f8 != NULL);
35     if ((err = cipher_is_valid(f8->cipher)) != CRYPT_OK) {
36         return err;
37     }
38
39     /* is blocklen/padlen valid? */
40     if (f8->blocklen < 0 || f8->blocklen > (int)sizeof(f8->IV) ||
41         f8->padlen < 0 || f8->padlen > (int)sizeof(f8->IV)) {
42         return CRYPT_INVALID_ARG;
43     }
44 }
```

```

43     }
44
45     zeromem(buf, sizeof(buf));
46
47     /* make sure the pad is empty */
48     if (f8->padlen == f8->blocklen) {
49         /* xor of IV, MIV and blockcnt == what goes into cipher */
50         STORE32H(f8->blockcnt, (buf+(f8->blocklen-4)));
51         ++(f8->blockcnt);
52         for (x = 0; x < f8->blocklen; x++) {
53             f8->IV[x] ^= f8->MIV[x] ^ buf[x];
54         }
55         if ((err = cipher_descriptor[f8->cipher].ecb_encrypt(f8->IV, f8->IV, &f8->key)) != CRYPT_OK) {
56             return err;
57         }
58         f8->padlen = 0;
59     }
60
61 #ifdef LTC_FAST
62     if (f8->padlen == 0) {
63         while (len >= (unsigned long)f8->blocklen) {
64             STORE32H(f8->blockcnt, (buf+(f8->blocklen-4)));
65             ++(f8->blockcnt);
66             for (x = 0; x < f8->blocklen; x += sizeof(LTC_FAST_TYPE)) {
67                 *((LTC_FAST_TYPE*)&ct[x]) = *((LTC_FAST_TYPE*)&pt[x]) ^ *((LTC_FAST_TYPE*)&f8->IV[x])
68                 *((LTC_FAST_TYPE*)&f8->IV[x]) ^= *((LTC_FAST_TYPE*)&f8->MIV[x]) ^ *((LTC_FAST_TYPE*)&f8->IV[x]);
69             }
70             if ((err = cipher_descriptor[f8->cipher].ecb_encrypt(f8->IV, f8->IV, &f8->key)) != CRYPT_OK) {
71                 return err;
72             }
73             len -= x;
74             pt += x;
75             ct += x;
76         }
77     }
78 #endif
79
80     while (len > 0) {
81         if (f8->padlen == f8->blocklen) {
82             /* xor of IV, MIV and blockcnt == what goes into cipher */
83             STORE32H(f8->blockcnt, (buf+(f8->blocklen-4)));
84             ++(f8->blockcnt);
85             for (x = 0; x < f8->blocklen; x++) {
86                 f8->IV[x] ^= f8->MIV[x] ^ buf[x];
87             }
88             if ((err = cipher_descriptor[f8->cipher].ecb_encrypt(f8->IV, f8->IV, &f8->key)) != CRYPT_OK)
89                 return err;
90         }
91         f8->padlen = 0;
92     }
93     *ct++ = *pt++ ^ f8->IV[f8->padlen++];
94     --len;
95 }
96 return CRYPT_OK;
97 }

```

Here is the call graph for this function:

## 5.188 modes/f8/f8\_getiv.c File Reference

```
#include "tomcrypt.h"
```

Include dependency graph for f8\_getiv.c:

### Functions

- `int f8_getiv` (unsigned char \**IV*, unsigned long \**len*, symmetric\_F8 \**f8*)

*Get the current initial vector.*

### 5.188.1 Function Documentation

#### 5.188.1.1 `int f8_getiv` (unsigned char \* *IV*, unsigned long \* *len*, symmetric\_F8 \* *f8*)

Get the current initial vector.

##### Parameters:

*IV* [out] The destination of the initial vector

*len* [in/out] The max size and resulting size of the initial vector

*f8* The F8 state

##### Returns:

CRYPT\_OK if successful

Definition at line 27 of file f8\_getiv.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_OK, LTC\_ARGCHK, and XMEMCPY.

```
28 {
29     LTC_ARGCHK(IV != NULL);
30     LTC_ARGCHK(len != NULL);
31     LTC_ARGCHK(f8 != NULL);
32     if ((unsigned long)f8->blocklen > *len) {
33         *len = f8->blocklen;
34         return CRYPT_BUFFER_OVERFLOW;
35     }
36     XMEMCPY(IV, f8->IV, f8->blocklen);
37     *len = f8->blocklen;
38
39     return CRYPT_OK;
40 }
```

## 5.189 modes/f8/f8\_setiv.c File Reference

### 5.189.1 Detailed Description

F8 implementation, set IV, Tom St Denis.

Definition in file [f8\\_setiv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for f8\_setiv.c:

### Functions

- [int f8\\_setiv](#) (const unsigned char \*IV, unsigned long [len](#), symmetric\_F8 \*f8)  
*Set an initial vector.*

### 5.189.2 Function Documentation

#### 5.189.2.1 int f8\_setiv (const unsigned char \* IV, unsigned long len, symmetric\_F8 \*f8)

Set an initial vector.

##### Parameters:

- IV* The initial vector
- len* The length of the vector (in octets)
- f8* The F8 state

##### Returns:

CRYPT\_OK if successful

Definition at line 27 of file f8\_setiv.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), and [LTC\\_ARGCHK](#).

```
28 {
29     int err;
30
31     LTC_ARGCHK(IV != NULL);
32     LTC_ARGCHK(f8 != NULL);
33
34     if ((err = cipher_is_valid(f8->cipher)) != CRYPT_OK) {
35         return err;
36     }
37
38     if (len != (unsigned long)f8->blocklen) {
39         return CRYPT_INVALID_ARG;
40     }
41
42     /* force next block */
43     f8->padlen = 0;
44     return cipher_descriptor[f8->cipher].ecb_encrypt(IV, f8->IV, &f8->key);
45 }
```

Here is the call graph for this function:

## 5.190 modes/f8/f8\_start.c File Reference

### 5.190.1 Detailed Description

F8 implementation, start chain, Tom St Denis.

Definition in file [f8\\_start.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for f8\_start.c:

### Functions

- [int f8\\_start](#) (int cipher, const unsigned char \*IV, const unsigned char \*key, int keylen, const unsigned char \*salt\_key, int skeylen, int num\_rounds, symmetric\_F8 \*f8)

*Initialize an F8 context.*

### 5.190.2 Function Documentation

**5.190.2.1** [int f8\\_start](#) (int *cipher*, const unsigned char \* *IV*, const unsigned char \* *key*, int *keylen*, const unsigned char \* *salt\_key*, int *skeylen*, int *num\_rounds*, symmetric\_F8 \* *f8*)

Initialize an F8 context.

#### Parameters:

*cipher* The index of the cipher desired

*IV* The initial vector

*key* The secret key

*keylen* The length of the secret key (octets)

*salt\_key* The salting key for the IV

*skeylen* The length of the salting key (octets)

*num\_rounds* Number of rounds in the cipher desired (0 for default)

*f8* The F8 state to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 33 of file f8\_start.c.

References [ltc\\_cipher\\_descriptor::block\\_length](#), [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), [MAXBLOCKSIZE](#), and [zeromem\(\)](#).

Referenced by [f8\\_test\\_mode\(\)](#).

```
37 {
38     int                x, err;
39     unsigned char tkey[MAXBLOCKSIZE];
40
41     LTC_ARGCHK(IV      != NULL);
42     LTC_ARGCHK(key     != NULL);
43     LTC_ARGCHK(salt_key != NULL);
```

```

44     LTC_ARGCHK(f8      != NULL);
45
46     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
47         return err;
48     }
49
50 #ifdef LTC_FAST
51     if (cipher_descriptor[cipher].block_length % sizeof(LTC_FAST_TYPE)) {
52         return CRYPT_INVALID_ARG;
53     }
54 #endif
55
56     /* copy details */
57     f8->blockcnt = 0;
58     f8->cipher    = cipher;
59     f8->blocklen  = cipher_descriptor[cipher].block_length;
60     f8->padlen    = f8->blocklen;
61
62     /* now get key ^ salt_key [extend salt_ket with 0x55 as required to match length] */
63     zeromem(tkey, sizeof(tkey));
64     for (x = 0; x < keylen && x < (int)sizeof(tkey); x++) {
65         tkey[x] = key[x];
66     }
67     for (x = 0; x < skeylen && x < (int)sizeof(tkey); x++) {
68         tkey[x] ^= salt_key[x];
69     }
70     for (; x < keylen && x < (int)sizeof(tkey); x++) {
71         tkey[x] ^= 0x55;
72     }
73
74     /* now encrypt with tkey[0..keylen-1] the IV and use that as the IV */
75     if ((err = cipher_descriptor[cipher].setup(tkey, keylen, num_rounds, &f8->key)) != CRYPT_OK) {
76         return err;
77     }
78
79     /* encrypt IV */
80     if ((err = cipher_descriptor[f8->cipher].ecb_encrypt(IV, f8->MIV, &f8->key)) != CRYPT_OK) {
81         cipher_descriptor[f8->cipher].done(&f8->key);
82         return err;
83     }
84     zeromem(tkey, sizeof(tkey));
85     zeromem(f8->IV, sizeof(f8->IV));
86
87     /* terminate this cipher */
88     cipher_descriptor[f8->cipher].done(&f8->key);
89
90     /* init the cipher */
91     return cipher_descriptor[cipher].setup(key, keylen, num_rounds, &f8->key);
92 }

```

Here is the call graph for this function:

## 5.191 modes/f8/f8\_test\_mode.c File Reference

### 5.191.1 Detailed Description

F8 implementation, test, Tom St Denis.

Definition in file [f8\\_test\\_mode.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for f8\_test\_mode.c:

### Functions

- [int f8\\_test\\_mode](#) (void)

### 5.191.2 Function Documentation

#### 5.191.2.1 int f8\_test\_mode (void)

Definition at line 21 of file f8\_test\_mode.c.

References [CRYPT\\_FAIL\\_TESTVECTOR](#), [CRYPT\\_NOP](#), [CRYPT\\_OK](#), [f8\\_done\(\)](#), [f8\\_encrypt\(\)](#), [f8\\_start\(\)](#), [find\\_cipher\(\)](#), and [XMEMCMP](#).

```
22 {
23 #ifndef LTC_TEST
24     return CRYPT_NOP;
25 #else
26     static const unsigned char key[16] = { 0x23, 0x48, 0x29, 0x00, 0x84, 0x67, 0xbe, 0x18,
27                                             0x6c, 0x3d, 0xe1, 0x4a, 0xae, 0x72, 0xd6, 0x2c };
28     static const unsigned char salt[4] = { 0x32, 0xf2, 0x87, 0x0d };
29     static const unsigned char IV[16] = { 0x00, 0x6e, 0x5c, 0xba, 0x50, 0x68, 0x1d, 0xe5,
30                                             0x5c, 0x62, 0x15, 0x99, 0xd4, 0x62, 0x56, 0x4a };
31     static const unsigned char pt[39] = { 0x70, 0x73, 0x65, 0x75, 0x64, 0x6f, 0x72, 0x61,
32                                             0x6e, 0x64, 0x6f, 0x6d, 0x6e, 0x65, 0x73, 0x73,
33                                             0x20, 0x69, 0x73, 0x20, 0x74, 0x68, 0x65, 0x20,
34                                             0x6e, 0x65, 0x78, 0x74, 0x20, 0x62, 0x65, 0x73,
35                                             0x74, 0x20, 0x74, 0x68, 0x69, 0x6e, 0x67 };
36     static const unsigned char ct[39] = { 0x01, 0x9c, 0xe7, 0xa2, 0x6e, 0x78, 0x54, 0x01,
37                                             0x4a, 0x63, 0x66, 0xaa, 0x95, 0xd4, 0xee, 0xfd,
38                                             0x1a, 0xd4, 0x17, 0x2a, 0x14, 0xf9, 0xfa, 0xf4,
39                                             0x55, 0xb7, 0xf1, 0xd4, 0xb6, 0x2b, 0xd0, 0x8f,
40                                             0x56, 0x2c, 0x0e, 0xef, 0x7c, 0x48, 0x02 };
41     unsigned char buf[39];
42     symmetric_F8 f8;
43     int err, idx;
44
45     idx = find_cipher("aes");
46     if (idx == -1) {
47         idx = find_cipher("rijndael");
48         if (idx == -1) return CRYPT_NOP;
49     }
50
51     /* initialize the context */
52     if ((err = f8_start(idx, IV, key, sizeof(key), salt, sizeof(salt), 0, &f8)) != CRYPT_OK) {
53         return err;
54     }
55
56     /* encrypt block */
57     if ((err = f8_encrypt(pt, buf, sizeof(pt), &f8)) != CRYPT_OK) {
```

```
58     f8_done(&f8);
59     return err;
60 }
61 f8_done(&f8);
62
63 /* compare */
64 if (XMEMCMP(buf, ct, sizeof(ct))) {
65     return CRYPT_FAIL_TESTVECTOR;
66 }
67
68 return CRYPT_OK;
69 #endif
70 }
```

Here is the call graph for this function:



## 5.192 modes/lrw/lrw\_decrypt.c File Reference

### 5.192.1 Detailed Description

LRW\_MODE implementation, Decrypt blocks, Tom St Denis.

Definition in file [lrw\\_decrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `lrw_decrypt.c`:

### Functions

- `int lrw_decrypt` (const unsigned char \*ct, unsigned char \*pt, unsigned long len, symmetric\_LRW \*lrw)  
*LRW decrypt blocks.*

### 5.192.2 Function Documentation

#### 5.192.2.1 `int lrw_decrypt` (const unsigned char \* ct, unsigned char \* pt, unsigned long len, symmetric\_LRW \* lrw)

LRW decrypt blocks.

#### Parameters:

- ct* The ciphertext
- pt* [out] The plaintext
- len* The length in octets, must be a multiple of 16
- lrw* The LRW state

Definition at line 27 of file `lrw_decrypt.c`.

References `ltc_cipher_descriptor::accel_lrw_decrypt`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_OK`, `lrw_process()`, and `LTC_ARGCHK`.

```
28 {
29     int err;
30
31     LTC_ARGCHK(pt != NULL);
32     LTC_ARGCHK(ct != NULL);
33     LTC_ARGCHK(lrw != NULL);
34
35     if ((err = cipher_is_valid(lrw->cipher)) != CRYPT_OK) {
36         return err;
37     }
38
39     if (cipher_descriptor[lrw->cipher].accel_lrw_decrypt != NULL) {
40         return cipher_descriptor[lrw->cipher].accel_lrw_decrypt(ct, pt, len, lrw->IV, lrw->tweak, &lrw->state);
41     }
42
43     return lrw_process(ct, pt, len, LRW_DECRYPT, lrw);
44 }
```

Here is the call graph for this function:

## 5.193 modes/lrw/lrw\_done.c File Reference

### 5.193.1 Detailed Description

LRW\_MODE implementation, Free resources, Tom St Denis.

Definition in file [lrw\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for lrw\_done.c:

### Functions

- int [lrw\\_done](#) (symmetric\_LRW \*lrw)  
*Terminate a LRW state.*

### 5.193.2 Function Documentation

#### 5.193.2.1 int lrw\_done (symmetric\_LRW \* lrw)

Terminate a LRW state.

#### Parameters:

*lrw* The state to terminate

#### Returns:

CRYPT\_OK if successful

Definition at line 25 of file lrw\_done.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::done](#), and [LTC\\_ARGCHK](#).

```
26 {  
27     int err;  
28  
29     LTC_ARGCHK(lrw != NULL);  
30  
31     if ((err = cipher_is_valid(lrw->cipher)) != CRYPT_OK) {  
32         return err;  
33     }  
34     cipher_descriptor[lrw->cipher].done(&lrw->key);  
35  
36     return CRYPT_OK;  
37 }
```

Here is the call graph for this function:

## 5.194 modes/lrw/lrw\_encrypt.c File Reference

### 5.194.1 Detailed Description

LRW\_MODE implementation, Encrypt blocks, Tom St Denis.

Definition in file [lrw\\_encrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `lrw_encrypt.c`:

### Functions

- `int lrw_encrypt` (`const unsigned char *pt`, `unsigned char *ct`, `unsigned long len`, `symmetric_LRW *lrw`)  
*LRW encrypt blocks.*

### 5.194.2 Function Documentation

#### 5.194.2.1 `int lrw_encrypt` (`const unsigned char *pt`, `unsigned char *ct`, `unsigned long len`, `symmetric_LRW *lrw`)

LRW encrypt blocks.

#### Parameters:

- pt* The plaintext
- ct* [out] The ciphertext
- len* The length in octets, must be a multiple of 16
- lrw* The LRW state

Definition at line 27 of file `lrw_encrypt.c`.

References `ltc_cipher_descriptor::accel_lrw_encrypt`, `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_OK`, `lrw_process()`, and `LTC_ARGCHK`.

```
28 {
29     int err;
30
31     LTC_ARGCHK(pt != NULL);
32     LTC_ARGCHK(ct != NULL);
33     LTC_ARGCHK(lrw != NULL);
34
35     if ((err = cipher_is_valid(lrw->cipher)) != CRYPT_OK) {
36         return err;
37     }
38
39     if (cipher_descriptor[lrw->cipher].accel_lrw_encrypt != NULL) {
40         return cipher_descriptor[lrw->cipher].accel_lrw_encrypt(pt, ct, len, lrw->IV, lrw->tweak, &lrw->state);
41     }
42
43     return lrw_process(pt, ct, len, LRW_ENCRYPT, lrw);
44 }
```

Here is the call graph for this function:

## 5.195 modes/lrw/lrw\_getiv.c File Reference

### 5.195.1 Detailed Description

LRW\_MODE implementation, Retrieve the current IV, Tom St Denis.

Definition in file [lrw\\_getiv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for lrw\_getiv.c:

### Functions

- [int lrw\\_getiv](#) (unsigned char \*IV, unsigned long \*len, symmetric\_LRW \*lrw)  
*Get the IV for LRW.*

### 5.195.2 Function Documentation

#### 5.195.2.1 int lrw\_getiv (unsigned char \*IV, unsigned long \*len, symmetric\_LRW \*lrw)

Get the IV for LRW.

#### Parameters:

- IV** [out] The IV, must be 16 octets  
**len** Length ... must be at least 16 :-)  
**lrw** The LRW state to read

#### Returns:

CRYPT\_OK if successful

Definition at line 27 of file lrw\_getiv.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_OK, LTC\_ARGCHK, and XMEMCPY.

```
28 {
29     LTC_ARGCHK(IV != NULL);
30     LTC_ARGCHK(len != NULL);
31     LTC_ARGCHK(lrw != NULL);
32     if (*len < 16) {
33         *len = 16;
34         return CRYPT_BUFFER_OVERFLOW;
35     }
36
37     XMEMCPY(IV, lrw->IV, 16);
38     *len = 16;
39     return CRYPT_OK;
40 }
```

## 5.196 modes/lrw/lrw\_process.c File Reference

### 5.196.1 Detailed Description

LRW\_MODE implementation, Encrypt/decrypt blocks, Tom St Denis.

Definition in file [lrw\\_process.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for lrw\_process.c:

### Functions

- int [lrw\\_process](#) (const unsigned char \*pt, unsigned char \*ct, unsigned long len, int mode, symmetric\_LRW \*lrw)

*Process blocks with LRW, since decrypt/encrypt are largely the same they share this code.*

### 5.196.2 Function Documentation

#### 5.196.2.1 int lrw\_process (const unsigned char \*pt, unsigned char \*ct, unsigned long len, int mode, symmetric\_LRW \*lrw)

Process blocks with LRW, since decrypt/encrypt are largely the same they share this code.

#### Parameters:

*pt* The "input" data

*ct* [out] The "output" data

*len* The length of the input, must be a multiple of 128-bits (16 octets)

*mode* LRW\_ENCRYPT or LRW\_DECRYPT

*lrw* The LRW state

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file lrw\_process.c.

References CRYPT\_INVALID\_ARG, LTC\_ARGCHK, and XMEMCPY.

Referenced by lrw\_decrypt(), and lrw\_encrypt().

```
30 {
31     unsigned char prod[16];
32     int          x, err;
33 #ifdef LRW_TABLES
34     int          y;
35 #endif
36
37     LTC_ARGCHK(pt != NULL);
38     LTC_ARGCHK(ct != NULL);
39     LTC_ARGCHK(lrw != NULL);
40
41     if (len & 15) {
42         return CRYPT_INVALID_ARG;
```

```

43     }
44
45     while (len) {
46         /* copy pad */
47         XMEMCPY(prod, lrw->pad, 16);
48
49         /* increment IV */
50         for (x = 15; x >= 0; x--) {
51             lrw->IV[x] = (lrw->IV[x] + 1) & 255;
52             if (lrw->IV[x]) {
53                 break;
54             }
55         }
56
57         /* update pad */
58 #ifdef LRW_TABLES
59         /* for each byte changed we undo it's affect on the pad then add the new product */
60         for (; x < 16; x++) {
61 #ifdef LTC_FAST
62             for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
63                 *((LTC_FAST_TYPE *) (lrw->pad + y)) ^= *((LTC_FAST_TYPE *) (&lrw->PC[x][lrw->IV[x]][y])) ^
64             }
65 #else
66             for (y = 0; y < 16; y++) {
67                 lrw->pad[y] ^= lrw->PC[x][lrw->IV[x]][y] ^ lrw->PC[x][(lrw->IV[x]-1)&255][y];
68             }
69 #endif
70         }
71 #else
72         gcm_gf_mult(lrw->tweak, lrw->IV, lrw->pad);
73 #endif
74
75         /* xor prod */
76 #ifdef LTC_FAST
77         for (x = 0; x < 16; x += sizeof(LTC_FAST_TYPE)) {
78             *((LTC_FAST_TYPE *) (ct + x)) = *((LTC_FAST_TYPE *) (pt + x)) ^ *((LTC_FAST_TYPE *) (prod + x))
79         }
80 #else
81         for (x = 0; x < 16; x++) {
82             ct[x] = pt[x] ^ prod[x];
83         }
84 #endif
85
86         /* send through cipher */
87         if (mode == LRW_ENCRYPT) {
88             if ((err = cipher_descriptor[lrw->cipher].ecb_encrypt(ct, ct, &lrw->key)) != CRYPT_OK) {
89                 return err;
90             }
91         } else {
92             if ((err = cipher_descriptor[lrw->cipher].ecb_decrypt(ct, ct, &lrw->key)) != CRYPT_OK) {
93                 return err;
94             }
95         }
96
97         /* xor prod */
98 #ifdef LTC_FAST
99         for (x = 0; x < 16; x += sizeof(LTC_FAST_TYPE)) {
100             *((LTC_FAST_TYPE *) (ct + x)) = *((LTC_FAST_TYPE *) (ct + x)) ^ *((LTC_FAST_TYPE *) (prod + x))
101         }
102 #else
103         for (x = 0; x < 16; x++) {
104             ct[x] = ct[x] ^ prod[x];
105         }
106 #endif
107
108         /* move to next */
109         pt += 16;

```

---

```
110         ct  += 16;
111         len -= 16;
112     }
113
114     return CRYPT_OK;
115 }
```

## 5.197 modes/lrw/lrw\_setiv.c File Reference

### 5.197.1 Detailed Description

LRW\_MODE implementation, Set the current IV, Tom St Denis.

Definition in file [lrw\\_setiv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for lrw\_setiv.c:

### Functions

- int [lrw\\_setiv](#) (const unsigned char \*IV, unsigned long [len](#), symmetric\_LRW \*lrw)

*Set the IV for LRW.*

### 5.197.2 Function Documentation

#### 5.197.2.1 int lrw\_setiv (const unsigned char \*IV, unsigned long len, symmetric\_LRW \*lrw)

Set the IV for LRW.

#### Parameters:

**IV** The IV, must be 16 octets

**len** Length ... must be 16 :-)

**lrw** The LRW state to update

#### Returns:

CRYPT\_OK if successful

Definition at line 27 of file lrw\_setiv.c.

References [ltc\\_cipher\\_descriptor::accel\\_lrw\\_decrypt](#), [ltc\\_cipher\\_descriptor::accel\\_lrw\\_encrypt](#), [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), and [XMEMCPY](#).

```
28 {
29     int          err;
30 #ifdef LRW_TABLES
31     unsigned char T[16];
32     int          x, y;
33 #endif
34     LTC_ARGCHK(IV != NULL);
35     LTC_ARGCHK(lrw != NULL);
36
37     if (len != 16) {
38         return CRYPT_INVALID_ARG;
39     }
40
41     if ((err = cipher_is_valid(lrw->cipher)) != CRYPT_OK) {
42         return err;
43     }
44
45     /* copy the IV */
46     XMEMCPY(lrw->IV, IV, 16);
```



```
47
48     /* check if we have to actually do work */
49     if (cipher_descriptor[lrw->cipher].accel_lrw_encrypt != NULL && cipher_descriptor[lrw->cipher].accel
50         /* we have accelerators, let's bail since they don't use lrw->pad anyways */
51         return CRYPT_OK;
52     }
53
54 #ifdef LRW_TABLES
55     XMEMCPY(T, &lrw->PC[0][IV[0]][0], 16);
56     for (x = 1; x < 16; x++) {
57 #ifdef LTC_FAST
58         for (y = 0; y < 16; y += sizeof(LTC_FAST_TYPE)) {
59             *((LTC_FAST_TYPE *) (T + y)) ^= *((LTC_FAST_TYPE *) (&lrw->PC[x][IV[x]][y]));
60         }
61 #else
62         for (y = 0; y < 16; y++) {
63             T[y] ^= lrw->PC[x][IV[x]][y];
64         }
65 #endif
66     }
67     XMEMCPY(lrw->pad, T, 16);
68 #else
69     gcm_gf_mult(lrw->tweak, IV, lrw->pad);
70 #endif
71
72     return CRYPT_OK;
73 }
```

Here is the call graph for this function:

## 5.198 modes/lrw/lrw\_start.c File Reference

### 5.198.1 Detailed Description

LRW\_MODE implementation, start mode, Tom St Denis.

Definition in file [lrw\\_start.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for lrw\_start.c:

### Functions

- [int lrw\\_start](#) (int cipher, const unsigned char \*IV, const unsigned char \*key, int keylen, const unsigned char \*tweak, int num\_rounds, symmetric\_LRW \*lrw)

*Initialize the LRW context.*

### 5.198.2 Function Documentation

#### 5.198.2.1 int lrw\_start (int cipher, const unsigned char \* IV, const unsigned char \* key, int keylen, const unsigned char \* tweak, int num\_rounds, symmetric\_LRW \* lrw)

Initialize the LRW context.

#### Parameters:

**cipher** The cipher desired, must be a 128-bit block cipher

**IV** The index value, must be 128-bits

**key** The cipher key

**keylen** The length of the cipher key in octets

**tweak** The tweak value (second key), must be 128-bits

**num\_rounds** The number of rounds for the cipher (0 == default)

**lrw** [out] The LRW state

#### Returns:

CRYPT\_OK on success.

Definition at line 31 of file lrw\_start.c.

References [B](#), [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_INVALID\\_CIPHER](#), [CRYPT\\_OK](#), [gcm\\_gf\\_mult\(\)](#), [LTC\\_ARGCHK](#), [XMEMCPY](#), and [zeromem\(\)](#).

Referenced by [lrw\\_test\(\)](#).

```
37 {
38     int          err;
39 #ifdef LRW_TABLES
40     unsigned char B[16];
41     int          x, y, z, t;
42 #endif
43
44     LTC_ARGCHK(IV != NULL);
```

```

45 LTC_ARGCHK(key    != NULL);
46 LTC_ARGCHK(tweak  != NULL);
47 LTC_ARGCHK(lrw    != NULL);
48
49 #ifdef LTC_FAST
50     if (16 % sizeof(LTC_FAST_TYPE)) {
51         return CRYPT_INVALID_ARG;
52     }
53 #endif
54
55     /* is cipher valid? */
56     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
57         return err;
58     }
59     if (cipher_descriptor[cipher].block_length != 16) {
60         return CRYPT_INVALID_CIPHER;
61     }
62
63     /* schedule key */
64     if ((err = cipher_descriptor[cipher].setup(key, keylen, num_rounds, &lrw->key)) != CRYPT_OK) {
65         return err;
66     }
67     lrw->cipher = cipher;
68
69     /* copy the IV and tweak */
70     XMEMCPY(lrw->tweak, tweak, 16);
71
72 #ifdef LRW_TABLES
73     /* setup tables */
74     /* generate the first table as it has no shifting (from which we make the other tables) */
75     zeromem(B, 16);
76     for (y = 0; y < 256; y++) {
77         B[0] = y;
78         gcm_gf_mult(tweak, B, &lrw->PC[0][y][0]);
79     }
80
81     /* now generate the rest of the tables based the previous table */
82     for (x = 1; x < 16; x++) {
83         for (y = 0; y < 256; y++) {
84             /* now shift it right by 8 bits */
85             t = lrw->PC[x-1][y][15];
86             for (z = 15; z > 0; z--) {
87                 lrw->PC[x][y][z] = lrw->PC[x-1][y][z-1];
88             }
89             lrw->PC[x][y][0] = gcm_shift_table[t<<1];
90             lrw->PC[x][y][1] ^= gcm_shift_table[(t<<1)+1];
91         }
92     }
93 #endif
94
95     /* generate first pad */
96     return lrw_setiv(IV, 16, lrw);
97 }

```

Here is the call graph for this function:

## 5.199 modes/lrw/lrw\_test.c File Reference

### 5.199.1 Detailed Description

LRW\_MODE implementation, test LRW, Tom St Denis.

Definition in file [lrw\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for lrw\_test.c:

### Functions

- [int lrw\\_test](#) (void)  
*Test LRW against specs.*

### 5.199.2 Function Documentation

#### 5.199.2.1 int lrw\_test (void)

Test LRW against specs.

#### Returns:

CRYPT\_OK if goodly

Definition at line 24 of file lrw\_test.c.

References CRYPT\_NOP, CRYPT\_OK, find\_cipher(), and lrw\_start().

```
25 {
26 #ifndef LTC_TEST
27     return CRYPT_NOP;
28 #else
29     static const struct {
30         unsigned char key[16], tweak[16], IV[16], P[16], expected_tweak[16], C[16];
31     } tests[] = {
32
33 {
34 { 0x45, 0x62, 0xac, 0x25, 0xf8, 0x28, 0x17, 0x6d, 0x4c, 0x26, 0x84, 0x14, 0xb5, 0x68, 0x01, 0x85 },
35 { 0x25, 0x8e, 0x2a, 0x05, 0xe7, 0x3e, 0x9d, 0x03, 0xee, 0x5a, 0x83, 0x0c, 0xcc, 0x09, 0x4c, 0x87 },
36 { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
37 { 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46 },
38 { 0x25, 0x8e, 0x2a, 0x05, 0xe7, 0x3e, 0x9d, 0x03, 0xee, 0x5a, 0x83, 0x0c, 0xcc, 0x09, 0x4c, 0x87 },
39 { 0xf1, 0xb2, 0x73, 0xcd, 0x65, 0xa3, 0xdf, 0x5f, 0xe9, 0x5d, 0x48, 0x92, 0x54, 0x63, 0x4e, 0xb8 },
40 },
41
42 {
43 { 0x59, 0x70, 0x47, 0x14, 0xf5, 0x57, 0x47, 0x8c, 0xd7, 0x79, 0xe8, 0x0f, 0x54, 0x88, 0x79, 0x44 },
44 { 0x35, 0x23, 0xc2, 0xde, 0xc5, 0x69, 0x4f, 0xa8, 0x72, 0xa9, 0xac, 0xa7, 0x0b, 0x2b, 0xee, 0xbc },
45 { 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
46 { 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46 },
47 { 0x1a, 0x91, 0xe1, 0x6f, 0x62, 0xb4, 0xa7, 0xd4, 0x39, 0x54, 0xd6, 0x53, 0x85, 0x95, 0xf7, 0x5e },
48 { 0x00, 0xc8, 0x2b, 0xae, 0x95, 0xbb, 0xcd, 0xe5, 0x27, 0x4f, 0x07, 0x69, 0xb2, 0x60, 0xe1, 0x36 },
49 },
50
51 {
52 { 0x59, 0x70, 0x47, 0x14, 0xf5, 0x57, 0x47, 0x8c, 0xd7, 0x79, 0xe8, 0x0f, 0x54, 0x88, 0x79, 0x44 },
```

```

53 { 0x67, 0x53, 0xc9, 0x0c, 0xb7, 0xd8, 0xcd, 0xe5, 0x06, 0xa0, 0x47, 0x78, 0x1a, 0xad, 0x85, 0x11 },
54 { 0x00, 0x00, 0x00, 0x00, 0x00, 0xd8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02 },
55 { 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46 },
56 { 0x1a, 0x91, 0xe1, 0x6f, 0x62, 0xb4, 0xa7, 0xd4, 0x39, 0x54, 0xd6, 0x53, 0x85, 0x95, 0xf7, 0x5e },
57 { 0x00, 0xc8, 0x2b, 0xae, 0x95, 0xbb, 0xcd, 0xe5, 0x27, 0x4f, 0x07, 0x69, 0xb2, 0x60, 0xe1, 0x36 },
58 },
59
60 {
61
62 { 0xd8, 0x2a, 0x91, 0x34, 0xb2, 0x6a, 0x56, 0x50, 0x30, 0xfe, 0x69, 0xe2, 0x37, 0x7f, 0x98, 0x47 },
63 { 0x4e, 0xb5, 0x5d, 0x31, 0x05, 0x97, 0x3a, 0x3f, 0x5e, 0x23, 0xda, 0xfb, 0x5a, 0x45, 0xd6, 0xc0 },
64 { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00 },
65 { 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46 },
66 { 0x18, 0xc9, 0x1f, 0x6d, 0x60, 0x1a, 0x1a, 0x37, 0x5d, 0x0b, 0x0e, 0xf7, 0x3a, 0xd5, 0x74, 0xc4 },
67 { 0x76, 0x32, 0x21, 0x83, 0xed, 0x8f, 0xf1, 0x82, 0xf9, 0x59, 0x62, 0x03, 0x69, 0x0e, 0x5e, 0x01 },
68
69 }
70 };
71
72 int idx, err, x;
73 symmetric_LRW lrw;
74 unsigned char buf[2][16];
75
76 idx = find_cipher("aes");
77 if (idx == -1) {
78     idx = find_cipher("rijndael");
79     if (idx == -1) {
80         return CRYPT_NOP;
81     }
82 }
83
84 for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
85     /* schedule it */
86     if ((err = lrw_start(idx, tests[x].IV, tests[x].key, 16, tests[x].tweak, 0, &lrw)) != CRYPT_OK) {
87         return err;
88     }
89
90     /* check pad against expected tweak */
91     if (XMEMCMP(tests[x].expected_tweak, lrw.pad, 16)) {
92         lrw_done(&lrw);
93         return CRYPT_FAIL_TESTVECTOR;
94     }
95
96     /* process block */
97     if ((err = lrw_encrypt(tests[x].P, buf[0], 16, &lrw)) != CRYPT_OK) {
98         lrw_done(&lrw);
99         return err;
100     }
101
102     if (XMEMCMP(buf[0], tests[x].C, 16)) {
103         lrw_done(&lrw);
104         return CRYPT_FAIL_TESTVECTOR;
105     }
106
107     /* process block */
108     if ((err = lrw_setiv(tests[x].IV, 16, &lrw)) != CRYPT_OK) {
109         lrw_done(&lrw);
110         return err;
111     }
112
113     if ((err = lrw_decrypt(buf[0], buf[1], 16, &lrw)) != CRYPT_OK) {
114         lrw_done(&lrw);
115         return err;
116     }
117
118     if (XMEMCMP(buf[1], tests[x].P, 16)) {
119         lrw_done(&lrw);

```

```
120         return CRYPT_FAIL_TESTVECTOR;
121     }
122     if ((err = lrw_done(&lrw)) != CRYPT_OK) {
123         return err;
124     }
125 }
126 return CRYPT_OK;
127 #endif
128 }
```

Here is the call graph for this function:

## 5.200 modes/ofb/ofb\_decrypt.c File Reference

### 5.200.1 Detailed Description

OFB implementation, decrypt data, Tom St Denis.

Definition in file [ofb\\_decrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ofb\_decrypt.c:

### Functions

- `int ofb_decrypt` (const unsigned char \*ct, unsigned char \*pt, unsigned long len, symmetric\_OFB \*ofb)  
*OFB decrypt.*

### 5.200.2 Function Documentation

**5.200.2.1** `int ofb_decrypt` (const unsigned char \* *ct*, unsigned char \* *pt*, unsigned long *len*, symmetric\_OFB \* *ofb*)

OFB decrypt.

#### Parameters:

*ct* Ciphertext  
*pt* [out] Plaintext  
*len* Length of ciphertext (octets)  
*ofb* OFB state

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file ofb\_decrypt.c.

References LTC\_ARGCHK, and ofb\_encrypt().

```
29 {  
30     LTC_ARGCHK(pt != NULL);  
31     LTC_ARGCHK(ct != NULL);  
32     LTC_ARGCHK(ofb != NULL);  
33     return ofb_encrypt(ct, pt, len, ofb);  
34 }
```

Here is the call graph for this function:

## 5.201 modes/ofb/ofb\_done.c File Reference

### 5.201.1 Detailed Description

OFB implementation, finish chain, Tom St Denis.

Definition in file [ofb\\_done.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ofb\_done.c:

### Functions

- `int ofb_done (symmetric_OFB *ofb)`  
*Terminate the chain.*

### 5.201.2 Function Documentation

#### 5.201.2.1 `int ofb_done (symmetric_OFB * ofb)`

Terminate the chain.

#### Parameters:

*ofb* The OFB chain to terminate

#### Returns:

CRYPT\_OK on success

Definition at line 24 of file ofb\_done.c.

References `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_OK`, `ltc_cipher_descriptor::done`, and `LTC_ARGCHK`.

```
25 {
26     int err;
27     LTC_ARGCHK(ofb != NULL);
28
29     if ((err = cipher_is_valid(ofb->cipher)) != CRYPT_OK) {
30         return err;
31     }
32     cipher_descriptor[ofb->cipher].done(&ofb->key);
33     return CRYPT_OK;
34 }
```

Here is the call graph for this function:



## 5.202 modes/ofb/ofb\_encrypt.c File Reference

### 5.202.1 Detailed Description

OFB implementation, encrypt data, Tom St Denis.

Definition in file [ofb\\_encrypt.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ofb\_encrypt.c:

### Functions

- `int ofb_encrypt` (const unsigned char \*pt, unsigned char \*ct, unsigned long len, symmetric\_OFB \*ofb)  
*OFB encrypt.*

### 5.202.2 Function Documentation

#### 5.202.2.1 `int ofb_encrypt` (const unsigned char \*pt, unsigned char \*ct, unsigned long len, symmetric\_OFB \*ofb)

OFB encrypt.

#### Parameters:

*pt* Plaintext  
*ct* [out] Ciphertext  
*len* Length of plaintext (octets)  
*ofb* OFB state

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file ofb\_encrypt.c.

References `cipher_descriptor`, `cipher_is_valid()`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `ltc_cipher_descriptor::ecb_encrypt`, and `LTC_ARGCHK`.

Referenced by `ofb_decrypt()`.

```
29 {
30     int err;
31     LTC_ARGCHK(pt != NULL);
32     LTC_ARGCHK(ct != NULL);
33     LTC_ARGCHK(ofb != NULL);
34     if ((err = cipher_is_valid(ofb->cipher)) != CRYPT_OK) {
35         return err;
36     }
37
38     /* is blocklen/padlen valid? */
39     if (ofb->blocklen < 0 || ofb->blocklen > (int)sizeof(ofb->IV) ||
40         ofb->padlen < 0 || ofb->padlen > (int)sizeof(ofb->IV)) {
41         return CRYPT_INVALID_ARG;
42     }
43 }
```

```
42     }
43
44     while (len-- > 0) {
45         if (ofb->padlen == ofb->blocklen) {
46             if ((err = cipher_descriptor[ofb->cipher].ecb_encrypt(ofb->IV, ofb->IV, &ofb->key)) != CRYPT_
47                 return err;
48             }
49             ofb->padlen = 0;
50         }
51         *ct++ = *pt++ ^ ofb->IV[ofb->padlen++];
52     }
53     return CRYPT_OK;
54 }
```

Here is the call graph for this function:

## 5.203 modes/ofb/ofb\_getiv.c File Reference

### 5.203.1 Detailed Description

F8 implementation, get IV, Tom St Denis.

Definition in file [ofb\\_getiv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ofb\_getiv.c:

### Functions

- [int ofb\\_getiv](#) (unsigned char \*IV, unsigned long \*len, symmetric\_OFB \*ofb)

*Get the current initial vector.*

### 5.203.2 Function Documentation

#### 5.203.2.1 int ofb\_getiv (unsigned char \* IV, unsigned long \* len, symmetric\_OFB \* ofb)

Get the current initial vector.

##### Parameters:

*IV* [out] The destination of the initial vector

*len* [in/out] The max size and resulting size of the initial vector

*ofb* The OFB state

##### Returns:

CRYPT\_OK if successful

Definition at line 27 of file ofb\_getiv.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_OK, LTC\_ARGCHK, and XMEMCPY.

```
28 {
29     LTC_ARGCHK(IV != NULL);
30     LTC_ARGCHK(len != NULL);
31     LTC_ARGCHK(ofb != NULL);
32     if ((unsigned long)ofb->blocklen > *len) {
33         *len = ofb->blocklen;
34         return CRYPT_BUFFER_OVERFLOW;
35     }
36     XMEMCPY(IV, ofb->IV, ofb->blocklen);
37     *len = ofb->blocklen;
38
39     return CRYPT_OK;
40 }
```

## 5.204 modes/ofb/ofb\_setiv.c File Reference

### 5.204.1 Detailed Description

OFB implementation, set IV, Tom St Denis.

Definition in file [ofb\\_setiv.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ofb\_setiv.c:

### Functions

- [int ofb\\_setiv](#) (const unsigned char \*IV, unsigned long [len](#), symmetric\_OFB \*ofb)  
*Set an initial vector.*

### 5.204.2 Function Documentation

#### 5.204.2.1 int ofb\_setiv (const unsigned char \*IV, unsigned long len, symmetric\_OFB \* ofb)

Set an initial vector.

#### Parameters:

- IV* The initial vector
- len* The length of the vector (in octets)
- ofb* The OFB state

#### Returns:

CRYPT\_OK if successful

Definition at line 27 of file ofb\_setiv.c.

References [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [ltc\\_cipher\\_descriptor::ecb\\_encrypt](#), and [LTC\\_ARGCHK](#).

```
28 {
29     int err;
30
31     LTC_ARGCHK(IV != NULL);
32     LTC_ARGCHK(ofb != NULL);
33
34     if ((err = cipher_is_valid(ofb->cipher)) != CRYPT_OK) {
35         return err;
36     }
37
38     if (len != (unsigned long)ofb->blocklen) {
39         return CRYPT_INVALID_ARG;
40     }
41
42     /* force next block */
43     ofb->padlen = 0;
44     return cipher_descriptor[ofb->cipher].ecb_encrypt(IV, ofb->IV, &ofb->key);
45 }
```

Here is the call graph for this function:

## 5.205 modes/ofb/ofb\_start.c File Reference

### 5.205.1 Detailed Description

OFB implementation, start chain, Tom St Denis.

Definition in file [ofb\\_start.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ofb\_start.c:

### Functions

- [int ofb\\_start](#) (int cipher, const unsigned char \*IV, const unsigned char \*key, int keylen, int num\_rounds, symmetric\_OFB \*ofb)

*Initialize a OFB context.*

### 5.205.2 Function Documentation

#### 5.205.2.1 int ofb\_start (int cipher, const unsigned char \*IV, const unsigned char \*key, int keylen, int num\_rounds, symmetric\_OFB \*ofb)

Initialize a OFB context.

#### Parameters:

***cipher*** The index of the cipher desired

***IV*** The initial vector

***key*** The secret key

***keylen*** The length of the secret key (octets)

***num\_rounds*** Number of rounds in the cipher desired (0 for default)

***ofb*** The OFB state to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 31 of file ofb\_start.c.

References [ltc\\_cipher\\_descriptor::block\\_length](#), [cipher\\_descriptor](#), [cipher\\_is\\_valid\(\)](#), [CRYPT\\_OK](#), and [LTC\\_ARGCHK](#).

```
33 {
34     int x, err;
35
36     LTC_ARGCHK(IV != NULL);
37     LTC_ARGCHK(key != NULL);
38     LTC_ARGCHK(ofb != NULL);
39
40     if ((err = cipher_is_valid(cipher)) != CRYPT_OK) {
41         return err;
42     }
43
44     /* copy details */
```

```
45     ofb->cipher = cipher;
46     ofb->blocklen = cipher_descriptor[cipher].block_length;
47     for (x = 0; x < ofb->blocklen; x++) {
48         ofb->IV[x] = IV[x];
49     }
50
51     /* init the cipher */
52     ofb->padlen = ofb->blocklen;
53     return cipher_descriptor[cipher].setup(key, keylen, num_rounds, &ofb->key);
54 }
```

Here is the call graph for this function:

## 5.206 pk/asn1/der/bit/der\_decode\_bit\_string.c File Reference

### 5.206.1 Detailed Description

ASN.1 DER, encode a BIT STRING, Tom St Denis.

Definition in file [der\\_decode\\_bit\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_decode\_bit\_string.c:

### Functions

- int [der\\_decode\\_bit\\_string](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)  
*Store a BIT STRING.*

### 5.206.2 Function Documentation

#### 5.206.2.1 int der\_decode\_bit\_string (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

Store a BIT STRING.

#### Parameters:

- in* The DER encoded BIT STRING  
*inlen* The size of the DER BIT STRING  
*out* [out] The array of bits stored (one per char)  
*outlen* [in/out] The number of bits stored

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file der\_decode\_bit\_string.c.

References [CRYPT\\_BUFFER\\_OVERFLOW](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_INVALID\\_PACKET](#), and [LTC\\_ARGCHK](#).

Referenced by [der\\_decode\\_sequence\\_flexi\(\)](#).

```
31 {  
32     unsigned long dlen, blen, x, y;  
33  
34     LTC_ARGCHK(in != NULL);  
35     LTC_ARGCHK(out != NULL);  
36     LTC_ARGCHK(outlen != NULL);  
37  
38     /* packet must be at least 4 bytes */  
39     if (inlen < 4) {  
40         return CRYPT_INVALID_ARG;  
41     }  
42  
43     /* check for 0x03 */
```

```
44     if ((in[0]&0x1F) != 0x03) {
45         return CRYPT_INVALID_PACKET;
46     }
47
48     /* offset in the data */
49     x = 1;
50
51     /* get the length of the data */
52     if (in[x] & 0x80) {
53         /* long format get number of length bytes */
54         y = in[x++] & 0x7F;
55
56         /* invalid if 0 or > 2 */
57         if (y == 0 || y > 2) {
58             return CRYPT_INVALID_PACKET;
59         }
60
61         /* read the data len */
62         dlen = 0;
63         while (y-- > 0) {
64             dlen = (dlen << 8) | (unsigned long)in[x++];
65         }
66     } else {
67         /* short format */
68         dlen = in[x++] & 0x7F;
69     }
70
71     /* is the data len too long or too short? */
72     if ((dlen == 0) || (dlen + x > inlen)) {
73         return CRYPT_INVALID_PACKET;
74     }
75
76     /* get padding count */
77     blen = ((dlen - 1) << 3) - (in[x++] & 7);
78
79     /* too many bits? */
80     if (blen > *outlen) {
81         *outlen = blen;
82         return CRYPT_BUFFER_OVERFLOW;
83     }
84
85     /* decode/store the bits */
86     for (y = 0; y < blen; y++) {
87         out[y] = (in[x] & (1 << (7 - (y & 7)))) ? 1 : 0;
88         if ((y & 7) == 7) {
89             ++x;
90         }
91     }
92
93     /* we done */
94     *outlen = blen;
95     return CRYPT_OK;
96 }
```



## 5.207 pk/asn1/der/bit/der\_encode\_bit\_string.c File Reference

### 5.207.1 Detailed Description

ASN.1 DER, encode a BIT STRING, Tom St Denis.

Definition in file [der\\_encode\\_bit\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_encode\_bit\_string.c:

### Functions

- [int der\\_encode\\_bit\\_string](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*Store a BIT STRING.*

### 5.207.2 Function Documentation

#### 5.207.2.1 int der\_encode\_bit\_string (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

Store a BIT STRING.

#### Parameters:

*in* The array of bits to store (one per char)

*inlen* The number of bits to store

*out* [out] The destination for the DER encoded BIT STRING

*outlen* [in/out] The max size and resulting size of the DER BIT STRING

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file der\_encode\_bit\_string.c.

References [CRYPT\\_BUFFER\\_OVERFLOW](#), [CRYPT\\_OK](#), [der\\_length\\_bit\\_string\(\)](#), [len](#), and [LTC\\_ARGCHK](#).

```
31 {
32     unsigned long len, x, y, buf;
33     int          err;
34
35     LTC_ARGCHK(in      != NULL);
36     LTC_ARGCHK(out     != NULL);
37     LTC_ARGCHK(outlen != NULL);
38
39     /* avoid overflows */
40     if ((err = der_length_bit_string(inlen, &len)) != CRYPT_OK) {
41         return err;
42     }
43
44     if (len > *outlen) {
45         *outlen = len;
```

```
46     return CRYPT_BUFFER_OVERFLOW;
47 }
48
49 /* store header (include bit padding count in length) */
50 x = 0;
51 y = (inlen >> 3) + ((inlen&7) ? 1 : 0) + 1;
52
53 out[x++] = 0x03;
54 if (y < 128) {
55     out[x++] = y;
56 } else if (y < 256) {
57     out[x++] = 0x81;
58     out[x++] = y;
59 } else if (y < 65536) {
60     out[x++] = 0x82;
61     out[x++] = (y>>8)&255;
62     out[x++] = y&255;
63 }
64
65 /* store number of zero padding bits */
66 out[x++] = (8 - inlen) & 7;
67
68 /* store the bits in big endian format */
69 for (y = buf = 0; y < inlen; y++) {
70     buf |= (in[y] ? 1 : 0) << (7 - (y & 7));
71     if ((y & 7) == 7) {
72         out[x++] = buf;
73         buf = 0;
74     }
75 }
76 /* store last byte */
77 if (inlen & 7) {
78     out[x++] = buf;
79 }
80 *outlen = x;
81 return CRYPT_OK;
82 }
```

Here is the call graph for this function:

## 5.208 pk/asn1/der/bit/der\_length\_bit\_string.c File Reference

### 5.208.1 Detailed Description

ASN.1 DER, get length of BIT STRING, Tom St Denis.

Definition in file [der\\_length\\_bit\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_length\_bit\_string.c:

### Functions

- [int der\\_length\\_bit\\_string](#) (unsigned long nbits, unsigned long \*outlen)  
*Gets length of DER encoding of BIT STRING.*

### 5.208.2 Function Documentation

#### 5.208.2.1 int der\_length\_bit\_string (unsigned long nbits, unsigned long \* outlen)

Gets length of DER encoding of BIT STRING.

##### Parameters:

- nbits** The number of bits in the string to encode
- outlen** [out] The length of the DER encoding for the given string

##### Returns:

CRYPT\_OK if successful

Definition at line 25 of file der\_length\_bit\_string.c.

References CRYPT\_INVALID\_ARG, CRYPT\_OK, and LTC\_ARGCHK.

Referenced by der\_decode\_sequence\_flexi(), and der\_encode\_bit\_string().

```

26 {
27     unsigned long nbytes;
28     LTC_ARGCHK(outlen != NULL);
29
30     /* get the number of the bytes */
31     nbytes = (nbits >> 3) + ((nbits & 7) ? 1 : 0) + 1;
32
33     if (nbytes < 128) {
34         /* 03 LL PP DD DD DD ... */
35         *outlen = 2 + nbytes;
36     } else if (nbytes < 256) {
37         /* 03 81 LL PP DD DD DD ... */
38         *outlen = 3 + nbytes;
39     } else if (nbytes < 65536) {
40         /* 03 82 LL LL PP DD DD DD ... */
41         *outlen = 4 + nbytes;
42     } else {
43         return CRYPT_INVALID_ARG;
44     }
45
46     return CRYPT_OK;
47 }
```

## 5.209 pk/asn1/der/boolean/der\_decode\_boolean.c File Reference

### 5.209.1 Detailed Description

ASN.1 DER, decode a BOOLEAN, Tom St Denis.

Definition in file [der\\_decode\\_boolean.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_decode\_boolean.c:

### Functions

- int [der\\_decode\\_boolean](#) (const unsigned char \**in*, unsigned long *inlen*, int \**out*)  
*Read a BOOLEAN.*

### 5.209.2 Function Documentation

#### 5.209.2.1 int der\_decode\_boolean (const unsigned char \* *in*, unsigned long *inlen*, int \* *out*)

Read a BOOLEAN.

#### Parameters:

*in* The destination for the DER encoded BOOLEAN

*inlen* The size of the DER BOOLEAN

*out* [out] The boolean to decode

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file der\_decode\_boolean.c.

References [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), and [LTC\\_ARGCHK](#).

Referenced by [der\\_decode\\_sequence\\_flexi\(\)](#).

```
30 {
31     LTC_ARGCHK(in != NULL);
32     LTC_ARGCHK(out != NULL);
33
34     if (inlen != 3 || in[0] != 0x01 || in[1] != 0x01 || (in[2] != 0x00 && in[2] != 0xFF)) {
35         return CRYPT_INVALID_ARG;
36     }
37
38     *out = (in[2]==0xFF) ? 1 : 0;
39
40     return CRYPT_OK;
41 }
```

## 5.210 pk/asn1/der/boolean/der\_encode\_boolean.c File Reference

### 5.210.1 Detailed Description

ASN.1 DER, encode a BOOLEAN, Tom St Denis.

Definition in file [der\\_encode\\_boolean.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_encode\_boolean.c:

### Functions

- int [der\\_encode\\_boolean](#) (int *in*, unsigned char \*out, unsigned long \*outlen)  
*Store a BOOLEAN.*

### 5.210.2 Function Documentation

#### 5.210.2.1 int der\_encode\_boolean (int *in*, unsigned char \* *out*, unsigned long \* *outlen*)

Store a BOOLEAN.

##### Parameters:

*in* The boolean to encode

*out* [out] The destination for the DER encoded BOOLEAN

*outlen* [in/out] The max size and resulting size of the DER BOOLEAN

##### Returns:

CRYPT\_OK if successful

Definition at line 28 of file der\_encode\_boolean.c.

References [CRYPT\\_BUFFER\\_OVERFLOW](#), [CRYPT\\_OK](#), and [LTC\\_ARGCHK](#).

```
30 {
31     LTC_ARGCHK(outlen != NULL);
32     LTC_ARGCHK(out != NULL);
33
34     if (*outlen < 3) {
35         *outlen = 3;
36         return CRYPT_BUFFER_OVERFLOW;
37     }
38
39     *outlen = 3;
40     out[0] = 0x01;
41     out[1] = 0x01;
42     out[2] = in ? 0xFF : 0x00;
43
44     return CRYPT_OK;
45 }
```

## 5.211 pk/asn1/der/boolean/der\_length\_boolean.c File Reference

### 5.211.1 Detailed Description

ASN.1 DER, get length of a BOOLEAN, Tom St Denis.

Definition in file [der\\_length\\_boolean.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_length\_boolean.c:

### Functions

- int [der\\_length\\_boolean](#) (unsigned long \*outlen)  
*Gets length of DER encoding of a BOOLEAN.*

### 5.211.2 Function Documentation

#### 5.211.2.1 int der\_length\_boolean (unsigned long \* outlen)

Gets length of DER encoding of a BOOLEAN.

#### Parameters:

*outlen* [out] The length of the DER encoding

#### Returns:

CRYPT\_OK if successful

Definition at line 24 of file der\_length\_boolean.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by der\_decode\_sequence\_flexi().

```
25 {  
26     LTC_ARGCHK(outlen != NULL);  
27     *outlen = 3;  
28     return CRYPT_OK;  
29 }
```

## 5.212 pk/asn1/der/choice/der\_decode\_choice.c File Reference

### 5.212.1 Detailed Description

ASN.1 DER, decode a CHOICE, Tom St Denis.

Definition in file [der\\_decode\\_choice.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_decode\_choice.c:

### Functions

- [int der\\_decode\\_choice](#) (const unsigned char \**in*, unsigned long \*inlen, ltc\_asn1\_list \*list, unsigned long outlen)  
*Decode a CHOICE.*

### 5.212.2 Function Documentation

#### 5.212.2.1 int der\_decode\_choice (const unsigned char \* *in*, unsigned long \* *inlen*, ltc\_asn1\_list \* *list*, unsigned long *outlen*)

Decode a CHOICE.

#### Parameters:

- in* The DER encoded input
- inlen* [in/out] The size of the input and resulting size of read type
- list* The list of items to decode
- outlen* The number of items in the list

#### Returns:

- CRYPT\_OK on success

Definition at line 28 of file der\_decode\_choice.c.

References CRYPT\_INVALID\_PACKET, and LTC\_ARGCHK.

```
30 {
31     unsigned long size, x, z;
32     void          *data;
33
34     LTC_ARGCHK(in    != NULL);
35     LTC_ARGCHK(inlen != NULL);
36     LTC_ARGCHK(list  != NULL);
37
38     /* get blk size */
39     if (*inlen < 2) {
40         return CRYPT_INVALID_PACKET;
41     }
42
43     /* set all of the "used" flags to zero */
44     for (x = 0; x < outlen; x++) {
45         list[x].used = 0;
46     }
```

```
47
48 /* now scan until we have a winner */
49 for (x = 0; x < outlen; x++) {
50     size = list[x].size;
51     data = list[x].data;
52
53     switch (list[x].type) {
54         case LTC_ASN1_INTEGER:
55             if (der_decode_integer(in, *inlen, data) == CRYPT_OK) {
56                 if (der_length_integer(data, &z) == CRYPT_OK) {
57                     list[x].used = 1;
58                     *inlen = z;
59                     return CRYPT_OK;
60                 }
61             }
62             break;
63
64         case LTC_ASN1_SHORT_INTEGER:
65             if (der_decode_short_integer(in, *inlen, data) == CRYPT_OK) {
66                 if (der_length_short_integer(size, &z) == CRYPT_OK) {
67                     list[x].used = 1;
68                     *inlen = z;
69                     return CRYPT_OK;
70                 }
71             }
72             break;
73
74         case LTC_ASN1_BIT_STRING:
75             if (der_decode_bit_string(in, *inlen, data, &size) == CRYPT_OK) {
76                 if (der_length_bit_string(size, &z) == CRYPT_OK) {
77                     list[x].used = 1;
78                     list[x].size = size;
79                     *inlen = z;
80                     return CRYPT_OK;
81                 }
82             }
83             break;
84
85         case LTC_ASN1_OCTET_STRING:
86             if (der_decode_octet_string(in, *inlen, data, &size) == CRYPT_OK) {
87                 if (der_length_octet_string(size, &z) == CRYPT_OK) {
88                     list[x].used = 1;
89                     list[x].size = size;
90                     *inlen = z;
91                     return CRYPT_OK;
92                 }
93             }
94             break;
95
96         case LTC_ASN1_NULL:
97             if (*inlen == 2 && in[x] == 0x05 && in[x+1] == 0x00) {
98                 *inlen = 2;
99                 list[x].used = 1;
100                 return CRYPT_OK;
101             }
102             break;
103
104         case LTC_ASN1_OBJECT_IDENTIFIER:
105             if (der_decode_object_identifier(in, *inlen, data, &size) == CRYPT_OK) {
106                 if (der_length_object_identifier(data, size, &z) == CRYPT_OK) {
107                     list[x].used = 1;
108                     list[x].size = size;
109                     *inlen = z;
110                     return CRYPT_OK;
111                 }
112             }
113             break;
```



```
114
115     case LTC_ASN1_IA5_STRING:
116         if (der_decode_ia5_string(in, *inlen, data, &size) == CRYPT_OK) {
117             if (der_length_ia5_string(data, size, &z) == CRYPT_OK) {
118                 list[x].used = 1;
119                 list[x].size = size;
120                 *inlen = z;
121                 return CRYPT_OK;
122             }
123         }
124         break;
125
126
127     case LTC_ASN1_PRINTABLE_STRING:
128         if (der_decode_printable_string(in, *inlen, data, &size) == CRYPT_OK) {
129             if (der_length_printable_string(data, size, &z) == CRYPT_OK) {
130                 list[x].used = 1;
131                 list[x].size = size;
132                 *inlen = z;
133                 return CRYPT_OK;
134             }
135         }
136         break;
137
138     case LTC_ASN1_UTCTIME:
139         z = *inlen;
140         if (der_decode_utctime(in, &z, data) == CRYPT_OK) {
141             list[x].used = 1;
142             *inlen = z;
143             return CRYPT_OK;
144         }
145         break;
146
147     case LTC_ASN1_SET:
148     case LTC_ASN1_SETOF:
149     case LTC_ASN1_SEQUENCE:
150         if (der_decode_sequence(in, *inlen, data, size) == CRYPT_OK) {
151             if (der_length_sequence(data, size, &z) == CRYPT_OK) {
152                 list[x].used = 1;
153                 *inlen = z;
154                 return CRYPT_OK;
155             }
156         }
157         break;
158
159     default:
160         return CRYPT_INVALID_ARG;
161 }
162 }
163
164 return CRYPT_INVALID_PACKET;
165 }
```

## 5.213 pk/asn1/der/ia5/der\_decode\_ia5\_string.c File Reference

### 5.213.1 Detailed Description

ASN.1 DER, encode a IA5 STRING, Tom St Denis.

Definition in file [der\\_decode\\_ia5\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_decode\_ia5\_string.c:

### Functions

- int [der\\_decode\\_ia5\\_string](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*Store a IA5 STRING.*

### 5.213.2 Function Documentation

#### 5.213.2.1 int der\_decode\_ia5\_string (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

Store a IA5 STRING.

#### Parameters:

- in* The DER encoded IA5 STRING  
*inlen* The size of the DER IA5 STRING  
*out* [out] The array of octets stored (one per char)  
*outlen* [in/out] The number of octets stored

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file der\_decode\_ia5\_string.c.

References [CRYPT\\_BUFFER\\_OVERFLOW](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_INVALID\\_PACKET](#), [der\\_ia5\\_value\\_decode\(\)](#), [len](#), and [LTC\\_ARGCHK](#).

Referenced by [der\\_decode\\_sequence\\_flexi\(\)](#).

```
31 {
32     unsigned long x, y, len;
33     int          t;
34
35     LTC_ARGCHK(in      != NULL);
36     LTC_ARGCHK(out     != NULL);
37     LTC_ARGCHK(outlen != NULL);
38
39     /* must have header at least */
40     if (inlen < 2) {
41         return CRYPT_INVALID_PACKET;
42     }
43 }
```

```
44  /* check for 0x16 */
45  if ((in[0] & 0x1F) != 0x16) {
46      return CRYPT_INVALID_PACKET;
47  }
48  x = 1;
49
50  /* decode the length */
51  if (in[x] & 0x80) {
52      /* valid # of bytes in length are 1,2,3 */
53      y = in[x] & 0x7F;
54      if ((y == 0) || (y > 3) || ((x + y) > inlen)) {
55          return CRYPT_INVALID_PACKET;
56      }
57
58      /* read the length in */
59      len = 0;
60      ++x;
61      while (y--) {
62          len = (len << 8) | in[x++];
63      }
64  } else {
65      len = in[x++] & 0x7F;
66  }
67
68  /* is it too long? */
69  if (len > *outlen) {
70      *outlen = len;
71      return CRYPT_BUFFER_OVERFLOW;
72  }
73
74  if (len + x > inlen) {
75      return CRYPT_INVALID_PACKET;
76  }
77
78  /* read the data */
79  for (y = 0; y < len; y++) {
80      t = der_ia5_value_decode(in[x++]);
81      if (t == -1) {
82          return CRYPT_INVALID_ARG;
83      }
84      out[y] = t;
85  }
86
87  *outlen = y;
88
89  return CRYPT_OK;
90 }
```

Here is the call graph for this function:

## 5.214 pk/asn1/der/ia5/der\_encode\_ia5\_string.c File Reference

### 5.214.1 Detailed Description

ASN.1 DER, encode a IA5 STRING, Tom St Denis.

Definition in file [der\\_encode\\_ia5\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_encode\_ia5\_string.c:

### Functions

- [int der\\_encode\\_ia5\\_string](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*Store an IA5 STRING.*

### 5.214.2 Function Documentation

#### 5.214.2.1 int der\_encode\_ia5\_string (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

Store an IA5 STRING.

#### Parameters:

*in* The array of IA5 to store (one per char)

*inlen* The number of IA5 to store

*out* [out] The destination for the DER encoded IA5 STRING

*outlen* [in/out] The max size and resulting size of the DER IA5 STRING

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file der\_encode\_ia5\_string.c.

References [CRYPT\\_BUFFER\\_OVERFLOW](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [der\\_ia5\\_char\\_encode\(\)](#), [der\\_length\\_ia5\\_string\(\)](#), [len](#), and [LTC\\_ARGCHK](#).

```
30 {
31     unsigned long x, y, len;
32     int          err;
33
34     LTC_ARGCHK(in      != NULL);
35     LTC_ARGCHK(out     != NULL);
36     LTC_ARGCHK(outlen != NULL);
37
38     /* get the size */
39     if ((err = der_length_ia5_string(in, inlen, &len)) != CRYPT_OK) {
40         return err;
41     }
42
43     /* too big? */
44     if (len > *outlen) {
```

```
45     *outlen = len;
46     return CRYPT_BUFFER_OVERFLOW;
47 }
48
49 /* encode the header+len */
50 x = 0;
51 out[x++] = 0x16;
52 if (inlen < 128) {
53     out[x++] = inlen;
54 } else if (inlen < 256) {
55     out[x++] = 0x81;
56     out[x++] = inlen;
57 } else if (inlen < 65536UL) {
58     out[x++] = 0x82;
59     out[x++] = (inlen>>8)&255;
60     out[x++] = inlen&255;
61 } else if (inlen < 16777216UL) {
62     out[x++] = 0x83;
63     out[x++] = (inlen>>16)&255;
64     out[x++] = (inlen>>8)&255;
65     out[x++] = inlen&255;
66 } else {
67     return CRYPT_INVALID_ARG;
68 }
69
70 /* store octets */
71 for (y = 0; y < inlen; y++) {
72     out[x++] = der_ia5_char_encode(in[y]);
73 }
74
75 /* return length */
76 *outlen = x;
77
78 return CRYPT_OK;
79 }
```

Here is the call graph for this function:

## 5.215 pk/asn1/der/ia5/der\_length\_ia5\_string.c File Reference

### 5.215.1 Detailed Description

ASN.1 DER, get length of IA5 STRING, Tom St Denis.

Definition in file [der\\_length\\_ia5\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [der\\_length\\_ia5\\_string.c](#):

### Functions

- [int der\\_ia5\\_char\\_encode](#) (int c)
- [int der\\_ia5\\_value\\_decode](#) (int v)
- [int der\\_length\\_ia5\\_string](#) (const unsigned char \*octets, unsigned long noctets, unsigned long \*outlen)

*Gets length of DER encoding of IA5 STRING.*

### Variables

- struct {  
    int [len](#)  
    unsigned char [poly\\_div](#) [MAXBLOCKSIZE]  
    unsigned char [poly\\_mul](#) [MAXBLOCKSIZE]  
    int [code](#)  
    int [value](#)  
} [ia5\\_table](#) []

### 5.215.2 Function Documentation

#### 5.215.2.1 int der\_ia5\_char\_encode (int c)

Definition at line 127 of file [der\\_length\\_ia5\\_string.c](#).

References [code](#), [ia5\\_table](#), and [value](#).

Referenced by [der\\_encode\\_ia5\\_string\(\)](#), [der\\_encode\\_utctime\(\)](#), and [der\\_length\\_ia5\\_string\(\)](#).

```
128 {  
129     int x;  
130     for (x = 0; x < (int)(sizeof(ia5_table)/sizeof(ia5_table[0])); x++) {  
131         if (ia5_table[x].code == c) {  
132             return ia5_table[x].value;  
133         }  
134     }  
135     return -1;  
136 }
```

### 5.215.2.2 int der\_ia5\_value\_decode (int v)

Definition at line 138 of file der\_length\_ia5\_string.c.

References code, ia5\_table, and value.

Referenced by der\_decode\_ia5\_string(), and der\_decode\_utctime().

```
139 {
140     int x;
141     for (x = 0; x < (int)(sizeof(ia5_table)/sizeof(ia5_table[0])); x++) {
142         if (ia5_table[x].value == v) {
143             return ia5_table[x].code;
144         }
145     }
146     return -1;
147 }
```

### 5.215.2.3 int der\_length\_ia5\_string (const unsigned char \* octets, unsigned long noctets, unsigned long \* outlen)

Gets length of DER encoding of IA5 STRING.

#### Parameters:

*octets* The values you want to encode

*noctets* The number of octets in the string to encode

*outlen* [out] The length of the DER encoding for the given string

#### Returns:

CRYPT\_OK if successful

Definition at line 156 of file der\_length\_ia5\_string.c.

References CRYPT\_INVALID\_ARG, der\_ia5\_char\_encode(), and LTC\_ARGCHK.

Referenced by der\_decode\_sequence\_flexi(), and der\_encode\_ia5\_string().

```
157 {
158     unsigned long x;
159
160     LTC_ARGCHK(outlen != NULL);
161     LTC_ARGCHK(octets != NULL);
162
163     /* scan string for validity */
164     for (x = 0; x < noctets; x++) {
165         if (der_ia5_char_encode(octets[x]) == -1) {
166             return CRYPT_INVALID_ARG;
167         }
168     }
169
170     if (noctets < 128) {
171         /* 16 LL DD DD DD ... */
172         *outlen = 2 + noctets;
173     } else if (noctets < 256) {
174         /* 16 81 LL DD DD DD ... */
175         *outlen = 3 + noctets;
176     } else if (noctets < 65536UL) {
177         /* 16 82 LL LL DD DD DD ... */
178         *outlen = 4 + noctets;
```

```
179     } else if (noctets < 16777216UL) {
180         /* 16 83 LL LL LL DD DD DD ... */
181         *outlen = 5 + noctets;
182     } else {
183         return CRYPT_INVALID_ARG;
184     }
185
186     return CRYPT_OK;
187 }
```

Here is the call graph for this function:

### 5.215.3 Variable Documentation

#### 5.215.3.1 `int code`

Definition at line 21 of file `der_length_ia5_string.c`.

Referenced by `der_ia5_char_encode()`, `der_ia5_value_decode()`, `der_printable_char_encode()`, and `der_printable_value_decode()`.

#### 5.215.3.2 `const { ... } ia5_table[]` `[static]`

Referenced by `der_ia5_char_encode()`, and `der_ia5_value_decode()`.

#### 5.215.3.3 `int value`

Definition at line 21 of file `der_length_ia5_string.c`.

Referenced by `der_ia5_char_encode()`, `der_ia5_value_decode()`, `der_printable_char_encode()`, and `der_printable_value_decode()`.



## 5.216 pk/asn1/der/integer/der\_decode\_integer.c File Reference

### 5.216.1 Detailed Description

ASN.1 DER, decode an integer, Tom St Denis.

Definition in file [der\\_decode\\_integer.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_decode\_integer.c:

### Functions

- [int der\\_decode\\_integer](#) (const unsigned char \**in*, unsigned long *inlen*, void \**num*)  
*Read a mp\_int integer.*

### 5.216.2 Function Documentation

#### 5.216.2.1 int der\_decode\_integer (const unsigned char \* *in*, unsigned long *inlen*, void \* *num*)

Read a mp\_int integer.

#### Parameters:

- in* The DER encoded data
- inlen* Size of DER encoded data
- num* The first mp\_int to decode

#### Returns:

- CRYPT\_OK if successful

Definition at line 28 of file der\_decode\_integer.c.

References CRYPT\_INVALID\_PACKET, CRYPT\_MEM, CRYPT\_OK, and LTC\_ARGCHK.

Referenced by der\_decode\_sequence\_flexi().

```
29 {
30     unsigned long x, y, z;
31     int          err;
32
33     LTC_ARGCHK(num    != NULL);
34     LTC_ARGCHK(in     != NULL);
35
36     /* min DER INTEGER is 0x02 01 00 == 0 */
37     if (inlen < (1 + 1 + 1)) {
38         return CRYPT_INVALID_PACKET;
39     }
40
41     /* ok expect 0x02 when we AND with 0001 1111 [1F] */
42     x = 0;
43     if ((in[x++] & 0x1F) != 0x02) {
44         return CRYPT_INVALID_PACKET;
45     }
46
47     /* now decode the len stuff */
```

```
48     z = in[x++];
49
50     if ((z & 0x80) == 0x00) {
51         /* short form */
52
53         /* will it overflow? */
54         if (x + z > inlen) {
55             return CRYPT_INVALID_PACKET;
56         }
57
58         /* no so read it */
59         if ((err = mp_read_unsigned_bin(num, (unsigned char *)in + x, z)) != CRYPT_OK) {
60             return err;
61         }
62     } else {
63         /* long form */
64         z &= 0x7F;
65
66         /* will number of length bytes overflow? (or > 4) */
67         if (((x + z) > inlen) || (z > 4) || (z == 0)) {
68             return CRYPT_INVALID_PACKET;
69         }
70
71         /* now read it in */
72         y = 0;
73         while (z-- > 0) {
74             y = ((unsigned long)(in[x++])) | (y << 8);
75         }
76
77         /* now will reading y bytes overrun? */
78         if ((x + y) > inlen) {
79             return CRYPT_INVALID_PACKET;
80         }
81
82         /* no so read it */
83         if ((err = mp_read_unsigned_bin(num, (unsigned char *)in + x, y)) != CRYPT_OK) {
84             return err;
85         }
86     }
87
88     /* see if it's negative */
89     if (in[x] & 0x80) {
90         void *tmp;
91         if (mp_init(&tmp) != CRYPT_OK) {
92             return CRYPT_MEM;
93         }
94
95         if (mp_2expt(tmp, mp_count_bits(num)) != CRYPT_OK || mp_sub(num, tmp, num) != CRYPT_OK) {
96             mp_clear(tmp);
97             return CRYPT_MEM;
98         }
99         mp_clear(tmp);
100     }
101
102     return CRYPT_OK;
103 }
104 }
```

## 5.217 pk/asn1/der/integer/der\_encode\_integer.c File Reference

### 5.217.1 Detailed Description

ASN.1 DER, encode an integer, Tom St Denis.

Definition in file [der\\_encode\\_integer.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_encode\_integer.c:

### Functions

- [int der\\_encode\\_integer](#) (void \*num, unsigned char \*out, unsigned long \*outlen)  
*Store a mp\_int integer.*

### 5.217.2 Function Documentation

#### 5.217.2.1 int der\_encode\_integer (void \* num, unsigned char \* out, unsigned long \* outlen)

Store a mp\_int integer.

#### Parameters:

- num** The first mp\_int to encode
- out** [out] The destination for the DER encoded integers
- outlen** [in/out] The max size and resulting size of the DER encoded integers

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file der\_encode\_integer.c.

References [CRYPT\\_BUFFER\\_OVERFLOW](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [der\\_length\\_integer\(\)](#), [LTC\\_ARGCHK](#), [LTC\\_MP\\_GT](#), [LTC\\_MP\\_LT](#), and [LTC\\_MP\\_YES](#).

```
30 {
31     unsigned long tmlen, y;
32     int          err, leading_zero;
33
34     LTC_ARGCHK(num    != NULL);
35     LTC_ARGCHK(out    != NULL);
36     LTC_ARGCHK(outlen != NULL);
37
38     /* find out how big this will be */
39     if ((err = der_length_integer(num, &tmlen)) != CRYPT_OK) {
40         return err;
41     }
42
43     if (*outlen < tmlen) {
44         *outlen = tmlen;
45         return CRYPT_BUFFER_OVERFLOW;
46     }
47
48     if (mp_cmp_d(num, 0) != LTC_MP_LT) {
```

```

49     /* we only need a leading zero if the msb of the first byte is one */
50     if ((mp_count_bits(num) & 7) == 0 || mp_iszero(num) == LTC_MP_YES) {
51         leading_zero = 1;
52     } else {
53         leading_zero = 0;
54     }
55
56     /* get length of num in bytes (plus 1 since we force the msbyte to zero) */
57     y = mp_unsigned_bin_size(num) + leading_zero;
58 } else {
59     leading_zero = 0;
60     y = mp_count_bits(num);
61     y = y + (8 - (y & 7));
62     y = y >> 3;
63     if (((mp_cnt_lsb(num)+1)==mp_count_bits(num)) && ((mp_count_bits(num)&7)==0)) --y;
64 }
65
66 /* now store initial data */
67 *out++ = 0x02;
68 if (y < 128) {
69     /* short form */
70     *out++ = (unsigned char)y;
71 } else if (y < 256) {
72     *out++ = 0x81;
73     *out++ = y;
74 } else if (y < 65536UL) {
75     *out++ = 0x82;
76     *out++ = (y>>8)&255;
77     *out++ = y;
78 } else if (y < 16777216UL) {
79     *out++ = 0x83;
80     *out++ = (y>>16)&255;
81     *out++ = (y>>8)&255;
82     *out++ = y;
83 } else {
84     return CRYPT_INVALID_ARG;
85 }
86
87 /* now store msbyte of zero if num is non-zero */
88 if (leading_zero) {
89     *out++ = 0x00;
90 }
91
92 /* if it's not zero store it as big endian */
93 if (mp_cmp_d(num, 0) == LTC_MP_GT) {
94     /* now store the mpint */
95     if ((err = mp_to_unsigned_bin(num, out)) != CRYPT_OK) {
96         return err;
97     }
98 } else if (mp_iszero(num) != LTC_MP_YES) {
99     void *tmp;
100
101     /* negative */
102     if (mp_init(&tmp) != CRYPT_OK) {
103         return CRYPT_MEM;
104     }
105
106     /* 2^roundup and subtract */
107     y = mp_count_bits(num);
108     y = y + (8 - (y & 7));
109     if (((mp_cnt_lsb(num)+1)==mp_count_bits(num)) && ((mp_count_bits(num)&7)==0)) y -= 8;
110     if (mp_2expt(tmp, y) != CRYPT_OK || mp_add(tmp, num, tmp) != CRYPT_OK) {
111         mp_clear(tmp);
112         return CRYPT_MEM;
113     }
114     if ((err = mp_to_unsigned_bin(tmp, out)) != CRYPT_OK) {
115         mp_clear(tmp);

```

```
116         return err;
117     }
118     mp_clear(tmp);
119 }
120
121 /* we good */
122 *outlen = tmplen;
123 return CRYPT_OK;
124 }
```

Here is the call graph for this function:

## 5.218 pk/asn1/der/integer/der\_length\_integer.c File Reference

### 5.218.1 Detailed Description

ASN.1 DER, get length of encoding, Tom St Denis.

Definition in file [der\\_length\\_integer.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_length\_integer.c:

### Functions

- int [der\\_length\\_integer](#) (void \*num, unsigned long \*outlen)

*Gets length of DER encoding of num.*

### 5.218.2 Function Documentation

#### 5.218.2.1 int der\_length\_integer (void \* num, unsigned long \* outlen)

Gets length of DER encoding of num.

#### Parameters:

**num** The int to get the size of

**outlen** [out] The length of the DER encoding for the given integer

#### Returns:

CRYPT\_OK if successful

Definition at line 26 of file der\_length\_integer.c.

References CRYPT\_OK, len, LTC\_ARGCHK, LTC\_MP\_LT, and LTC\_MP\_YES.

Referenced by der\_decode\_sequence\_flexi(), and der\_encode\_integer().

```
27 {
28     unsigned long z, len;
29     int          leading_zero;
30
31     LTC_ARGCHK(num      != NULL);
32     LTC_ARGCHK(outlen   != NULL);
33
34     if (mp_cmp_d(num, 0) != LTC_MP_LT) {
35         /* positive */
36
37         /* we only need a leading zero if the msb of the first byte is one */
38         if ((mp_count_bits(num) & 7) == 0 || mp_iszero(num) == LTC_MP_YES) {
39             leading_zero = 1;
40         } else {
41             leading_zero = 0;
42         }
43
44         /* size for bignum */
45         z = len = leading_zero + mp_unsigned_bin_size(num);
46     } else {
```

```
47     /* it's negative */
48     /* find power of 2 that is a multiple of eight and greater than count bits */
49     leading_zero = 0;
50     z = mp_count_bits(num);
51     z = z + (8 - (z & 7));
52     if (((mp_cnt_lsb(num)+1)==mp_count_bits(num)) && ((mp_count_bits(num)&7)==0)) --z;
53     len = z = z >> 3;
54 }
55
56 /* now we need a length */
57 if (z < 128) {
58     /* short form */
59     ++len;
60 } else {
61     /* long form (relies on z != 0), assumes length bytes < 128 */
62     ++len;
63
64     while (z) {
65         ++len;
66         z >>= 8;
67     }
68 }
69
70 /* we need a 0x02 to indicate it's INTEGER */
71 ++len;
72
73 /* return length */
74 *outlen = len;
75 return CRYPT_OK;
76 }
```

## 5.219 pk/asn1/der/object\_identifier/der\_decode\_object\_identifier.c File Reference

### 5.219.1 Detailed Description

ASN.1 DER, Decode Object Identifier, Tom St Denis.

Definition in file [der\\_decode\\_object\\_identifier.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_decode\_object\_identifier.c:

### Functions

- int [der\\_decode\\_object\\_identifier](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned long \**words*, unsigned long \**outlen*)

*Decode OID data and store the array of integers in words.*

### 5.219.2 Function Documentation

#### 5.219.2.1 int der\_decode\_object\_identifier (const unsigned char \* *in*, unsigned long *inlen*, unsigned long \* *words*, unsigned long \* *outlen*)

Decode OID data and store the array of integers in words.

#### Parameters:

- in* The OID DER encoded data
- inlen* The length of the OID data
- words* [out] The destination of the OID words
- outlen* [in/out] The number of OID words

#### Returns:

- CRYPT\_OK if successful

Definition at line 27 of file der\_decode\_object\_identifier.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_INVALID\_PACKET, CRYPT\_OK, len, and LTC\_ARGCHK.

Referenced by der\_decode\_sequence\_flexi().

```
29 {
30     unsigned long x, y, t, len;
31
32     LTC_ARGCHK(in != NULL);
33     LTC_ARGCHK(words != NULL);
34     LTC_ARGCHK(outlen != NULL);
35
36     /* header is at least 3 bytes */
37     if (inlen < 3) {
38         return CRYPT_INVALID_PACKET;
39     }
```



```
40
41  /* must be room for at least two words */
42  if (*outlen < 2) {
43      return CRYPT_BUFFER_OVERFLOW;
44  }
45
46  /* decode the packet header */
47  x = 0;
48  if ((in[x++] & 0x1F) != 0x06) {
49      return CRYPT_INVALID_PACKET;
50  }
51
52  /* get the length */
53  if (in[x] < 128) {
54      len = in[x++];
55  } else {
56      if (in[x] < 0x81 || in[x] > 0x82) {
57          return CRYPT_INVALID_PACKET;
58      }
59      y = in[x++] & 0x7F;
60      len = 0;
61      while (y-- > 0) {
62          len = (len << 8) | (unsigned long)in[x++];
63      }
64  }
65
66  if (len < 1 || (len + x) > inlen) {
67      return CRYPT_INVALID_PACKET;
68  }
69
70  /* decode words */
71  y = 0;
72  t = 0;
73  while (len-- > 0) {
74      t = (t << 7) | (in[x] & 0x7F);
75      if (!(in[x++] & 0x80)) {
76          /* store t */
77          if (y >= *outlen) {
78              return CRYPT_BUFFER_OVERFLOW;
79          }
80          if (y == 0) {
81              words[0] = t / 40;
82              words[1] = t % 40;
83              y = 2;
84          } else {
85              words[y++] = t;
86          }
87          t = 0;
88      }
89  }
90
91  *outlen = y;
92  return CRYPT_OK;
93 }
```

## 5.220 pk/asn1/der/object\_identifier/der\_encode\_object\_identifier.c File Reference

### 5.220.1 Detailed Description

ASN.1 DER, Encode Object Identifier, Tom St Denis.

Definition in file [der\\_encode\\_object\\_identifier.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_encode\_object\_identifier.c:

### Functions

- `int der_encode_object_identifier` (unsigned long \*words, unsigned long nwords, unsigned char \*out, unsigned long \*outlen)

*Encode an OID.*

### 5.220.2 Function Documentation

#### 5.220.2.1 `int der_encode_object_identifier` (unsigned long \* words, unsigned long nwords, unsigned char \* out, unsigned long \* outlen)

Encode an OID.

#### Parameters:

**words** The words to encode (upto 32-bits each)

**nwords** The number of words in the OID

**out** [out] Destination of OID data

**outlen** [in/out] The max and resulting size of the OID

#### Returns:

CRYPT\_OK if successful

Definition at line 27 of file der\_encode\_object\_identifier.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_OK, der\_length\_object\_identifier(), der\_object\_identifier\_bits(), LTC\_ARGCHK, and mask.

```
29 {
30     unsigned long i, x, y, z, t, mask, wordbuf;
31     int          err;
32
33     LTC_ARGCHK(words != NULL);
34     LTC_ARGCHK(out   != NULL);
35     LTC_ARGCHK(outlen != NULL);
36
37     /* check length */
38     if ((err = der_length_object_identifier(words, nwords, &x)) != CRYPT_OK) {
39         return err;
40     }
41     if (x > *outlen) {
```

```

42     *outlen = x;
43     return CRYPT_BUFFER_OVERFLOW;
44 }
45
46 /* compute length to store OID data */
47 z = 0;
48 wordbuf = words[0] * 40 + words[1];
49 for (y = 1; y < nwords; y++) {
50     t = der_object_identifier_bits(wordbuf);
51     z += t/7 + ((t%7) ? 1 : 0) + (wordbuf == 0 ? 1 : 0);
52     if (y < nwords - 1) {
53         wordbuf = words[y + 1];
54     }
55 }
56
57 /* store header + length */
58 x = 0;
59 out[x++] = 0x06;
60 if (z < 128) {
61     out[x++] = z;
62 } else if (z < 256) {
63     out[x++] = 0x81;
64     out[x++] = z;
65 } else if (z < 65536UL) {
66     out[x++] = 0x82;
67     out[x++] = (z>>8)&255;
68     out[x++] = z&255;
69 } else {
70     return CRYPT_INVALID_ARG;
71 }
72
73 /* store first byte */
74 wordbuf = words[0] * 40 + words[1];
75 for (i = 1; i < nwords; i++) {
76     /* store 7 bit words in little endian */
77     t = wordbuf & 0xFFFFFFFF;
78     if (t) {
79         y = x;
80         mask = 0;
81         while (t) {
82             out[x++] = (t & 0x7F) | mask;
83             t >>= 7;
84             mask |= 0x80; /* upper bit is set on all but the last byte */
85         }
86         /* now swap bytes y...x-1 */
87         z = x - 1;
88         while (y < z) {
89             t = out[y]; out[y] = out[z]; out[z] = t;
90             ++y;
91             --z;
92         }
93     } else {
94         /* zero word */
95         out[x++] = 0x00;
96     }
97
98     if (i < nwords - 1) {
99         wordbuf = words[i + 1];
100     }
101 }
102
103 *outlen = x;
104 return CRYPT_OK;
105 }

```

Here is the call graph for this function:

## 5.221 pk/asn1/der/object\_identifier/der\_length\_object\_identifier.c File Reference

### 5.221.1 Detailed Description

ASN.1 DER, get length of Object Identifier, Tom St Denis.

Definition in file [der\\_length\\_object\\_identifier.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_length\_object\_identifier.c:

### Functions

- unsigned long [der\\_object\\_identifier\\_bits](#) (unsigned long x)
- int [der\\_length\\_object\\_identifier](#) (unsigned long \*words, unsigned long nwords, unsigned long \*outlen)

*Gets length of DER encoding of Object Identifier.*

### 5.221.2 Function Documentation

#### 5.221.2.1 int der\_length\_object\_identifier (unsigned long \* words, unsigned long nwords, unsigned long \* outlen)

Gets length of DER encoding of Object Identifier.

#### Parameters:

**nwords** The number of OID words

**words** The actual OID words to get the size of

**outlen** [out] The length of the DER encoding for the given string

#### Returns:

CRYPT\_OK if successful

Definition at line 40 of file der\_length\_object\_identifier.c.

References CRYPT\_INVALID\_ARG, der\_object\_identifier\_bits(), and LTC\_ARGCHK.

Referenced by der\_decode\_sequence\_flexi(), and der\_encode\_object\_identifier().

```
41 {
42     unsigned long y, z, t, wordbuf;
43
44     LTC_ARGCHK(words != NULL);
45     LTC_ARGCHK(outlen != NULL);
46
47
48     /* must be >= 2 words */
49     if (nwords < 2) {
50         return CRYPT_INVALID_ARG;
51     }
52
53     /* word1 = 0,1,2,3 and word2 0..39 */
```

```
54     if (words[0] > 3 || (words[0] < 2 && words[1] > 39)) {
55         return CRYPT_INVALID_ARG;
56     }
57
58     /* leading word is the first two */
59     z = 0;
60     wordbuf = words[0] * 40 + words[1];
61     for (y = 1; y < nwords; y++) {
62         t = der_object_identifier_bits(wordbuf);
63         z += t/7 + ((t%7) ? 1 : 0) + (wordbuf == 0 ? 1 : 0);
64         if (y < nwords - 1) {
65             /* grab next word */
66             wordbuf = words[y+1];
67         }
68     }
69
70     /* now depending on the length our length encoding changes */
71     if (z < 128) {
72         z += 2;
73     } else if (z < 256) {
74         z += 3;
75     } else if (z < 65536UL) {
76         z += 4;
77     } else {
78         return CRYPT_INVALID_ARG;
79     }
80
81     *outlen = z;
82     return CRYPT_OK;
83 }
```

Here is the call graph for this function:

#### 5.221.2.2 unsigned long der\_object\_identifier\_bits (unsigned long x)

Definition at line 20 of file der\_length\_object\_identifier.c.

References c.

Referenced by der\_encode\_object\_identifier(), and der\_length\_object\_identifier().

```
21 {
22     unsigned long c;
23     x &= 0xFFFFFFFF;
24     c = 0;
25     while (x) {
26         ++c;
27         x >>= 1;
28     }
29     return c;
30 }
```

## 5.222 pk/asn1/der/octet/der\_decode\_octet\_string.c File Reference

### 5.222.1 Detailed Description

ASN.1 DER, encode a OCTET STRING, Tom St Denis.

Definition in file [der\\_decode\\_octet\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_decode\_octet\_string.c:

### Functions

- [int der\\_decode\\_octet\\_string](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*Store a OCTET STRING.*

### 5.222.2 Function Documentation

#### 5.222.2.1 int der\_decode\_octet\_string (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

Store a OCTET STRING.

#### Parameters:

*in* The DER encoded OCTET STRING

*inlen* The size of the DER OCTET STRING

*out* [out] The array of octets stored (one per char)

*outlen* [in/out] The number of octets stored

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file der\_decode\_octet\_string.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_INVALID\_PACKET, *len*, and LTC\_ARGCHK.

Referenced by der\_decode\_sequence\_flexi().

```
31 {
32     unsigned long x, y, len;
33
34     LTC_ARGCHK(in != NULL);
35     LTC_ARGCHK(out != NULL);
36     LTC_ARGCHK(outlen != NULL);
37
38     /* must have header at least */
39     if (inlen < 2) {
40         return CRYPT_INVALID_PACKET;
41     }
42
43     /* check for 0x04 */
44     if ((in[0] & 0x1F) != 0x04) {
45         return CRYPT_INVALID_PACKET;
```

```
46     }
47     x = 1;
48
49     /* decode the length */
50     if (in[x] & 0x80) {
51         /* valid # of bytes in length are 1,2,3 */
52         y = in[x] & 0x7F;
53         if ((y == 0) || (y > 3) || ((x + y) > inlen)) {
54             return CRYPT_INVALID_PACKET;
55         }
56
57         /* read the length in */
58         len = 0;
59         ++x;
60         while (y--) {
61             len = (len << 8) | in[x++];
62         }
63     } else {
64         len = in[x++] & 0x7F;
65     }
66
67     /* is it too long? */
68     if (len > *outlen) {
69         *outlen = len;
70         return CRYPT_BUFFER_OVERFLOW;
71     }
72
73     if (len + x > inlen) {
74         return CRYPT_INVALID_PACKET;
75     }
76
77     /* read the data */
78     for (y = 0; y < len; y++) {
79         out[y] = in[x++];
80     }
81
82     *outlen = y;
83
84     return CRYPT_OK;
85 }
```

## 5.223 pk/asn1/der/octet/der\_encode\_octet\_string.c File Reference

### 5.223.1 Detailed Description

ASN.1 DER, encode a OCTET STRING, Tom St Denis.

Definition in file [der\\_encode\\_octet\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_encode\_octet\_string.c:

### Functions

- [int der\\_encode\\_octet\\_string](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*Store an OCTET STRING.*

### 5.223.2 Function Documentation

#### 5.223.2.1 int der\_encode\_octet\_string (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

Store an OCTET STRING.

#### Parameters:

*in* The array of OCTETS to store (one per char)

*inlen* The number of OCTETS to store

*out* [out] The destination for the DER encoded OCTET STRING

*outlen* [in/out] The max size and resulting size of the DER OCTET STRING

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file der\_encode\_octet\_string.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_INVALID\_ARG, CRYPT\_OK, der\_length\_octet\_string(), len, and LTC\_ARGCHK.

```
31 {
32     unsigned long x, y, len;
33     int          err;
34
35     LTC_ARGCHK(in      != NULL);
36     LTC_ARGCHK(out     != NULL);
37     LTC_ARGCHK(outlen != NULL);
38
39     /* get the size */
40     if ((err = der_length_octet_string(inlen, &len)) != CRYPT_OK) {
41         return err;
42     }
43
44     /* too big? */
45     if (len > *outlen) {
```



```
46     *outlen = len;
47     return CRYPT_BUFFER_OVERFLOW;
48 }
49
50 /* encode the header+len */
51 x = 0;
52 out[x++] = 0x04;
53 if (inlen < 128) {
54     out[x++] = inlen;
55 } else if (inlen < 256) {
56     out[x++] = 0x81;
57     out[x++] = inlen;
58 } else if (inlen < 65536UL) {
59     out[x++] = 0x82;
60     out[x++] = (inlen>>8)&255;
61     out[x++] = inlen&255;
62 } else if (inlen < 16777216UL) {
63     out[x++] = 0x83;
64     out[x++] = (inlen>>16)&255;
65     out[x++] = (inlen>>8)&255;
66     out[x++] = inlen&255;
67 } else {
68     return CRYPT_INVALID_ARG;
69 }
70
71 /* store octets */
72 for (y = 0; y < inlen; y++) {
73     out[x++] = in[y];
74 }
75
76 /* return length */
77 *outlen = x;
78
79 return CRYPT_OK;
80 }
```

Here is the call graph for this function:

## 5.224 pk/asn1/der/octet/der\_length\_octet\_string.c File Reference

### 5.224.1 Detailed Description

ASN.1 DER, get length of OCTET STRING, Tom St Denis.

Definition in file [der\\_length\\_octet\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_length\_octet\_string.c:

### Functions

- [int der\\_length\\_octet\\_string](#) (unsigned long noctets, unsigned long \*outlen)  
*Gets length of DER encoding of OCTET STRING.*

### 5.224.2 Function Documentation

#### 5.224.2.1 int der\_length\_octet\_string (unsigned long noctets, unsigned long \* outlen)

Gets length of DER encoding of OCTET STRING.

##### Parameters:

**noctets** The number of octets in the string to encode

**outlen** [out] The length of the DER encoding for the given string

##### Returns:

CRYPT\_OK if successful

Definition at line 25 of file der\_length\_octet\_string.c.

References CRYPT\_INVALID\_ARG, CRYPT\_OK, and LTC\_ARGCHK.

Referenced by der\_decode\_sequence\_flexi(), and der\_encode\_octet\_string().

```
26 {
27     LTC_ARGCHK(outlen != NULL);
28
29     if (noctets < 128) {
30         /* 04 LL DD DD DD ... */
31         *outlen = 2 + noctets;
32     } else if (noctets < 256) {
33         /* 04 81 LL DD DD DD ... */
34         *outlen = 3 + noctets;
35     } else if (noctets < 65536UL) {
36         /* 04 82 LL LL DD DD DD ... */
37         *outlen = 4 + noctets;
38     } else if (noctets < 16777216UL) {
39         /* 04 83 LL LL LL DD DD DD ... */
40         *outlen = 5 + noctets;
41     } else {
42         return CRYPT_INVALID_ARG;
43     }
44
45     return CRYPT_OK;
46 }
```

## 5.225 pk/asn1/der/printable\_string/der\_decode\_printable\_string.c File Reference

### 5.225.1 Detailed Description

ASN.1 DER, encode a printable STRING, Tom St Denis.

Definition in file [der\\_decode\\_printable\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_decode\_printable\_string.c:

### Functions

- `int der_decode_printable_string` (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*Store a printable STRING.*

### 5.225.2 Function Documentation

#### 5.225.2.1 `int der_decode_printable_string` (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

Store a printable STRING.

##### Parameters:

- in* The DER encoded printable STRING
- inlen* The size of the DER printable STRING
- out* [out] The array of octets stored (one per char)
- outlen* [in/out] The number of octets stored

##### Returns:

CRYPT\_OK if successful

Definition at line 29 of file der\_decode\_printable\_string.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_INVALID\_ARG, CRYPT\_INVALID\_PACKET, der\_printable\_value\_decode(), len, and LTC\_ARGCHK.

Referenced by der\_decode\_sequence\_flexi().

```
31 {
32     unsigned long x, y, len;
33     int          t;
34
35     LTC_ARGCHK(in      != NULL);
36     LTC_ARGCHK(out     != NULL);
37     LTC_ARGCHK(outlen != NULL);
38
39     /* must have header at least */
40     if (inlen < 2) {
41         return CRYPT_INVALID_PACKET;
```

```
42     }
43
44     /* check for 0x13 */
45     if ((in[0] & 0x1F) != 0x13) {
46         return CRYPT_INVALID_PACKET;
47     }
48     x = 1;
49
50     /* decode the length */
51     if (in[x] & 0x80) {
52         /* valid # of bytes in length are 1,2,3 */
53         y = in[x] & 0x7F;
54         if ((y == 0) || (y > 3) || ((x + y) > inlen)) {
55             return CRYPT_INVALID_PACKET;
56         }
57
58         /* read the length in */
59         len = 0;
60         ++x;
61         while (y-- > 0) {
62             len = (len << 8) | in[x++];
63         }
64     } else {
65         len = in[x++] & 0x7F;
66     }
67
68     /* is it too long? */
69     if (len > *outlen) {
70         *outlen = len;
71         return CRYPT_BUFFER_OVERFLOW;
72     }
73
74     if (len + x > inlen) {
75         return CRYPT_INVALID_PACKET;
76     }
77
78     /* read the data */
79     for (y = 0; y < len; y++) {
80         t = der_printable_value_decode(in[x++]);
81         if (t == -1) {
82             return CRYPT_INVALID_ARG;
83         }
84         out[y] = t;
85     }
86
87     *outlen = y;
88
89     return CRYPT_OK;
90 }
```

Here is the call graph for this function:

## 5.226 pk/asn1/der/printable\_string/der\_encode\_printable\_string.c File Reference

### 5.226.1 Detailed Description

ASN.1 DER, encode a printable STRING, Tom St Denis.

Definition in file [der\\_encode\\_printable\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `der_encode_printable_string.c`:

### Functions

- `int der_encode_printable_string` (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*)

*Store an printable STRING.*

### 5.226.2 Function Documentation

#### 5.226.2.1 `int der_encode_printable_string` (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*)

Store an printable STRING.

#### Parameters:

*in* The array of printable to store (one per char)

*inlen* The number of printable to store

*out* [out] The destination for the DER encoded printable STRING

*outlen* [in/out] The max size and resulting size of the DER printable STRING

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file `der_encode_printable_string.c`.

References `CRYPT_BUFFER_OVERFLOW`, `CRYPT_INVALID_ARG`, `CRYPT_OK`, `der_length_printable_string()`, `der_printable_char_encode()`, `len`, and `LTC_ARGCHK`.

```
30 {
31     unsigned long x, y, len;
32     int          err;
33
34     LTC_ARGCHK(in      != NULL);
35     LTC_ARGCHK(out     != NULL);
36     LTC_ARGCHK(outlen != NULL);
37
38     /* get the size */
39     if ((err = der_length_printable_string(in, inlen, &len)) != CRYPT_OK) {
40         return err;
41     }
42 }
```

```
43  /* too big? */
44  if (len > *outlen) {
45      *outlen = len;
46      return CRYPT_BUFFER_OVERFLOW;
47  }
48
49  /* encode the header+len */
50  x = 0;
51  out[x++] = 0x13;
52  if (inlen < 128) {
53      out[x++] = inlen;
54  } else if (inlen < 256) {
55      out[x++] = 0x81;
56      out[x++] = inlen;
57  } else if (inlen < 65536UL) {
58      out[x++] = 0x82;
59      out[x++] = (inlen>>8)&255;
60      out[x++] = inlen&255;
61  } else if (inlen < 16777216UL) {
62      out[x++] = 0x83;
63      out[x++] = (inlen>>16)&255;
64      out[x++] = (inlen>>8)&255;
65      out[x++] = inlen&255;
66  } else {
67      return CRYPT_INVALID_ARG;
68  }
69
70  /* store octets */
71  for (y = 0; y < inlen; y++) {
72      out[x++] = der_printable_char_encode(in[y]);
73  }
74
75  /* return length */
76  *outlen = x;
77
78  return CRYPT_OK;
79 }
```

Here is the call graph for this function:

## 5.227 pk/asn1/der/printable\_string/der\_length\_printable\_string.c File Reference

### 5.227.1 Detailed Description

ASN.1 DER, get length of Printable STRING, Tom St Denis.

Definition in file [der\\_length\\_printable\\_string.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_length\_printable\_string.c:

### Functions

- int [der\\_printable\\_char\\_encode](#) (int c)
- int [der\\_printable\\_value\\_decode](#) (int v)
- int [der\\_length\\_printable\\_string](#) (const unsigned char \*octets, unsigned long noctets, unsigned long \*outlen)

*Gets length of DER encoding of Printable STRING.*

### Variables

- struct {  
    int [len](#)  
    unsigned char [poly\\_div](#) [MAXBLOCKSIZE]  
    unsigned char [poly\\_mul](#) [MAXBLOCKSIZE]  
    int [code](#)  
    int [value](#)  
} [printable\\_table](#) []

### 5.227.2 Function Documentation

#### 5.227.2.1 int der\_length\_printable\_string (const unsigned char \* octets, unsigned long noctets, unsigned long \* outlen)

Gets length of DER encoding of Printable STRING.

##### Parameters:

*octets* The values you want to encode

*noctets* The number of octets in the string to encode

*outlen* [out] The length of the DER encoding for the given string

##### Returns:

CRYPT\_OK if successful

Definition at line 128 of file der\_length\_printable\_string.c.

References [CRYPT\\_INVALID\\_ARG](#), [der\\_printable\\_char\\_encode\(\)](#), and [LTC\\_ARGCHK](#).

Referenced by [der\\_decode\\_sequence\\_flexi\(\)](#), and [der\\_encode\\_printable\\_string\(\)](#).

```

129 {
130     unsigned long x;
131
132     LTC_ARGCHK(outlen != NULL);
133     LTC_ARGCHK(octets != NULL);
134
135     /* scan string for validity */
136     for (x = 0; x < noctets; x++) {
137         if (der_printable_char_encode(octets[x]) == -1) {
138             return CRYPT_INVALID_ARG;
139         }
140     }
141
142     if (noctets < 128) {
143         /* 16 LL DD DD DD ... */
144         *outlen = 2 + noctets;
145     } else if (noctets < 256) {
146         /* 16 81 LL DD DD DD ... */
147         *outlen = 3 + noctets;
148     } else if (noctets < 65536UL) {
149         /* 16 82 LL LL DD DD DD ... */
150         *outlen = 4 + noctets;
151     } else if (noctets < 16777216UL) {
152         /* 16 83 LL LL LL DD DD DD ... */
153         *outlen = 5 + noctets;
154     } else {
155         return CRYPT_INVALID_ARG;
156     }
157
158     return CRYPT_OK;
159 }

```

Here is the call graph for this function:

#### 5.227.2.2 int der\_printable\_char\_encode(int c)

Definition at line 99 of file der\_length\_printable\_string.c.

References code, printable\_table, and value.

Referenced by der\_encode\_printable\_string(), and der\_length\_printable\_string().

```

100 {
101     int x;
102     for (x = 0; x < (int)(sizeof(printable_table)/sizeof(printable_table[0])); x++) {
103         if (printable_table[x].code == c) {
104             return printable_table[x].value;
105         }
106     }
107     return -1;
108 }

```

#### 5.227.2.3 int der\_printable\_value\_decode(int v)

Definition at line 110 of file der\_length\_printable\_string.c.

References code, printable\_table, and value.

Referenced by der\_decode\_printable\_string().

```

111 {

```



```
112     int x;
113     for (x = 0; x < (int)(sizeof(printable_table)/sizeof(printable_table[0])); x++) {
114         if (printable_table[x].value == v) {
115             return printable_table[x].code;
116         }
117     }
118     return -1;
119 }
```

### 5.227.3 Variable Documentation

#### 5.227.3.1 int [code](#)

Definition at line 21 of file der\_length\_printable\_string.c.

#### 5.227.3.2 const { ... } [printable\\_table\[\]](#) [static]

Referenced by der\_printable\_char\_encode(), and der\_printable\_value\_decode().

#### 5.227.3.3 int [value](#)

Definition at line 21 of file der\_length\_printable\_string.c.

## 5.228 pk/asn1/der/sequence/der\_decode\_sequence\_ex.c File Reference

### 5.228.1 Detailed Description

ASN.1 DER, decode a SEQUENCE, Tom St Denis.

Definition in file [der\\_decode\\_sequence\\_ex.c](#).

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for `der_decode_sequence_ex.c`:

### Functions

- `int der_decode_sequence_ex` (const unsigned char \**in*, unsigned long *inlen*, ltc\_asn1\_list \**list*, unsigned long *outlen*, int *ordered*)

*Decode a SEQUENCE.*

### 5.228.2 Function Documentation

#### 5.228.2.1 `int der_decode_sequence_ex` (const unsigned char \**in*, unsigned long *inlen*, ltc\_asn1\_list \**list*, unsigned long *outlen*, int *ordered*)

Decode a SEQUENCE.

#### Parameters:

*in* The DER encoded input

*inlen* The size of the input

*list* The list of items to decode

*outlen* The number of items in the list

*ordered* Search an unordered or ordered list

#### Returns:

CRYPT\_OK on success

Definition at line 31 of file `der_decode_sequence_ex.c`.

References `CRYPT_INVALID_PACKET`, and `LTC_ARGCHK`.

```
33 {
34     int                err, type;
35     unsigned long size, x, y, z, i, blksize;
36     void                *data;
37
38     LTC_ARGCHK(in != NULL);
39     LTC_ARGCHK(list != NULL);
40
41     /* get blk size */
42     if (inlen < 2) {
43         return CRYPT_INVALID_PACKET;
```

```

44     }
45
46     /* sequence type? We allow 0x30 SEQUENCE and 0x31 SET since fundamentally they're the same structure
47     x = 0;
48     if (in[x] != 0x30 && in[x] != 0x31) {
49         return CRYPT_INVALID_PACKET;
50     }
51     ++x;
52
53     if (in[x] < 128) {
54         blksize = in[x++];
55     } else if (in[x] & 0x80) {
56         if (in[x] < 0x81 || in[x] > 0x83) {
57             return CRYPT_INVALID_PACKET;
58         }
59         y = in[x++] & 0x7F;
60
61         /* would reading the len bytes overrun? */
62         if (x + y > inlen) {
63             return CRYPT_INVALID_PACKET;
64         }
65
66         /* read len */
67         blksize = 0;
68         while (y-- > 0) {
69             blksize = (blksize << 8) | (unsigned long)in[x++];
70         }
71     }
72
73     /* would this blksize overflow? */
74     if (x + blksize > inlen) {
75         return CRYPT_INVALID_PACKET;
76     }
77
78     /* mark all as unused */
79     for (i = 0; i < outlen; i++) {
80         list[i].used = 0;
81     }
82
83     /* ok read data */
84     inlen = blksize;
85     for (i = 0; i < outlen; i++) {
86         z = 0;
87         type = list[i].type;
88         size = list[i].size;
89         data = list[i].data;
90         if (!ordered && list[i].used == 1) { continue; }
91
92         if (type == LTC_ASN1_EOL) {
93             break;
94         }
95
96         switch (type) {
97             case LTC_ASN1_BOOLEAN:
98                 z = inlen;
99                 if ((err = der_decode_boolean(in + x, z, ((int *)data))) != CRYPT_OK) {
100                     goto LBL_ERR;
101                 }
102                 if ((err = der_length_boolean(&z)) != CRYPT_OK) {
103                     goto LBL_ERR;
104                 }
105                 break;
106
107             case LTC_ASN1_INTEGER:
108                 z = inlen;
109                 if ((err = der_decode_integer(in + x, z, data)) != CRYPT_OK) {
110                     if (!ordered) { continue; }

```

```
111         goto LBL_ERR;
112     }
113     if ((err = der_length_integer(data, &z)) != CRYPT_OK) {
114         goto LBL_ERR;
115     }
116     break;
117
118     case LTC_ASN1_SHORT_INTEGER:
119         z = inlen;
120         if ((err = der_decode_short_integer(in + x, z, data)) != CRYPT_OK) {
121             if (!ordered) { continue; }
122             goto LBL_ERR;
123         }
124         if ((err = der_length_short_integer(((unsigned long*)data)[0], &z)) != CRYPT_OK) {
125             goto LBL_ERR;
126         }
127
128         break;
129
130     case LTC_ASN1_BIT_STRING:
131         z = inlen;
132         if ((err = der_decode_bit_string(in + x, z, data, &size)) != CRYPT_OK) {
133             if (!ordered) { continue; }
134             goto LBL_ERR;
135         }
136         list[i].size = size;
137         if ((err = der_length_bit_string(size, &z)) != CRYPT_OK) {
138             goto LBL_ERR;
139         }
140         break;
141
142     case LTC_ASN1_OCTET_STRING:
143         z = inlen;
144         if ((err = der_decode_octet_string(in + x, z, data, &size)) != CRYPT_OK) {
145             if (!ordered) { continue; }
146             goto LBL_ERR;
147         }
148         list[i].size = size;
149         if ((err = der_length_octet_string(size, &z)) != CRYPT_OK) {
150             goto LBL_ERR;
151         }
152         break;
153
154     case LTC_ASN1_NULL:
155         if (inlen < 2 || in[x] != 0x05 || in[x+1] != 0x00) {
156             if (!ordered) { continue; }
157             err = CRYPT_INVALID_PACKET;
158             goto LBL_ERR;
159         }
160         z = 2;
161         break;
162
163     case LTC_ASN1_OBJECT_IDENTIFIER:
164         z = inlen;
165         if ((err = der_decode_object_identifier(in + x, z, data, &size)) != CRYPT_OK) {
166             if (!ordered) { continue; }
167             goto LBL_ERR;
168         }
169         list[i].size = size;
170         if ((err = der_length_object_identifier(data, size, &z)) != CRYPT_OK) {
171             goto LBL_ERR;
172         }
173         break;
174
175     case LTC_ASN1_IA5_STRING:
176         z = inlen;
177         if ((err = der_decode_ia5_string(in + x, z, data, &size)) != CRYPT_OK) {
```

```
178         if (!ordered) { continue; }
179         goto LBL_ERR;
180     }
181     list[i].size = size;
182     if ((err = der_length_ia5_string(data, size, &z)) != CRYPT_OK) {
183         goto LBL_ERR;
184     }
185     break;
186
187
188     case LTC_ASN1_PRINTABLE_STRING:
189         z = inlen;
190         if ((err = der_decode_printable_string(in + x, z, data, &size)) != CRYPT_OK) {
191             if (!ordered) { continue; }
192             goto LBL_ERR;
193         }
194         list[i].size = size;
195         if ((err = der_length_printable_string(data, size, &z)) != CRYPT_OK) {
196             goto LBL_ERR;
197         }
198         break;
199
200     case LTC_ASN1_UTCTIME:
201         z = inlen;
202         if ((err = der_decode_utctime(in + x, &z, data)) != CRYPT_OK) {
203             if (!ordered) { continue; }
204             goto LBL_ERR;
205         }
206         break;
207
208     case LTC_ASN1_SET:
209         z = inlen;
210         if ((err = der_decode_set(in + x, z, data, size)) != CRYPT_OK) {
211             if (!ordered) { continue; }
212             goto LBL_ERR;
213         }
214         if ((err = der_length_sequence(data, size, &z)) != CRYPT_OK) {
215             goto LBL_ERR;
216         }
217         break;
218
219     case LTC_ASN1_SETOF:
220     case LTC_ASN1_SEQUENCE:
221         /* detect if we have the right type */
222         if ((type == LTC_ASN1_SETOF && (in[x] & 0x3F) != 0x31) || (type == LTC_ASN1_SEQUENCE &&
223             err = CRYPT_INVALID_PACKET;
224             goto LBL_ERR;
225         }
226
227         z = inlen;
228         if ((err = der_decode_sequence(in + x, z, data, size)) != CRYPT_OK) {
229             if (!ordered) { continue; }
230             goto LBL_ERR;
231         }
232         if ((err = der_length_sequence(data, size, &z)) != CRYPT_OK) {
233             goto LBL_ERR;
234         }
235         break;
236
237
238     case LTC_ASN1_CHOICE:
239         z = inlen;
240         if ((err = der_decode_choice(in + x, &z, data, size)) != CRYPT_OK) {
241             if (!ordered) { continue; }
242             goto LBL_ERR;
243         }
244         break;
```

```
245
246         default:
247             err = CRYPT_INVALID_ARG;
248             goto LBL_ERR;
249     }
250     x           += z;
251     inlen       -= z;
252     list[i].used = 1;
253     if (!ordered) {
254         /* restart the decoder */
255         i = -1;
256     }
257 }
258
259 for (i = 0; i < outlen; i++) {
260     if (list[i].used == 0) {
261         err = CRYPT_INVALID_PACKET;
262         goto LBL_ERR;
263     }
264 }
265 err = CRYPT_OK;
266
267 LBL_ERR:
268     return err;
269 }
```

## 5.229 pk/asn1/der/sequence/der\_decode\_sequence\_flexi.c File Reference

### 5.229.1 Detailed Description

ASN.1 DER, decode an array of ASN.1 types with a flexi parser, Tom St Denis.

Definition in file [der\\_decode\\_sequence\\_flexi.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `der_decode_sequence_flexi.c`:

### Functions

- static unsigned long [fetch\\_length](#) (const unsigned char \**in*, unsigned long *inlen*)
- int [der\\_decode\\_sequence\\_flexi](#) (const unsigned char \**in*, unsigned long \**inlen*, ltc\_asn1\_list \*\**out*)

*ASN.1 DER Flexi(ble) decoder will decode arbitrary DER packets and create a linked list of the decoded elements.*

### 5.229.2 Function Documentation

#### 5.229.2.1 int der\_decode\_sequence\_flexi (const unsigned char \* *in*, unsigned long \* *inlen*, ltc\_asn1\_list \*\* *out*)

ASN.1 DER Flexi(ble) decoder will decode arbitrary DER packets and create a linked list of the decoded elements.

#### Parameters:

*in* The input buffer

*inlen* [in/out] The length of the input buffer and on output the amount of decoded data

*out* [out] A pointer to the linked list

#### Returns:

CRYPT\_OK on success.

Definition at line 63 of file `der_decode_sequence_flexi.c`.

References `CRYPT_INVALID_PACKET`, `CRYPT_MEM`, `CRYPT_OK`, `der_decode_bit_string()`, `der_decode_boolean()`, `der_decode_ia5_string()`, `der_decode_integer()`, `der_decode_object_identifier()`, `der_decode_octet_string()`, `der_decode_printable_string()`, `der_decode_utctime()`, `der_length_bit_string()`, `der_length_boolean()`, `der_length_ia5_string()`, `der_length_integer()`, `der_length_object_identifier()`, `der_length_octet_string()`, `der_length_printable_string()`, `der_length_utctime()`, `der_sequence_free()`, `fetch_length()`, `len`, `LTC_ARGCHK`, `XALLOC`, `XFREE`, and `XREALLOC`.

```
64 {
65     ltc_asn1_list *l;
66     unsigned long err, type, len, totlen, x, y;
67     void          *realloc_tmp;
68
69     LTC_ARGCHK(in      != NULL);
70     LTC_ARGCHK(inlen  != NULL);
```

```

71 LTC_ARGCHK(out    != NULL);
72
73 l = NULL;
74 totlen = 0;
75
76 /* scan the input and and get lengths and what not */
77 while (*inlen) {
78     /* read the type byte */
79     type = *in;
80
81     /* fetch length */
82     len = fetch_length(in, *inlen);
83     if (len > *inlen) {
84         err = CRYPT_INVALID_PACKET;
85         goto error;
86     }
87
88     /* alloc new link */
89     if (l == NULL) {
90         l = XCALLOC(1, sizeof(*l));
91         if (l == NULL) {
92             err = CRYPT_MEM;
93             goto error;
94         }
95     } else {
96         l->next = XCALLOC(1, sizeof(*l));
97         if (l->next == NULL) {
98             err = CRYPT_MEM;
99             goto error;
100         }
101         l->next->prev = l;
102         l = l->next;
103     }
104
105     /* now switch on type */
106     switch (type) {
107         case 0x01: /* BOOLEAN */
108             l->type = LTC_ASN1_BOOLEAN;
109             l->size = 1;
110             l->data = XCALLOC(1, sizeof(int));
111
112             if ((err = der_decode_boolean(in, *inlen, l->data)) != CRYPT_OK) {
113                 goto error;
114             }
115
116             if ((err = der_length_boolean(&len)) != CRYPT_OK) {
117                 goto error;
118             }
119             break;
120
121         case 0x02: /* INTEGER */
122             /* init field */
123             l->type = LTC_ASN1_INTEGER;
124             l->size = 1;
125             if ((err = mp_init(&l->data)) != CRYPT_OK) {
126                 goto error;
127             }
128
129             /* decode field */
130             if ((err = der_decode_integer(in, *inlen, l->data)) != CRYPT_OK) {
131                 goto error;
132             }
133
134             /* calc length of object */
135             if ((err = der_length_integer(l->data, &len)) != CRYPT_OK) {
136                 goto error;
137             }

```



```
138         break;
139
140     case 0x03: /* BIT */
141         /* init field */
142         l->type = LTC_ASN1_BIT_STRING;
143         l->size = len * 8; /* *8 because we store decoded bits one per char and they are encoded 8
144
145         if ((l->data = XCALLOC(1, l->size)) == NULL) {
146             err = CRYPT_MEM;
147             goto error;
148         }
149
150         if ((err = der_decode_bit_string(in, *inlen, l->data, &l->size)) != CRYPT_OK) {
151             goto error;
152         }
153
154         if ((err = der_length_bit_string(l->size, &len)) != CRYPT_OK) {
155             goto error;
156         }
157         break;
158
159     case 0x04: /* OCTET */
160
161         /* init field */
162         l->type = LTC_ASN1_OCTET_STRING;
163         l->size = len;
164
165         if ((l->data = XCALLOC(1, l->size)) == NULL) {
166             err = CRYPT_MEM;
167             goto error;
168         }
169
170         if ((err = der_decode_octet_string(in, *inlen, l->data, &l->size)) != CRYPT_OK) {
171             goto error;
172         }
173
174         if ((err = der_length_octet_string(l->size, &len)) != CRYPT_OK) {
175             goto error;
176         }
177         break;
178
179     case 0x05: /* NULL */
180
181         /* valid NULL is 0x05 0x00 */
182         if (in[0] != 0x05 || in[1] != 0x00) {
183             err = CRYPT_INVALID_PACKET;
184             goto error;
185         }
186
187         /* simple to store ;- ) */
188         l->type = LTC_ASN1_NULL;
189         l->data = NULL;
190         l->size = 0;
191         len = 2;
192
193         break;
194
195     case 0x06: /* OID */
196
197         /* init field */
198         l->type = LTC_ASN1_OBJECT_IDENTIFIER;
199         l->size = len;
200
201         if ((l->data = XCALLOC(len, sizeof(unsigned long))) == NULL) {
202             err = CRYPT_MEM;
203             goto error;
204         }
```

```
205
206     if ((err = der_decode_object_identifier(in, *inlen, l->data, &l->size)) != CRYPT_OK) {
207         goto error;
208     }
209
210     if ((err = der_length_object_identifier(l->data, l->size, &len)) != CRYPT_OK) {
211         goto error;
212     }
213
214     /* resize it to save a bunch of mem */
215     if ((realloc_tmp = XREALLOC(l->data, l->size * sizeof(unsigned long))) == NULL) {
216         /* out of heap but this is not an error */
217         break;
218     }
219     l->data = realloc_tmp;
220     break;
221
222
223     case 0x13: /* PRINTABLE */
224
225         /* init field */
226         l->type = LTC_ASN1_PRINTABLE_STRING;
227         l->size = len;
228
229         if ((l->data = XCALLOC(1, l->size)) == NULL) {
230             err = CRYPT_MEM;
231             goto error;
232         }
233
234         if ((err = der_decode_printable_string(in, *inlen, l->data, &l->size)) != CRYPT_OK) {
235             goto error;
236         }
237
238         if ((err = der_length_printable_string(l->data, l->size, &len)) != CRYPT_OK) {
239             goto error;
240         }
241         break;
242
243     case 0x16: /* IA5 */
244
245         /* init field */
246         l->type = LTC_ASN1_IA5_STRING;
247         l->size = len;
248
249         if ((l->data = XCALLOC(1, l->size)) == NULL) {
250             err = CRYPT_MEM;
251             goto error;
252         }
253
254         if ((err = der_decode_ia5_string(in, *inlen, l->data, &l->size)) != CRYPT_OK) {
255             goto error;
256         }
257
258         if ((err = der_length_ia5_string(l->data, l->size, &len)) != CRYPT_OK) {
259             goto error;
260         }
261         break;
262
263     case 0x17: /* UTC TIME */
264
265         /* init field */
266         l->type = LTC_ASN1_UTCTIME;
267         l->size = 1;
268
269         if ((l->data = XCALLOC(1, sizeof(ltc_utctime))) == NULL) {
270             err = CRYPT_MEM;
271             goto error;
```

```

272     }
273
274     len = *inlen;
275     if ((err = der_decode_utctime(in, &len, l->data)) != CRYPT_OK) {
276         goto error;
277     }
278
279     if ((err = der_length_utctime(l->data, &len)) != CRYPT_OK) {
280         goto error;
281     }
282     break;
283
284     case 0x30: /* SEQUENCE */
285     case 0x31: /* SET */
286
287         /* init field */
288         l->type = (type == 0x30) ? LTC_ASN1_SEQUENCE : LTC_ASN1_SET;
289
290         /* we have to decode the SEQUENCE header and get it's length */
291
292         /* move past type */
293         ++in; --(*inlen);
294
295         /* read length byte */
296         x = *in++; --(*inlen);
297
298         /* smallest SEQUENCE/SET header */
299         y = 2;
300
301         /* now if it's > 127 the next bytes are the length of the length */
302         if (x > 128) {
303             x      &= 0x7F;
304             in      += x;
305             *inlen -= x;
306
307             /* update sequence header len */
308             y      += x;
309         }
310
311         /* Sequence elements go as child */
312         len = len - y;
313         if ((err = der_decode_sequence_flexi(in, &len, &(l->child))) != CRYPT_OK) {
314             goto error;
315         }
316
317         /* len update */
318         totlen += y;
319
320         /* link them up y0 */
321         l->child->parent = l;
322
323         break;
324     default:
325         /* invalid byte ... this is a soft error */
326         /* remove link */
327         l      = l->prev;
328         XFREE(l->next);
329         l->next = NULL;
330         goto outside;
331     }
332
333     /* advance pointers */
334     totlen += len;
335     in      += len;
336     *inlen -= len;
337 }
338

```

```

339 outside:
340
341     /* rewind l please */
342     while (l->prev != NULL || l->parent != NULL) {
343         if (l->parent != NULL) {
344             l = l->parent;
345         } else {
346             l = l->prev;
347         }
348     }
349
350     /* return */
351     *out    = l;
352     *inlen = totlen;
353     return CRYPT_OK;
354
355 error:
356     /* free list */
357     der_sequence_free(l);
358
359     return err;
360 }

```

Here is the call graph for this function:

#### 5.229.2.2 static unsigned long fetch\_length (const unsigned char \* *in*, unsigned long *inlen*) [static]

Definition at line 20 of file der\_decode\_sequence\_flexi.c.

Referenced by der\_decode\_sequence\_flexi().

```

21 {
22     unsigned long x, y, z;
23
24     y = 0;
25
26     /* skip type and read len */
27     if (inlen < 2) {
28         return 0xFFFFFFFF;
29     }
30     ++in; ++y;
31
32     /* read len */
33     x = *in++; ++y;
34
35     /* <128 means literal */
36     if (x < 128) {
37         return x+y;
38     }
39     x      &= 0x7F; /* the lower 7 bits are the length of the length */
40     inlen -= 2;
41
42     /* len means len of len! */
43     if (x == 0 || x > 4 || x > inlen) {
44         return 0xFFFFFFFF;
45     }
46
47     y += x;
48     z = 0;
49     while (x-- > 0) {
50         z = (z<<8) | ((unsigned long)*in);
51         ++in;
52     }

```

```
53     return z+y;
54 }
```

## 5.230 pk/asn1/der/sequence/der\_decode\_sequence\_multi.c File Reference

### 5.230.1 Detailed Description

ASN.1 DER, decode a SEQUENCE, Tom St Denis.

Definition in file [der\\_decode\\_sequence\\_multi.c](#).

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for `der_decode_sequence_multi.c`:

### Functions

- `int der_decode_sequence_multi` (const unsigned char \**in*, unsigned long *inlen*,...)

*Decode a SEQUENCE type using a VA list.*

### 5.230.2 Function Documentation

#### 5.230.2.1 `int der_decode_sequence_multi` (const unsigned char \* *in*, unsigned long *inlen*, ...)

Decode a SEQUENCE type using a VA list.

#### Parameters:

*in* Input buffer

*inlen* Length of input in octets

#### Remarks:

<...> is of the form <type, size, data> (int, unsigned long, void\*)

#### Returns:

CRYPT\_OK on success

Definition at line 29 of file `der_decode_sequence_multi.c`.

References `CRYPT_INVALID_ARG`, and `LTC_ARGCHK`.

Referenced by `dsa_import()`, `dsa_verify_hash()`, `ecc_import()`, and `ecc_verify_hash()`.

```
30 {
31     int            err, type;
32     unsigned long  size, x;
33     void           *data;
34     va_list        args;
35     ltc_asn1_list  *list;
36
37     LTC_ARGCHK(in != NULL);
38
39     /* get size of output that will be required */
40     va_start(args, inlen);
41     x = 0;
42     for (;;) {
```

```
43     type = va_arg(args, int);
44     size = va_arg(args, unsigned long);
45     data = va_arg(args, void*);
46
47     if (type == LTC_ASN1_EOL) {
48         break;
49     }
50
51     switch (type) {
52         case LTC_ASN1_BOOLEAN:
53         case LTC_ASN1_INTEGER:
54         case LTC_ASN1_SHORT_INTEGER:
55         case LTC_ASN1_BIT_STRING:
56         case LTC_ASN1_OCTET_STRING:
57         case LTC_ASN1_NULL:
58         case LTC_ASN1_OBJECT_IDENTIFIER:
59         case LTC_ASN1_IA5_STRING:
60         case LTC_ASN1_PRINTABLE_STRING:
61         case LTC_ASN1_UTCTIME:
62         case LTC_ASN1_SET:
63         case LTC_ASN1_SETOF:
64         case LTC_ASN1_SEQUENCE:
65         case LTC_ASN1_CHOICE:
66             ++x;
67             break;
68
69         default:
70             va_end(args);
71             return CRYPT_INVALID_ARG;
72     }
73 }
74 va_end(args);
75
76 /* allocate structure for x elements */
77 if (x == 0) {
78     return CRYPT_NOP;
79 }
80
81 list = XCALLOC(sizeof(*list), x);
82 if (list == NULL) {
83     return CRYPT_MEM;
84 }
85
86 /* fill in the structure */
87 va_start(args, inlen);
88 x = 0;
89 for (;;) {
90     type = va_arg(args, int);
91     size = va_arg(args, unsigned long);
92     data = va_arg(args, void*);
93
94     if (type == LTC_ASN1_EOL) {
95         break;
96     }
97
98     switch (type) {
99         case LTC_ASN1_BOOLEAN:
100         case LTC_ASN1_INTEGER:
101         case LTC_ASN1_SHORT_INTEGER:
102         case LTC_ASN1_BIT_STRING:
103         case LTC_ASN1_OCTET_STRING:
104         case LTC_ASN1_NULL:
105         case LTC_ASN1_OBJECT_IDENTIFIER:
106         case LTC_ASN1_IA5_STRING:
107         case LTC_ASN1_PRINTABLE_STRING:
108         case LTC_ASN1_UTCTIME:
109         case LTC_ASN1_SEQUENCE:
```

```
110         case LTC_ASN1_SET:
111         case LTC_ASN1_SETOF:
112         case LTC_ASN1_CHOICE:
113             list[x].type = type;
114             list[x].size = size;
115             list[x++].data = data;
116             break;
117
118         default:
119             va_end(args);
120             err = CRYPT_INVALID_ARG;
121             goto LBL_ERR;
122     }
123 }
124 va_end(args);
125
126 err = der_decode_sequence(in, inlen, list, x);
127 LBL_ERR:
128     XFREE(list);
129     return err;
130 }
```



## 5.231 pk/asn1/der/sequence/der\_encode\_sequence\_ex.c File Reference

### 5.231.1 Detailed Description

ASN.1 DER, encode a SEQUENCE, Tom St Denis.

Definition in file [der\\_encode\\_sequence\\_ex.c](#).

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for der\_encode\_sequence\_ex.c:

### Functions

- [int der\\_encode\\_sequence\\_ex](#) (ltc\_asn1\_list \*list, unsigned long inlen, unsigned char \*out, unsigned long \*outlen, int type\_of)  
*Encode a SEQUENCE.*

### 5.231.2 Function Documentation

#### 5.231.2.1 int der\_encode\_sequence\_ex (ltc\_asn1\_list \* list, unsigned long inlen, unsigned char \* out, unsigned long \* outlen, int type\_of)

Encode a SEQUENCE.

#### Parameters:

*list* The list of items to encode

*inlen* The number of items in the list

*out* [out] The destination

*outlen* [in/out] The size of the output

*type\_of* LTC\_ASN1\_SEQUENCE or LTC\_ASN1\_SET/LTC\_ASN1\_SETOF

#### Returns:

CRYPT\_OK on success

Definition at line 31 of file der\_encode\_sequence\_ex.c.

References LTC\_ARGCHK.

```
33 {
34     int            err, type;
35     unsigned long size, x, y, z, i;
36     void           *data;
37
38     LTC_ARGCHK(list    != NULL);
39     LTC_ARGCHK(out     != NULL);
40     LTC_ARGCHK(outlen  != NULL);
41
42     /* get size of output that will be required */
43     y = 0;
```

```
44     for (i = 0; i < inlen; i++) {
45         type = list[i].type;
46         size = list[i].size;
47         data = list[i].data;
48
49         if (type == LTC_ASN1_EOL) {
50             break;
51         }
52
53         switch (type) {
54             case LTC_ASN1_BOOLEAN:
55                 if ((err = der_length_boolean(&x)) != CRYPT_OK) {
56                     goto LBL_ERR;
57                 }
58                 y += x;
59                 break;
60
61             case LTC_ASN1_INTEGER:
62                 if ((err = der_length_integer(data, &x)) != CRYPT_OK) {
63                     goto LBL_ERR;
64                 }
65                 y += x;
66                 break;
67
68             case LTC_ASN1_SHORT_INTEGER:
69                 if ((err = der_length_short_integer(*((unsigned long*)data), &x)) != CRYPT_OK) {
70                     goto LBL_ERR;
71                 }
72                 y += x;
73                 break;
74
75             case LTC_ASN1_BIT_STRING:
76                 if ((err = der_length_bit_string(size, &x)) != CRYPT_OK) {
77                     goto LBL_ERR;
78                 }
79                 y += x;
80                 break;
81
82             case LTC_ASN1_OCTET_STRING:
83                 if ((err = der_length_octet_string(size, &x)) != CRYPT_OK) {
84                     goto LBL_ERR;
85                 }
86                 y += x;
87                 break;
88
89             case LTC_ASN1_NULL:
90                 y += 2;
91                 break;
92
93             case LTC_ASN1_OBJECT_IDENTIFIER:
94                 if ((err = der_length_object_identifier(data, size, &x)) != CRYPT_OK) {
95                     goto LBL_ERR;
96                 }
97                 y += x;
98                 break;
99
100             case LTC_ASN1_IA5_STRING:
101                 if ((err = der_length_ia5_string(data, size, &x)) != CRYPT_OK) {
102                     goto LBL_ERR;
103                 }
104                 y += x;
105                 break;
106
107             case LTC_ASN1_PRINTABLE_STRING:
108                 if ((err = der_length_printable_string(data, size, &x)) != CRYPT_OK) {
109                     goto LBL_ERR;
110                 }
111         }
```

```
111         y += x;
112         break;
113
114     case LTC_ASN1_UTCTIME:
115         if ((err = der_length_utctime(data, &x)) != CRYPT_OK) {
116             goto LBL_ERR;
117         }
118         y += x;
119         break;
120
121     case LTC_ASN1_SET:
122     case LTC_ASN1_SETOF:
123     case LTC_ASN1_SEQUENCE:
124         if ((err = der_length_sequence(data, size, &x)) != CRYPT_OK) {
125             goto LBL_ERR;
126         }
127         y += x;
128         break;
129
130     default:
131         err = CRYPT_INVALID_ARG;
132         goto LBL_ERR;
133     }
134 }
135
136 /* calc header size */
137 z = y;
138 if (y < 128) {
139     y += 2;
140 } else if (y < 256) {
141     /* 0x30 0x81 LL */
142     y += 3;
143 } else if (y < 65536UL) {
144     /* 0x30 0x82 LL LL */
145     y += 4;
146 } else if (y < 16777216UL) {
147     /* 0x30 0x83 LL LL LL */
148     y += 5;
149 } else {
150     err = CRYPT_INVALID_ARG;
151     goto LBL_ERR;
152 }
153
154 /* too big ? */
155 if (*outlen < y) {
156     *outlen = y;
157     err = CRYPT_BUFFER_OVERFLOW;
158     goto LBL_ERR;
159 }
160
161 /* store header */
162 x = 0;
163 out[x++] = (type_of == LTC_ASN1_SEQUENCE) ? 0x30 : 0x31;
164
165 if (z < 128) {
166     out[x++] = z;
167 } else if (z < 256) {
168     out[x++] = 0x81;
169     out[x++] = z;
170 } else if (z < 65536UL) {
171     out[x++] = 0x82;
172     out[x++] = (z >> 8UL) & 255;
173     out[x++] = z & 255;
174 } else if (z < 16777216UL) {
175     out[x++] = 0x83;
176     out[x++] = (z >> 16UL) & 255;
177     out[x++] = (z >> 8UL) & 255;
```

```
178     out[x++] = z&255;
179 }
180
181 /* store data */
182 *outlen -= x;
183 for (i = 0; i < inlen; i++) {
184     type = list[i].type;
185     size = list[i].size;
186     data = list[i].data;
187
188     if (type == LTC_ASN1_EOL) {
189         break;
190     }
191
192     switch (type) {
193     case LTC_ASN1_BOOLEAN:
194         z = *outlen;
195         if ((err = der_encode_boolean(*((int *)data), out + x, &z)) != CRYPT_OK) {
196             goto LBL_ERR;
197         }
198         x += z;
199         *outlen -= z;
200         break;
201
202     case LTC_ASN1_INTEGER:
203         z = *outlen;
204         if ((err = der_encode_integer(data, out + x, &z)) != CRYPT_OK) {
205             goto LBL_ERR;
206         }
207         x += z;
208         *outlen -= z;
209         break;
210
211     case LTC_ASN1_SHORT_INTEGER:
212         z = *outlen;
213         if ((err = der_encode_short_integer(*((unsigned long*)data), out + x, &z)) != CRYPT_OK) {
214             goto LBL_ERR;
215         }
216         x += z;
217         *outlen -= z;
218         break;
219
220     case LTC_ASN1_BIT_STRING:
221         z = *outlen;
222         if ((err = der_encode_bit_string(data, size, out + x, &z)) != CRYPT_OK) {
223             goto LBL_ERR;
224         }
225         x += z;
226         *outlen -= z;
227         break;
228
229     case LTC_ASN1_OCTET_STRING:
230         z = *outlen;
231         if ((err = der_encode_octet_string(data, size, out + x, &z)) != CRYPT_OK) {
232             goto LBL_ERR;
233         }
234         x += z;
235         *outlen -= z;
236         break;
237
238     case LTC_ASN1_NULL:
239         out[x++] = 0x05;
240         out[x++] = 0x00;
241         *outlen -= 2;
242         break;
243
244     case LTC_ASN1_OBJECT_IDENTIFIER:
```

```
245         z = *outlen;
246         if ((err = der_encode_object_identifier(data, size, out + x, &z)) != CRYPT_OK) {
247             goto LBL_ERR;
248         }
249         x += z;
250         *outlen -= z;
251         break;
252
253     case LTC_ASN1_IA5_STRING:
254         z = *outlen;
255         if ((err = der_encode_ia5_string(data, size, out + x, &z)) != CRYPT_OK) {
256             goto LBL_ERR;
257         }
258         x += z;
259         *outlen -= z;
260         break;
261
262     case LTC_ASN1_PRINTABLE_STRING:
263         z = *outlen;
264         if ((err = der_encode_printable_string(data, size, out + x, &z)) != CRYPT_OK) {
265             goto LBL_ERR;
266         }
267         x += z;
268         *outlen -= z;
269         break;
270
271     case LTC_ASN1_UTCTIME:
272         z = *outlen;
273         if ((err = der_encode_utctime(data, out + x, &z)) != CRYPT_OK) {
274             goto LBL_ERR;
275         }
276         x += z;
277         *outlen -= z;
278         break;
279
280     case LTC_ASN1_SET:
281         z = *outlen;
282         if ((err = der_encode_set(data, size, out + x, &z)) != CRYPT_OK) {
283             goto LBL_ERR;
284         }
285         x += z;
286         *outlen -= z;
287         break;
288
289     case LTC_ASN1_SETOF:
290         z = *outlen;
291         if ((err = der_encode_setof(data, size, out + x, &z)) != CRYPT_OK) {
292             goto LBL_ERR;
293         }
294         x += z;
295         *outlen -= z;
296         break;
297
298     case LTC_ASN1_SEQUENCE:
299         z = *outlen;
300         if ((err = der_encode_sequence_ex(data, size, out + x, &z, type)) != CRYPT_OK) {
301             goto LBL_ERR;
302         }
303         x += z;
304         *outlen -= z;
305         break;
306
307     default:
308         err = CRYPT_INVALID_ARG;
309         goto LBL_ERR;
310 }
311 }
```

```
312     *outlen = x;
313     err = CRYPT_OK;
314
315 LBL_ERR:
316     return err;
317 }
```

## 5.232 pk/asn1/der/sequence/der\_encode\_sequence\_multi.c File Reference

### 5.232.1 Detailed Description

ASN.1 DER, encode a SEQUENCE, Tom St Denis.

Definition in file [der\\_encode\\_sequence\\_multi.c](#).

```
#include "tomcrypt.h"
```

```
#include <stdarg.h>
```

Include dependency graph for `der_encode_sequence_multi.c`:

### Functions

- `int der_encode_sequence_multi` (unsigned char \*out, unsigned long \*outlen,...)  
*Encode a SEQUENCE type using a VA list.*

### 5.232.2 Function Documentation

#### 5.232.2.1 int der\_encode\_sequence\_multi (unsigned char \* out, unsigned long \* outlen, ...)

Encode a SEQUENCE type using a VA list.

#### Parameters:

*out* [out] Destination for data

*outlen* [in/out] Length of buffer and resulting length of output

#### Remarks:

<...> is of the form <type, size, data> (int, unsigned long, void\*)

#### Returns:

CRYPT\_OK on success

Definition at line 29 of file `der_encode_sequence_multi.c`.

References `CRYPT_INVALID_ARG`, and `LTC_ARGCHK`.

Referenced by `dsa_export()`, `dsa_sign_hash()`, `ecc_export()`, and `rsa_export()`.

```
30 {
31     int            err, type;
32     unsigned long  size, x;
33     void           *data;
34     va_list        args;
35     ltc_asn1_list  *list;
36
37     LTC_ARGCHK(out != NULL);
38     LTC_ARGCHK(outlen != NULL);
39
40     /* get size of output that will be required */
41     va_start(args, outlen);
42     x = 0;
```

```
43     for (;;) {
44         type = va_arg(args, int);
45         size = va_arg(args, unsigned long);
46         data = va_arg(args, void*);
47
48         if (type == LTC_ASN1_EOL) {
49             break;
50         }
51
52         switch (type) {
53             case LTC_ASN1_BOOLEAN:
54             case LTC_ASN1_INTEGER:
55             case LTC_ASN1_SHORT_INTEGER:
56             case LTC_ASN1_BIT_STRING:
57             case LTC_ASN1_OCTET_STRING:
58             case LTC_ASN1_NULL:
59             case LTC_ASN1_OBJECT_IDENTIFIER:
60             case LTC_ASN1_IA5_STRING:
61             case LTC_ASN1_PRINTABLE_STRING:
62             case LTC_ASN1_UTCTIME:
63             case LTC_ASN1_SEQUENCE:
64             case LTC_ASN1_SET:
65             case LTC_ASN1_SETOF:
66                 ++x;
67                 break;
68
69             default:
70                 va_end(args);
71                 return CRYPT_INVALID_ARG;
72         }
73     }
74     va_end(args);
75
76     /* allocate structure for x elements */
77     if (x == 0) {
78         return CRYPT_NOP;
79     }
80
81     list = XCALLOC(sizeof(*list), x);
82     if (list == NULL) {
83         return CRYPT_MEM;
84     }
85
86     /* fill in the structure */
87     va_start(args, outlen);
88     x = 0;
89     for (;;) {
90         type = va_arg(args, int);
91         size = va_arg(args, unsigned long);
92         data = va_arg(args, void*);
93
94         if (type == LTC_ASN1_EOL) {
95             break;
96         }
97
98         switch (type) {
99             case LTC_ASN1_BOOLEAN:
100             case LTC_ASN1_INTEGER:
101             case LTC_ASN1_SHORT_INTEGER:
102             case LTC_ASN1_BIT_STRING:
103             case LTC_ASN1_OCTET_STRING:
104             case LTC_ASN1_NULL:
105             case LTC_ASN1_OBJECT_IDENTIFIER:
106             case LTC_ASN1_IA5_STRING:
107             case LTC_ASN1_PRINTABLE_STRING:
108             case LTC_ASN1_UTCTIME:
109             case LTC_ASN1_SEQUENCE:
```



```
110         case LTC_ASN1_SET:
111         case LTC_ASN1_SETOF:
112             list[x].type = type;
113             list[x].size = size;
114             list[x++].data = data;
115             break;
116
117         default:
118             va_end(args);
119             err = CRYPT_INVALID_ARG;
120             goto LBL_ERR;
121     }
122 }
123 va_end(args);
124
125 err = der_encode_sequence(list, x, out, outlen);
126 LBL_ERR:
127 XFREE(list);
128 return err;
129 }
```

## 5.233 pk/asn1/der/sequence/der\_length\_sequence.c File Reference

### 5.233.1 Detailed Description

ASN.1 DER, length a SEQUENCE, Tom St Denis.

Definition in file [der\\_length\\_sequence.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_length\_sequence.c:

### Functions

- [int der\\_length\\_sequence](#) (ltc\_asn1\_list \*list, unsigned long inlen, unsigned long \*outlen)  
*Get the length of a DER sequence.*

### 5.233.2 Function Documentation

#### 5.233.2.1 int der\_length\_sequence (ltc\_asn1\_list \* list, unsigned long inlen, unsigned long \* outlen)

Get the length of a DER sequence.

#### Parameters:

- list* The sequences of items in the SEQUENCE
- inlen* The number of items
- outlen* [out] The length required in octets to store it

#### Returns:

CRYPT\_OK on success

Definition at line 27 of file der\_length\_sequence.c.

References LTC\_ARGCHK.

```
29 {
30     int            err, type;
31     unsigned long size, x, y, z, i;
32     void           *data;
33
34     LTC_ARGCHK(list != NULL);
35     LTC_ARGCHK(outlen != NULL);
36
37     /* get size of output that will be required */
38     y = 0;
39     for (i = 0; i < inlen; i++) {
40         type = list[i].type;
41         size = list[i].size;
42         data = list[i].data;
43
44         if (type == LTC_ASN1_EOL) {
45             break;
46         }
47
48         switch (type) {
```

```
49         case LTC_ASN1_BOOLEAN:
50             if ((err = der_length_boolean(&x)) != CRYPT_OK) {
51                 goto LBL_ERR;
52             }
53             y += x;
54             break;
55
56         case LTC_ASN1_INTEGER:
57             if ((err = der_length_integer(data, &x)) != CRYPT_OK) {
58                 goto LBL_ERR;
59             }
60             y += x;
61             break;
62
63         case LTC_ASN1_SHORT_INTEGER:
64             if ((err = der_length_short_integer*((unsigned long *)data), &x)) != CRYPT_OK) {
65                 goto LBL_ERR;
66             }
67             y += x;
68             break;
69
70         case LTC_ASN1_BIT_STRING:
71             if ((err = der_length_bit_string(size, &x)) != CRYPT_OK) {
72                 goto LBL_ERR;
73             }
74             y += x;
75             break;
76
77         case LTC_ASN1_OCTET_STRING:
78             if ((err = der_length_octet_string(size, &x)) != CRYPT_OK) {
79                 goto LBL_ERR;
80             }
81             y += x;
82             break;
83
84         case LTC_ASN1_NULL:
85             y += 2;
86             break;
87
88         case LTC_ASN1_OBJECT_IDENTIFIER:
89             if ((err = der_length_object_identifier(data, size, &x)) != CRYPT_OK) {
90                 goto LBL_ERR;
91             }
92             y += x;
93             break;
94
95         case LTC_ASN1_IA5_STRING:
96             if ((err = der_length_ia5_string(data, size, &x)) != CRYPT_OK) {
97                 goto LBL_ERR;
98             }
99             y += x;
100             break;
101
102         case LTC_ASN1_PRINTABLE_STRING:
103             if ((err = der_length_printable_string(data, size, &x)) != CRYPT_OK) {
104                 goto LBL_ERR;
105             }
106             y += x;
107             break;
108
109         case LTC_ASN1_UTCTIME:
110             if ((err = der_length_utctime(data, &x)) != CRYPT_OK) {
111                 goto LBL_ERR;
112             }
113             y += x;
114             break;
115
```

```
116         case LTC_ASN1_SET:
117         case LTC_ASN1_SETOF:
118         case LTC_ASN1_SEQUENCE:
119             if ((err = der_length_sequence(data, size, &x)) != CRYPT_OK) {
120                 goto LBL_ERR;
121             }
122             y += x;
123             break;
124
125
126         default:
127             err = CRYPT_INVALID_ARG;
128             goto LBL_ERR;
129     }
130 }
131
132 /* calc header size */
133 z = y;
134 if (y < 128) {
135     y += 2;
136 } else if (y < 256) {
137     /* 0x30 0x81 LL */
138     y += 3;
139 } else if (y < 65536UL) {
140     /* 0x30 0x82 LL LL */
141     y += 4;
142 } else if (y < 16777216UL) {
143     /* 0x30 0x83 LL LL LL */
144     y += 5;
145 } else {
146     err = CRYPT_INVALID_ARG;
147     goto LBL_ERR;
148 }
149
150 /* store size */
151 *outlen = y;
152 err = CRYPT_OK;
153
154 LBL_ERR:
155     return err;
156 }
```

## 5.234 pk/asn1/der/sequence/der\_sequence\_free.c File Reference

### 5.234.1 Detailed Description

ASN.1 DER, free's a structure allocated by [der\\_decode\\_sequence\\_flexi\(\)](#), Tom St Denis.

Definition in file [der\\_sequence\\_free.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [der\\_sequence\\_free.c](#):

### Functions

- void [der\\_sequence\\_free](#) (ltc\_asn1\_list \**in*)  
*Free memory allocated by [der\\_decode\\_sequence\\_flexi\(\)](#).*

### 5.234.2 Function Documentation

#### 5.234.2.1 void der\_sequence\_free (ltc\_asn1\_list \**in*)

Free memory allocated by [der\\_decode\\_sequence\\_flexi\(\)](#).

#### Parameters:

*in* The list to free

Definition at line 24 of file [der\\_sequence\\_free.c](#).

References [XFREE](#).

Referenced by [der\\_decode\\_sequence\\_flexi\(\)](#).

```

25 {
26     ltc_asn1_list *l;
27
28     /* walk to the start of the chain */
29     while (in->prev != NULL || in->parent != NULL) {
30         if (in->parent != NULL) {
31             in = in->parent;
32         } else {
33             in = in->prev;
34         }
35     }
36
37     /* now walk the list and free stuff */
38     while (in != NULL) {
39         /* is there a child? */
40         if (in->child) {
41             /* disconnect */
42             in->child->parent = NULL;
43             der_sequence_free(in->child);
44         }
45
46         switch (in->type) {
47             case LTC_ASN1_SET:
48             case LTC_ASN1_SETOF:
49             case LTC_ASN1_SEQUENCE: break;
50             case LTC_ASN1_INTEGER : if (in->data != NULL) { mp_clear(in->data); } break;

```

```
51         default                : if (in->data != NULL) { XFREE(in->data); }
52     }
53
54     /* move to next and free current */
55     l = in->next;
56     free(in);
57     in = l;
58 }
59 }
```

## 5.235 pk/asn1/der/set/der\_encode\_set.c File Reference

### 5.235.1 Detailed Description

ASN.1 DER, Encode a SET, Tom St Denis.

Definition in file [der\\_encode\\_set.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `der_encode_set.c`:

### Functions

- static int [ltc\\_to\\_asn1](#) (int v)
- static int [qsort\\_helper](#) (const void \*a, const void \*b)
- int [der\\_encode\\_set](#) (ltc\_asn1\_list \*list, unsigned long inlen, unsigned char \*out, unsigned long \*outlen)

### 5.235.2 Function Documentation

#### 5.235.2.1 int der\_encode\_set (ltc\_asn1\_list \* list, unsigned long inlen, unsigned char \* out, unsigned long \* outlen)

Definition at line 66 of file `der_encode_set.c`.

References `CRYPT_MEM`, and `XCALLOC`.

```
68 {
69     ltc_asn1_list  *copy;
70     unsigned long  x;
71     int            err;
72
73     /* make copy of list */
74     copy = XCALLOC(inlen, sizeof(*copy));
75     if (copy == NULL) {
76         return CRYPT_MEM;
77     }
78
79     /* fill in used member with index so we can fully sort it */
80     for (x = 0; x < inlen; x++) {
81         copy[x] = list[x];
82         copy[x].used = x;
83     }
84
85     /* sort it by the "type" field */
86     XQSORT(copy, inlen, sizeof(*copy), &qsort_helper);
87
88     /* call der_encode_sequence_ex() */
89     err = der_encode_sequence_ex(copy, inlen, out, outlen, LTC_ASN1_SET);
90
91     /* free list */
92     XFREE(copy);
93
94     return err;
95 }
```

### 5.235.2.2 static int ltc\_to\_asn1 (int v) [static]

Definition at line 21 of file der\_encode\_set.c.

Referenced by qsort\_helper().

```
22 {
23     switch (v) {
24         case LTC_ASN1_BOOLEAN:           return 0x01;
25         case LTC_ASN1_INTEGER:           return 0x02;
26         case LTC_ASN1_SHORT_INTEGER:     return 0x03;
27         case LTC_ASN1_BIT_STRING:        return 0x04;
28         case LTC_ASN1_OCTET_STRING:      return 0x05;
29         case LTC_ASN1_NULL:              return 0x06;
30         case LTC_ASN1_OBJECT_IDENTIFIER: return 0x07;
31         case LTC_ASN1_PRINTABLE_STRING:  return 0x08;
32         case LTC_ASN1_IA5_STRING:        return 0x09;
33         case LTC_ASN1_UTCTIME:           return 0x0A;
34         case LTC_ASN1_SEQUENCE:          return 0x0B;
35         case LTC_ASN1_SET:                return 0x0C;
36         case LTC_ASN1_SETOF:             return 0x0D;
37         default: return -1;
38     }
39 }
```

### 5.235.2.3 static int qsort\_helper (const void \*a, const void \*b) [static]

Definition at line 42 of file der\_encode\_set.c.

References B, and ltc\_to\_asn1().

```
43 {
44     ltc_asn1_list *A = (ltc_asn1_list *)a, *B = (ltc_asn1_list *)b;
45     int r;
46
47     r = ltc_to_asn1(A->type) - ltc_to_asn1(B->type);
48
49     /* for QSORT the order is UNDEFINED if they are "equal" which means it is NOT DETERMINISTIC. So we
50     if (r == 0) {
51         /* their order in the original list now determines the position */
52         return A->used - B->used;
53     } else {
54         return r;
55     }
56 }
```

Here is the call graph for this function:



## 5.236 pk/asn1/der/set/der\_encode\_setof.c File Reference

### 5.236.1 Detailed Description

ASN.1 DER, Encode SET OF, Tom St Denis.

Definition in file [der\\_encode\\_setof.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_encode\_setof.c:

### Data Structures

- struct [edge](#)

### Functions

- static int [qsort\\_helper](#) (const void \*a, const void \*b)
- int [der\\_encode\\_setof](#) (ltc\_asn1\_list \*list, unsigned long inlen, unsigned char \*out, unsigned long \*outlen)

*Encode a SETOF structure.*

### 5.236.2 Function Documentation

#### 5.236.2.1 int der\_encode\_setof (ltc\_asn1\_list \* list, unsigned long inlen, unsigned char \* out, unsigned long \* outlen)

Encode a SETOF structure.

#### Parameters:

- list** The list of items to encode
- inlen** The number of items in the list
- out** [out] The destination
- outlen** [in/out] The size of the output

#### Returns:

CRYPT\_OK on success

Definition at line 61 of file der\_encode\_setof.c.

References CRYPT\_INVALID\_ARG.

```
63 {
64     unsigned long  x, y, z, hdrlen;
65     int            err;
66     struct edge    *edges;
67     unsigned char *ptr, *buf;
68
69     /* check that they're all the same type */
70     for (x = 1; x < inlen; x++) {
71         if (list[x].type != list[x-1].type) {
```

```
72         return CRYPT_INVALID_ARG;
73     }
74 }
75
76 /* alloc buffer to store copy of output */
77 buf = XCALLOC(1, *outlen);
78 if (buf == NULL) {
79     return CRYPT_MEM;
80 }
81
82 /* encode list */
83 if ((err = der_encode_sequence_ex(list, inlen, buf, outlen, LTC_ASN1_SETOF)) != CRYPT_OK) {
84     XFREE(buf);
85     return err;
86 }
87
88 /* allocate edges */
89 edges = XCALLOC(inlen, sizeof(*edges));
90 if (edges == NULL) {
91     XFREE(buf);
92     return CRYPT_MEM;
93 }
94
95 /* skip header */
96 ptr = buf + 1;
97
98 /* now skip length data */
99 x = *ptr++;
100 if (x >= 0x80) {
101     ptr += (x & 0x7F);
102 }
103
104 /* get the size of the static header */
105 hdrlen = ((unsigned long)ptr) - ((unsigned long)buf);
106
107
108 /* scan for edges */
109 x = 0;
110 while (ptr < (buf + *outlen)) {
111     /* store start */
112     edges[x].start = ptr;
113
114     /* skip type */
115     z = 1;
116
117     /* parse length */
118     y = ptr[z++];
119     if (y < 128) {
120         edges[x].size = y;
121     } else {
122         y &= 0x7F;
123         edges[x].size = 0;
124         while (y-- > 0) {
125             edges[x].size = (edges[x].size << 8) | ((unsigned long)ptr[z++]);
126         }
127     }
128
129     /* skip content */
130     edges[x].size += z;
131     ptr += edges[x].size;
132     ++x;
133 }
134
135 /* sort based on contents (using edges) */
136 XQSORT(edges, inlen, sizeof(*edges), &qsort_helper);
137
138 /* copy static header */
```

```

139     XMEMCPY(out, buf, hdrlen);
140
141     /* copy+sort using edges+indecies to output from buffer */
142     for (y = hdrlen, x = 0; x < inlen; x++) {
143         XMEMCPY(out+y, edges[x].start, edges[x].size);
144         y += edges[x].size;
145     }
146
147 #ifdef LTC_CLEAN_STACK
148     zeromem(buf, *outlen);
149 #endif
150
151     /* free buffers */
152     XFREE(edges);
153     XFREE(buf);
154
155     return CRYPT_OK;
156 }

```

### 5.236.2.2 static int qsort\_helper (const void \*a, const void \*b) [static]

Definition at line 25 of file der\_encode\_setof.c.

References B, MIN, edge::size, edge::start, and XMEMCMP.

```

26 {
27     struct edge  *A = (struct edge *)a, *B = (struct edge *)b;
28     int          r;
29     unsigned long x;
30
31     /* compare min length */
32     r = XMEMCMP(A->start, B->start, MIN(A->size, B->size));
33
34     if (r == 0 && A->size != B->size) {
35         if (A->size > B->size) {
36             for (x = B->size; x < A->size; x++) {
37                 if (A->start[x]) {
38                     return 1;
39                 }
40             }
41         } else {
42             for (x = A->size; x < B->size; x++) {
43                 if (B->start[x]) {
44                     return -1;
45                 }
46             }
47         }
48     }
49
50     return r;
51 }

```

## 5.237 pk/asn1/der/short\_integer/der\_decode\_short\_integer.c File Reference

### 5.237.1 Detailed Description

ASN.1 DER, decode an integer, Tom St Denis.

Definition in file [der\\_decode\\_short\\_integer.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `der_decode_short_integer.c`:

### Functions

- `int der_decode_short_integer` (const unsigned char \**in*, unsigned long *inlen*, unsigned long \**num*)  
*Read a short integer.*

### 5.237.2 Function Documentation

#### 5.237.2.1 `int der_decode_short_integer` (const unsigned char \* *in*, unsigned long *inlen*, unsigned long \* *num*)

Read a short integer.

#### Parameters:

- in* The DER encoded data
- inlen* Size of data
- num* [out] The integer to decode

#### Returns:

- CRYPT\_OK if successful

Definition at line 28 of file `der_decode_short_integer.c`.

References `CRYPT_INVALID_PACKET`, `CRYPT_OK`, `len`, and `LTC_ARGCHK`.

```
29 {
30     unsigned long len, x, y;
31
32     LTC_ARGCHK(num != NULL);
33     LTC_ARGCHK(in != NULL);
34
35     /* check length */
36     if (inlen < 2) {
37         return CRYPT_INVALID_PACKET;
38     }
39
40     /* check header */
41     x = 0;
42     if ((in[x++] & 0x1F) != 0x02) {
43         return CRYPT_INVALID_PACKET;
44     }
45 }
```

```
46  /* get the packet len */
47  len = in[x++];
48
49  if (x + len > inlen) {
50      return CRYPT_INVALID_PACKET;
51  }
52
53  /* read number */
54  y = 0;
55  while (len-- > 0) {
56      y = (y<<8) | (unsigned long)in[x++];
57  }
58  *num = y;
59
60  return CRYPT_OK;
61
62 }
```

## 5.238 pk/asn1/der/short\_integer/der\_encode\_short\_integer.c File Reference

### 5.238.1 Detailed Description

ASN.1 DER, encode an integer, Tom St Denis.

Definition in file [der\\_encode\\_short\\_integer.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [der\\_encode\\_short\\_integer.c](#):

### Functions

- [int der\\_encode\\_short\\_integer](#) (unsigned long num, unsigned char \*out, unsigned long \*outlen)  
*Store a short integer in the range  $(0, 2^{32}-1)$ .*

### 5.238.2 Function Documentation

#### 5.238.2.1 [int der\\_encode\\_short\\_integer](#) (unsigned long num, unsigned char \* out, unsigned long \* outlen)

Store a short integer in the range  $(0, 2^{32}-1)$ .

#### Parameters:

**num** The integer to encode

**out** [out] The destination for the DER encoded integers

**outlen** [in/out] The max size and resulting size of the DER encoded integers

#### Returns:

CRYPT\_OK if successful

Definition at line 28 of file [der\\_encode\\_short\\_integer.c](#).

References [CRYPT\\_BUFFER\\_OVERFLOW](#), [CRYPT\\_OK](#), [der\\_length\\_short\\_integer\(\)](#), [len](#), and [LTC\\_ARGCHK](#).

```
29 {
30     unsigned long len, x, y, z;
31     int          err;
32
33     LTC_ARGCHK(out != NULL);
34     LTC_ARGCHK(outlen != NULL);
35
36     /* force to 32 bits */
37     num &= 0xFFFFFFFFUL;
38
39     /* find out how big this will be */
40     if ((err = der_length_short_integer(num, &len)) != CRYPT_OK) {
41         return err;
42     }
43
44     if (*outlen < len) {
```

```
45     *outlen = len;
46     return CRYPT_BUFFER_OVERFLOW;
47 }
48
49 /* get len of output */
50 z = 0;
51 y = num;
52 while (y) {
53     ++z;
54     y >>= 8;
55 }
56
57 /* handle zero */
58 if (z == 0) {
59     z = 1;
60 }
61
62 /* see if msb is set */
63 z += (num & (1UL << ((z < 3) - 1))) ? 1 : 0;
64
65 /* adjust the number so the msB is non-zero */
66 for (x = 0; (z <= 4) && (x < (4 - z)); x++) {
67     num <<= 8;
68 }
69
70 /* store header */
71 x = 0;
72 out[x++] = 0x02;
73 out[x++] = z;
74
75 /* if 31st bit is set output a leading zero and decrement count */
76 if (z == 5) {
77     out[x++] = 0;
78     --z;
79 }
80
81 /* store values */
82 for (y = 0; y < z; y++) {
83     out[x++] = (num >> 24) & 0xFF;
84     num <<= 8;
85 }
86
87 /* we good */
88 *outlen = x;
89
90 return CRYPT_OK;
91 }
```

Here is the call graph for this function:

## 5.239 pk/asn1/der/short\_integer/der\_length\_short\_integer.c File Reference

### 5.239.1 Detailed Description

ASN.1 DER, get length of encoding, Tom St Denis.

Definition in file [der\\_length\\_short\\_integer.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `der_length_short_integer.c`:

### Functions

- `int der_length_short_integer` (unsigned long *num*, unsigned long \**outlen*)  
*Gets length of DER encoding of num.*

### 5.239.2 Function Documentation

#### 5.239.2.1 `int der_length_short_integer` (unsigned long *num*, unsigned long \* *outlen*)

Gets length of DER encoding of *num*.

#### Parameters:

- num* The integer to get the size of  
*outlen* [out] The length of the DER encoding for the given integer

#### Returns:

CRYPT\_OK if successful

Definition at line 26 of file `der_length_short_integer.c`.

References `CRYPT_OK`, `len`, and `LTC_ARGCHK`.

Referenced by `der_encode_short_integer()`.

```
27 {
28     unsigned long z, y, len;
29
30     LTC_ARGCHK(outlen != NULL);
31
32     /* force to 32 bits */
33     num &= 0xFFFFFFFFUL;
34
35     /* get the number of bytes */
36     z = 0;
37     y = num;
38     while (y) {
39         ++z;
40         y >>= 8;
41     }
42
43     /* handle zero */
44     if (z == 0) {
```



```
45     z = 1;
46 }
47
48 /* we need a 0x02 to indicate it's INTEGER */
49 len = 1;
50
51 /* length byte */
52 ++len;
53
54 /* bytes in value */
55 len += z;
56
57 /* see if msb is set */
58 len += (num & (1UL << ((z << 3) - 1))) ? 1 : 0;
59
60 /* return length */
61 *outlen = len;
62
63 return CRYPT_OK;
64 }
```

## 5.240 pk/asn1/der/utctime/der\_decode\_utctime.c File Reference

### 5.240.1 Detailed Description

ASN.1 DER, decode a UTCTIME, Tom St Denis.

Definition in file [der\\_decode\\_utctime.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `der_decode_utctime.c`:

### Defines

- #define [DECODE\\_V](#)(y, max)

### Functions

- static int [char\\_to\\_int](#) (unsigned char x)
- int [der\\_decode\\_utctime](#) (const unsigned char \*in, unsigned long \*inlen, ltc\_utctime \*out)

*Decodes a UTC time structure in DER format (reads all 6 valid encoding formats).*

### 5.240.2 Define Documentation

#### 5.240.2.1 #define DECODE\_V(y, max)

#### Value:

```
y = char_to_int(buf[x])*10 + char_to_int(buf[x+1]); \
if (y >= max) return CRYPT_INVALID_PACKET;      \
x += 2;
```

Definition at line 37 of file `der_decode_utctime.c`.

### 5.240.3 Function Documentation

#### 5.240.3.1 static int char\_to\_int (unsigned char x) [static]

Definition at line 20 of file `der_decode_utctime.c`.

```
21 {
22     switch (x) {
23         case '0': return 0;
24         case '1': return 1;
25         case '2': return 2;
26         case '3': return 3;
27         case '4': return 4;
28         case '5': return 5;
29         case '6': return 6;
30         case '7': return 7;
31         case '8': return 8;
32         case '9': return 9;
33     }
```

```

34     return 100;
35 }

```

### 5.240.3.2 int der\_decode\_utctime (const unsigned char \* *in*, unsigned long \* *inlen*, ltc\_utctime \* *out*)

Decodes a UTC time structure in DER format (reads all 6 valid encoding formats).

#### Parameters:

*in* Input buffer  
*inlen* Length of input buffer in octets  
*out* [out] Destination of UTC time structure

#### Returns:

CRYPT\_OK if successful

Definition at line 49 of file der\_decode\_utctime.c.

References CRYPT\_INVALID\_PACKET, der\_ia5\_value\_decode(), and LTC\_ARGCHK.

Referenced by der\_decode\_sequence\_flexi().

```

51 {
52     unsigned char buf[32];
53     unsigned long x;
54     int          y;
55
56     LTC_ARGCHK(in    != NULL);
57     LTC_ARGCHK(inlen != NULL);
58     LTC_ARGCHK(out   != NULL);
59
60     /* check header */
61     if (*inlen < 2UL || (in[1] >= sizeof(buf)) || ((in[1] + 2UL) > *inlen)) {
62         return CRYPT_INVALID_PACKET;
63     }
64
65     /* decode the string */
66     for (x = 0; x < in[1]; x++) {
67         y = der_ia5_value_decode(in[x+2]);
68         if (y == -1) {
69             return CRYPT_INVALID_PACKET;
70         }
71         buf[x] = y;
72     }
73     *inlen = 2 + x;
74
75
76     /* possible encodings are
77     YYMMDDhhmmZ
78     YYMMDDhhmm+hh'mm'
79     YYMMDDhhmm-hh'mm'
80     YYMMDDhhmmssZ
81     YYMMDDhhmmss+hh'mm'
82     YYMMDDhhmmss-hh'mm'
83
84     So let's do a trivial decode upto [including] mm
85     */
86
87     x = 0;
88     DECODE_V(out->YY, 100);

```

```
89     DECODE_V(out->MM, 13);
90     DECODE_V(out->DD, 32);
91     DECODE_V(out->hh, 24);
92     DECODE_V(out->mm, 60);
93
94     /* clear timezone and seconds info */
95     out->off_dir = out->off_hh = out->off_mm = out->ss = 0;
96
97     /* now is it Z, +, - or 0-9 */
98     if (buf[x] == 'Z') {
99         return CRYPT_OK;
100     } else if (buf[x] == '+' || buf[x] == '-') {
101         out->off_dir = (buf[x++] == '+') ? 0 : 1;
102         DECODE_V(out->off_hh, 24);
103         DECODE_V(out->off_mm, 60);
104         return CRYPT_OK;
105     }
106
107     /* decode seconds */
108     DECODE_V(out->ss, 60);
109
110     /* now is it Z, +, - */
111     if (buf[x] == 'Z') {
112         return CRYPT_OK;
113     } else if (buf[x] == '+' || buf[x] == '-') {
114         out->off_dir = (buf[x++] == '+') ? 0 : 1;
115         DECODE_V(out->off_hh, 24);
116         DECODE_V(out->off_mm, 60);
117         return CRYPT_OK;
118     } else {
119         return CRYPT_INVALID_PACKET;
120     }
121 }
```

Here is the call graph for this function:

## 5.241 pk/asn1/der/utctime/der\_encode\_utctime.c File Reference

### 5.241.1 Detailed Description

ASN.1 DER, encode a UTCTIME, Tom St Denis.

Definition in file [der\\_encode\\_utctime.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_encode\_utctime.c:

### Defines

- #define [STORE\\_V](#)(y)

### Functions

- int [der\\_encode\\_utctime](#) (ltc\_utctime \*utctime, unsigned char \*out, unsigned long \*outlen)  
*Encodes a UTC time structure in DER format.*

### Variables

- static const char \* [baseten](#) = "0123456789"

### 5.241.2 Define Documentation

#### 5.241.2.1 #define STORE\_V(y)

#### Value:

```
out[x++] = der_ia5_char_encode(baseten[(y/10) % 10]); \
out[x++] = der_ia5_char_encode(baseten[y % 10]);
```

Definition at line 22 of file der\_encode\_utctime.c.

Referenced by der\_encode\_utctime().

### 5.241.3 Function Documentation

#### 5.241.3.1 int der\_encode\_utctime (ltc\_utctime \* *utctime*, unsigned char \* *out*, unsigned long \* *outlen*)

Encodes a UTC time structure in DER format.

#### Parameters:

- utctime* The UTC time structure to encode
- out* The destination of the DER encoding of the UTC time structure
- outlen* [in/out] The length of the DER encoding

**Returns:**

CRYPT\_OK if successful

Definition at line 33 of file der\_encode\_utctime.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_OK, der\_ia5\_char\_encode(), der\_length\_utctime(), LTC\_ARGCHK, and STORE\_V.

```

35 {
36     unsigned long x, tmlen;
37     int          err;
38
39     LTC_ARGCHK(utctime != NULL);
40     LTC_ARGCHK(out      != NULL);
41     LTC_ARGCHK(outlen   != NULL);
42
43     if ((err = der_length_utctime(utctime, &tmlen)) != CRYPT_OK) {
44         return err;
45     }
46     if (tmlen > *outlen) {
47         *outlen = tmlen;
48         return CRYPT_BUFFER_OVERFLOW;
49     }
50
51     /* store header */
52     out[0] = 0x17;
53
54     /* store values */
55     x = 2;
56     STORE_V(utctime->YY);
57     STORE_V(utctime->MM);
58     STORE_V(utctime->DD);
59     STORE_V(utctime->hh);
60     STORE_V(utctime->mm);
61     STORE_V(utctime->ss);
62
63     if (utctime->off_mm || utctime->off_hh) {
64         out[x++] = der_ia5_char_encode(utctime->off_dir ? '-' : '+');
65         STORE_V(utctime->off_hh);
66         STORE_V(utctime->off_mm);
67     } else {
68         out[x++] = der_ia5_char_encode('Z');
69     }
70
71     /* store length */
72     out[1] = x - 2;
73
74     /* all good let's return */
75     *outlen = x;
76     return CRYPT_OK;
77 }

```

Here is the call graph for this function:

**5.241.4 Variable Documentation****5.241.4.1** `const char* baseten = "0123456789"` [static]

Definition at line 20 of file der\_encode\_utctime.c.

## 5.242 pk/asn1/der/utctime/der\_length\_utctime.c File Reference

### 5.242.1 Detailed Description

ASN.1 DER, get length of UTCTIME, Tom St Denis.

Definition in file [der\\_length\\_utctime.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for der\_length\_utctime.c:

### Functions

- [int der\\_length\\_utctime](#) (ltc\_utctime \*utctime, unsigned long \*outlen)  
*Gets length of DER encoding of UTCTIME.*

### 5.242.2 Function Documentation

#### 5.242.2.1 int der\_length\_utctime (ltc\_utctime \* utctime, unsigned long \* outlen)

Gets length of DER encoding of UTCTIME.

##### Parameters:

- utctime** The UTC time structure to get the size of
- outlen** [out] The length of the DER encoding

##### Returns:

CRYPT\_OK if successful

Definition at line 26 of file der\_length\_utctime.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by der\_decode\_sequence\_flexi(), and der\_encode\_utctime().

```
27 {
28     LTC_ARGCHK(outlen != NULL);
29     LTC_ARGCHK(utctime != NULL);
30
31     if (utctime->off_hh == 0 && utctime->off_mm == 0) {
32         /* we encode as YYMMDDhhmmssZ */
33         *outlen = 2 + 13;
34     } else {
35         /* we encode as YYMMDDhhmmss{+|-}hh'mm' */
36         *outlen = 2 + 17;
37     }
38
39     return CRYPT_OK;
40 }
```

## 5.243 pk/dsa/dsa\_decrypt\_key.c File Reference

### 5.243.1 Detailed Description

DSA Crypto, Tom St Denis.

Definition in file [dsa\\_decrypt\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for dsa\_decrypt\_key.c:

### Functions

- [int dsa\\_decrypt\\_key](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, dsa\_key \**key*)

*Decrypt an DSA encrypted key.*

### 5.243.2 Function Documentation

#### 5.243.2.1 int dsa\_decrypt\_key (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*, dsa\_key \* *key*)

Decrypt an DSA encrypted key.

#### Parameters:

*in* The ciphertext

*inlen* The length of the ciphertext (octets)

*out* [out] The plaintext

*outlen* [in/out] The max size and resulting size of the plaintext

*key* The corresponding private DSA key

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file dsa\_decrypt\_key.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_INVALID\_PACKET, CRYPT\_MEM, CRYPT\_OK, CRYPT\_PK\_NOT\_PRIVATE, dsa\_shared\_secret(), find\_hash\_oid(), hash\_is\_valid(), hash\_memory(), LTC\_ARGCHK, MAXBLOCKSIZE, MIN, PK\_PRIVATE, edge::size, XFREE, and XMAL-LOC.

```
32 {
33     unsigned char    *skey, *expt;
34     void             *g_pub;
35     unsigned long    x, y, hashOID[32];
36     int              hash, err;
37     ltc_asn1_list     decode[3];
38
39     LTC_ARGCHK(in      != NULL);
40     LTC_ARGCHK(out     != NULL);
41     LTC_ARGCHK(outlen  != NULL);
42     LTC_ARGCHK(key     != NULL);
```



```

43
44  /* right key type? */
45  if (key->type != PK_PRIVATE) {
46      return CRYPT_PK_NOT_PRIVATE;
47  }
48
49  /* decode to find out hash */
50  LTC_SET_ASN1(decode, 0, LTC_ASN1_OBJECT_IDENTIFIER, hashOID, sizeof(hashOID)/sizeof(hashOID[0]));
51
52  if ((err = der_decode_sequence(in, inlen, decode, 1)) != CRYPT_OK) {
53      return err;
54  }
55
56  hash = find_hash_oid(hashOID, decode[0].size);
57  if (hash_is_valid(hash) != CRYPT_OK) {
58      return CRYPT_INVALID_PACKET;
59  }
60
61  /* we now have the hash! */
62
63  if ((err = mp_init(&g_pub)) != CRYPT_OK) {
64      return err;
65  }
66
67  /* allocate memory */
68  expt = XMALLOC(mp_unsigned_bin_size(key->p) + 1);
69  skey = XMALLOC(MAXBLOCKSIZE);
70  if (expt == NULL || skey == NULL) {
71      if (expt != NULL) {
72          XFREE(expt);
73      }
74      if (skey != NULL) {
75          XFREE(skey);
76      }
77      mp_clear(g_pub);
78      return CRYPT_MEM;
79  }
80
81  LTC_SET_ASN1(decode, 1, LTC_ASN1_INTEGER, g_pub, 1UL);
82  LTC_SET_ASN1(decode, 2, LTC_ASN1_OCTET_STRING, skey, MAXBLOCKSIZE);
83
84  /* read the structure in now */
85  if ((err = der_decode_sequence(in, inlen, decode, 3)) != CRYPT_OK) {
86      goto LBL_ERR;
87  }
88
89  /* make shared key */
90  x = mp_unsigned_bin_size(key->p) + 1;
91  if ((err = dsa_shared_secret(key->x, g_pub, key, expt, &x)) != CRYPT_OK) {
92      goto LBL_ERR;
93  }
94
95  y = MIN(mp_unsigned_bin_size(key->p) + 1, MAXBLOCKSIZE);
96  if ((err = hash_memory(hash, expt, x, expt, &y)) != CRYPT_OK) {
97      goto LBL_ERR;
98  }
99
100  /* ensure the hash of the shared secret is at least as big as the encrypt itself */
101  if (decode[2].size > y) {
102      err = CRYPT_INVALID_PACKET;
103      goto LBL_ERR;
104  }
105
106  /* avoid buffer overflow */
107  if (*outlen < decode[2].size) {
108      *outlen = decode[2].size;
109      err = CRYPT_BUFFER_OVERFLOW;

```

```
110     goto LBL_ERR;
111 }
112
113 /* Decrypt the key */
114 for (x = 0; x < decode[2].size; x++) {
115     out[x] = expt[x] ^ skey[x];
116 }
117 *outlen = x;
118
119 err = CRYPT_OK;
120 LBL_ERR:
121 #ifdef LTC_CLEAN_STACK
122     zeromem(expt,    mp_unsigned_bin_size(key->p) + 1);
123     zeromem(skey,    MAXBLOCKSIZE);
124 #endif
125
126     XFREE(expt);
127     XFREE(skey);
128
129     mp_clear(g_pub);
130
131     return err;
132 }
```

Here is the call graph for this function:

## 5.244 pk/dsa/dsa\_encrypt\_key.c File Reference

### 5.244.1 Detailed Description

DSA Crypto, Tom St Denis.

Definition in file [dsa\\_encrypt\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for dsa\_encrypt\_key.c:

### Functions

- int [dsa\\_encrypt\\_key](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, [prng\\_state](#) \**prng*, int *wprng*, int *hash*, dsa\_key \**key*)

*Encrypt a symmetric key with DSA.*

### 5.244.2 Function Documentation

- 5.244.2.1** int [dsa\\_encrypt\\_key](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, [prng\\_state](#) \**prng*, int *wprng*, int *hash*, dsa\_key \**key*)

Encrypt a symmetric key with DSA.

#### Parameters:

- in* The symmetric key you want to encrypt
- inlen* The length of the key to encrypt (octets)
- out* [out] The destination for the ciphertext
- outlen* [in/out] The max size and resulting size of the ciphertext
- prng* An active PRNG state
- wprng* The index of the PRNG you wish to use
- hash* The index of the hash you want to use
- key* The DSA key you want to encrypt to

#### Returns:

- CRYPT\_OK if successful

Definition at line 32 of file dsa\_encrypt\_key.c.

References [CRYPT\\_ERROR\\_READPRNG](#), [CRYPT\\_INVALID\\_HASH](#), [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [dsa\\_shared\\_secret\(\)](#), [hash\\_descriptor](#), [hash\\_is\\_valid\(\)](#), [hash\\_memory\(\)](#), [LTC\\_ARGCHK](#), [MAXBLOCKSIZE](#), [prng\\_descriptor](#), [prng\\_is\\_valid\(\)](#), [XFREE](#), and [XMALLOC](#).

```
36 {
37     unsigned char *expt, *skey;
38     void          *g_pub, *g_priv;
39     unsigned long  x, y;
40     int            err;
41
42     LTC_ARGCHK(in      != NULL);
```

```

43     LTC_ARGCHK(out      != NULL);
44     LTC_ARGCHK(outlen   != NULL);
45     LTC_ARGCHK(key      != NULL);
46
47     /* check that wprng/cipher/hash are not invalid */
48     if ((err = prng_is_valid(wprng)) != CRYPT_OK) {
49         return err;
50     }
51
52     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
53         return err;
54     }
55
56     if (inlen > hash_descriptor[hash].hashsize) {
57         return CRYPT_INVALID_HASH;
58     }
59
60     /* make a random key and export the public copy */
61     if ((err = mp_init_multi(&g_pub, &g_priv, NULL)) != CRYPT_OK) {
62         return err;
63     }
64
65     expt      = XMALLOC(mp_unsigned_bin_size(key->p) + 1);
66     skey      = XMALLOC(MAXBLOCKSIZE);
67     if (expt == NULL || skey == NULL) {
68         if (expt != NULL) {
69             XFREE(expt);
70         }
71         if (skey != NULL) {
72             XFREE(skey);
73         }
74         mp_clear_multi(g_pub, g_priv, NULL);
75         return CRYPT_MEM;
76     }
77
78     /* make a random x, g^x pair */
79     x = mp_unsigned_bin_size(key->q);
80     if (prng_descriptor[wprng].read(expt, x, prng) != x) {
81         err = CRYPT_ERROR_READPRNG;
82         goto LBL_ERR;
83     }
84
85     /* load x */
86     if ((err = mp_read_unsigned_bin(g_priv, expt, x)) != CRYPT_OK) {
87         goto LBL_ERR;
88     }
89
90     /* compute y */
91     if ((err = mp_exptmod(key->g, g_priv, key->p, g_pub)) != CRYPT_OK) {
92         goto LBL_ERR;
93     }
94
95     /* make random key */
96     x = mp_unsigned_bin_size(key->p) + 1;
97     if ((err = dsa_shared_secret(g_priv, key->y, key, expt, &x)) != CRYPT_OK) {
98         goto LBL_ERR;
99     }
100
101     y = MAXBLOCKSIZE;
102     if ((err = hash_memory(hash, expt, x, skey, &y)) != CRYPT_OK) {
103         goto LBL_ERR;
104     }
105
106     /* Encrypt key */
107     for (x = 0; x < inlen; x++) {
108         skey[x] ^= in[x];
109     }

```

```
110
111     err = der_encode_sequence_multi(out, outlen,
112                                     LTC_ASN1_OBJECT_IDENTIFIER, hash_descriptor[hash].OIDlen, hash_
113                                     LTC_ASN1_INTEGER, 1UL, g_pub
114                                     LTC_ASN1_OCTET_STRING, inlen, skey,
115                                     LTC_ASN1_EOL, 0UL, NULL)
116
117 LBL_ERR:
118 #ifdef LTC_CLEAN_STACK
119     /* clean up */
120     zeromem(expt, mp_unsigned_bin_size(key->p) + 1);
121     zeromem(skey, MAXBLOCKSIZE);
122 #endif
123
124     XFREE(skey);
125     XFREE(expt);
126
127     mp_clear_multi(g_pub, g_priv, NULL);
128
129     return err;
130 }
```

Here is the call graph for this function:

## 5.245 pk/dsa/dsa\_export.c File Reference

### 5.245.1 Detailed Description

DSA implementation, export key, Tom St Denis.

Definition in file [dsa\\_export.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for dsa\_export.c:

### Functions

- `int dsa_export` (unsigned char \*out, unsigned long \*outlen, int type, dsa\_key \*key)  
*Export a DSA key to a binary packet.*

### 5.245.2 Function Documentation

#### 5.245.2.1 int dsa\_export (unsigned char \* out, unsigned long \* outlen, int type, dsa\_key \* key)

Export a DSA key to a binary packet.

#### Parameters:

- out** [out] Where to store the packet
- outlen** [in/out] The max size and resulting size of the packet
- type** The type of key to export (PK\_PRIVATE or PK\_PUBLIC)
- key** The key to export

#### Returns:

- CRYPT\_OK if successful

Definition at line 28 of file dsa\_export.c.

References CRYPT\_INVALID\_ARG, CRYPT\_PK\_TYPE\_MISMATCH, der\_encode\_sequence\_multi(), LTC\_ARGCHK, PK\_PRIVATE, and PK\_PUBLIC.

```
29 {
30     unsigned char flags[1];
31
32     LTC_ARGCHK(out != NULL);
33     LTC_ARGCHK(outlen != NULL);
34     LTC_ARGCHK(key != NULL);
35
36     /* can we store the static header? */
37     if (type == PK_PRIVATE && key->type != PK_PRIVATE) {
38         return CRYPT_PK_TYPE_MISMATCH;
39     }
40
41     if (type != PK_PUBLIC && type != PK_PRIVATE) {
42         return CRYPT_INVALID_ARG;
43     }
44
45     flags[0] = (type != PK_PUBLIC) ? 1 : 0;
```

```
46
47     if (type == PK_PRIVATE) {
48         return der_encode_sequence_multi(out, outlen,
49                                         LTC_ASN1_BIT_STRING, 1UL, flags,
50                                         LTC_ASN1_INTEGER, 1UL, key->g,
51                                         LTC_ASN1_INTEGER, 1UL, key->p,
52                                         LTC_ASN1_INTEGER, 1UL, key->q,
53                                         LTC_ASN1_INTEGER, 1UL, key->y,
54                                         LTC_ASN1_INTEGER, 1UL, key->x,
55                                         LTC_ASN1_EOL, 0UL, NULL);
56     } else {
57         return der_encode_sequence_multi(out, outlen,
58                                         LTC_ASN1_BIT_STRING, 1UL, flags,
59                                         LTC_ASN1_INTEGER, 1UL, key->g,
60                                         LTC_ASN1_INTEGER, 1UL, key->p,
61                                         LTC_ASN1_INTEGER, 1UL, key->q,
62                                         LTC_ASN1_INTEGER, 1UL, key->y,
63                                         LTC_ASN1_EOL, 0UL, NULL);
64     }
65 }
```

Here is the call graph for this function:

## 5.246 pk/dsa/dsa\_free.c File Reference

### 5.246.1 Detailed Description

DSA implementation, free a DSA key, Tom St Denis.

Definition in file [dsa\\_free.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for dsa\_free.c:

### Functions

- void [dsa\\_free](#) (dsa\_key \*key)

*Free a DSA key.*

### 5.246.2 Function Documentation

#### 5.246.2.1 void dsa\_free (dsa\_key \*key)

Free a DSA key.

#### Parameters:

*key* The key to free from memory

Definition at line 24 of file dsa\_free.c.

References LTC\_ARGCHKVD.

```
25 {  
26     LTC_ARGCHKVD(key != NULL);  
27     mp_clear_multi(key->g, key->q, key->p, key->x, key->y, NULL);  
28 }
```



## 5.247 pk/dsa/dsa\_import.c File Reference

### 5.247.1 Detailed Description

DSA implementation, import a DSA key, Tom St Denis.

Definition in file [dsa\\_import.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for dsa\_import.c:

### Functions

- `int dsa_import (const unsigned char *in, unsigned long inlen, dsa_key *key)`  
*Import a DSA key.*

### 5.247.2 Function Documentation

#### 5.247.2.1 `int dsa_import (const unsigned char *in, unsigned long inlen, dsa_key *key)`

Import a DSA key.

##### Parameters:

- in** The binary packet to import from
- inlen** The length of the binary packet
- key** [out] Where to store the imported key

##### Returns:

CRYPT\_OK if successful, upon error this function will free all allocated memory

Definition at line 27 of file dsa\_import.c.

References CRYPT\_INVALID\_PACKET, CRYPT\_MEM, CRYPT\_OK, der\_decode\_sequence\_multi(), LTC\_ARGCHK, ltc\_mp, ltc\_math\_descriptor::name, PK\_PRIVATE, and PK\_PUBLIC.

```
28 {
29     unsigned char flags[1];
30     int          err;
31
32     LTC_ARGCHK(in  != NULL);
33     LTC_ARGCHK(key != NULL);
34     LTC_ARGCHK(ltc_mp.name != NULL);
35
36     /* init key */
37     if (mp_init_multi(&key->p, &key->g, &key->q, &key->x, &key->y, NULL) != CRYPT_OK) {
38         return CRYPT_MEM;
39     }
40
41     /* get key type */
42     if ((err = der_decode_sequence_multi(in, inlen,
43                                         LTC_ASN1_BIT_STRING, 1UL, flags,
44                                         LTC_ASN1_EOL, 0UL, NULL)) != CRYPT_OK) {
45         goto error;
46     }
```

```

47
48     if (flags[0] == 1) {
49         if ((err = der_decode_sequence_multi(in, inlen,
50             LTC_ASN1_BIT_STRING,    1UL, flags,
51             LTC_ASN1_INTEGER,      1UL, key->q,
52             LTC_ASN1_INTEGER,      1UL, key->p,
53             LTC_ASN1_INTEGER,      1UL, key->q,
54             LTC_ASN1_INTEGER,      1UL, key->y,
55             LTC_ASN1_INTEGER,      1UL, key->x,
56             LTC_ASN1_EOL,          0UL, NULL)) != CRYPT_OK) {
57             goto error;
58         }
59         key->type = PK_PRIVATE;
60     } else {
61         if ((err = der_decode_sequence_multi(in, inlen,
62             LTC_ASN1_BIT_STRING,    1UL, flags,
63             LTC_ASN1_INTEGER,      1UL, key->q,
64             LTC_ASN1_INTEGER,      1UL, key->p,
65             LTC_ASN1_INTEGER,      1UL, key->q,
66             LTC_ASN1_INTEGER,      1UL, key->y,
67             LTC_ASN1_EOL,          0UL, NULL)) != CRYPT_OK) {
68             goto error;
69         }
70         key->type = PK_PUBLIC;
71     }
72     key->qord = mp_unsigned_bin_size(key->q);
73
74     if (key->qord >= MDSA_MAX_GROUP || key->qord <= 15 ||
75         (unsigned long)key->qord >= mp_unsigned_bin_size(key->p) || (mp_unsigned_bin_size(key->p) - key->qord >= 1))
76         err = CRYPT_INVALID_PACKET;
77         goto error;
78     }
79
80     return CRYPT_OK;
81 error:
82     mp_clear_multi(key->p, key->q, key->q, key->x, key->y, NULL);
83     return err;
84 }

```

Here is the call graph for this function:

## 5.248 pk/dsa/dsa\_make\_key.c File Reference

### 5.248.1 Detailed Description

DSA implementation, generate a DSA key, Tom St Denis.

Definition in file [dsa\\_make\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for dsa\_make\_key.c:

### Functions

- int [dsa\\_make\\_key](#) ([prng\\_state](#) \*prng, int wprng, int group\_size, int modulus\_size, dsa\_key \*key)  
*Create a DSA key.*

### 5.248.2 Function Documentation

#### 5.248.2.1 int dsa\_make\_key ([prng\\_state](#) \*prng, int wprng, int group\_size, int modulus\_size, dsa\_key \*key)

Create a DSA key.

#### Parameters:

- prng* An active PRNG state
- wprng* The index of the PRNG desired
- group\_size* Size of the multiplicative group (octets)
- modulus\_size* Size of the modulus (octets)
- key* [out] Where to store the created key

#### Returns:

CRYPT\_OK if successful, upon error this function will free all allocated memory

Definition at line 29 of file dsa\_make\_key.c.

References [CRYPT\\_ERROR\\_READPRNG](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), [ltc\\_mp](#), [ltc\\_math\\_descriptor::name](#), [prng\\_descriptor](#), [prng\\_is\\_valid\(\)](#), [rand\\_prime\(\)](#), and [XMALLOC](#).

```
30 {
31     void          *tmp, *tmp2;
32     int           err, res;
33     unsigned char *buf;
34
35     LTC_ARGCHK(key != NULL);
36     LTC_ARGCHK(ltc_mp.name != NULL);
37
38     /* check prng */
39     if ((err = prng_is_valid(wprng)) != CRYPT_OK) {
40         return err;
41     }
42 }
```

```

43  /* check size */
44  if (group_size >= MDSA_MAX_GROUP || group_size <= 15 ||
45      group_size >= modulus_size || (modulus_size - group_size) >= MDSA_DELTA) {
46      return CRYPT_INVALID_ARG;
47  }
48
49  /* allocate ram */
50  buf = XMALLOC(MDSA_DELTA);
51  if (buf == NULL) {
52      return CRYPT_MEM;
53  }
54
55  /* init mp_ints */
56  if ((err = mp_init_multi(&tmp, &tmp2, &key->q, &key->q, &key->p, &key->x, &key->y, NULL)) != CRYPT_OK)
57      goto LBL_ERR;
58  }
59
60  /* make our prime q */
61  if ((err = rand_prime(key->q, group_size, prng, wprng)) != CRYPT_OK) { goto LBL_ERR; }
62
63  /* double q */
64  if ((err = mp_add(key->q, key->q, tmp)) != CRYPT_OK) { goto error; }
65
66  /* now make a random string and multiply it against q */
67  if (prng_descriptor[wprng].read(buf+1, modulus_size - group_size, prng) != (unsigned long)(modulus_size - group_size))
68      err = CRYPT_ERROR_READPRNG;
69  goto LBL_ERR;
70  }
71
72  /* force magnitude */
73  buf[0] |= 0xC0;
74
75  /* force even */
76  buf[modulus_size - group_size - 1] &= ~1;
77
78  if ((err = mp_read_unsigned_bin(tmp2, buf, modulus_size - group_size)) != CRYPT_OK) { goto error; }
79  if ((err = mp_mul(key->q, tmp2, key->p)) != CRYPT_OK) { goto error; }
80  if ((err = mp_add_d(key->p, 1, key->p)) != CRYPT_OK) { goto error; }
81
82  /* now loop until p is prime */
83  for (;;) {
84      if ((err = mp_prime_is_prime(key->p, 8, &res)) != CRYPT_OK) { goto LBL_ERR; }
85      if (res == LTC_MP_YES) break;
86
87      /* add 2q to p and 2 to tmp2 */
88      if ((err = mp_add(tmp, key->p, key->p)) != CRYPT_OK) { goto error; }
89      if ((err = mp_add_d(tmp2, 2, tmp2)) != CRYPT_OK) { goto error; }
90  }
91
92  /* now p = (q * tmp2) + 1 is prime, find a value g for which g^tmp2 != 1 */
93  mp_set(key->g, 1);
94
95  do {
96      if ((err = mp_add_d(key->g, 1, key->g)) != CRYPT_OK) { goto error; }
97      if ((err = mp_exptmod(key->g, tmp2, key->p, tmp)) != CRYPT_OK) { goto error; }
98  } while (mp_cmp_d(tmp, 1) == LTC_MP_EQ);
99
100  /* at this point tmp generates a group of order q mod p */
101  mp_exch(tmp, key->g);
102
103  /* so now we have our DH structure, generator g, order q, modulus p
104   * Now we need a random exponent [mod q] and it's power g^x mod p
105   */
106  do {
107      if (prng_descriptor[wprng].read(buf, group_size, prng) != (unsigned long)group_size) {
108          err = CRYPT_ERROR_READPRNG;
109          goto LBL_ERR;

```

```
110     }
111     if ((err = mp_read_unsigned_bin(key->x, buf, group_size)) != CRYPT_OK)      { goto error; }
112 } while (mp_cmp_d(key->x, 1) != LTC_MP_GT);
113 if ((err = mp_exptmod(key->g, key->x, key->p, key->y)) != CRYPT_OK)      { goto error; }
114
115 key->type = PK_PRIVATE;
116 key->qord = group_size;
117
118 #ifdef LTC_CLEAN_STACK
119     zeromem(buf, MDSA_DELTA);
120 #endif
121
122     err = CRYPT_OK;
123     goto done;
124 error:
125 LBL_ERR:
126     mp_clear_multi(key->g, key->q, key->p, key->x, key->y, NULL);
127 done:
128     mp_clear_multi(tmp, tmp2, NULL);
129
130     XFREE(buf);
131     return err;
132 }
```

Here is the call graph for this function:

## 5.249 pk/dsa/dsa\_shared\_secret.c File Reference

### 5.249.1 Detailed Description

DSA Crypto, Tom St Denis.

Definition in file [dsa\\_shared\\_secret.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for dsa\_shared\_secret.c:

### Functions

- [int dsa\\_shared\\_secret](#) (void \*private\_key, void \*base, dsa\_key \*public\_key, unsigned char \*out, unsigned long \*outlen)

*Create a DSA shared secret between two keys.*

### 5.249.2 Function Documentation

#### 5.249.2.1 int dsa\_shared\_secret (void \*private\_key, void \*base, dsa\_key \*public\_key, unsigned char \*out, unsigned long \*outlen)

Create a DSA shared secret between two keys.

#### Parameters:

**private\_key** The private DSA key (the exponent)

**base** The base of the exponentiation (allows this to be used for both encrypt and decrypt)

**public\_key** The public key

**out** [out] Destination of the shared secret

**outlen** [in/out] The max size and resulting size of the shared secret

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file dsa\_shared\_secret.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_OK, LTC\_ARGCHK, and zeromem().

Referenced by dsa\_decrypt\_key(), and dsa\_encrypt\_key().

```
32 {
33     unsigned long x;
34     void *res;
35     int err;
36
37     LTC_ARGCHK(private_key != NULL);
38     LTC_ARGCHK(public_key != NULL);
39     LTC_ARGCHK(out != NULL);
40     LTC_ARGCHK(outlen != NULL);
41
42     /* make new point */
43     if ((err = mp_init(&res)) != CRYPT_OK) {
44         return err;
45     }
```

```
45     }
46
47     if ((err = mp_exptmod(base, private_key, public_key->p, res)) != CRYPT_OK) {
48         mp_clear(res);
49         return err;
50     }
51
52     x = (unsigned long)mp_unsigned_bin_size(res);
53     if (*outlen < x) {
54         *outlen = x;
55         err = CRYPT_BUFFER_OVERFLOW;
56         goto done;
57     }
58     zeromem(out, x);
59     if ((err = mp_to_unsigned_bin(res, out + (x - mp_unsigned_bin_size(res)))) != CRYPT_OK)
60
61     err = CRYPT_OK;
62     *outlen = x;
63 done:
64     mp_clear(res);
65     return err;
66 }
```

Here is the call graph for this function:

## 5.250 pk/dsa/dsa\_sign\_hash.c File Reference

### 5.250.1 Detailed Description

DSA implementation, sign a hash, Tom St Denis.

Definition in file [dsa\\_sign\\_hash.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for dsa\_sign\_hash.c:

### Functions

- [int dsa\\_sign\\_hash\\_raw](#) (const unsigned char \**in*, unsigned long *inlen*, void \**r*, void \**s*, [prng\\_state](#) \**prng*, int *wprng*, dsa\_key \**key*)  
*Sign a hash with DSA.*
- [int dsa\\_sign\\_hash](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, [prng\\_state](#) \**prng*, int *wprng*, dsa\_key \**key*)  
*Sign a hash with DSA.*

### 5.250.2 Function Documentation

**5.250.2.1** [int dsa\\_sign\\_hash](#) (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*, [prng\\_state](#) \* *prng*, int *wprng*, dsa\_key \* *key*)

Sign a hash with DSA.

#### Parameters:

- in* The hash to sign
- inlen* The length of the hash to sign
- out* [out] Where to store the signature
- outlen* [in/out] The max size and resulting size of the signature
- prng* An active PRNG state
- wprng* The index of the PRNG desired
- key* A private DSA key

#### Returns:

- CRYPT\_OK if successful

Definition at line 124 of file dsa\_sign\_hash.c.

References CRYPT\_MEM, CRYPT\_OK, der\_encode\_sequence\_multi(), dsa\_sign\_hash\_raw(), and LTC\_ARGCHK.

```
127 {
128     void      *r, *s;
129     int        err;
130
131     LTC_ARGCHK(in != NULL);
```



```

132     LTC_ARGCHK(out      != NULL);
133     LTC_ARGCHK(outlen   != NULL);
134     LTC_ARGCHK(key      != NULL);
135
136     if (mp_init_multi(&r, &s, NULL) != CRYPT_OK) {
137         return CRYPT_MEM;
138     }
139
140     if ((err = dsa_sign_hash_raw(in, inlen, r, s, prng, wprng, key)) != CRYPT_OK) {
141         goto LBL_ERR;
142     }
143
144     err = der_encode_sequence_multi(out, outlen,
145                                     LTC_ASN1_INTEGER, 1UL, r,
146                                     LTC_ASN1_INTEGER, 1UL, s,
147                                     LTC_ASN1_EOL,     0UL, NULL);
148
149 LBL_ERR:
150     mp_clear_multi(r, s, NULL);
151     return err;
152 }

```

Here is the call graph for this function:

### 5.250.2.2 int dsa\_sign\_hash\_raw (const unsigned char \* *in*, unsigned long *inlen*, void \* *r*, void \* *s*, *prng\_state* \* *prng*, int *wprng*, dsa\_key \* *key*)

Sign a hash with DSA.

#### Parameters:

- in* The hash to sign
- inlen* The length of the hash to sign
- r* The "r" integer of the signature (caller must initialize with mp\_init() first)
- s* The "s" integer of the signature (caller must initialize with mp\_init() first)
- prng* An active PRNG state
- wprng* The index of the PRNG desired
- key* A private DSA key

#### Returns:

CRYPT\_OK if successful

Definition at line 31 of file dsa\_sign\_hash.c.

References CRYPT\_ERROR\_READPRNG, CRYPT\_INVALID\_ARG, CRYPT\_MEM, CRYPT\_OK, CRYPT\_PK\_NOT\_PRIVATE, LTC\_ARGCHK, LTC\_MP\_EQ, LTC\_MP\_GT, LTC\_MP\_YES, PK\_PRIVATE, prng\_descriptor, prng\_is\_valid(), XFREE, XMALLOC, and zeromem().

Referenced by dsa\_sign\_hash().

```

34 {
35     void          *k, *kinv, *tmp;
36     unsigned char *buf;
37     int           err;
38
39     LTC_ARGCHK(in  != NULL);
40     LTC_ARGCHK(r   != NULL);
41     LTC_ARGCHK(s   != NULL);

```

```

42 LTC_ARGCHK(key != NULL);
43
44 if ((err = prng_is_valid(wprng)) != CRYPT_OK) {
45     return err;
46 }
47 if (key->type != PK_PRIVATE) {
48     return CRYPT_PK_NOT_PRIVATE;
49 }
50
51 /* check group order size */
52 if (key->qord >= MDSA_MAX_GROUP) {
53     return CRYPT_INVALID_ARG;
54 }
55
56 buf = XMALLOC(MDSA_MAX_GROUP);
57 if (buf == NULL) {
58     return CRYPT_MEM;
59 }
60
61 /* Init our temps */
62 if ((err = mp_init_multi(&k, &kinv, &tmp, NULL)) != CRYPT_OK) { goto error; }
63
64 retry:
65
66 do {
67     /* gen random k */
68     if (prng_descriptor[wprng].read(buf, key->qord, prng) != (unsigned long)key->qord) {
69         err = CRYPT_ERROR_READPRNG;
70         goto LBL_ERR;
71     }
72
73     /* read k */
74     if ((err = mp_read_unsigned_bin(k, buf, key->qord)) != CRYPT_OK) { goto error; }
75
76     /* k > 1 ? */
77     if (mp_cmp_d(k, 1) != LTC_MP_GT) { goto retry; }
78
79     /* test gcd */
80     if ((err = mp_gcd(k, key->q, tmp)) != CRYPT_OK) { goto error; }
81 } while (mp_cmp_d(tmp, 1) != LTC_MP_EQ);
82
83 /* now find 1/k mod q */
84 if ((err = mp_invmod(k, key->q, kinv)) != CRYPT_OK) { goto error; }
85
86 /* now find r = g^k mod p mod q */
87 if ((err = mp_exptmod(key->g, k, key->p, r)) != CRYPT_OK) { goto error; }
88 if ((err = mp_mod(r, key->q, r)) != CRYPT_OK) { goto error; }
89
90 if (mp_iszero(r) == LTC_MP_YES) { goto retry; }
91
92 /* now find s = (in + xr)/k mod q */
93 if ((err = mp_read_unsigned_bin(tmp, (unsigned char *)in, inlen)) != CRYPT_OK) { goto error; }
94 if ((err = mp_mul(key->x, r, s)) != CRYPT_OK) { goto error; }
95 if ((err = mp_add(s, tmp, s)) != CRYPT_OK) { goto error; }
96 if ((err = mp_mulmod(s, kinv, key->q, s)) != CRYPT_OK) { goto error; }
97
98 if (mp_iszero(s) == LTC_MP_YES) { goto retry; }
99
100 err = CRYPT_OK;
101 goto LBL_ERR;
102
103 error:
104 LBL_ERR:
105     mp_clear_multi(k, kinv, tmp, NULL);
106 #ifdef LTC_CLEAN_STACK
107     zeromem(buf, MDSA_MAX_GROUP);
108 #endif

```

```
109     XFREE(buf);  
110     return err;  
111 }
```

Here is the call graph for this function:

## 5.251 pk/dsa/dsa\_verify\_hash.c File Reference

### 5.251.1 Detailed Description

DSA implementation, verify a signature, Tom St Denis.

Definition in file [dsa\\_verify\\_hash.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for dsa\_verify\_hash.c:

### Functions

- int [dsa\\_verify\\_hash\\_raw](#) (void \*r, void \*s, const unsigned char \*hash, unsigned long hashlen, int \*stat, dsa\_key \*key)  
*Verify a DSA signature.*
- int [dsa\\_verify\\_hash](#) (const unsigned char \*sig, unsigned long siglen, const unsigned char \*hash, unsigned long hashlen, int \*stat, dsa\_key \*key)  
*Verify a DSA signature.*

### 5.251.2 Function Documentation

#### 5.251.2.1 int dsa\_verify\_hash (const unsigned char \*sig, unsigned long siglen, const unsigned char \*hash, unsigned long hashlen, int \*stat, dsa\_key \*key)

Verify a DSA signature.

#### Parameters:

- sig** The signature  
**siglen** The length of the signature (octets)  
**hash** The hash that was signed  
**hashlen** The length of the hash that was signed  
**stat** [out] The result of the signature verification, 1==valid, 0==invalid  
**key** The corresponding public DH key

#### Returns:

CRYPT\_OK if successful (even if the signature is invalid)

Definition at line 96 of file dsa\_verify\_hash.c.

References CRYPT\_MEM, CRYPT\_OK, der\_decode\_sequence\_multi(), and dsa\_verify\_hash\_raw().

```
99 {
100     int     err;
101     void    *r, *s;
102
103     if ((err = mp_init_multi(&r, &s, NULL)) != CRYPT_OK) {
104         return CRYPT_MEM;
105     }
106 }
```

```

107  /* decode the sequence */
108  if ((err = der_decode_sequence_multi(sig, siglen,
109                                     LTC_ASN1_INTEGER, 1UL, r,
110                                     LTC_ASN1_INTEGER, 1UL, s,
111                                     LTC_ASN1_EOL, 0UL, NULL)) != CRYPT_OK) {
112      goto LBL_ERR;
113  }
114
115  /* do the op */
116  err = dsa_verify_hash_raw(r, s, hash, hashlen, stat, key);
117
118 LBL_ERR:
119  mp_clear_multi(r, s, NULL);
120  return err;
121 }

```

Here is the call graph for this function:

### 5.251.2.2 int dsa\_verify\_hash\_raw (void \*r, void \*s, const unsigned char \*hash, unsigned long hashlen, int \*stat, dsa\_key \*key)

Verify a DSA signature.

#### Parameters:

*r* DSA "r" parameter

*s* DSA "s" parameter

*hash* The hash that was signed

*hashlen* The length of the hash that was signed

*stat* [out] The result of the signature verification, 1==valid, 0==invalid

*key* The corresponding public DH key

#### Returns:

CRYPT\_OK if successful (even if the signature is invalid)

Definition at line 31 of file dsa\_verify\_hash.c.

References CRYPT\_INVALID\_PACKET, CRYPT\_OK, LTC\_ARGCHK, LTC\_MP\_EQ, LTC\_MP\_LT, and LTC\_MP\_YES.

Referenced by dsa\_verify\_hash().

```

34 {
35     void          *w, *v, *u1, *u2;
36     int           err;
37
38     LTC_ARGCHK(r    != NULL);
39     LTC_ARGCHK(s    != NULL);
40     LTC_ARGCHK(stat != NULL);
41     LTC_ARGCHK(key  != NULL);
42
43     /* default to invalid signature */
44     *stat = 0;
45
46     /* init our variables */
47     if ((err = mp_init_multi(&w, &v, &u1, &u2, NULL)) != CRYPT_OK) {
48         return err;
49     }
50

```

```

51  /* neither r or s can be null or >q*/
52  if (mp_iszero(r) == LTC_MP_YES || mp_iszero(s) == LTC_MP_YES || mp_cmp(r, key->q) != LTC_MP_LT || mp
53      err = CRYPT_INVALID_PACKET;
54      goto done;
55  }
56
57  /* w = 1/s mod q */
58  if ((err = mp_invmod(s, key->q, w)) != CRYPT_OK) { goto error; }
59
60  /* u1 = m * w mod q */
61  if ((err = mp_read_unsigned_bin(u1, (unsigned char *)hash, hashlen)) != CRYPT_OK) { goto error; }
62  if ((err = mp_mulmod(u1, w, key->q, u1)) != CRYPT_OK) { goto error; }
63
64  /* u2 = r*w mod q */
65  if ((err = mp_mulmod(r, w, key->q, u2)) != CRYPT_OK) { goto error; }
66
67  /* v = g^u1 * y^u2 mod p mod q */
68  if ((err = mp_exptmod(key->g, u1, key->p, u1)) != CRYPT_OK) { goto error; }
69  if ((err = mp_exptmod(key->y, u2, key->p, u2)) != CRYPT_OK) { goto error; }
70  if ((err = mp_mulmod(u1, u2, key->p, v)) != CRYPT_OK) { goto error; }
71  if ((err = mp_mod(v, key->q, v)) != CRYPT_OK) { goto error; }
72
73  /* if r = v then we're set */
74  if (mp_cmp(r, v) == LTC_MP_EQ) {
75      *stat = 1;
76  }
77
78  err = CRYPT_OK;
79  goto done;
80
81 error :
82 done : mp_clear_multi(w, v, u1, u2, NULL);
83 return err;
84 }

```

## 5.252 pk/dsa/dsa\_verify\_key.c File Reference

### 5.252.1 Detailed Description

DSA implementation, verify a key, Tom St Denis.

Definition in file [dsa\\_verify\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for dsa\_verify\_key.c:

### Functions

- [int dsa\\_verify\\_key](#) (dsa\_key \*key, int \*stat)  
*Verify a DSA key for validity.*

### 5.252.2 Function Documentation

#### 5.252.2.1 int dsa\_verify\_key (dsa\_key \*key, int \*stat)

Verify a DSA key for validity.

#### Parameters:

*key* The key to verify

*stat* [out] Result of test, 1==valid, 0==invalid

#### Returns:

CRYPT\_OK if successful

Definition at line 26 of file dsa\_verify\_key.c.

References CRYPT\_OK, LTC\_ARGCHK, LTC\_MP\_EQ, LTC\_MP\_GT, LTC\_MP\_LT, and LTC\_MP\_YES.

```
27 {
28     void    *tmp, *tmp2;
29     int      res, err;
30
31     LTC_ARGCHK(key != NULL);
32     LTC_ARGCHK(stat != NULL);
33
34     /* default to an invalid key */
35     *stat = 0;
36
37     /* first make sure key->q and key->p are prime */
38     if ((err = mp_prime_is_prime(key->q, 8, &res)) != CRYPT_OK) {
39         return err;
40     }
41     if (res == 0) {
42         return CRYPT_OK;
43     }
44
45     if ((err = mp_prime_is_prime(key->p, 8, &res)) != CRYPT_OK) {
46         return err;
47     }
```

```
48     if (res == 0) {
49         return CRYPT_OK;
50     }
51
52     /* now make sure that g is not -1, 0 or 1 and <p */
53     if (mp_cmp_d(key->g, 0) == LTC_MP_EQ || mp_cmp_d(key->g, 1) == LTC_MP_EQ) {
54         return CRYPT_OK;
55     }
56     if ((err = mp_init_multi(&tmp, &tmp2, NULL)) != CRYPT_OK) { goto error; }
57     if ((err = mp_sub_d(key->p, 1, tmp)) != CRYPT_OK) { goto error; }
58     if (mp_cmp(tmp, key->g) == LTC_MP_EQ || mp_cmp(key->g, key->p) != LTC_MP_LT) {
59         err = CRYPT_OK;
60         goto done;
61     }
62
63     /* 1 < y < p-1 */
64     if (!(mp_cmp_d(key->y, 1) == LTC_MP_GT && mp_cmp(key->y, tmp) == LTC_MP_LT)) {
65         err = CRYPT_OK;
66         goto done;
67     }
68
69     /* now we have to make sure that g^q = 1, and that p-1/q gives 0 remainder */
70     if ((err = mp_div(tmp, key->q, tmp, tmp2)) != CRYPT_OK) { goto error; }
71     if (mp_iszero(tmp2) != LTC_MP_YES) {
72         err = CRYPT_OK;
73         goto done;
74     }
75
76     if ((err = mp_exptmod(key->g, key->q, key->p, tmp)) != CRYPT_OK) { goto error; }
77     if (mp_cmp_d(tmp, 1) != LTC_MP_EQ) {
78         err = CRYPT_OK;
79         goto done;
80     }
81
82     /* now we have to make sure that y^q = 1, this makes sure y \in g^x mod p */
83     if ((err = mp_exptmod(key->y, key->q, key->p, tmp)) != CRYPT_OK) { goto error; }
84     if (mp_cmp_d(tmp, 1) != LTC_MP_EQ) {
85         err = CRYPT_OK;
86         goto done;
87     }
88
89     /* at this point we are out of tests ;-( */
90     err = CRYPT_OK;
91     *stat = 1;
92     goto done;
93 error:
94 done : mp_clear_multi(tmp, tmp2, NULL);
95     return err;
96 }
```



## 5.253 pk/ecc/ecc.c File Reference

### 5.253.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc.c:

### Variables

- const ltc\_ecc\_set\_type [ltc\\_ecc\\_sets](#) []

### 5.253.2 Variable Documentation

#### 5.253.2.1 const ltc\_ecc\_set\_type [ltc\\_ecc\\_sets](#) []

Definition at line 27 of file ecc.c.

Referenced by [ecc\\_ansi\\_x963\\_export\(\)](#), [ecc\\_ansi\\_x963\\_import\(\)](#), [ecc\\_export\(\)](#), [ecc\\_get\\_size\(\)](#), [ecc\\_import\(\)](#), [ecc\\_make\\_key\(\)](#), [ecc\\_shared\\_secret\(\)](#), [ecc\\_sign\\_hash\(\)](#), [ecc\\_sizes\(\)](#), [ecc\\_test\(\)](#), [ecc\\_verify\\_hash\(\)](#), [is\\_point\(\)](#), and [ltc\\_ecc\\_is\\_valid\\_idx\(\)](#).

## 5.254 pk/ecc/ecc\_ansi\_x963\_export.c File Reference

### 5.254.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_ansi\\_x963\\_export.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_ansi\_x963\_export.c:

### Functions

- [int ecc\\_ansi\\_x963\\_export](#) (ecc\_key \*key, unsigned char \*out, unsigned long \*outlen)  
*ECC X9.63 (Sec.*

### 5.254.2 Function Documentation

#### 5.254.2.1 int ecc\_ansi\_x963\_export (ecc\_key \*key, unsigned char \*out, unsigned long \*outlen)

ECC X9.63 (Sec.

4.3.6) uncompressed export

#### Parameters:

*key* Key to export

*out* [out] destination of export

*outlen* [in/out] Length of destination and final output size Return CRYPT\_OK on success

Definition at line 32 of file ecc\_ansi\_x963\_export.c.

References [CRYPT\\_BUFFER\\_OVERFLOW](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), [ltc\\_ecc\\_is\\_valid\\_idx\(\)](#), [ltc\\_ecc\\_sets](#), [XMEMCPY](#), and [zeromem\(\)](#).

```
33 {
34     unsigned char buf[128];
35     unsigned long numlen;
36
37     LTC_ARGCHK(key    != NULL);
38     LTC_ARGCHK(out    != NULL);
39     LTC_ARGCHK(outlen != NULL);
40
41     if (ltc_ecc_is_valid_idx(key->idx) == 0) {
42         return CRYPT_INVALID_ARG;
43     }
44     numlen = ltc_ecc_sets[key->idx].size;
45
46     if (*outlen < (1 + 2*numlen)) {
47         *outlen = 1 + 2*numlen;
48         return CRYPT_BUFFER_OVERFLOW;
49     }
50
51     /* store byte 0x04 */
52     out[0] = 0x04;
53
54     /* pad and store x */
```

```
55     zeromem(buf, sizeof(buf));
56     mp_to_unsigned_bin(key->pubkey.x, buf + (numlen - mp_unsigned_bin_size(key->pubkey.x)));
57     XMEMCPY(out+1, buf, numlen);
58
59     /* pad and store y */
60     zeromem(buf, sizeof(buf));
61     mp_to_unsigned_bin(key->pubkey.y, buf + (numlen - mp_unsigned_bin_size(key->pubkey.y)));
62     XMEMCPY(out+1+numlen, buf, numlen);
63
64     *outlen = 1 + 2*numlen;
65     return CRYPT_OK;
66 }
```

Here is the call graph for this function:

## 5.255 pk/ecc/ecc\_ansi\_x963\_import.c File Reference

### 5.255.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_ansi\\_x963\\_import.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_ansi\_x963\_import.c:

### Functions

- `int ecc_ansi_x963_import` (const unsigned char \**in*, unsigned long *inlen*, ecc\_key \**key*)  
*Import an ANSI X9.63 format public key.*

### 5.255.2 Function Documentation

#### 5.255.2.1 int ecc\_ansi\_x963\_import (const unsigned char \* *in*, unsigned long *inlen*, ecc\_key \* *key*)

Import an ANSI X9.63 format public key.

#### Parameters:

- in* The input data to read
- inlen* The length of the input data
- key* [out] destination to store imported key \

Definition at line 31 of file ecc\_ansi\_x963\_import.c.

References `CRYPT_INVALID_ARG`, `CRYPT_INVALID_PACKET`, `CRYPT_MEM`, `CRYPT_OK`, `LTC_ARGCHK`, `ltc_ecc_sets`, and `edge::size`.

```
32 {
33     int x, err;
34
35     LTC_ARGCHK(in != NULL);
36     LTC_ARGCHK(key != NULL);
37
38     /* must be odd */
39     if ((inlen & 1) == 0) {
40         return CRYPT_INVALID_ARG;
41     }
42
43     /* init key */
44     if (mp_init_multi(&key->pubkey.x, &key->pubkey.y, &key->pubkey.z, &key->k, NULL) != CRYPT_OK) {
45         return CRYPT_MEM;
46     }
47
48     /* check for 4, 6 or 7 */
49     if (in[0] != 4 && in[0] != 6 && in[0] != 7) {
50         err = CRYPT_INVALID_PACKET;
51         goto error;
52     }
53
54     /* read data */
```

```
55     if ((err = mp_read_unsigned_bin(key->pubkey.x, (unsigned char *)in+1, (inlen-1)>>1)) != CRYPT_OK) {
56         goto error;
57     }
58
59     if ((err = mp_read_unsigned_bin(key->pubkey.y, (unsigned char *)in+1+((inlen-1)>>1), (inlen-1)>>1)) != CRYPT_OK) {
60         goto error;
61     }
62     mp_set(key->pubkey.z, 1);
63
64     /* determine the idx */
65     for (x = 0; ltc_ecc_sets[x].size != 0; x++) {
66         if ((unsigned)ltc_ecc_sets[x].size >= ((inlen-1)>>1)) {
67             break;
68         }
69     }
70     if (ltc_ecc_sets[x].size == 0) {
71         err = CRYPT_INVALID_PACKET;
72         goto error;
73     }
74
75     /* set the idx */
76     key->idx = x;
77     key->type = PK_PUBLIC;
78
79     /* we're done */
80     return CRYPT_OK;
81 error:
82     mp_clear_multi(key->pubkey.x, key->pubkey.y, key->pubkey.z, key->k, NULL);
83     return err;
84 }
```

## 5.256 pk/ecc/ecc\_decrypt\_key.c File Reference

### 5.256.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_decrypt\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_decrypt\_key.c:

### Functions

- `int ecc_decrypt_key` (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, ecc\_key \**key*)

*Decrypt an ECC encrypted key.*

### 5.256.2 Function Documentation

#### 5.256.2.1 `int ecc_decrypt_key` (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, ecc\_key \**key*)

Decrypt an ECC encrypted key.

#### Parameters:

*in* The ciphertext

*inlen* The length of the ciphertext (octets)

*out* [out] The plaintext

*outlen* [in/out] The max size and resulting size of the plaintext

*key* The corresponding private ECC key

#### Returns:

CRYPT\_OK if successful

Definition at line 35 of file ecc\_decrypt\_key.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_INVALID\_PACKET, CRYPT\_MEM, CRYPT\_OK, CRYPT\_PK\_NOT\_PRIVATE, ecc\_free(), ecc\_import(), ecc\_shared\_secret(), find\_hash\_oid(), hash\_is\_valid(), hash\_memory(), LTC\_ARGCHK, MAXBLOCKSIZE, MIN, PK\_PRIVATE, edge::size, XFREE, and XMALLOC.

```
38 {
39     unsigned char *ecc_shared, *skey, *pub_expt;
40     unsigned long x, y, hashOID[32];
41     int          hash, err;
42     ecc_key      pubkey;
43     ltc_asn1_list decode[3];
44
45     LTC_ARGCHK(in      != NULL);
46     LTC_ARGCHK(out     != NULL);
47     LTC_ARGCHK(outlen  != NULL);
48     LTC_ARGCHK(key     != NULL);
```

```
49
50  /* right key type? */
51  if (key->type != PK_PRIVATE) {
52      return CRYPT_PK_NOT_PRIVATE;
53  }
54
55  /* decode to find out hash */
56  LTC_SET_ASN1(decode, 0, LTC_ASN1_OBJECT_IDENTIFIER, hashOID, sizeof(hashOID)/sizeof(hashOID[0]));
57
58  if ((err = der_decode_sequence(in, inlen, decode, 1)) != CRYPT_OK) {
59      return err;
60  }
61
62  hash = find_hash_oid(hashOID, decode[0].size);
63  if (hash_is_valid(hash) != CRYPT_OK) {
64      return CRYPT_INVALID_PACKET;
65  }
66
67  /* we now have the hash! */
68
69  /* allocate memory */
70  pub_expt = XMALLOC(ECC_BUF_SIZE);
71  ecc_shared = XMALLOC(ECC_BUF_SIZE);
72  skey = XMALLOC(MAXBLOCKSIZE);
73  if (pub_expt == NULL || ecc_shared == NULL || skey == NULL) {
74      if (pub_expt != NULL) {
75          XFREE(pub_expt);
76      }
77      if (ecc_shared != NULL) {
78          XFREE(ecc_shared);
79      }
80      if (skey != NULL) {
81          XFREE(skey);
82      }
83      return CRYPT_MEM;
84  }
85  LTC_SET_ASN1(decode, 1, LTC_ASN1_OCTET_STRING, pub_expt, ECC_BUF_SIZE);
86  LTC_SET_ASN1(decode, 2, LTC_ASN1_OCTET_STRING, skey, MAXBLOCKSIZE);
87
88  /* read the structure in now */
89  if ((err = der_decode_sequence(in, inlen, decode, 3)) != CRYPT_OK) {
90      goto LBL_ERR;
91  }
92
93  /* import ECC key from packet */
94  if ((err = ecc_import(decode[1].data, decode[1].size, &pubkey)) != CRYPT_OK) {
95      goto LBL_ERR;
96  }
97
98  /* make shared key */
99  x = ECC_BUF_SIZE;
100  if ((err = ecc_shared_secret(key, &pubkey, ecc_shared, &x)) != CRYPT_OK) {
101      ecc_free(&pubkey);
102      goto LBL_ERR;
103  }
104  ecc_free(&pubkey);
105
106  y = MIN(ECC_BUF_SIZE, MAXBLOCKSIZE);
107  if ((err = hash_memory(hash, ecc_shared, x, ecc_shared, &y)) != CRYPT_OK) {
108      goto LBL_ERR;
109  }
110
111  /* ensure the hash of the shared secret is at least as big as the encrypt itself */
112  if (decode[2].size > y) {
113      err = CRYPT_INVALID_PACKET;
114      goto LBL_ERR;
115  }
```

```
116
117     /* avoid buffer overflow */
118     if (*outlen < decode[2].size) {
119         *outlen = decode[2].size;
120         err = CRYPT_BUFFER_OVERFLOW;
121         goto LBL_ERR;
122     }
123
124     /* Decrypt the key */
125     for (x = 0; x < decode[2].size; x++) {
126         out[x] = skey[x] ^ ecc_shared[x];
127     }
128     *outlen = x;
129
130     err = CRYPT_OK;
131 LBL_ERR:
132 #ifdef LTC_CLEAN_STACK
133     zeromem(pub_expt, ECC_BUF_SIZE);
134     zeromem(ecc_shared, ECC_BUF_SIZE);
135     zeromem(skey, MAXBLOCKSIZE);
136 #endif
137
138     XFREE(pub_expt);
139     XFREE(ecc_shared);
140     XFREE(skey);
141
142     return err;
143 }
```

Here is the call graph for this function:



## 5.257 pk/ecc/ecc\_encrypt\_key.c File Reference

### 5.257.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_encrypt\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_encrypt\_key.c:

### Functions

- `int ecc_encrypt_key` (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, [prng\\_state](#) \**prng*, int *wprng*, int *hash*, ecc\_key \**key*)

*Encrypt a symmetric key with ECC.*

### 5.257.2 Function Documentation

- 5.257.2.1** `int ecc_encrypt_key` (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, [prng\\_state](#) \**prng*, int *wprng*, int *hash*, ecc\_key \**key*)

Encrypt a symmetric key with ECC.

#### Parameters:

- in* The symmetric key you want to encrypt
- inlen* The length of the key to encrypt (octets)
- out* [out] The destination for the ciphertext
- outlen* [in/out] The max size and resulting size of the ciphertext
- prng* An active PRNG state
- wprng* The index of the PRNG you wish to use
- hash* The index of the hash you want to use
- key* The ECC key you want to encrypt to

#### Returns:

- CRYPT\_OK if successful

Definition at line 38 of file ecc\_encrypt\_key.c.

References CRYPT\_INVALID\_HASH, CRYPT\_MEM, CRYPT\_OK, ecc\_export(), ecc\_free(), ecc\_get\_size(), ecc\_make\_key(), ecc\_shared\_secret(), hash\_descriptor, hash\_is\_valid(), hash\_memory(), LTC\_ARGCHK, MAXBLOCKSIZE, PK\_PUBLIC, prng\_is\_valid(), XFREE, and XMALLOC.

```
42 {
43     unsigned char *pub_expt, *ecc_shared, *skey;
44     ecc_key      pubkey;
45     unsigned long x, y, pubkeysize;
46     int          err;
47
48     LTC_ARGCHK(in != NULL);
```

```

49     LTC_ARGCHK(out      != NULL);
50     LTC_ARGCHK(outlen   != NULL);
51     LTC_ARGCHK(key      != NULL);
52
53     /* check that wprng/cipher/hash are not invalid */
54     if ((err = prng_is_valid(wprng)) != CRYPT_OK) {
55         return err;
56     }
57
58     if ((err = hash_is_valid(hash)) != CRYPT_OK) {
59         return err;
60     }
61
62     if (inlen > hash_descriptor[hash].hashsize) {
63         return CRYPT_INVALID_HASH;
64     }
65
66     /* make a random key and export the public copy */
67     if ((err = ecc_make_key(prng, wprng, ecc_get_size(key), &pubkey)) != CRYPT_OK) {
68         return err;
69     }
70
71     pub_expt = XMALLOC(ECC_BUF_SIZE);
72     ecc_shared = XMALLOC(ECC_BUF_SIZE);
73     skey = XMALLOC(MAXBLOCKSIZE);
74     if (pub_expt == NULL || ecc_shared == NULL || skey == NULL) {
75         if (pub_expt != NULL) {
76             XFREE(pub_expt);
77         }
78         if (ecc_shared != NULL) {
79             XFREE(ecc_shared);
80         }
81         if (skey != NULL) {
82             XFREE(skey);
83         }
84         ecc_free(&pubkey);
85         return CRYPT_MEM;
86     }
87
88     pubkeysize = ECC_BUF_SIZE;
89     if ((err = ecc_export(pub_expt, &pubkeysize, PK_PUBLIC, &pubkey)) != CRYPT_OK) {
90         ecc_free(&pubkey);
91         goto LBL_ERR;
92     }
93
94     /* make random key */
95     x = ECC_BUF_SIZE;
96     if ((err = ecc_shared_secret(&pubkey, key, ecc_shared, &x)) != CRYPT_OK) {
97         ecc_free(&pubkey);
98         goto LBL_ERR;
99     }
100     ecc_free(&pubkey);
101     y = MAXBLOCKSIZE;
102     if ((err = hash_memory(hash, ecc_shared, x, skey, &y)) != CRYPT_OK) {
103         goto LBL_ERR;
104     }
105
106     /* Encrypt key */
107     for (x = 0; x < inlen; x++) {
108         skey[x] ^= in[x];
109     }
110
111     err = der_encode_sequence_multi(out, outlen,
112                                     LTC_ASN1_OBJECT_IDENTIFIER, hash_descriptor[hash].OIDlen, hash_
113                                     LTC_ASN1_OCTET_STRING, pubkeysize, pub_e
114                                     LTC_ASN1_OCTET_STRING, inlen, skey,
115                                     LTC_ASN1_EOL, 0UL, NULL)

```

```
116
117 LBL_ERR:
118 #ifdef LTC_CLEAN_STACK
119     /* clean up */
120     zeromem(pub_expt, ECC_BUF_SIZE);
121     zeromem(ecc_shared, ECC_BUF_SIZE);
122     zeromem(skey, MAXBLOCKSIZE);
123 #endif
124
125     XFREE(skey);
126     XFREE(ecc_shared);
127     XFREE(pub_expt);
128
129     return err;
130 }
```

Here is the call graph for this function:

## 5.258 pk/ecc/ecc\_export.c File Reference

### 5.258.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_export.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_export.c:

### Functions

- `int ecc_export` (unsigned char \*out, unsigned long \*outlen, int type, ecc\_key \*key)  
*Export an ECC key as a binary packet.*

### 5.258.2 Function Documentation

#### 5.258.2.1 int ecc\_export (unsigned char \* out, unsigned long \* outlen, int type, ecc\_key \* key)

Export an ECC key as a binary packet.

#### Parameters:

- out** [out] Destination for the key
- outlen** [in/out] Max size and resulting size of the exported key
- type** The type of key you want to export (PK\_PRIVATE or PK\_PUBLIC)
- key** The key to export

#### Returns:

CRYPT\_OK if successful

Definition at line 34 of file ecc\_export.c.

References CRYPT\_INVALID\_ARG, CRYPT\_PK\_TYPE\_MISMATCH, der\_encode\_sequence\_multi(), LTC\_ARGCHK, ltc\_ecc\_is\_valid\_idx(), ltc\_ecc\_sets, and PK\_PRIVATE.

Referenced by ecc\_encrypt\_key().

```

35 {
36     int          err;
37     unsigned char flags[1];
38     unsigned long key_size;
39
40     LTC_ARGCHK(out    != NULL);
41     LTC_ARGCHK(outlen != NULL);
42     LTC_ARGCHK(key    != NULL);
43
44     /* type valid? */
45     if (key->type != PK_PRIVATE && type == PK_PRIVATE) {
46         return CRYPT_PK_TYPE_MISMATCH;
47     }
48
49     if (ltc_ecc_is_valid_idx(key->idx) == 0) {
50         return CRYPT_INVALID_ARG;

```

```
51     }
52
53     /* we store the NIST byte size */
54     key_size = ltc_ecc_sets[key->idx].size;
55
56     if (type == PK_PRIVATE) {
57         flags[0] = 1;
58         err = der_encode_sequence_multi(out, outlen,
59                                     LTC_ASN1_BIT_STRING,      1UL, flags,
60                                     LTC_ASN1_SHORT_INTEGER,   1UL, &key_size,
61                                     LTC_ASN1_INTEGER,          1UL, key->pubkey.x,
62                                     LTC_ASN1_INTEGER,          1UL, key->pubkey.y,
63                                     LTC_ASN1_INTEGER,          1UL, key->k,
64                                     LTC_ASN1_EOL,              0UL, NULL);
65     } else {
66         flags[0] = 0;
67         err = der_encode_sequence_multi(out, outlen,
68                                     LTC_ASN1_BIT_STRING,      1UL, flags,
69                                     LTC_ASN1_SHORT_INTEGER,   1UL, &key_size,
70                                     LTC_ASN1_INTEGER,          1UL, key->pubkey.x,
71                                     LTC_ASN1_INTEGER,          1UL, key->pubkey.y,
72                                     LTC_ASN1_EOL,              0UL, NULL);
73     }
74
75     return err;
76 }
```

Here is the call graph for this function:

## 5.259 pk/ecc/ecc\_free.c File Reference

### 5.259.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_free.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_free.c:

### Functions

- void [ecc\\_free](#) (ecc\_key \*key)  
*Free an ECC key from memory.*

### 5.259.2 Function Documentation

#### 5.259.2.1 void ecc\_free (ecc\_key \*key)

Free an ECC key from memory.

#### Parameters:

*key* The key you wish to free

Definition at line 30 of file ecc\_free.c.

References LTC\_ARGCHKVD.

Referenced by ecc\_decrypt\_key(), ecc\_encrypt\_key(), and ecc\_sign\_hash().

```
31 {  
32     LTC_ARGCHKVD(key != NULL);  
33     mp_clear_multi(key->pubkey.x, key->pubkey.y, key->pubkey.z, key->k, NULL);  
34 }
```

## 5.260 pk/ecc/ecc\_get\_size.c File Reference

### 5.260.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_get\\_size.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_get\_size.c:

### Functions

- [int ecc\\_get\\_size](#) (ecc\_key \*key)  
*Get the size of an ECC key.*

### 5.260.2 Function Documentation

#### 5.260.2.1 int ecc\_get\_size (ecc\_key \* key)

Get the size of an ECC key.

#### Parameters:

*key* The key to get the size of

#### Returns:

The size (octets) of the key or INT\_MAX on error

Definition at line 31 of file ecc\_get\_size.c.

References LTC\_ARGCHK, ltc\_ecc\_is\_valid\_idx(), and ltc\_ecc\_sets.

Referenced by ecc\_encrypt\_key(), and ecc\_sign\_hash().

```
32 {  
33     LTC_ARGCHK(key != NULL);  
34     if (ltc_ecc_is_valid_idx(key->idx))  
35         return ltc_ecc_sets[key->idx].size;  
36     else  
37         return INT_MAX; /* large value known to cause it to fail when passed to ecc_make_key() */  
38 }
```

Here is the call graph for this function:

## 5.261 pk/ecc/ecc\_import.c File Reference

### 5.261.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_import.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_import.c:

### Functions

- static int [is\\_point](#) (ecc\_key \*key)
- int [ecc\\_import](#) (const unsigned char \*in, unsigned long inlen, ecc\_key \*key)  
*Import an ECC key from a binary packet.*

### 5.261.2 Function Documentation

#### 5.261.2.1 int ecc\_import (const unsigned char \*in, unsigned long inlen, ecc\_key \*key)

Import an ECC key from a binary packet.

#### Parameters:

- in** The packet to import  
**inlen** The length of the packet  
**key** [out] The destination of the import

#### Returns:

CRYPT\_OK if successful, upon error all allocated memory will be freed

Definition at line 81 of file ecc\_import.c.

References [CRYPT\\_INVALID\\_PACKET](#), [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [der\\_decode\\_sequence\\_multi\(\)](#), [is\\_point\(\)](#), [LTC\\_ARGCHK](#), [ltc\\_ecc\\_sets](#), [ltc\\_mp](#), [ltc\\_math\\_descriptor::name](#), [PK\\_PRIVATE](#), and [PK\\_PUBLIC](#).

Referenced by [ecc\\_decrypt\\_key\(\)](#).

```
82 {
83     unsigned long key_size;
84     unsigned char flags[1];
85     int          err;
86
87     LTC_ARGCHK(in != NULL);
88     LTC_ARGCHK(key != NULL);
89     LTC_ARGCHK(ltc_mp.name != NULL);
90
91     /* init key */
92     if (mp_init_multi(&key->pubkey.x, &key->pubkey.y, &key->pubkey.z, &key->k, NULL) != CRYPT_OK) {
93         return CRYPT_MEM;
94     }
95
96     /* find out what type of key it is */
```



```

97     if ((err = der_decode_sequence_multi(in, inlen,
98                                         LTC_ASN1_BIT_STRING, 1UL, &flags,
99                                         LTC_ASN1_EOL, 0UL, NULL)) != CRYPT_OK) {
100         goto done;
101     }
102
103
104     if (flags[0] == 1) {
105         /* private key */
106         key->type = PK_PRIVATE;
107         if ((err = der_decode_sequence_multi(in, inlen,
108                                             LTC_ASN1_BIT_STRING, 1UL, flags,
109                                             LTC_ASN1_SHORT_INTEGER, 1UL, &key_size,
110                                             LTC_ASN1_INTEGER, 1UL, key->pubkey.x,
111                                             LTC_ASN1_INTEGER, 1UL, key->pubkey.y,
112                                             LTC_ASN1_INTEGER, 1UL, key->k,
113                                             LTC_ASN1_EOL, 0UL, NULL)) != CRYPT_OK) {
114             goto done;
115         }
116     } else {
117         /* public key */
118         key->type = PK_PUBLIC;
119         if ((err = der_decode_sequence_multi(in, inlen,
120                                             LTC_ASN1_BIT_STRING, 1UL, flags,
121                                             LTC_ASN1_SHORT_INTEGER, 1UL, &key_size,
122                                             LTC_ASN1_INTEGER, 1UL, key->pubkey.x,
123                                             LTC_ASN1_INTEGER, 1UL, key->pubkey.y,
124                                             LTC_ASN1_EOL, 0UL, NULL)) != CRYPT_OK) {
125             goto done;
126         }
127     }
128
129     /* find the idx */
130     for (key->idx = 0; ltc_ecc_sets[key->idx].size && (unsigned long)ltc_ecc_sets[key->idx].size != key->size; key->idx++)
131         if (ltc_ecc_sets[key->idx].size == 0) {
132             err = CRYPT_INVALID_PACKET;
133             goto done;
134         }
135
136     /* set z */
137     mp_set(key->pubkey.z, 1);
138
139     /* is it a point on the curve? */
140     if ((err = is_point(key)) != CRYPT_OK) {
141         goto done;
142     }
143
144     /* we're good */
145     return CRYPT_OK;
146 done:
147     mp_clear_multi(key->pubkey.x, key->pubkey.y, key->pubkey.z, key->k, NULL);
148     return err;
149 }

```

Here is the call graph for this function:

### 5.261.2.2 static int is\_point (ecc\_key \*key) [static]

Definition at line 26 of file ecc\_import.c.

References CRYPT\_INVALID\_PACKET, CRYPT\_OK, ltc\_ecc\_sets, LTC\_MP\_EQ, LTC\_MP\_LT, t1, and t2.

Referenced by ecc\_import().

```

27 {
28     void *prime, *b, *t1, *t2;
29     int err;
30
31     if ((err = mp_init_multi(&prime, &b, &t1, &t2, NULL)) != CRYPT_OK) {
32         return err;
33     }
34
35     /* load prime and b */
36     if ((err = mp_read_radix(prime, ltc_ecc_sets[key->idx].prime, 16)) != CRYPT_OK) { goto error; }
37     if ((err = mp_read_radix(b, ltc_ecc_sets[key->idx].B, 16)) != CRYPT_OK) { goto error; }
38
39     /* compute y^2 */
40     if ((err = mp_sqr(key->pubkey.y, t1)) != CRYPT_OK) { goto error; }
41
42     /* compute x^3 */
43     if ((err = mp_sqr(key->pubkey.x, t2)) != CRYPT_OK) { goto error; }
44     if ((err = mp_mod(t2, prime, t2)) != CRYPT_OK) { goto error; }
45     if ((err = mp_mul(key->pubkey.x, t2, t2)) != CRYPT_OK) { goto error; }
46
47     /* compute y^2 - x^3 */
48     if ((err = mp_sub(t1, t2, t1)) != CRYPT_OK) { goto error; }
49
50     /* compute y^2 - x^3 + 3x */
51     if ((err = mp_add(t1, key->pubkey.x, t1)) != CRYPT_OK) { goto error; }
52     if ((err = mp_add(t1, key->pubkey.x, t1)) != CRYPT_OK) { goto error; }
53     if ((err = mp_add(t1, key->pubkey.x, t1)) != CRYPT_OK) { goto error; }
54     if ((err = mp_mod(t1, prime, t1)) != CRYPT_OK) { goto error; }
55     while (mp_cmp_d(t1, 0) == LTC_MP_LT) {
56         if ((err = mp_add(t1, prime, t1)) != CRYPT_OK) { goto error; }
57     }
58     while (mp_cmp(t1, prime) != LTC_MP_LT) {
59         if ((err = mp_sub(t1, prime, t1)) != CRYPT_OK) { goto error; }
60     }
61
62     /* compare to b */
63     if (mp_cmp(t1, b) != LTC_MP_EQ) {
64         err = CRYPT_INVALID_PACKET;
65     } else {
66         err = CRYPT_OK;
67     }
68
69 error:
70     mp_clear_multi(prime, b, t1, t2, NULL);
71     return err;
72 }

```

## 5.262 pk/ecc/ecc\_make\_key.c File Reference

### 5.262.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_make\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_make\_key.c:

### Functions

- `int ecc_make_key(prng\_state *prng, int wprng, int keysize, ecc_key *key)`  
*Make a new ECC key.*

### 5.262.2 Function Documentation

#### 5.262.2.1 `int ecc_make_key(prng\_state *prng, int wprng, int keysize, ecc_key *key)`

Make a new ECC key.

#### Parameters:

- prng* An active PRNG state
- wprng* The index of the PRNG you wish to use
- keysize* The keysize for the new key (in octets from 20 to 65 bytes)
- key* [out] Destination of the newly created key

#### Returns:

CRYPT\_OK if successful, upon error all allocated memory will be freed

Definition at line 34 of file ecc\_make\_key.c.

References `CRYPT_INVALID_KEYSIZE`, `CRYPT_OK`, `LTC_ARGCHK`, `ltc_ecc_sets`, `ltc_mp`, `ltc_math_descriptor::name`, `prng_is_valid()`, and `edge::size`.

Referenced by `ecc_encrypt_key()`, and `ecc_sign_hash()`.

```
35 {
36     int            x, err;
37     ecc_point      *base;
38     void           *prime;
39     unsigned char *buf;
40
41     LTC_ARGCHK(key != NULL);
42     LTC_ARGCHK(ltc_mp.name != NULL);
43
44     /* good prng? */
45     if ((err = prng_is_valid(wprng)) != CRYPT_OK) {
46         return err;
47     }
48
49     /* find key size */
50     for (x = 0; (keysize > ltc_ecc_sets[x].size) && (ltc_ecc_sets[x].size != 0); x++);
```

```

51     keysize = ltc_ecc_sets[x].size;
52
53     if (keysize > ECC_MAXSIZE || ltc_ecc_sets[x].size == 0) {
54         return CRYPT_INVALID_KEYSIZE;
55     }
56     key->idx = x;
57
58     /* allocate ram */
59     base = NULL;
60     buf = XMALLOC(ECC_MAXSIZE);
61     if (buf == NULL) {
62         return CRYPT_MEM;
63     }
64
65     /* make up random string */
66     if (prng_descriptor[wprng].read(buf, (unsigned long)keysize, prng) != (unsigned long)keysize) {
67         err = CRYPT_ERROR_READPRNG;
68         goto LBL_ERR2;
69     }
70
71     /* setup the key variables */
72     if ((err = mp_init_multi(&key->pubkey.x, &key->pubkey.y, &key->pubkey.z, &key->k, &prime, NULL)) !=
73         goto done;
74     }
75     base = ltc_ecc_new_point();
76     if (base == NULL) {
77         mp_clear_multi(key->pubkey.x, key->pubkey.y, key->pubkey.z, key->k, prime, NULL);
78         err = CRYPT_MEM;
79         goto done;
80     }
81
82     /* read in the specs for this key */
83     if ((err = mp_read_radix(prime, (char *)ltc_ecc_sets[key->idx].prime, 16)) != CRYPT_OK) { goto
84     if ((err = mp_read_radix(base->x, (char *)ltc_ecc_sets[key->idx].Gx, 16)) != CRYPT_OK) { goto
85     if ((err = mp_read_radix(base->y, (char *)ltc_ecc_sets[key->idx].Gy, 16)) != CRYPT_OK) { goto
86     mp_set(base->z, 1);
87     if ((err = mp_read_unsigned_bin(key->k, (unsigned char *)buf, keysize)) != CRYPT_OK) { goto
88
89     /* make the public key */
90     if ((err = ltc_mp.ecc_ptmul(key->k, base, &key->pubkey, prime, 1)) != CRYPT_OK) { goto
91     key->type = PK_PRIVATE;
92
93     /* free up ram */
94     err = CRYPT_OK;
95 done:
96     ltc_ecc_del_point(base);
97     mp_clear(prime);
98 LBL_ERR2:
99 #ifdef LTC_CLEAN_STACK
100     zeromem(buf, ECC_MAXSIZE);
101 #endif
102
103     XFREE(buf);
104
105     return err;
106 }

```

Here is the call graph for this function:

## 5.263 pk/ecc/ecc\_shared\_secret.c File Reference

### 5.263.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_shared\\_secret.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_shared\_secret.c:

### Functions

- `int ecc_shared_secret` (`ecc_key *private_key`, `ecc_key *public_key`, `unsigned char *out`, `unsigned long *outlen`)

*Create an ECC shared secret between two keys.*

### 5.263.2 Function Documentation

#### 5.263.2.1 `int ecc_shared_secret` (`ecc_key *private_key`, `ecc_key *public_key`, `unsigned char *out`, `unsigned long *outlen`)

Create an ECC shared secret between two keys.

#### Parameters:

*private\_key* The private ECC key

*public\_key* The public key

*out* [out] Destination of the shared secret (Conforms to EC-DH from ANSI X9.63)

*outlen* [in/out] The max size and resulting size of the shared secret

#### Returns:

CRYPT\_OK if successful

Definition at line 34 of file ecc\_shared\_secret.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_INVALID\_ARG, CRYPT\_MEM, CRYPT\_OK, CRYPT\_PK\_NOT\_PRIVATE, CRYPT\_PK\_TYPE\_MISMATCH, ltc\_math\_descriptor::ecc\_ptmul, LTC\_ARGCHK, ltc\_ecc\_del\_point(), ltc\_ecc\_is\_valid\_idx(), ltc\_ecc\_new\_point(), ltc\_ecc\_sets, ltc\_mp, PK\_PRIVATE, and zeromem().

Referenced by ecc\_decrypt\_key(), and ecc\_encrypt\_key().

```
36 {
37     unsigned long x;
38     ecc_point *result;
39     void *prime;
40     int err;
41
42     LTC_ARGCHK(private_key != NULL);
43     LTC_ARGCHK(public_key != NULL);
44     LTC_ARGCHK(out != NULL);
45     LTC_ARGCHK(outlen != NULL);
46 }
```

```
47  /* type valid? */
48  if (private_key->type != PK_PRIVATE) {
49      return CRYPT_PK_NOT_PRIVATE;
50  }
51
52  if (ltc_ecc_is_valid_idx(private_key->idx) == 0) {
53      return CRYPT_INVALID_ARG;
54  }
55
56  if (private_key->idx != public_key->idx) {
57      return CRYPT_PK_TYPE_MISMATCH;
58  }
59
60  /* make new point */
61  result = ltc_ecc_new_point();
62  if (result == NULL) {
63      return CRYPT_MEM;
64  }
65
66  if ((err = mp_init(&prime)) != CRYPT_OK) {
67      ltc_ecc_del_point(result);
68      return err;
69  }
70
71  if ((err = mp_read_radix(prime, (char *)ltc_ecc_sets[private_key->idx].prime, 16)) != CRYPT_OK)
72  if ((err = ltc_mp.ecc_ptmul(private_key->k, &public_key->pubkey, result, prime, 1)) != CRYPT_OK)
73
74  x = (unsigned long)mp_unsigned_bin_size(prime);
75  if (*outlen < x) {
76      *outlen = x;
77      err = CRYPT_BUFFER_OVERFLOW;
78      goto done;
79  }
80  zeromem(out, x);
81  if ((err = mp_to_unsigned_bin(result->x, out + (x - mp_unsigned_bin_size(result->x)))) != CRYPT_OK)
82
83  err = CRYPT_OK;
84  *outlen = x;
85 done:
86  mp_clear(prime);
87  ltc_ecc_del_point(result);
88  return err;
89 }
```

Here is the call graph for this function:

## 5.264 pk/ecc/ecc\_sign\_hash.c File Reference

### 5.264.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_sign\\_hash.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_sign\_hash.c:

### Functions

- `int ecc_sign_hash` (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, [prng\\_state](#) \**prng*, int *wprng*, ecc\_key \**key*)

*Sign a message digest.*

### 5.264.2 Function Documentation

**5.264.2.1** `int ecc_sign_hash` (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, [prng\\_state](#) \**prng*, int *wprng*, ecc\_key \**key*)

Sign a message digest.

#### Parameters:

*in* The message digest to sign

*inlen* The length of the digest

*out* [out] The destination for the signature

*outlen* [in/out] The max size and resulting size of the signature

*prng* An active PRNG state

*wprng* The index of the PRNG you wish to use

*key* A private ECC key

#### Returns:

CRYPT\_OK if successful

Definition at line 37 of file ecc\_sign\_hash.c.

References CRYPT\_OK, CRYPT\_PK\_INVALID\_TYPE, CRYPT\_PK\_NOT\_PRIVATE, ecc\_free(), ecc\_get\_size(), ecc\_make\_key(), LTC\_ARGCHK, ltc\_ecc\_is\_valid\_idx(), ltc\_ecc\_sets, PK\_PRIVATE, and prng\_is\_valid().

```
40 {
41     ecc_key      pubkey;
42     void         *r, *s, *e, *p;
43     int          err;
44
45     LTC_ARGCHK(in      != NULL);
46     LTC_ARGCHK(out     != NULL);
47     LTC_ARGCHK(outlen  != NULL);
48     LTC_ARGCHK(key     != NULL);
```

```

49
50  /* is this a private key? */
51  if (key->type != PK_PRIVATE) {
52      return CRYPT_PK_NOT_PRIVATE;
53  }
54
55  /* is the IDX valid ? */
56  if (ltc_ecc_is_valid_idx(key->idx) != 1) {
57      return CRYPT_PK_INVALID_TYPE;
58  }
59
60  if ((err = prng_is_valid(wprng)) != CRYPT_OK) {
61      return err;
62  }
63
64  /* get the hash and load it as a bignum into 'e' */
65  /* init the bignums */
66  if ((err = mp_init_multi(&r, &s, &p, &e, NULL)) != CRYPT_OK) {
67      ecc_free(&pubkey);
68      goto LBL_ERR;
69  }
70  if ((err = mp_read_radix(p, (char *)ltc_ecc_sets[key->idx].order, 16)) != CRYPT_OK) { goto error; }
71  if ((err = mp_read_unsigned_bin(e, (unsigned char *)in, (int)inlen)) != CRYPT_OK) { goto error; }
72
73  /* make up a key and export the public copy */
74  for (;;) {
75      if ((err = ecc_make_key(prng, wprng, ecc_get_size(key), &pubkey)) != CRYPT_OK) {
76          return err;
77      }
78
79      /* find r = x1 mod n */
80      if ((err = mp_mod(pubkey.pubkey.x, p, r)) != CRYPT_OK) { goto error; }
81
82      if (mp_iszero(r)) {
83          ecc_free(&pubkey);
84      } else {
85          /* find s = (e + xr)/k */
86          if ((err = mp_invmod(pubkey.k, p, pubkey.k)) != CRYPT_OK) { goto error; } /* k = 1/k
87          if ((err = mp_mulmod(key->k, r, p, s)) != CRYPT_OK) { goto error; } /* s = xr
88          if ((err = mp_add(e, s, s)) != CRYPT_OK) { goto error; } /* s = e +
89          if ((err = mp_mod(s, p, s)) != CRYPT_OK) { goto error; } /* s = e +
90          if ((err = mp_mulmod(s, pubkey.k, p, s)) != CRYPT_OK) { goto error; } /* s = (e
91
92          if (mp_iszero(s)) {
93              ecc_free(&pubkey);
94          } else {
95              break;
96          }
97      }
98  }
99
100  /* store as SEQUENCE { r, s -- integer } */
101  err = der_encode_sequence_multi(out, outlen,
102                                LTC_ASN1_INTEGER, 1UL, r,
103                                LTC_ASN1_INTEGER, 1UL, s,
104                                LTC_ASN1_EOL, 0UL, NULL);
105  goto LBL_ERR;
106 error:
107 LBL_ERR:
108  mp_clear_multi(r, s, p, e, NULL);
109  ecc_free(&pubkey);
110
111  return err;
112 }

```

Here is the call graph for this function:



## 5.265 pk/ecc/ecc\_sizes.c File Reference

### 5.265.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_sizes.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_sizes.c:

### Functions

- void [ecc\\_sizes](#) (int \*low, int \*high)

### 5.265.2 Function Documentation

#### 5.265.2.1 void ecc\_sizes (int \* *low*, int \* *high*)

Definition at line 26 of file ecc\_sizes.c.

References LTC\_ARGCHKVD, ltc\_ecc\_sets, and edge::size.

```
27 {
28     int i;
29     LTC_ARGCHKVD(low != NULL);
30     LTC_ARGCHKVD(high != NULL);
31
32     *low = INT_MAX;
33     *high = 0;
34     for (i = 0; ltc_ecc_sets[i].size != 0; i++) {
35         if (ltc_ecc_sets[i].size < *low) {
36             *low = ltc_ecc_sets[i].size;
37         }
38         if (ltc_ecc_sets[i].size > *high) {
39             *high = ltc_ecc_sets[i].size;
40         }
41     }
42 }
```

## 5.266 pk/ecc/ecc\_test.c File Reference

### 5.266.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_test.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_test.c:

### Functions

- [int ecc\\_test](#) (void)  
*Perform on the ECC system.*

### 5.266.2 Function Documentation

#### 5.266.2.1 int ecc\_test (void)

Perform on the ECC system.

#### Returns:

CRYPT\_OK if successful

Definition at line 30 of file ecc\_test.c.

References CRYPT\_MEM, CRYPT\_OK, G, GG, ltc\_ecc\_del\_point(), ltc\_ecc\_new\_point(), ltc\_ecc\_sets, and edge::size.

```

31 {
32     void      *modulus, *order;
33     ecc_point  *G, *GG;
34     int i, err, primality;
35
36     if ((err = mp_init_multi(&modulus, &order, NULL)) != CRYPT_OK) {
37         return err;
38     }
39
40     G  = ltc_ecc_new_point();
41     GG = ltc_ecc_new_point();
42     if (G == NULL || GG == NULL) {
43         mp_clear_multi(modulus, order, NULL);
44         ltc_ecc_del_point(G);
45         ltc_ecc_del_point(GG);
46         return CRYPT_MEM;
47     }
48
49     for (i = 0; ltc_ecc_sets[i].size; i++) {
50         #if 0
51             printf("Testing %d\n", ltc_ecc_sets[i].size);
52         #endif
53         if ((err = mp_read_radix(modulus, (char *)ltc_ecc_sets[i].prime, 16)) != CRYPT_OK) { goto done; }
54         if ((err = mp_read_radix(order, (char *)ltc_ecc_sets[i].order, 16)) != CRYPT_OK) { goto done; }
55
56         /* is prime actually prime? */

```

```

57     if ((err = mp_prime_is_prime(modulus, 8, &primality)) != CRYPT_OK)           { goto done;
58     if (primality == 0) {
59         err = CRYPT_FAIL_TESTVECTOR;
60         goto done;
61     }
62
63     /* is order prime ? */
64     if ((err = mp_prime_is_prime(order, 8, &primality)) != CRYPT_OK)           { goto done;
65     if (primality == 0) {
66         err = CRYPT_FAIL_TESTVECTOR;
67         goto done;
68     }
69
70     if ((err = mp_read_radix(G->x, (char *)ltc_ecc_sets[i].Gx, 16)) != CRYPT_OK) { goto done;
71     if ((err = mp_read_radix(G->y, (char *)ltc_ecc_sets[i].Gy, 16)) != CRYPT_OK) { goto done;
72     mp_set(G->z, 1);
73
74     /* then we should have G == (order + 1)G */
75     if ((err = mp_add_d(order, 1, order)) != CRYPT_OK)                         { goto done;
76     if ((err = ltc_mp.ecc_ptmul(order, G, GG, modulus, 1)) != CRYPT_OK)         { goto done;
77     if (mp_cmp(G->x, GG->x) != LTC_MP_EQ || mp_cmp(G->y, GG->y) != LTC_MP_EQ) {
78         err = CRYPT_FAIL_TESTVECTOR;
79         goto done;
80     }
81 }
82     err = CRYPT_OK;
83     goto done;
84 done:
85     ltc_ecc_del_point(GG);
86     ltc_ecc_del_point(G);
87     mp_clear_multi(order, modulus, NULL);
88     return err;
89 }

```

Here is the call graph for this function:

## 5.267 pk/ecc/ecc\_verify\_hash.c File Reference

### 5.267.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ecc\\_verify\\_hash.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ecc\_verify\_hash.c:

### Functions

- [int ecc\\_verify\\_hash](#) (const unsigned char \*sig, unsigned long siglen, const unsigned char \*hash, unsigned long hashlen, int \*stat, ecc\_key \*key)

*Verify an ECC signature.*

### 5.267.2 Function Documentation

#### 5.267.2.1 int ecc\_verify\_hash (const unsigned char \*sig, unsigned long siglen, const unsigned char \*hash, unsigned long hashlen, int \*stat, ecc\_key \*key)

Verify an ECC signature.

#### Parameters:

- sig** The signature to verify
- siglen** The length of the signature (octets)
- hash** The hash (message digest) that was signed
- hashlen** The length of the hash (octets)
- stat** Result of signature, 1==valid, 0==invalid
- key** The corresponding public ECC key

#### Returns:

CRYPT\_OK if successful (even if the signature is not valid)

Definition at line 46 of file ecc\_verify\_hash.c.

References [CRYPT\\_INVALID\\_PACKET](#), [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [CRYPT\\_PK\\_INVALID\\_TYPE](#), [der\\_decode\\_sequence\\_multi\(\)](#), [ltc\\_math\\_descriptor::ecc\\_ptmul](#), [LTC\\_ARGCHK](#), [ltc\\_ecc\\_is\\_valid\\_idx\(\)](#), [ltc\\_ecc\\_new\\_point\(\)](#), [ltc\\_ecc\\_sets](#), [ltc\\_mp](#), and [LTC\\_MP\\_LT](#).

```
49 {
50     ecc_point    *mG, *mQ;
51     void         *r, *s, *v, *w, *u1, *u2, *e, *p, *m;
52     void         *mp;
53     int          err;
54
55     LTC_ARGCHK(sig != NULL);
56     LTC_ARGCHK(hash != NULL);
57     LTC_ARGCHK(stat != NULL);
58     LTC_ARGCHK(key != NULL);
```

```

59
60  /* default to invalid signature */
61  *stat = 0;
62  mp    = NULL;
63
64  /* is the IDX valid ? */
65  if (ltc_ecc_is_valid_idx(key->idx) != 1) {
66      return CRYPT_PK_INVALID_TYPE;
67  }
68
69  /* allocate ints */
70  if ((err = mp_init_multi(&r, &s, &v, &w, &u1, &u2, &p, &e, &m, NULL)) != CRYPT_OK) {
71      return CRYPT_MEM;
72  }
73
74  /* allocate points */
75  mG = ltc_ecc_new_point();
76  mQ = ltc_ecc_new_point();
77  if (mQ == NULL || mG == NULL) {
78      err = CRYPT_MEM;
79      goto done;
80  }
81
82  /* parse header */
83  if ((err = der_decode_sequence_multi(sig, siglen,
84                                     LTC_ASN1_INTEGER, 1UL, r,
85                                     LTC_ASN1_INTEGER, 1UL, s,
86                                     LTC_ASN1_EOL, 0UL, NULL)) != CRYPT_OK) {
87      goto done;
88  }
89
90  /* get the order */
91  if ((err = mp_read_radix(p, (char *)ltc_ecc_sets[key->idx].order, 16)) != CRYPT_OK)
92
93  /* get the modulus */
94  if ((err = mp_read_radix(m, (char *)ltc_ecc_sets[key->idx].prime, 16)) != CRYPT_OK)
95
96  /* check for zero */
97  if (mp_iszero(r) || mp_iszero(s) || mp_cmp(r, p) != LTC_MP_LT || mp_cmp(s, p) != LTC_MP_LT) {
98      err = CRYPT_INVALID_PACKET;
99      goto done;
100  }
101
102  /* read hash */
103  if ((err = mp_read_unsigned_bin(e, (unsigned char *)hash, (int)hashlen)) != CRYPT_OK)
104
105  /* w = s^-1 mod n */
106  if ((err = mp_invmod(s, p, w)) != CRYPT_OK)
107
108  /* u1 = ew */
109  if ((err = mp_mulmod(e, w, p, u1)) != CRYPT_OK)
110
111  /* u2 = rw */
112  if ((err = mp_mulmod(r, w, p, u2)) != CRYPT_OK)
113
114  /* find mG = u1*G */
115  if ((err = mp_read_radix(mG->x, (char *)ltc_ecc_sets[key->idx].Gx, 16)) != CRYPT_OK)
116  if ((err = mp_read_radix(mG->y, (char *)ltc_ecc_sets[key->idx].Gy, 16)) != CRYPT_OK)
117  mp_set(mG->z, 1);
118  if ((err = ltc_mp.ecc_ptmul(u1, mG, mG, m, 0)) != CRYPT_OK)
119
120  /* find mQ = u2*Q */
121  if ((err = mp_copy(key->pubkey.x, mQ->x)) != CRYPT_OK)
122  if ((err = mp_copy(key->pubkey.y, mQ->y)) != CRYPT_OK)
123  if ((err = mp_copy(key->pubkey.z, mQ->z)) != CRYPT_OK)
124  if ((err = ltc_mp.ecc_ptmul(u2, mQ, mQ, m, 0)) != CRYPT_OK)
125

```

```
126  /* find the montgomery mp */
127  if ((err = mp_montgomery_setup(m, &mp)) != CRYPT_OK)
128  /* add them */
129  if ((err = ltc_mp.ecc_ptadd(mQ, mG, mG, m, mp)) != CRYPT_OK)
130
131  /* reduce */
132  if ((err = ltc_mp.ecc_map(mG, m, mp)) != CRYPT_OK)
133
134  /* v = X_x1 mod n */
135  if ((err = mp_mod(mG->x, p, v)) != CRYPT_OK)
136
137  /* does v == r */
138  if (mp_cmp(v, r) == LTC_MP_EQ) {
139      *stat = 1;
140  }
141
142  /* clear up and return */
143  err = CRYPT_OK;
144  goto done;
145 error:
146 done:
147     ltc_ecc_del_point(mG);
148     ltc_ecc_del_point(mQ);
149     mp_clear_multi(r, s, v, w, u1, u2, p, e, m, NULL);
150     if (mp != NULL) {
151         mp_montgomery_free(mp);
152     }
153     return err;
154 }
```

Here is the call graph for this function:

## 5.268 pk/ecc/ltc\_ecc\_is\_valid\_idx.c File Reference

### 5.268.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ltc\\_ecc\\_is\\_valid\\_idx.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ltc\_ecc\_is\_valid\_idx.c:

### Functions

- [int ltc\\_ecc\\_is\\_valid\\_idx](#) (int *n*)  
*Returns whether an ECC idx is valid or not.*

### 5.268.2 Function Documentation

#### 5.268.2.1 int ltc\_ecc\_is\_valid\_idx (int *n*)

Returns whether an ECC idx is valid or not.

##### Parameters:

*n* The idx number to check

##### Returns:

1 if valid, 0 if not

Definition at line 30 of file ltc\_ecc\_is\_valid\_idx.c.

References [ltc\\_ecc\\_sets](#).

Referenced by [ecc\\_ansi\\_x963\\_export\(\)](#), [ecc\\_export\(\)](#), [ecc\\_get\\_size\(\)](#), [ecc\\_shared\\_secret\(\)](#), [ecc\\_sign\\_hash\(\)](#), and [ecc\\_verify\\_hash\(\)](#).

```
31 {  
32     int x;  
33  
34     for (x = 0; ltc_ecc_sets[x].size != 0; x++);  
35     if ((n < 0) || (n >= x)) {  
36         return 0;  
37     }  
38     return 1;  
39 }
```

## 5.269 pk/ecc/ltc\_ecc\_map.c File Reference

### 5.269.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ltc\\_ecc\\_map.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ltc\_ecc\_map.c:

### Functions

- `int ltc_ecc_map (ecc_point *P, void *modulus, void *mp)`

*Map a projective jacobian point back to affine space.*

### 5.269.2 Function Documentation

#### 5.269.2.1 `int ltc_ecc_map (ecc_point *P, void *modulus, void *mp)`

Map a projective jacobian point back to affine space.

#### Parameters:

**P** [in/out] The point to map

**modulus** The modulus of the field the ECC curve is in

**mp** The "b" value from montgomery\_setup()

#### Returns:

CRYPT\_OK on success

Definition at line 33 of file ltc\_ecc\_map.c.

References CRYPT\_MEM, CRYPT\_OK, LTC\_ARGCHK, t1, and t2.

```

34 {
35     void *t1, *t2;
36     int err;
37
38     LTC_ARGCHK(P != NULL);
39     LTC_ARGCHK(modulus != NULL);
40     LTC_ARGCHK(mp != NULL);
41
42     if ((err = mp_init_multi(&t1, &t2, NULL)) != CRYPT_OK) {
43         return CRYPT_MEM;
44     }
45
46     /* first map z back to normal */
47     if ((err = mp_montgomery_reduce(P->z, modulus, mp)) != CRYPT_OK) { goto done; }
48
49     /* get 1/z */
50     if ((err = mp_invmod(P->z, modulus, t1)) != CRYPT_OK) { goto done; }
51
52     /* get 1/z^2 and 1/z^3 */
53     if ((err = mp_sqr(t1, t2)) != CRYPT_OK) { goto done; }

```



```
54     if ((err = mp_mod(t2, modulus, t2)) != CRYPT_OK) { goto done; }
55     if ((err = mp_mul(t1, t2, t1)) != CRYPT_OK) { goto done; }
56     if ((err = mp_mod(t1, modulus, t1)) != CRYPT_OK) { goto done; }
57
58     /* multiply against x/y */
59     if ((err = mp_mul(P->x, t2, P->x)) != CRYPT_OK) { goto done; }
60     if ((err = mp_montgomery_reduce(P->x, modulus, mp)) != CRYPT_OK) { goto done; }
61     if ((err = mp_mul(P->y, t1, P->y)) != CRYPT_OK) { goto done; }
62     if ((err = mp_montgomery_reduce(P->y, modulus, mp)) != CRYPT_OK) { goto done; }
63     mp_set(P->z, 1);
64
65     err = CRYPT_OK;
66     goto done;
67 done:
68     mp_clear_multi(t1, t2, NULL);
69     return err;
70 }
```

## 5.270 pk/ecc/ltc\_ecc\_mulmod.c File Reference

### 5.270.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ltc\\_ecc\\_mulmod.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ltc\_ecc\_mulmod.c:

### Defines

- #define [WINSIZE](#) 4

### Functions

- int [ltc\\_ecc\\_mulmod](#) (void \*k, [ecc\\_point](#) \*G, [ecc\\_point](#) \*R, void \*modulus, int map)  
*Perform a point multiplication.*

### 5.270.2 Define Documentation

#### 5.270.2.1 #define WINSIZE 4

Definition at line 28 of file ltc\_ecc\_mulmod.c.

### 5.270.3 Function Documentation

#### 5.270.3.1 int ltc\_ecc\_mulmod (void \*k, [ecc\\_point](#) \*G, [ecc\\_point](#) \*R, void \*modulus, int map)

Perform a point multiplication.

#### Parameters:

*k* The scalar to multiply by

*G* The base point

*R* [out] Destination for kG

*modulus* The modulus of the field the ECC curve is in

*map* Boolean whether to map back to affine or not (1==map, 0 == leave in projective)

#### Returns:

CRYPT\_OK on success

Definition at line 39 of file ltc\_ecc\_mulmod.c.

References [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), [ltc\\_ecc\\_del\\_point\(\)](#), and [ltc\\_ecc\\_new\\_point\(\)](#).

```

40 {
41     ecc_point *tG, *M[8];
42     int        i, j, err;
43     void        *mu, *mp;
44     unsigned long buf;
45     int         first, bitbuf, bitcpy, bitcnt, mode, digidx;
46
47     LTC_ARGCHK(k        != NULL);
48     LTC_ARGCHK(G        != NULL);
49     LTC_ARGCHK(R        != NULL);
50     LTC_ARGCHK(modulus != NULL);
51
52     /* init montgomery reduction */
53     if ((err = mp_montgomery_setup(modulus, &mp)) != CRYPT_OK) {
54         return err;
55     }
56     if ((err = mp_init(&mu)) != CRYPT_OK) {
57         return err;
58     }
59     if ((err = mp_montgomery_normalization(mu, modulus)) != CRYPT_OK) {
60         mp_montgomery_free(mp);
61         mp_clear(mu);
62         return err;
63     }
64
65     /* alloc ram for window temps */
66     for (i = 0; i < 8; i++) {
67         M[i] = ltc_ecc_new_point();
68         if (M[i] == NULL) {
69             for (j = 0; j < i; j++) {
70                 ltc_ecc_del_point(M[j]);
71             }
72             mp_montgomery_free(mp);
73             mp_clear(mu);
74             return CRYPT_MEM;
75         }
76     }
77
78     /* make a copy of G incase R==G */
79     tG = ltc_ecc_new_point();
80     if (tG == NULL) {
81         { err = CRYPT_MEM; }
82     }
83     /* tG = G and convert to montgomery */
84     if (mp_cmp_d(mu, 1) == LTC_MP_EQ) {
85         if ((err = mp_copy(G->x, tG->x)) != CRYPT_OK) { goto done; }
86         if ((err = mp_copy(G->y, tG->y)) != CRYPT_OK) { goto done; }
87         if ((err = mp_copy(G->z, tG->z)) != CRYPT_OK) { goto done; }
88     } else {
89         if ((err = mp_mulmod(G->x, mu, modulus, tG->x)) != CRYPT_OK) { goto done; }
90         if ((err = mp_mulmod(G->y, mu, modulus, tG->y)) != CRYPT_OK) { goto done; }
91         if ((err = mp_mulmod(G->z, mu, modulus, tG->z)) != CRYPT_OK) { goto done; }
92     }
93     mp_clear(mu);
94
95     /* calc the M tab, which holds kG for k==8..15 */
96     /* M[0] == 8G */
97     if ((err = ltc_mp.ecc_ptdbl(tG, M[0], modulus, mp)) != CRYPT_OK) { goto done; }
98     if ((err = ltc_mp.ecc_ptdbl(M[0], M[0], modulus, mp)) != CRYPT_OK) { goto done; }
99     if ((err = ltc_mp.ecc_ptdbl(M[0], M[0], modulus, mp)) != CRYPT_OK) { goto done; }
100
101     /* now find (8+k)G for k=1..7 */
102     for (j = 9; j < 16; j++) {
103         if ((err = ltc_mp.ecc_ptadd(M[j-9], tG, M[j-8], modulus, mp)) != CRYPT_OK) { goto done; }
104     }
105     /* setup sliding window */
106     mode = 0;

```

```

107     bitcnt = 1;
108     buf    = 0;
109     digidx = mp_get_digit_count(k) - 1;
110     bitcpy = bitbuf = 0;
111     first  = 1;
112
113     /* perform ops */
114     for (;;) {
115         /* grab next digit as required */
116         if (--bitcnt == 0) {
117             if (digidx == -1) {
118                 break;
119             }
120             buf    = mp_get_digit(k, digidx);
121             bitcnt = (int) ltc_mp.bits_per_digit;
122             --digidx;
123         }
124
125         /* grab the next msb from the ltiplicand */
126         i = (buf >> (ltc_mp.bits_per_digit - 1)) & 1;
127         buf <<= 1;
128
129         /* skip leading zero bits */
130         if (mode == 0 && i == 0) {
131             continue;
132         }
133
134         /* if the bit is zero and mode == 1 then we double */
135         if (mode == 1 && i == 0) {
136             if ((err = ltc_mp.ecc_ptdbl(R, R, modulus, mp)) != CRYPT_OK)           { goto done; }
137             continue;
138         }
139
140         /* else we add it to the window */
141         bitbuf |= (i << (WINSIZE - ++bitcpy));
142         mode = 2;
143
144         if (bitcpy == WINSIZE) {
145             /* if this is the first window we do a simple copy */
146             if (first == 1) {
147                 /* R = kG [k = first window] */
148                 if ((err = mp_copy(M[bitbuf-8]->x, R->x)) != CRYPT_OK)           { goto done; }
149                 if ((err = mp_copy(M[bitbuf-8]->y, R->y)) != CRYPT_OK)           { goto done; }
150                 if ((err = mp_copy(M[bitbuf-8]->z, R->z)) != CRYPT_OK)           { goto done; }
151                 first = 0;
152             } else {
153                 /* normal window */
154                 /* ok window is filled so double as required and add */
155                 /* double first */
156                 for (j = 0; j < WINSIZE; j++) {
157                     if ((err = ltc_mp.ecc_ptdbl(R, R, modulus, mp)) != CRYPT_OK) { goto done; }
158                 }
159
160                 /* then add, bitbuf will be 8..15 [8..2^WINSIZE] guaranteed */
161                 if ((err = ltc_mp.ecc_ptadd(R, M[bitbuf-8], R, modulus, mp)) != CRYPT_OK) { goto done; }
162             }
163             /* empty window and reset */
164             bitcpy = bitbuf = 0;
165             mode = 1;
166         }
167     }
168
169     /* if bits remain then double/add */
170     if (mode == 2 && bitcpy > 0) {
171         /* double then add */
172         for (j = 0; j < bitcpy; j++) {
173             /* only double if we have had at least one add first */

```

```

174     if (first == 0) {
175         if ((err = ltc_mp.ecc_ptdbl(R, R, modulus, mp)) != CRYPT_OK)           { goto done; }
176     }
177
178     bitbuf <= 1;
179     if ((bitbuf & (1 << WINSIZE)) != 0) {
180         if (first == 1){
181             /* first add, so copy */
182             if ((err = mp_copy(tG->x, R->x)) != CRYPT_OK)                       { goto done; }
183             if ((err = mp_copy(tG->y, R->y)) != CRYPT_OK)                       { goto done; }
184             if ((err = mp_copy(tG->z, R->z)) != CRYPT_OK)                       { goto done; }
185             first = 0;
186         } else {
187             /* then add */
188             if ((err = ltc_mp.ecc_ptadd(R, tG, R, modulus, mp)) != CRYPT_OK)   { goto done; }
189         }
190     }
191 }
192 }
193
194 /* map R back from projective space */
195 if (map) {
196     err = ltc_ecc_map(R, modulus, mp);
197 } else {
198     err = CRYPT_OK;
199 }
200 done:
201 mp_montgomery_free(mp);
202 ltc_ecc_del_point(tG);
203 for (i = 0; i < 8; i++) {
204     ltc_ecc_del_point(M[i]);
205 }
206 return err;
207 }

```

Here is the call graph for this function:

## 5.271 pk/ecc/ltc\_ecc\_mulmod\_timing.c File Reference

### 5.271.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ltc\\_ecc\\_mulmod\\_timing.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ltc\_ecc\_mulmod\_timing.c:

## 5.272 pk/ecc/ltc\_ecc\_points.c File Reference

### 5.272.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ltc\\_ecc\\_points.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ltc\_ecc\_points.c:

### Functions

- [ecc\\_point](#) \* [ltc\\_ecc\\_new\\_point](#) (void)  
*Allocate a new ECC point.*
- void [ltc\\_ecc\\_del\\_point](#) ([ecc\\_point](#) \*p)  
*Free an ECC point from memory.*

### 5.272.2 Function Documentation

#### 5.272.2.1 void ltc\_ecc\_del\_point ([ecc\\_point](#) \*p)

Free an ECC point from memory.

#### Parameters:

*p* The point to free

Definition at line 47 of file ltc\_ecc\_points.c.

References XFREE.

Referenced by ecc\_shared\_secret(), ecc\_test(), and ltc\_ecc\_mulmod().

```
48 {
49     /* prevents free'ing null arguments */
50     if (p != NULL) {
51         mp_clear_multi(p->x, p->y, p->z, NULL); /* note: p->z may be NULL but that's ok with this function */
52         XFREE(p);
53     }
54 }
```

#### 5.272.2.2 [ecc\\_point](#)\* ltc\_ecc\_new\_point (void)

Allocate a new ECC point.

#### Returns:

A newly allocated point or NULL on error

Definition at line 30 of file ltc\_ecc\_points.c.

References CRYPT\_OK, XFREE, and XMALLOC.

Referenced by ecc\_shared\_secret(), ecc\_test(), ecc\_verify\_hash(), and ltc\_ecc\_mulmod().

```
31 {
32     ecc_point *p;
33     p = XMALLOC(sizeof(*p));
34     if (p == NULL) {
35         return NULL;
36     }
37     if (mp_init_multi(&p->x, &p->y, &p->z, NULL) != CRYPT_OK) {
38         XFREE(p);
39         return NULL;
40     }
41     return p;
42 }
```



## 5.273 pk/ecc/ltc\_ecc\_projective\_add\_point.c File Reference

### 5.273.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ltc\\_ecc\\_projective\\_add\\_point.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ltc\_ecc\_projective\_add\_point.c:

### Functions

- `int ltc_ecc_projective_add_point (ecc_point *P, ecc_point *Q, ecc_point *R, void *modulus, void *mp)`

*Add two ECC points.*

### 5.273.2 Function Documentation

#### 5.273.2.1 `int ltc_ecc_projective_add_point (ecc_point *P, ecc_point *Q, ecc_point *R, void *modulus, void *mp)`

Add two ECC points.

#### Parameters:

**P** The point to add

**Q** The point to add

**R** [out] The destination of the double

**modulus** The modulus of the field the ECC curve is in

**mp** The "b" value from montgomery\_setup()

#### Returns:

CRYPT\_OK on success

Definition at line 35 of file ltc\_ecc\_projective\_add\_point.c.

References CRYPT\_OK, LTC\_ARGCHK, ltc\_ecc\_projective\_dbl\_point(), LTC\_MP\_EQ, LTC\_MP\_LT, t1, and t2.

```
36 {
37     void *t1, *t2, *x, *y, *z;
38     int err;
39
40     LTC_ARGCHK(P != NULL);
41     LTC_ARGCHK(Q != NULL);
42     LTC_ARGCHK(R != NULL);
43     LTC_ARGCHK(modulus != NULL);
44     LTC_ARGCHK(mp != NULL);
45
46     if ((err = mp_init_multi(&t1, &t2, &x, &y, &z, NULL)) != CRYPT_OK) {
47         return err;
48     }
```

```

49
50 /* should we dbl instead? */
51 if ((err = mp_sub(modulus, Q->y, t1)) != CRYPT_OK) { goto done; }
52
53 if ( (mp_cmp(P->x, Q->x) == LTC_MP_EQ) &&
54      (Q->z != NULL && mp_cmp(P->z, Q->z) == LTC_MP_EQ) &&
55      (mp_cmp(P->y, Q->y) == LTC_MP_EQ || mp_cmp(P->y, t1) == LTC_MP_EQ)) {
56     mp_clear_multi(t1, t2, x, y, z, NULL);
57     return ltc_ecc_projective_dbl_point(P, R, modulus, mp);
58 }
59
60 if ((err = mp_copy(P->x, x)) != CRYPT_OK) { goto done; }
61 if ((err = mp_copy(P->y, y)) != CRYPT_OK) { goto done; }
62 if ((err = mp_copy(P->z, z)) != CRYPT_OK) { goto done; }
63
64 /* if Z is one then these are no-operations */
65 if (Q->z != NULL) {
66     /* T1 = Z' * Z' */
67     if ((err = mp_sqr(Q->z, t1)) != CRYPT_OK) { goto done; }
68     if ((err = mp_montgomery_reduce(t1, modulus, mp)) != CRYPT_OK) { goto done; }
69     /* X = X * T1 */
70     if ((err = mp_mul(t1, x, x)) != CRYPT_OK) { goto done; }
71     if ((err = mp_montgomery_reduce(x, modulus, mp)) != CRYPT_OK) { goto done; }
72     /* T1 = Z' * T1 */
73     if ((err = mp_mul(Q->z, t1, t1)) != CRYPT_OK) { goto done; }
74     if ((err = mp_montgomery_reduce(t1, modulus, mp)) != CRYPT_OK) { goto done; }
75     /* Y = Y * T1 */
76     if ((err = mp_mul(t1, y, y)) != CRYPT_OK) { goto done; }
77     if ((err = mp_montgomery_reduce(y, modulus, mp)) != CRYPT_OK) { goto done; }
78 }
79
80 /* T1 = Z*Z */
81 if ((err = mp_sqr(z, t1)) != CRYPT_OK) { goto done; }
82 if ((err = mp_montgomery_reduce(t1, modulus, mp)) != CRYPT_OK) { goto done; }
83 /* T2 = X' * T1 */
84 if ((err = mp_mul(Q->x, t1, t2)) != CRYPT_OK) { goto done; }
85 if ((err = mp_montgomery_reduce(t2, modulus, mp)) != CRYPT_OK) { goto done; }
86 /* T1 = Z * T1 */
87 if ((err = mp_mul(z, t1, t1)) != CRYPT_OK) { goto done; }
88 if ((err = mp_montgomery_reduce(t1, modulus, mp)) != CRYPT_OK) { goto done; }
89 /* T1 = Y' * T1 */
90 if ((err = mp_mul(Q->y, t1, t1)) != CRYPT_OK) { goto done; }
91 if ((err = mp_montgomery_reduce(t1, modulus, mp)) != CRYPT_OK) { goto done; }
92
93 /* Y = Y - T1 */
94 if ((err = mp_sub(y, t1, y)) != CRYPT_OK) { goto done; }
95 if (mp_cmp_d(y, 0) == LTC_MP_LT) {
96     if ((err = mp_add(y, modulus, y)) != CRYPT_OK) { goto done; }
97 }
98 /* T1 = 2T1 */
99 if ((err = mp_add(t1, t1, t1)) != CRYPT_OK) { goto done; }
100 if (mp_cmp(t1, modulus) != LTC_MP_LT) {
101     if ((err = mp_sub(t1, modulus, t1)) != CRYPT_OK) { goto done; }
102 }
103 /* T1 = Y + T1 */
104 if ((err = mp_add(t1, y, t1)) != CRYPT_OK) { goto done; }
105 if (mp_cmp(t1, modulus) != LTC_MP_LT) {
106     if ((err = mp_sub(t1, modulus, t1)) != CRYPT_OK) { goto done; }
107 }
108 /* X = X - T2 */
109 if ((err = mp_sub(x, t2, x)) != CRYPT_OK) { goto done; }
110 if (mp_cmp_d(x, 0) == LTC_MP_LT) {
111     if ((err = mp_add(x, modulus, x)) != CRYPT_OK) { goto done; }
112 }
113 /* T2 = 2T2 */
114 if ((err = mp_add(t2, t2, t2)) != CRYPT_OK) { goto done; }
115 if (mp_cmp(t2, modulus) != LTC_MP_LT) {

```

```

116     if ((err = mp_sub(t2, modulus, t2)) != CRYPT_OK) { goto done; }
117 }
118 /* T2 = X + T2 */
119 if ((err = mp_add(t2, x, t2)) != CRYPT_OK) { goto done; }
120 if (mp_cmp(t2, modulus) != LTC_MP_LT) {
121     if ((err = mp_sub(t2, modulus, t2)) != CRYPT_OK) { goto done; }
122 }
123
124 /* if Z' != 1 */
125 if (Q->z != NULL) {
126     /* Z = Z * Z' */
127     if ((err = mp_mul(z, Q->z, z)) != CRYPT_OK) { goto done; }
128     if ((err = mp_montgomery_reduce(z, modulus, mp)) != CRYPT_OK) { goto done; }
129 }
130
131 /* Z = Z * X */
132 if ((err = mp_mul(z, x, z)) != CRYPT_OK) { goto done; }
133 if ((err = mp_montgomery_reduce(z, modulus, mp)) != CRYPT_OK) { goto done; }
134
135 /* T1 = T1 * X */
136 if ((err = mp_mul(t1, x, t1)) != CRYPT_OK) { goto done; }
137 if ((err = mp_montgomery_reduce(t1, modulus, mp)) != CRYPT_OK) { goto done; }
138 /* X = X * X */
139 if ((err = mp_sqr(x, x)) != CRYPT_OK) { goto done; }
140 if ((err = mp_montgomery_reduce(x, modulus, mp)) != CRYPT_OK) { goto done; }
141 /* T2 = T2 * x */
142 if ((err = mp_mul(t2, x, t2)) != CRYPT_OK) { goto done; }
143 if ((err = mp_montgomery_reduce(t2, modulus, mp)) != CRYPT_OK) { goto done; }
144 /* T1 = T1 * X */
145 if ((err = mp_mul(t1, x, t1)) != CRYPT_OK) { goto done; }
146 if ((err = mp_montgomery_reduce(t1, modulus, mp)) != CRYPT_OK) { goto done; }
147
148 /* X = Y*Y */
149 if ((err = mp_sqr(y, x)) != CRYPT_OK) { goto done; }
150 if ((err = mp_montgomery_reduce(x, modulus, mp)) != CRYPT_OK) { goto done; }
151 /* X = X - T2 */
152 if ((err = mp_sub(x, t2, x)) != CRYPT_OK) { goto done; }
153 if (mp_cmp_d(x, 0) == LTC_MP_LT) {
154     if ((err = mp_add(x, modulus, x)) != CRYPT_OK) { goto done; }
155 }
156
157 /* T2 = T2 - X */
158 if ((err = mp_sub(t2, x, t2)) != CRYPT_OK) { goto done; }
159 if (mp_cmp_d(t2, 0) == LTC_MP_LT) {
160     if ((err = mp_add(t2, modulus, t2)) != CRYPT_OK) { goto done; }
161 }
162 /* T2 = T2 - X */
163 if ((err = mp_sub(t2, x, t2)) != CRYPT_OK) { goto done; }
164 if (mp_cmp_d(t2, 0) == LTC_MP_LT) {
165     if ((err = mp_add(t2, modulus, t2)) != CRYPT_OK) { goto done; }
166 }
167 /* T2 = T2 * Y */
168 if ((err = mp_mul(t2, y, t2)) != CRYPT_OK) { goto done; }
169 if ((err = mp_montgomery_reduce(t2, modulus, mp)) != CRYPT_OK) { goto done; }
170 /* Y = T2 - T1 */
171 if ((err = mp_sub(t2, t1, y)) != CRYPT_OK) { goto done; }
172 if (mp_cmp_d(y, 0) == LTC_MP_LT) {
173     if ((err = mp_add(y, modulus, y)) != CRYPT_OK) { goto done; }
174 }
175 /* Y = Y/2 */
176 if (mp_isodd(y)) {
177     if ((err = mp_add(y, modulus, y)) != CRYPT_OK) { goto done; }
178 }
179 if ((err = mp_div_2(y, y)) != CRYPT_OK) { goto done; }
180
181 if ((err = mp_copy(x, R->x)) != CRYPT_OK) { goto done; }
182 if ((err = mp_copy(y, R->y)) != CRYPT_OK) { goto done; }

```

```
183     if ((err = mp_copy(z, R->z)) != CRYPT_OK)                { goto done; }
184
185     err = CRYPT_OK;
186     goto done;
187 done:
188     mp_clear_multi(t1, t2, x, y, z, NULL);
189     return err;
190 }
```

Here is the call graph for this function:

## 5.274 pk/ecc/ltc\_ecc\_projective\_dbl\_point.c File Reference

### 5.274.1 Detailed Description

ECC Crypto, Tom St Denis.

Definition in file [ltc\\_ecc\\_projective\\_dbl\\_point.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for ltc\_ecc\_projective\_dbl\_point.c:

### Functions

- [int ltc\\_ecc\\_projective\\_dbl\\_point](#) ([ecc\\_point](#) \*P, [ecc\\_point](#) \*R, void \*modulus, void \*mp)  
*Double an ECC point.*

### 5.274.2 Function Documentation

#### 5.274.2.1 int ltc\_ecc\_projective\_dbl\_point ([ecc\\_point](#) \*P, [ecc\\_point](#) \*R, void \*modulus, void \*mp)

Double an ECC point.

#### Parameters:

- P** The point to double
- R** [out] The destination of the double
- modulus** The modulus of the field the ECC curve is in
- mp** The "b" value from montgomery\_setup()

#### Returns:

CRYPT\_OK on success

Definition at line 34 of file ltc\_ecc\_projective\_dbl\_point.c.

References CRYPT\_OK, LTC\_ARGCHK, LTC\_MP\_LT, t1, and t2.

Referenced by ltc\_ecc\_projective\_add\_point().

```

35 {
36     void *t1, *t2;
37     int err;
38
39     LTC_ARGCHK(P != NULL);
40     LTC_ARGCHK(R != NULL);
41     LTC_ARGCHK(modulus != NULL);
42     LTC_ARGCHK(mp != NULL);
43
44     if ((err = mp_init_multi(&t1, &t2, NULL)) != CRYPT_OK) {
45         return err;
46     }
47
48     if (P != R) {
49         if ((err = mp_copy(P->x, R->x)) != CRYPT_OK) { goto done; }
50         if ((err = mp_copy(P->y, R->y)) != CRYPT_OK) { goto done; }
51         if ((err = mp_copy(P->z, R->z)) != CRYPT_OK) { goto done; }

```

```

52     }
53
54     /* t1 = Z * Z */
55     if ((err = mp_sqr(R->z, t1)) != CRYPT_OK) { goto done; }
56     if ((err = mp_montgomery_reduce(t1, modulus, mp)) != CRYPT_OK) { goto done; }
57     /* Z = Y * Z */
58     if ((err = mp_mul(R->z, R->y, R->z)) != CRYPT_OK) { goto done; }
59     if ((err = mp_montgomery_reduce(R->z, modulus, mp)) != CRYPT_OK) { goto done; }
60     /* Z = 2Z */
61     if ((err = mp_add(R->z, R->z, R->z)) != CRYPT_OK) { goto done; }
62     if (mp_cmp(R->z, modulus) != LTC_MP_LT) {
63         if ((err = mp_sub(R->z, modulus, R->z)) != CRYPT_OK) { goto done; }
64     }
65
66     /* T2 = X - T1 */
67     if ((err = mp_sub(R->x, t1, t2)) != CRYPT_OK) { goto done; }
68     if (mp_cmp_d(t2, 0) == LTC_MP_LT) {
69         if ((err = mp_add(t2, modulus, t2)) != CRYPT_OK) { goto done; }
70     }
71     /* T1 = X + T1 */
72     if ((err = mp_add(t1, R->x, t1)) != CRYPT_OK) { goto done; }
73     if (mp_cmp(t1, modulus) != LTC_MP_LT) {
74         if ((err = mp_sub(t1, modulus, t1)) != CRYPT_OK) { goto done; }
75     }
76     /* T2 = T1 * T2 */
77     if ((err = mp_mul(t1, t2, t2)) != CRYPT_OK) { goto done; }
78     if ((err = mp_montgomery_reduce(t2, modulus, mp)) != CRYPT_OK) { goto done; }
79     /* T1 = 2T2 */
80     if ((err = mp_add(t2, t2, t1)) != CRYPT_OK) { goto done; }
81     if (mp_cmp(t1, modulus) != LTC_MP_LT) {
82         if ((err = mp_sub(t1, modulus, t1)) != CRYPT_OK) { goto done; }
83     }
84     /* T1 = T1 + T2 */
85     if ((err = mp_add(t1, t2, t1)) != CRYPT_OK) { goto done; }
86     if (mp_cmp(t1, modulus) != LTC_MP_LT) {
87         if ((err = mp_sub(t1, modulus, t1)) != CRYPT_OK) { goto done; }
88     }
89
90     /* Y = 2Y */
91     if ((err = mp_add(R->y, R->y, R->y)) != CRYPT_OK) { goto done; }
92     if (mp_cmp(R->y, modulus) != LTC_MP_LT) {
93         if ((err = mp_sub(R->y, modulus, R->y)) != CRYPT_OK) { goto done; }
94     }
95     /* Y = Y * Y */
96     if ((err = mp_sqr(R->y, R->y)) != CRYPT_OK) { goto done; }
97     if ((err = mp_montgomery_reduce(R->y, modulus, mp)) != CRYPT_OK) { goto done; }
98     /* T2 = Y * Y */
99     if ((err = mp_sqr(R->y, t2)) != CRYPT_OK) { goto done; }
100    if ((err = mp_montgomery_reduce(t2, modulus, mp)) != CRYPT_OK) { goto done; }
101    /* T2 = T2/2 */
102    if (mp_isodd(t2)) {
103        if ((err = mp_add(t2, modulus, t2)) != CRYPT_OK) { goto done; }
104    }
105    if ((err = mp_div_2(t2, t2)) != CRYPT_OK) { goto done; }
106    /* Y = Y * X */
107    if ((err = mp_mul(R->y, R->x, R->y)) != CRYPT_OK) { goto done; }
108    if ((err = mp_montgomery_reduce(R->y, modulus, mp)) != CRYPT_OK) { goto done; }
109
110    /* X = T1 * T1 */
111    if ((err = mp_sqr(t1, R->x)) != CRYPT_OK) { goto done; }
112    if ((err = mp_montgomery_reduce(R->x, modulus, mp)) != CRYPT_OK) { goto done; }
113    /* X = X - Y */
114    if ((err = mp_sub(R->x, R->y, R->x)) != CRYPT_OK) { goto done; }
115    if (mp_cmp_d(R->x, 0) == LTC_MP_LT) {
116        if ((err = mp_add(R->x, modulus, R->x)) != CRYPT_OK) { goto done; }
117    }
118    /* X = X - Y */

```

```
119     if ((err = mp_sub(R->x, R->y, R->x)) != CRYPT_OK)                { goto done; }
120     if (mp_cmp_d(R->x, 0) == LTC_MP_LT) {
121         if ((err = mp_add(R->x, modulus, R->x)) != CRYPT_OK)        { goto done; }
122     }
123
124     /* Y = Y - X */
125     if ((err = mp_sub(R->y, R->x, R->y)) != CRYPT_OK)                { goto done; }
126     if (mp_cmp_d(R->y, 0) == LTC_MP_LT) {
127         if ((err = mp_add(R->y, modulus, R->y)) != CRYPT_OK)        { goto done; }
128     }
129     /* Y = Y * T1 */
130     if ((err = mp_mul(R->y, t1, R->y)) != CRYPT_OK)                  { goto done; }
131     if ((err = mp_montgomery_reduce(R->y, modulus, mp)) != CRYPT_OK) { goto done; }
132     /* Y = Y - T2 */
133     if ((err = mp_sub(R->y, t2, R->y)) != CRYPT_OK)                  { goto done; }
134     if (mp_cmp_d(R->y, 0) == LTC_MP_LT) {
135         if ((err = mp_add(R->y, modulus, R->y)) != CRYPT_OK)        { goto done; }
136     }
137
138     err = CRYPT_OK;
139     goto done;
140 done:
141     mp_clear_multi(t1, t2, NULL);
142     return err;
143 }
```

## 5.275 pk/katja/katja\_decrypt\_key.c File Reference

### 5.275.1 Detailed Description

Katja PKCS #1 OAEP Decryption, Tom St Denis.

Definition in file [katja\\_decrypt\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for katja\_decrypt\_key.c:



## 5.276 pk/katja/katja\_encrypt\_key.c File Reference

### 5.276.1 Detailed Description

Katja PKCS-style OAEP encryption, Tom St Denis.

Definition in file [katja\\_encrypt\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for katja\_encrypt\_key.c:

## 5.277 pk/katja/katja\_export.c File Reference

### 5.277.1 Detailed Description

Export Katja PKCS-style keys, Tom St Denis.

Definition in file [katja\\_export.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for katja\_export.c:

## 5.278 pk/katja/katja\_exptmod.c File Reference

### 5.278.1 Detailed Description

Katja PKCS-style exptmod, Tom St Denis.

Definition in file [katja\\_exptmod.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for katja\_exptmod.c:

## 5.279 pk/katja/katja\_free.c File Reference

### 5.279.1 Detailed Description

Free an Katja key, Tom St Denis.

Definition in file [katja\\_free.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for katja\_free.c:

## 5.280 pk/katja/katja\_import.c File Reference

### 5.280.1 Detailed Description

Import a PKCS-style Katja key, Tom St Denis.

Definition in file [katja\\_import.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for katja\_import.c:

## 5.281 pk/katja/katja\_make\_key.c File Reference

### 5.281.1 Detailed Description

Katja key generation, Tom St Denis.

Definition in file [katja\\_make\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for katja\_make\_key.c:

## 5.282 pk/pkcs1/pkcs\_1\_i2osp.c File Reference

### 5.282.1 Detailed Description

Integer to Octet I2OSP, Tom St Denis.

Definition in file [pkcs\\_1\\_i2osp.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pkcs\_1\_i2osp.c:

### Functions

- [pkcs\\_1\\_i2osp](#) (void \*n, unsigned long modulus\_len, unsigned char \*out)  
*PKCS #1 Integer to binary.*

### 5.282.2 Function Documentation

#### 5.282.2.1 int pkcs\_1\_i2osp (void \* n, unsigned long modulus\_len, unsigned char \* out)

PKCS #1 Integer to binary.

##### Parameters:

- n* The integer to store  
*modulus\_len* The length of the RSA modulus  
*out* [out] The destination for the integer

##### Returns:

CRYPT\_OK if successful

Definition at line 31 of file pkcs\_1\_i2osp.c.

References CRYPT\_BUFFER\_OVERFLOW, edge::size, and zeromem().

```
32 {  
33     unsigned long size;  
34  
35     size = mp_unsigned_bin_size(n);  
36  
37     if (size > modulus_len) {  
38         return CRYPT_BUFFER_OVERFLOW;  
39     }  
40  
41     /* store it */  
42     zeromem(out, modulus_len);  
43     return mp_to_unsigned_bin(n, out+(modulus_len-size));  
44 }
```

Here is the call graph for this function:

## 5.283 pk/pkcs1/pkcs\_1\_mgf1.c File Reference

### 5.283.1 Detailed Description

The Mask Generation Function (MGF1) for PKCS #1, Tom St Denis.

Definition in file [pkcs\\_1\\_mgf1.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pkcs\_1\_mgf1.c:

### Functions

- [int pkcs\\_1\\_mgf1](#) (int *hash\_idx*, const unsigned char \**seed*, unsigned long *seedlen*, unsigned char \**mask*, unsigned long *masklen*)

*Perform PKCS #1 MGF1 (internal).*

### 5.283.2 Function Documentation

#### 5.283.2.1 int pkcs\_1\_mgf1 (int *hash\_idx*, const unsigned char \* *seed*, unsigned long *seedlen*, unsigned char \* *mask*, unsigned long *masklen*)

Perform PKCS #1 MGF1 (internal).

#### Parameters:

*seed* The seed for MGF1  
*seedlen* The length of the seed  
*hash\_idx* The index of the hash desired  
*mask* [out] The destination  
*masklen* The length of the mask desired

#### Returns:

CRYPT\_OK if successful

Definition at line 29 of file pkcs\_1\_mgf1.c.

References CRYPT\_MEM, CRYPT\_OK, hash\_descriptor, hash\_is\_valid(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, XFREE, and XMALLOC.

Referenced by pkcs\_1\_oaep\_decode(), pkcs\_1\_oaep\_encode(), pkcs\_1\_pss\_decode(), and pkcs\_1\_pss\_encode().

```
32 {
33     unsigned long hLen, x;
34     ulong32      counter;
35     int          err;
36     hash_state   *md;
37     unsigned char *buf;
38
39     LTC_ARGCHK(seed != NULL);
40     LTC_ARGCHK(mask != NULL);
41
```



```

42  /* ensure valid hash */
43  if ((err = hash_is_valid(hash_idx)) != CRYPT_OK) {
44      return err;
45  }
46
47  /* get hash output size */
48  hLen = hash_descriptor[hash_idx].hashsize;
49
50  /* allocate memory */
51  md = XMALLOC(sizeof(hash_state));
52  buf = XMALLOC(hLen);
53  if (md == NULL || buf == NULL) {
54      if (md != NULL) {
55          XFREE(md);
56      }
57      if (buf != NULL) {
58          XFREE(buf);
59      }
60      return CRYPT_MEM;
61  }
62
63  /* start counter */
64  counter = 0;
65
66  while (masklen > 0) {
67      /* handle counter */
68      STORE32H(counter, buf);
69      ++counter;
70
71      /* get hash of seed || counter */
72      if ((err = hash_descriptor[hash_idx].init(md)) != CRYPT_OK) {
73          goto LBL_ERR;
74      }
75      if ((err = hash_descriptor[hash_idx].process(md, seed, seedlen)) != CRYPT_OK) {
76          goto LBL_ERR;
77      }
78      if ((err = hash_descriptor[hash_idx].process(md, buf, 4)) != CRYPT_OK) {
79          goto LBL_ERR;
80      }
81      if ((err = hash_descriptor[hash_idx].done(md, buf)) != CRYPT_OK) {
82          goto LBL_ERR;
83      }
84
85      /* store it */
86      for (x = 0; x < hLen && masklen > 0; x++, masklen--) {
87          *mask++ = buf[x];
88      }
89  }
90
91  err = CRYPT_OK;
92 LBL_ERR:
93 #ifdef LTC_CLEAN_STACK
94     zeromem(buf, hLen);
95     zeromem(md, sizeof(hash_state));
96 #endif
97
98     XFREE(buf);
99     XFREE(md);
100
101     return err;
102 }

```

Here is the call graph for this function:

## 5.284 pk/pkcs1/pkcs\_1\_oaep\_decode.c File Reference

### 5.284.1 Detailed Description

OAEP Padding for PKCS #1, Tom St Denis.

Definition in file [pkcs\\_1\\_oaep\\_decode.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pkcs\_1\_oaep\_decode.c:

### Functions

- [int pkcs\\_1\\_oaep\\_decode](#) (const unsigned char \*msg, unsigned long msglen, const unsigned char \*lparam, unsigned long lparamlen, unsigned long modulus\_bitlen, int hash\_idx, unsigned char \*out, unsigned long \*outlen, int \*res)

*PKCS #1 v2.00 OAEP decode.*

### 5.284.2 Function Documentation

**5.284.2.1 int pkcs\_1\_oaep\_decode** (const unsigned char \* *msg*, unsigned long *msglen*, const unsigned char \* *lparam*, unsigned long *lparamlen*, unsigned long *modulus\_bitlen*, int *hash\_idx*, unsigned char \* *out*, unsigned long \* *outlen*, int \* *res*)

PKCS #1 v2.00 OAEP decode.

#### Parameters:

*msg* The encoded data to decode  
*msglen* The length of the encoded data (octets)  
*lparam* The session or system data (can be NULL)  
*lparamlen* The length of the lparam  
*modulus\_bitlen* The bit length of the RSA modulus  
*hash\_idx* The index of the hash desired  
*out* [out] Destination of decoding  
*outlen* [in/out] The max size and resulting size of the decoding  
*res* [out] Result of decoding, 1==valid, 0==invalid

#### Returns:

CRYPT\_OK if successful (even if invalid)

Definition at line 33 of file pkcs\_1\_oaep\_decode.c.

References CRYPT\_MEM, CRYPT\_OK, CRYPT\_PK\_INVALID\_SIZE, hash\_descriptor, hash\_is\_valid(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, mask, pkcs\_1\_mgf1(), XFREE, XMALLOC, and XMEMCPY.

Referenced by rsa\_decrypt\_key\_ex().

```

38 {
39     unsigned char *DB, *seed, *mask;
40     unsigned long hLen, x, y, modulus_len;
41     int          err;
42
43     LTC_ARGCHK(msg      != NULL);
44     LTC_ARGCHK(out      != NULL);
45     LTC_ARGCHK(outlen   != NULL);
46     LTC_ARGCHK(res      != NULL);
47
48     /* default to invalid packet */
49     *res = 0;
50
51     /* test valid hash */
52     if ((err = hash_is_valid(hash_idx)) != CRYPT_OK) {
53         return err;
54     }
55     hLen      = hash_descriptor[hash_idx].hashsize;
56     modulus_len = (modulus_bitlen >> 3) + (modulus_bitlen & 7 ? 1 : 0);
57
58     /* test hash/message size */
59     if ((2*hLen >= (modulus_len - 2)) || (msglen != modulus_len)) {
60         return CRYPT_PK_INVALID_SIZE;
61     }
62
63     /* allocate ram for DB/mask/salt of size modulus_len */
64     DB = XMALLOC(modulus_len);
65     mask = XMALLOC(modulus_len);
66     seed = XMALLOC(hLen);
67     if (DB == NULL || mask == NULL || seed == NULL) {
68         if (DB != NULL) {
69             XFREE(DB);
70         }
71         if (mask != NULL) {
72             XFREE(mask);
73         }
74         if (seed != NULL) {
75             XFREE(seed);
76         }
77         return CRYPT_MEM;
78     }
79
80     /* ok so it's now in the form
81
82         0x00 || maskedseed || maskedDB
83
84         1    || hLen      || modulus_len - hLen - 1
85
86     */
87
88     /* must have leading 0x00 byte */
89     if (msg[0] != 0x00) {
90         err = CRYPT_OK;
91         goto LBL_ERR;
92     }
93
94     /* now read the masked seed */
95     x = 1;
96     XMEMCPY(seed, msg + x, hLen);
97     x += hLen;
98
99     /* now read the masked DB */
100    XMEMCPY(DB, msg + x, modulus_len - hLen - 1);
101    x += modulus_len - hLen - 1;
102
103    /* compute MGF1 of maskedDB (hLen) */
104    if ((err = pkcs_1_mgf1(hash_idx, DB, modulus_len - hLen - 1, mask, hLen)) != CRYPT_OK) {

```

```

105     goto LBL_ERR;
106 }
107
108 /* XOR against seed */
109 for (y = 0; y < hLen; y++) {
110     seed[y] ^= mask[y];
111 }
112
113 /* compute MGF1 of seed (k - hlen - 1) */
114 if ((err = pkcs_1_mgf1(hash_idx, seed, hLen, mask, modulus_len - hLen - 1)) != CRYPT_OK) {
115     goto LBL_ERR;
116 }
117
118 /* xor against DB */
119 for (y = 0; y < (modulus_len - hLen - 1); y++) {
120     DB[y] ^= mask[y];
121 }
122
123 /* now DB == lhash || PS || 0x01 || M, PS == k - mlen - 2hlen - 2 zeroes */
124
125 /* compute lhash and store it in seed [reuse temps!] */
126 x = modulus_len;
127 if (lparam != NULL) {
128     if ((err = hash_memory(hash_idx, lparam, lparamlen, seed, &x)) != CRYPT_OK) {
129         goto LBL_ERR;
130     }
131 } else {
132     /* can't pass hash_memory a NULL so use DB with zero length */
133     if ((err = hash_memory(hash_idx, DB, 0, seed, &x)) != CRYPT_OK) {
134         goto LBL_ERR;
135     }
136 }
137
138 /* compare the lhash'es */
139 if (XMEMCMP(seed, DB, hLen) != 0) {
140     err = CRYPT_OK;
141     goto LBL_ERR;
142 }
143
144 /* now zeroes before a 0x01 */
145 for (x = hLen; x < (modulus_len - hLen - 1) && DB[x] == 0x00; x++) {
146     /* step... */
147 }
148
149 /* error out if wasn't 0x01 */
150 if (x == (modulus_len - hLen - 1) || DB[x] != 0x01) {
151     err = CRYPT_INVALID_PACKET;
152     goto LBL_ERR;
153 }
154
155 /* rest is the message (and skip 0x01) */
156 if ((modulus_len - hLen - 1 - ++x) > *outlen) {
157     *outlen = modulus_len - hLen - 1 - x;
158     err = CRYPT_BUFFER_OVERFLOW;
159     goto LBL_ERR;
160 }
161
162 /* copy message */
163 *outlen = modulus_len - hLen - 1 - x;
164 XMEMCPY(out, DB + x, modulus_len - hLen - 1 - x);
165 x += modulus_len - hLen - 1;
166
167 /* valid packet */
168 *res = 1;
169
170 err = CRYPT_OK;
171 LBL_ERR:

```

```
172 #ifdef LTC_CLEAN_STACK
173     zeromem(DB, modulus_len);
174     zeromem(seed, hLen);
175     zeromem(mask, modulus_len);
176 #endif
177
178     XFREE(seed);
179     XFREE(mask);
180     XFREE(DB);
181
182     return err;
183 }
```

Here is the call graph for this function:

## 5.285 pk/pkcs1/pkcs\_1\_oaep\_encode.c File Reference

### 5.285.1 Detailed Description

OAEP Padding for PKCS #1, Tom St Denis.

Definition in file [pkcs\\_1\\_oaep\\_encode.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pkcs\_1\_oaep\_encode.c:

### Functions

- [int pkcs\\_1\\_oaep\\_encode](#) (const unsigned char \*msg, unsigned long msglen, const unsigned char \*lparam, unsigned long lparamlen, unsigned long modulus\_bitlen, [prng\\_state](#) \*prng, int prng\_idx, int hash\_idx, unsigned char \*out, unsigned long \*outlen)

*PKCS #1 v2.00 OAEP encode.*

### 5.285.2 Function Documentation

- 5.285.2.1** [int pkcs\\_1\\_oaep\\_encode](#) (const unsigned char \* *msg*, unsigned long *msglen*, const unsigned char \* *lparam*, unsigned long *lparamlen*, unsigned long *modulus\_bitlen*, [prng\\_state](#) \* *prng*, int *prng\_idx*, int *hash\_idx*, unsigned char \* *out*, unsigned long \* *outlen*)

PKCS #1 v2.00 OAEP encode.

#### Parameters:

*msg* The data to encode  
*msglen* The length of the data to encode (octets)  
*lparam* A session or system parameter (can be NULL)  
*lparamlen* The length of the lparam data  
*modulus\_bitlen* The bit length of the RSA modulus  
*prng* An active PRNG state  
*prng\_idx* The index of the PRNG desired  
*hash\_idx* The index of the hash desired  
*out* [out] The destination for the encoded data  
*outlen* [in/out] The max size and resulting size of the encoded data

#### Returns:

CRYPT\_OK if successful

Definition at line 34 of file pkcs\_1\_oaep\_encode.c.

References CRYPT\_ERROR\_READPRNG, CRYPT\_MEM, CRYPT\_OK, CRYPT\_PK\_INVALID\_SIZE, hash\_descriptor, hash\_is\_valid(), hash\_memory(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, mask, pkcs\_1\_mgf1(), prng\_descriptor, prng\_is\_valid(), XFREE, XMALLOC, XMEMCPY, and XMEMSET.

Referenced by rsa\_encrypt\_key\_ex().

```

39 {
40     unsigned char *DB, *seed, *mask;
41     unsigned long hLen, x, y, modulus_len;
42     int          err;
43
44     LTC_ARGCHK(msg      != NULL);
45     LTC_ARGCHK(out      != NULL);
46     LTC_ARGCHK(outlen != NULL);
47
48     /* test valid hash */
49     if ((err = hash_is_valid(hash_idx)) != CRYPT_OK) {
50         return err;
51     }
52
53     /* valid prng */
54     if ((err = prng_is_valid(prng_idx)) != CRYPT_OK) {
55         return err;
56     }
57
58     hLen      = hash_descriptor[hash_idx].hashsize;
59     modulus_len = (modulus_bitlen >> 3) + (modulus_bitlen & 7 ? 1 : 0);
60
61     /* test message size */
62     if ((2*hLen >= (modulus_len - 2)) || (msglen > (modulus_len - 2*hLen - 2))) {
63         return CRYPT_PK_INVALID_SIZE;
64     }
65
66     /* allocate ram for DB/mask/salt of size modulus_len */
67     DB = XMALLOC(modulus_len);
68     mask = XMALLOC(modulus_len);
69     seed = XMALLOC(hLen);
70     if (DB == NULL || mask == NULL || seed == NULL) {
71         if (DB != NULL) {
72             XFREE(DB);
73         }
74         if (mask != NULL) {
75             XFREE(mask);
76         }
77         if (seed != NULL) {
78             XFREE(seed);
79         }
80         return CRYPT_MEM;
81     }
82
83     /* get lhash */
84     /* DB == lhash || PS || 0x01 || M, PS == k - mlen - 2hlen - 2 zeroes */
85     x = modulus_len;
86     if (lparam != NULL) {
87         if ((err = hash_memory(hash_idx, lparam, lparamlen, DB, &x)) != CRYPT_OK) {
88             goto LBL_ERR;
89         }
90     } else {
91         /* can't pass hash_memory a NULL so use DB with zero length */
92         if ((err = hash_memory(hash_idx, DB, 0, DB, &x)) != CRYPT_OK) {
93             goto LBL_ERR;
94         }
95     }
96
97     /* append PS then 0x01 (to lhash) */
98     x = hLen;
99     y = modulus_len - msglen - 2*hLen - 2;
100     XMEMSET(DB+x, 0, y);
101     x += y;
102
103     /* 0x01 byte */
104     DB[x++] = 0x01;
105

```

```

106  /* message (length = msglen) */
107  XMEMCPY(DB+x, msg, msglen);
108  x += msglen;
109
110  /* now choose a random seed */
111  if (prng_descriptor[prng_idx].read(seed, hLen, prng) != hLen) {
112      err = CRYPT_ERROR_READPRNG;
113      goto LBL_ERR;
114  }
115
116  /* compute MGF1 of seed (k - hlen - 1) */
117  if ((err = pkcs_1_mgf1(hash_idx, seed, hLen, mask, modulus_len - hLen - 1)) != CRYPT_OK) {
118      goto LBL_ERR;
119  }
120
121  /* xor against DB */
122  for (y = 0; y < (modulus_len - hLen - 1); y++) {
123      DB[y] ^= mask[y];
124  }
125
126  /* compute MGF1 of maskedDB (hLen) */
127  if ((err = pkcs_1_mgf1(hash_idx, DB, modulus_len - hLen - 1, mask, hLen)) != CRYPT_OK) {
128      goto LBL_ERR;
129  }
130
131  /* XOR against seed */
132  for (y = 0; y < hLen; y++) {
133      seed[y] ^= mask[y];
134  }
135
136  /* create string of length modulus_len */
137  if (*outlen < modulus_len) {
138      *outlen = modulus_len;
139      err = CRYPT_BUFFER_OVERFLOW;
140      goto LBL_ERR;
141  }
142
143  /* start output which is 0x00 || maskedSeed || maskedDB */
144  x = 0;
145  out[x++] = 0x00;
146  XMEMCPY(out+x, seed, hLen);
147  x += hLen;
148  XMEMCPY(out+x, DB, modulus_len - hLen - 1);
149  x += modulus_len - hLen - 1;
150
151  *outlen = x;
152
153  err = CRYPT_OK;
154 LBL_ERR:
155 #ifdef LTC_CLEAN_STACK
156     zeromem(DB, modulus_len);
157     zeromem(seed, hLen);
158     zeromem(mask, modulus_len);
159 #endif
160
161     XFREE(seed);
162     XFREE(mask);
163     XFREE(DB);
164
165     return err;
166 }

```

Here is the call graph for this function:



## 5.286 pk/pkcs1/pkcs\_1\_os2ip.c File Reference

### 5.286.1 Detailed Description

Octet to Integer OS2IP, Tom St Denis.

Definition in file [pkcs\\_1\\_os2ip.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pkcs\_1\_os2ip.c:

### Functions

- int [pkcs\\_1\\_os2ip](#) (void \*n, unsigned char \*in, unsigned long inlen)

*Read a binary string into an mp\_int.*

### 5.286.2 Function Documentation

#### 5.286.2.1 int pkcs\_1\_os2ip (void \*n, unsigned char \*in, unsigned long inlen)

Read a binary string into an mp\_int.

#### Parameters:

*n* [out] The mp\_int destination

*in* The binary string to read

*inlen* The length of the binary string

#### Returns:

CRYPT\_OK if successful

Definition at line 26 of file pkcs\_1\_os2ip.c.

```
27 {  
28     return mp_read_unsigned_bin(n, in, inlen);  
29 }
```

## 5.287 pk/pkcs1/pkcs\_1\_pss\_decode.c File Reference

### 5.287.1 Detailed Description

PKCS #1 PSS Signature Padding, Tom St Denis.

Definition in file [pkcs\\_1\\_pss\\_decode.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pkcs\_1\_pss\_decode.c:

### Functions

- [pkcs\\_1\\_pss\\_decode](#) (const unsigned char \*msghash, unsigned long msghashlen, const unsigned char \*sig, unsigned long siglen, unsigned long saltlen, int hash\_idx, unsigned long modulus\_bitlen, int \*res)

*PKCS #1 v2.00 PSS decode.*

### 5.287.2 Function Documentation

- 5.287.2.1** `int pkcs_1_pss_decode (const unsigned char *msghash, unsigned long msghashlen, const unsigned char *sig, unsigned long siglen, unsigned long saltlen, int hash_idx, unsigned long modulus_bitlen, int *res)`

PKCS #1 v2.00 PSS decode.

#### Parameters:

*msghash* The hash to verify  
*msghashlen* The length of the hash (octets)  
*sig* The signature data (encoded data)  
*siglen* The length of the signature data (octets)  
*saltlen* The length of the salt used (octets)  
*hash\_idx* The index of the hash desired  
*modulus\_bitlen* The bit length of the RSA modulus  
*res* [out] The result of the comparison, 1==valid, 0==invalid

#### Returns:

CRYPT\_OK if successful (even if the comparison failed)

Definition at line 32 of file pkcs\_1\_pss\_decode.c.

References CRYPT\_MEM, CRYPT\_OK, CRYPT\_PK\_INVALID\_SIZE, hash\_descriptor, hash\_is\_valid(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, mask, pkcs\_1\_mgf1(), XFREE, XMALLOC, and XMEMCPY.

Referenced by rsa\_verify\_hash\_ex().

```
36 {
37     unsigned char *DB, *mask, *salt, *hash;
38     unsigned long x, y, hLen, modulus_len;
```

```

39     int          err;
40     hash_state   md;
41
42     LTC_ARGCHK(msghash != NULL);
43     LTC_ARGCHK(res      != NULL);
44
45     /* default to invalid */
46     *res = 0;
47
48     /* ensure hash is valid */
49     if ((err = hash_is_valid(hash_idx)) != CRYPT_OK) {
50         return err;
51     }
52
53     hLen      = hash_descriptor[hash_idx].hashsize;
54     modulus_len = (modulus_bitlen >> 3) + (modulus_bitlen & 7 ? 1 : 0);
55
56     /* check sizes */
57     if ((saltlen > modulus_len) ||
58         (modulus_len < hLen + saltlen + 2) || (siglen != modulus_len)) {
59         return CRYPT_PK_INVALID_SIZE;
60     }
61
62     /* allocate ram for DB/mask/salt/hash of size modulus_len */
63     DB = XMALLOC(modulus_len);
64     mask = XMALLOC(modulus_len);
65     salt = XMALLOC(modulus_len);
66     hash = XMALLOC(modulus_len);
67     if (DB == NULL || mask == NULL || salt == NULL || hash == NULL) {
68         if (DB != NULL) {
69             XFREE(DB);
70         }
71         if (mask != NULL) {
72             XFREE(mask);
73         }
74         if (salt != NULL) {
75             XFREE(salt);
76         }
77         if (hash != NULL) {
78             XFREE(hash);
79         }
80         return CRYPT_MEM;
81     }
82
83     /* ensure the 0xBC byte */
84     if (sig[siglen-1] != 0xBC) {
85         err = CRYPT_OK;
86         goto LBL_ERR;
87     }
88
89     /* copy out the DB */
90     x = 0;
91     XMEMCPY(DB, sig + x, modulus_len - hLen - 1);
92     x += modulus_len - hLen - 1;
93
94     /* copy out the hash */
95     XMEMCPY(hash, sig + x, hLen);
96     x += hLen;
97
98     /* check the MSB */
99     if ((sig[0] & ~(0xFF >> ((modulus_len << 3) - (modulus_bitlen - 1)))) != 0) {
100         err = CRYPT_OK;
101         goto LBL_ERR;
102     }
103
104     /* generate mask of length modulus_len - hLen - 1 from hash */
105     if ((err = pkcs_1_mgf1(hash_idx, hash, hLen, mask, modulus_len - hLen - 1)) != CRYPT_OK) {

```

```

106     goto LBL_ERR;
107 }
108
109 /* xor against DB */
110 for (y = 0; y < (modulus_len - hLen - 1); y++) {
111     DB[y] ^= mask[y];
112 }
113
114 /* now clear the first byte [make sure smaller than modulus] */
115 DB[0] &= 0xFF >> ((modulus_len<<3) - (modulus_bitlen-1));
116
117 /* DB = PS || 0x01 || salt, PS == modulus_len - saltlen - hLen - 2 zero bytes */
118
119 /* check for zeroes and 0x01 */
120 for (x = 0; x < modulus_len - saltlen - hLen - 2; x++) {
121     if (DB[x] != 0x00) {
122         err = CRYPT_OK;
123         goto LBL_ERR;
124     }
125 }
126
127 /* check for the 0x01 */
128 if (DB[x++] != 0x01) {
129     err = CRYPT_OK;
130     goto LBL_ERR;
131 }
132
133 /* M = (eight) 0x00 || msghash || salt, mask = H(M) */
134 if ((err = hash_descriptor[hash_idx].init(&md)) != CRYPT_OK) {
135     goto LBL_ERR;
136 }
137 zeromem(mask, 8);
138 if ((err = hash_descriptor[hash_idx].process(&md, mask, 8)) != CRYPT_OK) {
139     goto LBL_ERR;
140 }
141 if ((err = hash_descriptor[hash_idx].process(&md, msghash, msghashlen)) != CRYPT_OK) {
142     goto LBL_ERR;
143 }
144 if ((err = hash_descriptor[hash_idx].process(&md, DB+x, saltlen)) != CRYPT_OK) {
145     goto LBL_ERR;
146 }
147 if ((err = hash_descriptor[hash_idx].done(&md, mask)) != CRYPT_OK) {
148     goto LBL_ERR;
149 }
150
151 /* mask == hash means valid signature */
152 if (XMEMCMP(mask, hash, hLen) == 0) {
153     *res = 1;
154 }
155
156 err = CRYPT_OK;
157 LBL_ERR:
158 #ifdef LTC_CLEAN_STACK
159     zeromem(DB, modulus_len);
160     zeromem(mask, modulus_len);
161     zeromem(salt, modulus_len);
162     zeromem(hash, modulus_len);
163 #endif
164
165     XFREE(hash);
166     XFREE(salt);
167     XFREE(mask);
168     XFREE(DB);
169
170     return err;
171 }

```

Here is the call graph for this function:

## 5.288 pk/pkcs1/pkcs\_1\_pss\_encode.c File Reference

### 5.288.1 Detailed Description

PKCS #1 PSS Signature Padding, Tom St Denis.

Definition in file [pkcs\\_1\\_pss\\_encode.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pkcs\_1\_pss\_encode.c:

### Functions

- [int pkcs\\_1\\_pss\\_encode](#) (const unsigned char \*msghash, unsigned long msghashlen, unsigned long saltlen, [prng\\_state](#) \*prng, int prng\_idx, int hash\_idx, unsigned long modulus\_bitlen, unsigned char \*out, unsigned long \*outlen)

*PKCS #1 v2.00 Signature Encoding.*

### 5.288.2 Function Documentation

**5.288.2.1 int pkcs\_1\_pss\_encode** (const unsigned char \* *msghash*, unsigned long *msghashlen*, unsigned long *saltlen*, [prng\\_state](#) \* *prng*, int *prng\_idx*, int *hash\_idx*, unsigned long *modulus\_bitlen*, unsigned char \* *out*, unsigned long \* *outlen*)

PKCS #1 v2.00 Signature Encoding.

#### Parameters:

*msghash* The hash to encode  
*msghashlen* The length of the hash (octets)  
*saltlen* The length of the salt desired (octets)  
*prng* An active PRNG context  
*prng\_idx* The index of the PRNG desired  
*hash\_idx* The index of the hash desired  
*modulus\_bitlen* The bit length of the RSA modulus  
*out* [out] The destination of the encoding  
*outlen* [in/out] The max size and resulting size of the encoded data

#### Returns:

CRYPT\_OK if successful

Definition at line 33 of file pkcs\_1\_pss\_encode.c.

References CRYPT\_ERROR\_READPRNG, CRYPT\_MEM, CRYPT\_OK, CRYPT\_PK\_INVALID\_SIZE, hash\_descriptor, hash\_is\_valid(), ltc\_hash\_descriptor::hashsize, LTC\_ARGCHK, mask, pkcs\_1\_mgf1(), prng\_descriptor, prng\_is\_valid(), XFREE, XMALLOC, XMEMCPY, XMEMSET, and zeromem().

Referenced by rsa\_sign\_hash\_ex().

```

38 {
39     unsigned char *DB, *mask, *salt, *hash;
40     unsigned long x, y, hLen, modulus_len;
41     int          err;
42     hash_state   md;
43
44     LTC_ARGCHK(msghash != NULL);
45     LTC_ARGCHK(out      != NULL);
46     LTC_ARGCHK(outlen   != NULL);
47
48     /* ensure hash and PRNG are valid */
49     if ((err = hash_is_valid(hash_idx)) != CRYPT_OK) {
50         return err;
51     }
52     if ((err = prng_is_valid(prng_idx)) != CRYPT_OK) {
53         return err;
54     }
55
56     hLen      = hash_descriptor[hash_idx].hashsize;
57     modulus_len = (modulus_bitlen >> 3) + (modulus_bitlen & 7 ? 1 : 0);
58
59     /* check sizes */
60     if ((saltlen > modulus_len) || (modulus_len < hLen + saltlen + 2)) {
61         return CRYPT_PK_INVALID_SIZE;
62     }
63
64     /* allocate ram for DB/mask/salt/hash of size modulus_len */
65     DB = XMALLOC(modulus_len);
66     mask = XMALLOC(modulus_len);
67     salt = XMALLOC(modulus_len);
68     hash = XMALLOC(modulus_len);
69     if (DB == NULL || mask == NULL || salt == NULL || hash == NULL) {
70         if (DB != NULL) {
71             XFREE(DB);
72         }
73         if (mask != NULL) {
74             XFREE(mask);
75         }
76         if (salt != NULL) {
77             XFREE(salt);
78         }
79         if (hash != NULL) {
80             XFREE(hash);
81         }
82         return CRYPT_MEM;
83     }
84
85
86     /* generate random salt */
87     if (saltlen > 0) {
88         if (prng_descriptor[prng_idx].read(salt, saltlen, prng) != saltlen) {
89             err = CRYPT_ERROR_READPRNG;
90             goto LBL_ERR;
91         }
92     }
93
94     /* M = (eight) 0x00 || msghash || salt, hash = H(M) */
95     if ((err = hash_descriptor[hash_idx].init(&md)) != CRYPT_OK) {
96         goto LBL_ERR;
97     }
98     zeromem(DB, 8);
99     if ((err = hash_descriptor[hash_idx].process(&md, DB, 8)) != CRYPT_OK) {
100         goto LBL_ERR;
101     }
102     if ((err = hash_descriptor[hash_idx].process(&md, msghash, msghashlen)) != CRYPT_OK) {
103         goto LBL_ERR;
104     }

```

```

105     if ((err = hash_descriptor[hash_idx].process(&md, salt, saltlen)) != CRYPT_OK) {
106         goto LBL_ERR;
107     }
108     if ((err = hash_descriptor[hash_idx].done(&md, hash)) != CRYPT_OK) {
109         goto LBL_ERR;
110     }
111
112     /* generate DB = PS || 0x01 || salt, PS == modulus_len - saltlen - hLen - 2 zero bytes */
113     x = 0;
114     XMEMSET(DB + x, 0, modulus_len - saltlen - hLen - 2);
115     x += modulus_len - saltlen - hLen - 2;
116     DB[x++] = 0x01;
117     XMEMCPY(DB + x, salt, saltlen);
118     x += saltlen;
119
120     /* generate mask of length modulus_len - hLen - 1 from hash */
121     if ((err = pkcs_1_mgf1(hash_idx, hash, hLen, mask, modulus_len - hLen - 1)) != CRYPT_OK) {
122         goto LBL_ERR;
123     }
124
125     /* xor against DB */
126     for (y = 0; y < (modulus_len - hLen - 1); y++) {
127         DB[y] ^= mask[y];
128     }
129
130     /* output is DB || hash || 0xBC */
131     if (*outlen < modulus_len) {
132         *outlen = modulus_len;
133         err = CRYPT_BUFFER_OVERFLOW;
134         goto LBL_ERR;
135     }
136
137     /* DB len = modulus_len - hLen - 1 */
138     y = 0;
139     XMEMCPY(out + y, DB, modulus_len - hLen - 1);
140     y += modulus_len - hLen - 1;
141
142     /* hash */
143     XMEMCPY(out + y, hash, hLen);
144     y += hLen;
145
146     /* 0xBC */
147     out[y] = 0xBC;
148
149     /* now clear the 8*modulus_len - modulus_bitlen most significant bits */
150     out[0] &= 0xFF >> ((modulus_len<<3) - (modulus_bitlen-1));
151
152     /* store output size */
153     *outlen = modulus_len;
154     err = CRYPT_OK;
155 LBL_ERR:
156 #ifdef LTC_CLEAN_STACK
157     zeromem(DB, modulus_len);
158     zeromem(mask, modulus_len);
159     zeromem(salt, modulus_len);
160     zeromem(hash, modulus_len);
161 #endif
162
163     XFREE(hash);
164     XFREE(salt);
165     XFREE(mask);
166     XFREE(DB);
167
168     return err;
169 }

```

Here is the call graph for this function:



## 5.289 pk/pkcs1/pkcs\_1\_v1\_5\_decode.c File Reference

### 5.289.1 Detailed Description

PKCS #1 v1.5 Padding. (Andreas Lange)

Definition in file [pkcs\\_1\\_v1\\_5\\_decode.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pkcs\_1\_v1\_5\_decode.c:

### Functions

- [pkcs\\_1\\_v1\\_5\\_decode](#) (const unsigned char \*msg, unsigned long msglen, int block\_type, unsigned long modulus\_bitlen, unsigned char \*out, unsigned long \*outlen, int \*is\_valid)  
*PKCS #1 v1.5 decode.*

### 5.289.2 Function Documentation

**5.289.2.1** `int pkcs_1_v1_5_decode (const unsigned char * msg, unsigned long msglen, int block_type, unsigned long modulus_bitlen, unsigned char * out, unsigned long * outlen, int * is_valid)`

PKCS #1 v1.5 decode.

#### Parameters:

*msg* The encoded data to decode  
*msglen* The length of the encoded data (octets)  
*block\_type* Block type to use in padding (

#### See also:

ltc\_pkcs\_1\_v1\_5\_blocks)

#### Parameters:

*modulus\_bitlen* The bit length of the RSA modulus  
*out* [out] Destination of decoding  
*outlen* [in/out] The max size and resulting size of the decoding

#### Returns:

CRYPT\_OK if successful (even if invalid)

Definition at line 31 of file pkcs\_1\_v1\_5\_decode.c.

References CRYPT\_INVALID\_PACKET, and CRYPT\_PK\_INVALID\_SIZE.

Referenced by rsa\_decrypt\_key\_ex(), and rsa\_verify\_hash\_ex().

```
38 {
39     unsigned long modulus_len, ps_len, i;
40     int result;
41
```

```
42  /* default to invalid packet */
43  *is_valid = 0;
44
45  modulus_len = (modulus_bitlen >> 3) + (modulus_bitlen & 7 ? 1 : 0);
46
47  /* test message size */
48
49  if ((msglen > modulus_len) || (modulus_len < 11)) {
50      return CRYPT_PK_INVALID_SIZE;
51  }
52
53  /* separate encoded message */
54
55  if ((msg[0] != 0x00) || (msg[1] != (unsigned char)block_type)) {
56      result = CRYPT_INVALID_PACKET;
57      goto bail;
58  }
59
60  if (block_type == LTC_PKCS_1_EME) {
61      for (i = 2; i < modulus_len; i++) {
62          /* separator */
63          if (msg[i] == 0x00) { break; }
64      }
65      ps_len = i - 2;
66
67      if ((i >= modulus_len) || (ps_len < 8)) {
68          /* There was no octet with hexadecimal value 0x00 to separate ps from m,
69           * or the length of ps is less than 8 octets.
70           */
71          result = CRYPT_INVALID_PACKET;
72          goto bail;
73      }
74  } else {
75      for (i = 2; i < modulus_len - 1; i++) {
76          if (msg[i] != 0xFF) { break; }
77      }
78
79      /* separator check */
80      if (msg[i] != 0) {
81          /* There was no octet with hexadecimal value 0x00 to separate ps from m. */
82          result = CRYPT_INVALID_PACKET;
83          goto bail;
84      }
85
86      ps_len = i - 2;
87  }
88
89  if (*outlen < (msglen - (2 + ps_len + 1))) {
90      *outlen = msglen - (2 + ps_len + 1);
91      result = CRYPT_BUFFER_OVERFLOW;
92      goto bail;
93  }
94
95  *outlen = (msglen - (2 + ps_len + 1));
96  XMEMCPY(out, &msg[2 + ps_len + 1], *outlen);
97
98  /* valid packet */
99  *is_valid = 1;
100  result = CRYPT_OK;
101 bail:
102  return result;
103 } /* pkcs_1_v1_5_decode */
```

## 5.290 pk/pkcs1/pkcs\_1\_v1\_5\_encode.c File Reference

### 5.290.1 Detailed Description

PKCS #1 v1.5 Padding (Andreas Lange)

Definition in file [pkcs\\_1\\_v1\\_5\\_encode.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for pkcs\_1\_v1\_5\_encode.c:

### Functions

- `int pkcs\_1\_v1\_5\_encode(const unsigned char *msg, unsigned long msglen, int block_type, unsigned long modulus_bitlen, prng\_state *prng, int prng_idx, unsigned char *out, unsigned long *outlen)`  
*PKCS #1 v1.5 encode.*

### 5.290.2 Function Documentation

**5.290.2.1** `int pkcs\_1\_v1\_5\_encode(const unsigned char *msg, unsigned long msglen, int block_type, unsigned long modulus_bitlen, prng\_state *prng, int prng_idx, unsigned char *out, unsigned long *outlen)`

PKCS #1 v1.5 encode.

#### Parameters:

*msg* The data to encode  
*msglen* The length of the data to encode (octets)  
*block\_type* Block type to use in padding (

#### See also:

`ltc_pkcs_1_v1_5_blocks`)

#### Parameters:

*modulus\_bitlen* The bit length of the RSA modulus  
*prng* An active PRNG state (only for LTC\_PKCS\_1\_EME)  
*prng\_idx* The index of the PRNG desired (only for LTC\_PKCS\_1\_EME)  
*out* [out] The destination for the encoded data  
*outlen* [in/out] The max size and resulting size of the encoded data

#### Returns:

CRYPT\_OK if successful

Definition at line 33 of file `pkcs_1_v1_5_encode.c`.

References `CRYPT_BUFFER_OVERFLOW`, `CRYPT_ERROR_READPRNG`, `CRYPT_OK`, `CRYPT_PK_INVALID_PADDING`, `CRYPT_PK_INVALID_SIZE`, `prng_descriptor`, and `prng_is_valid()`.

Referenced by `rsa_encrypt_key_ex()`, and `rsa_sign_hash_ex()`.

```
41 {
42     unsigned long modulus_len, ps_len, i;
43     unsigned char *ps;
44     int result;
45
46     /* valid block_type? */
47     if ((block_type != LTC_PKCS_1_EMSA) &&
48         (block_type != LTC_PKCS_1_EME)) {
49         return CRYPT_PK_INVALID_PADDING;
50     }
51
52     if (block_type == LTC_PKCS_1_EME) { /* encryption padding, we need a valid PRNG */
53         if ((result = prng_is_valid(prng_idx)) != CRYPT_OK) {
54             return result;
55         }
56     }
57
58     modulus_len = (modulus_bitlen >> 3) + (modulus_bitlen & 7 ? 1 : 0);
59
60     /* test message size */
61     if ((msglen + 11) > modulus_len) {
62         return CRYPT_PK_INVALID_SIZE;
63     }
64
65     if (*outlen < modulus_len) {
66         *outlen = modulus_len;
67         result = CRYPT_BUFFER_OVERFLOW;
68         goto bail;
69     }
70
71     /* generate an octets string PS */
72     ps = &out[2];
73     ps_len = modulus_len - msglen - 3;
74
75     if (block_type == LTC_PKCS_1_EME) {
76         /* now choose a random ps */
77         if (prng_descriptor[prng_idx].read(ps, ps_len, prng) != ps_len) {
78             result = CRYPT_ERROR_READPRNG;
79             goto bail;
80         }
81
82         /* transform zero bytes (if any) to non-zero random bytes */
83         for (i = 0; i < ps_len; i++) {
84             while (ps[i] == 0) {
85                 if (prng_descriptor[prng_idx].read(&ps[i], 1, prng) != 1) {
86                     result = CRYPT_ERROR_READPRNG;
87                     goto bail;
88                 }
89             }
90         }
91     } else {
92         XMEMSET(ps, 0xFF, ps_len);
93     }
94
95     /* create string of length modulus_len */
96     out[0] = 0x00;
97     out[1] = (unsigned char)block_type; /* block_type 1 or 2 */
98     out[2 + ps_len] = 0x00;
99     XMEMCPY(&out[2 + ps_len + 1], msg, msglen);
100     *outlen = modulus_len;
101
102     result = CRYPT_OK;
103 bail:
104     return result;
105 } /* pkcs_1_v1_5_encode */
```

Here is the call graph for this function:

## 5.291 pk/rsa/rsa\_decrypt\_key.c File Reference

### 5.291.1 Detailed Description

RSA PKCS #1 Decryption, Tom St Denis and Andreas Lange.

Definition in file [rsa\\_decrypt\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for rsa\_decrypt\_key.c:

### Functions

- [int rsa\\_decrypt\\_key\\_ex](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, const unsigned char \**lparam*, unsigned long *lparamlen*, int *hash\_idx*, int *padding*, int \**stat*, [rsa\\_key](#) \**key*)

*PKCS #1 decrypt then v1.5 or OAEP depad.*

### 5.291.2 Function Documentation

**5.291.2.1** [int rsa\\_decrypt\\_key\\_ex](#) (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*, const unsigned char \* *lparam*, unsigned long *lparamlen*, int *hash\_idx*, int *padding*, int \* *stat*, [rsa\\_key](#) \* *key*)

PKCS #1 decrypt then v1.5 or OAEP depad.

#### Parameters:

- in* The ciphertext
- inlen* The length of the ciphertext (octets)
- out* [out] The plaintext
- outlen* [in/out] The max size and resulting size of the plaintext (octets)
- lparam* The system "lparam" value
- lparamlen* The length of the lparam value (octets)
- hash\_idx* The index of the hash desired
- padding* Type of padding (LTC\_PKCS\_1\_OAEP or LTC\_PKCS\_1\_V1\_5)
- stat* [out] Result of the decryption, 1==valid, 0==invalid
- key* The corresponding private RSA key

#### Returns:

- CRYPT\_OK if successful (even if invalid)

Definition at line 34 of file [rsa\\_decrypt\\_key.c](#).

References [CRYPT\\_INVALID\\_PACKET](#), [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [CRYPT\\_PK\\_INVALID\\_PADDING](#), [hash\\_is\\_valid\(\)](#), [LTC\\_ARGCHK](#), [ltc\\_mp](#), [PK\\_PRIVATE](#), [pkcs\\_1\\_oaep\\_decode\(\)](#), [pkcs\\_1\\_v1\\_5\\_decode\(\)](#), [ltc\\_math\\_descriptor::rsa\\_me](#), [XFREE](#), and [XMALLOC](#).

```
39 {
40   unsigned long modulus_bitlen, modulus_bytelen, x;
41   int          err;
42   unsigned char *tmp;
43
44   LTC_ARGCHK(out    != NULL);
45   LTC_ARGCHK(outlen != NULL);
46   LTC_ARGCHK(key    != NULL);
47   LTC_ARGCHK(stat    != NULL);
48
49   /* default to invalid */
50   *stat = 0;
51
52   /* valid padding? */
53
54   if ((padding != LTC_PKCS_1_V1_5) &&
55       (padding != LTC_PKCS_1_OAEP)) {
56     return CRYPT_PK_INVALID_PADDING;
57   }
58
59   if (padding == LTC_PKCS_1_OAEP) {
60     /* valid hash ? */
61     if ((err = hash_is_valid(hash_idx)) != CRYPT_OK) {
62       return err;
63     }
64   }
65
66   /* get modulus len in bits */
67   modulus_bitlen = mp_count_bits( (key->N));
68
69   /* outlen must be at least the size of the modulus */
70   modulus_bytelen = mp_unsigned_bin_size( (key->N));
71   if (modulus_bytelen != inlen) {
72     return CRYPT_INVALID_PACKET;
73   }
74
75   /* allocate ram */
76   tmp = XMALLOC(inlen);
77   if (tmp == NULL) {
78     return CRYPT_MEM;
79   }
80
81   /* rsa decode the packet */
82   x = inlen;
83   if ((err = ltc_mp.rsa_me(in, inlen, tmp, &x, PK_PRIVATE, key)) != CRYPT_OK) {
84     XFREE(tmp);
85     return err;
86   }
87
88   if (padding == LTC_PKCS_1_OAEP) {
89     /* now OAEP decode the packet */
90     err = pkcs_1_oaep_decode(tmp, x, lparam, lparamlen, modulus_bitlen, hash_idx,
91                             out, outlen, stat);
92   } else {
93     /* now PKCS #1 v1.5 depad the packet */
94     err = pkcs_1_v1_5_decode(tmp, x, LTC_PKCS_1_EME, modulus_bitlen, out, outlen, stat);
95   }
96
97   XFREE(tmp);
98   return err;
99 }
```

Here is the call graph for this function:

## 5.292 pk/rsa/rsa\_encrypt\_key.c File Reference

### 5.292.1 Detailed Description

RSA PKCS #1 encryption, Tom St Denis and Andreas Lange.

Definition in file [rsa\\_encrypt\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [rsa\\_encrypt\\_key.c](#):

### Functions

- [int rsa\\_encrypt\\_key\\_ex](#) (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, const unsigned char \**lparam*, unsigned long *lparamlen*, [prng\\_state](#) \**prng*, int *prng\_idx*, int *hash\_idx*, int *padding*, [rsa\\_key](#) \**key*)

(PKCS #1 v2.0) OAEP pad then encrypt

### 5.292.2 Function Documentation

**5.292.2.1** [int rsa\\_encrypt\\_key\\_ex](#) (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*, const unsigned char \* *lparam*, unsigned long *lparamlen*, [prng\\_state](#) \* *prng*, int *prng\_idx*, int *hash\_idx*, int *padding*, [rsa\\_key](#) \* *key*)

(PKCS #1 v2.0) OAEP pad then encrypt

#### Parameters:

*in* The plaintext

*inlen* The length of the plaintext (octets)

*out* [out] The ciphertext

*outlen* [in/out] The max size and resulting size of the ciphertext

*lparam* The system "lparam" for the encryption

*lparamlen* The length of lparam (octets)

*prng* An active PRNG

*prng\_idx* The index of the desired prng

*hash\_idx* The index of the desired hash

*padding* Type of padding (LTC\_PKCS\_1\_OAEP or LTC\_PKCS\_1\_V1\_5)

*key* The RSA key to encrypt to

#### Returns:

CRYPT\_OK if successful

Definition at line 35 of file [rsa\\_encrypt\\_key.c](#).

References [CRYPT\\_BUFFER\\_OVERFLOW](#), [CRYPT\\_OK](#), [CRYPT\\_PK\\_INVALID\\_PADDING](#), [hash\\_is\\_valid\(\)](#), [LTC\\_ARGCHK](#), [ltc\\_mp](#), [PK\\_PUBLIC](#), [pkcs\\_1\\_oaep\\_encode\(\)](#), [pkcs\\_1\\_v1\\_5\\_encode\(\)](#), [prng\\_is\\_valid\(\)](#), and [ltc\\_math\\_descriptor::rsa\\_me](#).

```
39 {
40     unsigned long modulus_bitlen, modulus_bytelen, x;
41     int          err;
42
43     LTC_ARGCHK(in      != NULL);
44     LTC_ARGCHK(out     != NULL);
45     LTC_ARGCHK(outlen  != NULL);
46     LTC_ARGCHK(key     != NULL);
47
48     /* valid padding? */
49     if ((padding != LTC_PKCS_1_V1_5) &&
50         (padding != LTC_PKCS_1_OAEP)) {
51         return CRYPT_PK_INVALID_PADDING;
52     }
53
54     /* valid prng? */
55     if ((err = prng_is_valid(prng_idx)) != CRYPT_OK) {
56         return err;
57     }
58
59     if (padding == LTC_PKCS_1_OAEP) {
60         /* valid hash? */
61         if ((err = hash_is_valid(hash_idx)) != CRYPT_OK) {
62             return err;
63         }
64     }
65
66     /* get modulus len in bits */
67     modulus_bitlen = mp_count_bits( (key->N));
68
69     /* outlen must be at least the size of the modulus */
70     modulus_bytelen = mp_unsigned_bin_size( (key->N));
71     if (modulus_bytelen > *outlen) {
72         *outlen = modulus_bytelen;
73         return CRYPT_BUFFER_OVERFLOW;
74     }
75
76     if (padding == LTC_PKCS_1_OAEP) {
77         /* OAEP pad the key */
78         x = *outlen;
79         if ((err = pkcs_1_oaep_encode(in, inlen, lparam,
80                                     lparamlen, modulus_bitlen, prng, prng_idx, hash_idx,
81                                     out, &x)) != CRYPT_OK) {
82             return err;
83         }
84     } else {
85         /* PKCS #1 v1.5 pad the key */
86         x = *outlen;
87         if ((err = pkcs_1_v1_5_encode(in, inlen, LTC_PKCS_1_EME,
88                                     modulus_bitlen, prng, prng_idx,
89                                     out, &x)) != CRYPT_OK) {
90             return err;
91         }
92     }
93
94     /* rsa exptmod the OAEP or PKCS #1 v1.5 pad */
95     return ltc_mp.rsa_me(out, x, out, outlen, PK_PUBLIC, key);
96 }
```

Here is the call graph for this function:



## 5.293 pk/rsa/rsa\_export.c File Reference

### 5.293.1 Detailed Description

Export RSA PKCS keys, Tom St Denis.

Definition in file [rsa\\_export.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `rsa_export.c`:

### Functions

- `int rsa_export` (unsigned char \*out, unsigned long \*outlen, int type, [rsa\\_key](#) \*key)

*This will export either an RSAPublicKey or RSAPrivateKey [defined in PKCS #1 v2.1].*

### 5.293.2 Function Documentation

#### 5.293.2.1 `int rsa_export` (unsigned char \* out, unsigned long \* outlen, int type, [rsa\\_key](#) \* key)

This will export either an RSAPublicKey or RSAPrivateKey [defined in PKCS #1 v2.1].

#### Parameters:

- out** [out] Destination of the packet
- outlen** [in/out] The max size and resulting size of the packet
- type** The type of exported key (PK\_PRIVATE or PK\_PUBLIC)
- key** The RSA key to export

#### Returns:

- CRYPT\_OK if successful

Definition at line 28 of file `rsa_export.c`.

References `CRYPT_PK_INVALID_TYPE`, `der_encode_sequence_multi()`, `LTC_ARGCHK`, and `PK_PRIVATE`.

```
29 {
30     unsigned long zero=0;
31     LTC_ARGCHK(out != NULL);
32     LTC_ARGCHK(outlen != NULL);
33     LTC_ARGCHK(key != NULL);
34
35     /* type valid? */
36     if (!(key->type == PK_PRIVATE) && (type == PK_PRIVATE)) {
37         return CRYPT_PK_INVALID_TYPE;
38     }
39
40     if (type == PK_PRIVATE) {
41         /* private key */
42         /* output is
43            Version, n, e, d, p, q, d mod (p-1), d mod (q - 1), 1/q mod p
44            */
45         return der_encode_sequence_multi(out, outlen,
```

```
46             LTC_ASN1_SHORT_INTEGER, 1UL, &zero,
47             LTC_ASN1_INTEGER, 1UL, key->N,
48             LTC_ASN1_INTEGER, 1UL, key->e,
49             LTC_ASN1_INTEGER, 1UL, key->d,
50             LTC_ASN1_INTEGER, 1UL, key->p,
51             LTC_ASN1_INTEGER, 1UL, key->q,
52             LTC_ASN1_INTEGER, 1UL, key->dP,
53             LTC_ASN1_INTEGER, 1UL, key->dQ,
54             LTC_ASN1_INTEGER, 1UL, key->qP,
55             LTC_ASN1_EOL, 0UL, NULL);
56     } else {
57         /* public key */
58         return der_encode_sequence_multi(out, outlen,
59             LTC_ASN1_INTEGER, 1UL, key->N,
60             LTC_ASN1_INTEGER, 1UL, key->e,
61             LTC_ASN1_EOL, 0UL, NULL);
62     }
63 }
```

Here is the call graph for this function:

## 5.294 pk/rsa/rsa\_exptmod.c File Reference

### 5.294.1 Detailed Description

RSA PKCS exptmod, Tom St Denis.

Definition in file [rsa\\_exptmod.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for rsa\_exptmod.c:

### Functions

- `int rsa_exptmod` (const unsigned char \**in*, unsigned long *inlen*, unsigned char \**out*, unsigned long \**outlen*, int *which*, [rsa\\_key](#) \**key*)

*Compute an RSA modular exponentiation.*

### 5.294.2 Function Documentation

#### 5.294.2.1 `int rsa_exptmod` (const unsigned char \* *in*, unsigned long *inlen*, unsigned char \* *out*, unsigned long \* *outlen*, int *which*, [rsa\\_key](#) \* *key*)

Compute an RSA modular exponentiation.

#### Parameters:

*in* The input data to send into RSA

*inlen* The length of the input (octets)

*out* [out] The destination

*outlen* [in/out] The max size and resulting size of the output

*which* Which exponent to use, e.g. PK\_PRIVATE or PK\_PUBLIC

*key* The RSA key to use

#### Returns:

CRYPT\_OK if successful

Definition at line 30 of file `rsa_exptmod.c`.

References `CRYPT_BUFFER_OVERFLOW`, `CRYPT_ERROR`, `CRYPT_OK`, `CRYPT_PK_INVALID_SIZE`, `CRYPT_PK_INVALID_TYPE`, `CRYPT_PK_NOT_PRIVATE`, `LTC_ARGCHK`, `LTC_MP_LT`, `PK_PRIVATE`, `PK_PUBLIC`, and `zeromem()`.

```
33 {
34     void          *tmp, *tmpa, *tmpb;
35     unsigned long x;
36     int           err;
37
38     LTC_ARGCHK(in    != NULL);
39     LTC_ARGCHK(out   != NULL);
40     LTC_ARGCHK(outlen != NULL);
41     LTC_ARGCHK(key   != NULL);
42 }
```

```

43  /* is the key of the right type for the operation? */
44  if (which == PK_PRIVATE && (key->type != PK_PRIVATE)) {
45      return CRYPT_PK_NOT_PRIVATE;
46  }
47
48  /* must be a private or public operation */
49  if (which != PK_PRIVATE && which != PK_PUBLIC) {
50      return CRYPT_PK_INVALID_TYPE;
51  }
52
53  /* init and copy into tmp */
54  if ((err = mp_init_multi(&tmp, &tmpa, &tmpb, NULL)) != CRYPT_OK)
55  if ((err = mp_read_unsigned_bin(tmp, (unsigned char *)in, (int)inlen)) != CRYPT_OK)
56
57  /* sanity check on the input */
58  if (mp_cmp(key->N, tmp) == LTC_MP_LT) {
59      err = CRYPT_PK_INVALID_SIZE;
60      goto done;
61  }
62
63  /* are we using the private exponent and is the key optimized? */
64  if (which == PK_PRIVATE) {
65      /* tmpa = tmp^dP mod p */
66      if ((err = mp_exptmod(tmp, key->dP, key->p, tmpa)) != CRYPT_OK) { g
67
68      /* tmpb = tmp^dQ mod q */
69      if ((err = mp_exptmod(tmp, key->dQ, key->q, tmpb)) != CRYPT_OK) { g
70
71      /* tmp = (tmpa - tmpb) * qInv (mod p) */
72      if ((err = mp_sub(tmpa, tmpb, tmp)) != CRYPT_OK) { g
73      if ((err = mp_mulmod(tmp, key->qP, key->p, tmp)) != CRYPT_OK) { go
74
75      /* tmp = tmpb + q * tmp */
76      if ((err = mp_mul(tmp, key->q, tmp)) != CRYPT_OK) { g
77      if ((err = mp_add(tmp, tmpb, tmp)) != CRYPT_OK) { g
78  } else {
79      /* exptmod it */
80      if ((err = mp_exptmod(tmp, key->e, key->N, tmp)) != CRYPT_OK) { go
81  }
82
83  /* read it back */
84  x = (unsigned long)mp_unsigned_bin_size(key->N);
85  if (x > *outlen) {
86      *outlen = x;
87      err = CRYPT_BUFFER_OVERFLOW;
88      goto done;
89  }
90
91  /* this should never happen ... */
92  if (mp_unsigned_bin_size(tmp) > mp_unsigned_bin_size(key->N)) {
93      err = CRYPT_ERROR;
94      goto done;
95  }
96  *outlen = x;
97
98  /* convert it */
99  zeromem(out, x);
100  if ((err = mp_to_unsigned_bin(tmp, out+(x-mp_unsigned_bin_size(tmp)))) != CRYPT_OK) {
101
102  /* clean up and return */
103  err = CRYPT_OK;
104  goto done;
105 error:
106 done:
107  mp_clear_multi(tmp, tmpa, tmpb, NULL);
108  return err;
109 }

```

Here is the call graph for this function:

## 5.295 pk/rsa/rsa\_free.c File Reference

### 5.295.1 Detailed Description

Free an RSA key, Tom St Denis.

Definition in file [rsa\\_free.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `rsa_free.c`:

### Functions

- void [rsa\\_free](#) ([rsa\\_key](#) \*key)  
*Free an RSA key from memory.*

### 5.295.2 Function Documentation

#### 5.295.2.1 void [rsa\\_free](#) ([rsa\\_key](#) \*key)

Free an RSA key from memory.

#### Parameters:

*key* The RSA key to free

Definition at line 24 of file `rsa_free.c`.

References `LTC_ARGCHKVD`.

```
25 {  
26     LTC_ARGCHKVD(key != NULL);  
27     mp_clear_multi( key->e,  key->d,  key->N,  key->dQ,  key->dP,  
28                   key->qP,  key->p,  key->q,  NULL);  
29 }
```

## 5.296 pk/rsa/rsa\_import.c File Reference

### 5.296.1 Detailed Description

Import a PKCS RSA key, Tom St Denis.

Definition in file [rsa\\_import.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `rsa_import.c`:

### Functions

- `int rsa_import` (const unsigned char \**in*, unsigned long *inlen*, [rsa\\_key](#) \**key*)  
*Import an RSAPublicKey or RSAPrivateKey [two-prime only, only support >= 1024-bit keys, defined in PKCS #1 v2.1].*

### 5.296.2 Function Documentation

#### 5.296.2.1 `int rsa_import` (const unsigned char \**in*, unsigned long *inlen*, [rsa\\_key](#) \**key*)

Import an RSAPublicKey or RSAPrivateKey [two-prime only, only support >= 1024-bit keys, defined in PKCS #1 v2.1].

#### Parameters:

- in* The packet to import from
- inlen* It's length (octets)
- key* [out] Destination for newly imported key

#### Returns:

CRYPT\_OK if successful, upon error allocated memory is freed

Definition at line 27 of file `rsa_import.c`.

References `CRYPT_MEM`, `CRYPT_OK`, `LTC_ARGCHK`, `ltc_mp`, `ltc_math_descriptor::name`, and `XCALLOC`.

```
28 {
29     int          err;
30     void          *zero;
31     unsigned char *tmpbuf;
32     unsigned long t, x, y, z, tmpoid[16];
33     ltc_asn1_list ssl_pubkey_hashoid[2];
34     ltc_asn1_list ssl_pubkey[2];
35
36     LTC_ARGCHK(in != NULL);
37     LTC_ARGCHK(key != NULL);
38     LTC_ARGCHK(ltc_mp.name != NULL);
39
40     /* init key */
41     if ((err = mp_init_multi(&key->e, &key->d, &key->N, &key->dQ,
42                             &key->dP, &key->qP, &key->p, &key->q, NULL)) != CRYPT_OK) {
43         return err;
44     }
```

```

45
46  /* see if the OpenSSL DER format RSA public key will work */
47  tmpbuf = XCALLOC(1, MAX_RSA_SIZE*8);
48  if (tmpbuf == NULL) {
49      err = CRYPT_MEM;
50      goto LBL_ERR;
51  }
52
53  /* this includes the internal hash ID and optional params (NULL in this case) */
54  LTC_SET_ASN1(ssl_pubkey_hashoid, 0, LTC_ASN1_OBJECT_IDENTIFIER, tmpoid, sizeof(tmpoid));
55  LTC_SET_ASN1(ssl_pubkey_hashoid, 1, LTC_ASN1_NULL, NULL, 0);
56
57  /* the actual format of the SSL DER key is odd, it stores a RSAPublicKey in a **BIT** string ... so
58     then proceed to convert bit to octet
59     */
60  LTC_SET_ASN1(ssl_pubkey, 0, LTC_ASN1_SEQUENCE, &ssl_pubkey_hashoid, 2);
61  LTC_SET_ASN1(ssl_pubkey, 1, LTC_ASN1_BIT_STRING, tmpbuf, MAX_RSA_SIZE*8);
62
63  if (der_decode_sequence(in, inlen,
64                          ssl_pubkey, 2UL) == CRYPT_OK) {
65
66      /* ok now we have to reassemble the BIT STRING to an OCTET STRING.  Thanks OpenSSL... */
67      for (t = y = z = x = 0; x < ssl_pubkey[1].size; x++) {
68          y = (y << 1) | tmpbuf[x];
69          if (++z == 8) {
70              tmpbuf[t++] = y;
71              y = 0;
72              z = 0;
73          }
74      }
75
76      /* now it should be SEQUENCE { INTEGER, INTEGER } */
77      if ((err = der_decode_sequence_multi(tmpbuf, t,
78                                          LTC_ASN1_INTEGER, 1UL, key->N,
79                                          LTC_ASN1_INTEGER, 1UL, key->e,
80                                          LTC_ASN1_EOL, 0UL, NULL)) != CRYPT_OK) {
81          XFREE(tmpbuf);
82          goto LBL_ERR;
83      }
84      XFREE(tmpbuf);
85      key->type = PK_PUBLIC;
86      return CRYPT_OK;
87  }
88  XFREE(tmpbuf);
89
90  /* not SSL public key, try to match against PKCS #1 standards */
91  if ((err = der_decode_sequence_multi(in, inlen,
92                                      LTC_ASN1_INTEGER, 1UL, key->N,
93                                      LTC_ASN1_EOL, 0UL, NULL)) != CRYPT_OK) {
94      goto LBL_ERR;
95  }
96
97  if (mp_cmp_d(key->N, 0) == LTC_MP_EQ) {
98      if ((err = mp_init(&zero)) != CRYPT_OK) {
99          goto LBL_ERR;
100      }
101      /* it's a private key */
102      if ((err = der_decode_sequence_multi(in, inlen,
103                                          LTC_ASN1_INTEGER, 1UL, zero,
104                                          LTC_ASN1_INTEGER, 1UL, key->N,
105                                          LTC_ASN1_INTEGER, 1UL, key->e,
106                                          LTC_ASN1_INTEGER, 1UL, key->d,
107                                          LTC_ASN1_INTEGER, 1UL, key->p,
108                                          LTC_ASN1_INTEGER, 1UL, key->q,
109                                          LTC_ASN1_INTEGER, 1UL, key->dP,
110                                          LTC_ASN1_INTEGER, 1UL, key->dQ,
111                                          LTC_ASN1_INTEGER, 1UL, key->qP,

```



```
112             LTC_ASN1_EOL,      0UL, NULL)) != CRYPT_OK) {
113         mp_clear(zero);
114         goto LBL_ERR;
115     }
116     mp_clear(zero);
117     key->type = PK_PRIVATE;
118 } else if (mp_cmp_d(key->N, 1) == LTC_MP_EQ) {
119     /* we don't support multi-prime RSA */
120     err = CRYPT_PK_INVALID_TYPE;
121     goto LBL_ERR;
122 } else {
123     /* it's a public key and we lack e */
124     if ((err = der_decode_sequence_multi(in, inlen,
125                                         LTC_ASN1_INTEGER, 1UL, key->N,
126                                         LTC_ASN1_INTEGER, 1UL, key->e,
127                                         LTC_ASN1_EOL,      0UL, NULL)) != CRYPT_OK) {
128         goto LBL_ERR;
129     }
130     key->type = PK_PUBLIC;
131 }
132 return CRYPT_OK;
133 LBL_ERR:
134 mp_clear_multi(key->d, key->e, key->N, key->dQ, key->dP,
135                key->qP, key->p, key->q, NULL);
136 return err;
137 }
```

## 5.297 pk/rsa/rsa\_make\_key.c File Reference

### 5.297.1 Detailed Description

RSA key generation, Tom St Denis.

Definition in file [rsa\\_make\\_key.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [rsa\\_make\\_key.c](#):

### Functions

- `int rsa\_make\_key (prng\_state *prng, int wprng, int size, long e, rsa\_key *key)`  
*Create an RSA key.*

### 5.297.2 Function Documentation

#### 5.297.2.1 `int rsa\_make\_key (prng\_state *prng, int wprng, int size, long e, rsa\_key *key)`

Create an RSA key.

#### Parameters:

- prng* An active PRNG state
- wprng* The index of the PRNG desired
- size* The size of the modulus (key size) desired (octets)
- e* The "e" value (public key). e==65537 is a good choice
- key* [out] Destination of a newly created private key pair

#### Returns:

CRYPT\_OK if successful, upon error all allocated ram is freed

Definition at line 29 of file [rsa\\_make\\_key.c](#).

References [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_INVALID\\_KEYSIZE](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), [ltc\\_mp](#), [ltc\\_math\\_descriptor::name](#), [PK\\_PRIVATE](#), [prng\\_is\\_valid\(\)](#), and [rand\\_prime\(\)](#).

```
30 {
31     void *p, *q, *tmp1, *tmp2, *tmp3;
32     int     err;
33
34     LTC_ARGCHK(ltc_mp.name != NULL);
35     LTC_ARGCHK(key != NULL);
36
37     if ((size < (MIN_RSA_SIZE/8)) || (size > (MAX_RSA_SIZE/8))) {
38         return CRYPT_INVALID_KEYSIZE;
39     }
40
41     if ((e < 3) || ((e & 1) == 0)) {
42         return CRYPT_INVALID_ARG;
43     }
44
45     if ((err = prng_is_valid(wprng)) != CRYPT_OK) {
```

```

46     return err;
47 }
48
49 if ((err = mp_init_multi(&p, &q, &tmp1, &tmp2, &tmp3, NULL)) != CRYPT_OK) {
50     return err;
51 }
52
53 /* make primes p and q (optimization provided by Wayne Scott) */
54 if ((err = mp_set_int(tmp3, e)) != CRYPT_OK) { goto error; }          /* tmp3 = e */
55
56 /* make prime "p" */
57 do {
58     if ((err = rand_prime(p, size/2, prng, wprng)) != CRYPT_OK) { goto done; }
59     if ((err = mp_sub_d(p, 1, tmp1)) != CRYPT_OK) { goto error; } /* tmp1 = p-1 */
60     if ((err = mp_gcd(tmp1, tmp3, tmp2)) != CRYPT_OK) { goto error; } /* tmp2 = gcd(p-1,
61 } while (mp_cmp_d(tmp2, 1) != 0);                                     /* while e divides
62
63 /* make prime "q" */
64 do {
65     if ((err = rand_prime(q, size/2, prng, wprng)) != CRYPT_OK) { goto done; }
66     if ((err = mp_sub_d(q, 1, tmp1)) != CRYPT_OK) { goto error; } /* tmp1 = q-1 */
67     if ((err = mp_gcd(tmp1, tmp3, tmp2)) != CRYPT_OK) { goto error; } /* tmp2 = gcd(q-1,
68 } while (mp_cmp_d(tmp2, 1) != 0);                                     /* while e divides
69
70 /* tmp1 = lcm(p-1, q-1) */
71 if ((err = mp_sub_d(p, 1, tmp2)) != CRYPT_OK) { goto error; } /* tmp2 = p-1 */
72 /* tmp1 = q-1 (previous do/while loop) */
73 if ((err = mp_lcm(tmp1, tmp2, tmp1)) != CRYPT_OK) { goto error; } /* tmp1 = lcm(p-1,
74
75 /* make key */
76 if ((err = mp_init_multi(&key->e, &key->d, &key->N, &key->dQ, &key->dP,
77     &key->qP, &key->p, &key->q, NULL)) != CRYPT_OK) {
78     goto error;
79 }
80
81 if ((err = mp_set_int(key->e, e)) != CRYPT_OK) { goto error2; } /* key->e = e
82 if ((err = mp_invmod(key->e, tmp1, key->d)) != CRYPT_OK) { goto error2; } /* key->d = 1/e
83 if ((err = mp_mul(p, q, key->N)) != CRYPT_OK) { goto error2; } /* key->N = pq
84
85 /* optimize for CRT now */
86 /* find d mod q-1 and d mod p-1 */
87 if ((err = mp_sub_d(p, 1, tmp1)) != CRYPT_OK) { goto error2; } /* tmp1 = q-1
88 if ((err = mp_sub_d(q, 1, tmp2)) != CRYPT_OK) { goto error2; } /* tmp2 = p-1
89 if ((err = mp_mod(key->d, tmp1, key->dP)) != CRYPT_OK) { goto error2; } /* dP = d mod p
90 if ((err = mp_mod(key->d, tmp2, key->dQ)) != CRYPT_OK) { goto error2; } /* dQ = d mod q
91 if ((err = mp_invmod(q, p, key->qP)) != CRYPT_OK) { goto error2; } /* qP = 1/q mod p
92
93 if ((err = mp_copy(p, key->p)) != CRYPT_OK) { goto error2; }
94 if ((err = mp_copy(q, key->q)) != CRYPT_OK) { goto error2; }
95
96 /* set key type (in this case it's CRT optimized) */
97 key->type = PK_PRIVATE;
98
99 /* return ok and free temps */
100 err = CRYPT_OK;
101 goto done;
102 error2:
103     mp_clear_multi(key->d, key->e, key->N, key->dQ, key->dP,
104         key->qP, key->p, key->q, NULL);
105 error:
106 done:
107     mp_clear_multi(tmp3, tmp2, tmp1, p, q, NULL);
108     return err;
109 }

```

Here is the call graph for this function:

## 5.298 pk/rsa/rsa\_sign\_hash.c File Reference

### 5.298.1 Detailed Description

RSA PKCS #1 v1.5 and v2 PSS sign hash, Tom St Denis and Andreas Lange.

Definition in file [rsa\\_sign\\_hash.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [rsa\\_sign\\_hash.c](#):

### Functions

- [int rsa\\_sign\\_hash\\_ex](#) (const unsigned char \*[in](#), unsigned long [inlen](#), unsigned char \*[out](#), unsigned long \*[outlen](#), int [padding](#), [prng\\_state](#) \*[prng](#), int [prng\\_idx](#), int [hash\\_idx](#), unsigned long [saltlen](#), [rsa\\_key](#) \*[key](#))

*PKCS #1 pad then sign.*

### 5.298.2 Function Documentation

**5.298.2.1** [int rsa\\_sign\\_hash\\_ex](#) (const unsigned char \* [in](#), unsigned long [inlen](#), unsigned char \* [out](#), unsigned long \* [outlen](#), int [padding](#), [prng\\_state](#) \* [prng](#), int [prng\\_idx](#), int [hash\\_idx](#), unsigned long [saltlen](#), [rsa\\_key](#) \* [key](#))

PKCS #1 pad then sign.

#### Parameters:

- [in](#)* The hash to sign
- [inlen](#)* The length of the hash to sign (octets)
- [out](#)* [out] The signature
- [outlen](#)* [in/out] The max size and resulting size of the signature
- [padding](#)* Type of padding (LTC\_PKCS\_1\_PSS or LTC\_PKCS\_1\_V1\_5)
- [prng](#)* An active PRNG state
- [prng\\_idx](#)* The index of the PRNG desired
- [hash\\_idx](#)* The index of the hash desired
- [saltlen](#)* The length of the salt desired (octets)
- [key](#)* The private RSA key to use

#### Returns:

- CRYPT\_OK if successful

Definition at line 34 of file [rsa\\_sign\\_hash.c](#).

References [CRYPT\\_BUFFER\\_OVERFLOW](#), [CRYPT\\_INVALID\\_ARG](#), [CRYPT\\_MEM](#), [CRYPT\\_OK](#), [CRYPT\\_PK\\_INVALID\\_PADDING](#), [hash\\_descriptor](#), [hash\\_is\\_valid\(\)](#), [LTC\\_ARGCHK](#), [ltc\\_mp](#), [PK\\_PRIVATE](#), [pkcs\\_1\\_pss\\_encode\(\)](#), [pkcs\\_1\\_v1\\_5\\_encode\(\)](#), [prng\\_is\\_valid\(\)](#), [ltc\\_math\\_descriptor::rsa\\_me](#), [XFREE](#), and [XMALLOC](#).

```

40 {
41     unsigned long modulus_bitlen, modulus_bytelen, x, y;
42     int          err;
43
44     LTC_ARGCHK(in      != NULL);
45     LTC_ARGCHK(out     != NULL);
46     LTC_ARGCHK(outlen  != NULL);
47     LTC_ARGCHK(key     != NULL);
48
49     /* valid padding? */
50     if ((padding != LTC_PKCS_1_V1_5) && (padding != LTC_PKCS_1_PSS)) {
51         return CRYPT_PK_INVALID_PADDING;
52     }
53
54     if (padding == LTC_PKCS_1_PSS) {
55         /* valid prng and hash ? */
56         if ((err = prng_is_valid(prng_idx)) != CRYPT_OK) {
57             return err;
58         }
59         if ((err = hash_is_valid(hash_idx)) != CRYPT_OK) {
60             return err;
61         }
62     }
63
64     /* get modulus len in bits */
65     modulus_bitlen = mp_count_bits((key->N));
66
67     /* outlen must be at least the size of the modulus */
68     modulus_bytelen = mp_unsigned_bin_size((key->N));
69     if (modulus_bytelen > *outlen) {
70         *outlen = modulus_bytelen;
71         return CRYPT_BUFFER_OVERFLOW;
72     }
73
74     if (padding == LTC_PKCS_1_PSS) {
75         /* PSS pad the key */
76         x = *outlen;
77         if ((err = pkcs_1_pss_encode(in, inlen, saltlen, prng, prng_idx,
78                                     hash_idx, modulus_bitlen, out, &x)) != CRYPT_OK) {
79             return err;
80         }
81     } else {
82         /* PKCS #1 v1.5 pad the hash */
83         unsigned char *tmpin;
84         ltc_asn1_list digestinfo[2], siginfo[2];
85
86         /* not all hashes have OIDs... so sad */
87         if (hash_descriptor[hash_idx].OIDlen == 0) {
88             return CRYPT_INVALID_ARG;
89         }
90
91         /* construct the SEQUENCE
92         SEQUENCE {
93             SEQUENCE {hashoid OID
94                       blah      NULL
95             }
96             hash      OCTET STRING
97         }
98         */
99         LTC_SET_ASN1(digestinfo, 0, LTC_ASN1_OBJECT_IDENTIFIER, hash_descriptor[hash_idx].OID, hash_descrip
100         LTC_SET_ASN1(digestinfo, 1, LTC_ASN1_NULL, NULL, 0);
101         LTC_SET_ASN1(siginfo, 0, LTC_ASN1_SEQUENCE, digestinfo, 2);
102         LTC_SET_ASN1(siginfo, 1, LTC_ASN1_OCTET_STRING, in, inlen);
103
104         /* allocate memory for the encoding */
105         y = mp_unsigned_bin_size(key->N);
106         tmpin = XMALLOC(y);

```

```
107     if (tmpin == NULL) {
108         return CRYPT_MEM;
109     }
110
111     if ((err = der_encode_sequence(siginfo, 2, tmpin, &y)) != CRYPT_OK) {
112         XFREE(tmpin);
113         return err;
114     }
115
116     x = *outlen;
117     if ((err = pkcs_1_v1_5_encode(tmpin, y, LTC_PKCS_1_EMSA,
118                                  modulus_bitlen, NULL, 0,
119                                  out, &x)) != CRYPT_OK) {
120         XFREE(tmpin);
121         return err;
122     }
123     XFREE(tmpin);
124 }
125
126 /* RSA encode it */
127 return ltc_mp.rsa_me(out, x, out, outlen, PK_PRIVATE, key);
128 }
```

Here is the call graph for this function:

## 5.299 pk/rsa/rsa\_verify\_hash.c File Reference

### 5.299.1 Detailed Description

RSA PKCS #1 v1.5 or v2 PSS signature verification, Tom St Denis and Andreas Lange.

Definition in file [rsa\\_verify\\_hash.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for `rsa_verify_hash.c`:

### Functions

- `int rsa\_verify\_hash\_ex (const unsigned char *sig, unsigned long siglen, const unsigned char *hash, unsigned long hashlen, int padding, int hash_idx, unsigned long saltlen, int *stat, rsa\_key *key)`  
*PKCS #1 de-sign then v1.5 or PSS depad.*

### 5.299.2 Function Documentation

- 5.299.2.1** `int rsa\_verify\_hash\_ex (const unsigned char *sig, unsigned long siglen, const unsigned char *hash, unsigned long hashlen, int padding, int hash_idx, unsigned long saltlen, int *stat, rsa\_key *key)`

PKCS #1 de-sign then v1.5 or PSS depad.

#### Parameters:

- sig* The signature data
- siglen* The length of the signature data (octets)
- hash* The hash of the message that was signed
- hashlen* The length of the hash of the message that was signed (octets)
- padding* Type of padding (LTC\_PKCS\_1\_PSS or LTC\_PKCS\_1\_V1\_5)
- hash\_idx* The index of the desired hash
- saltlen* The length of the salt used during signature
- stat* [out] The result of the signature comparison, 1==valid, 0==invalid
- key* The public RSA key corresponding to the key that performed the signature

#### Returns:

- CRYPT\_OK on success (even if the signature is invalid)

Definition at line 33 of file `rsa_verify_hash.c`.

References `CRYPT_INVALID_ARG`, `CRYPT_INVALID_PACKET`, `CRYPT_MEM`, `CRYPT_OK`, `CRYPT_PK_INVALID_PADDING`, `hash_descriptor`, `hash_is_valid()`, `LTC_ARGCHK`, `ltc_mp`, `PK_PUBLIC`, `pkcs_1_pss_decode()`, `pkcs_1_v1_5_decode()`, `ltc_math_descriptor::rsa_me`, `edge::size`, `XFREE`, `XMALLOC`, `XMEMCMP`, and `zeromem()`.

```
38 {
39     unsigned long modulus_bitlen, modulus_bytelen, x;
40     int          err;
```

```

41 unsigned char *tmpbuf;
42
43 LTC_ARGCHK(hash != NULL);
44 LTC_ARGCHK(sig != NULL);
45 LTC_ARGCHK(stat != NULL);
46 LTC_ARGCHK(key != NULL);
47
48 /* default to invalid */
49 *stat = 0;
50
51 /* valid padding? */
52
53 if ((padding != LTC_PKCS_1_V1_5) &&
54     (padding != LTC_PKCS_1_PSS)) {
55     return CRYPT_PK_INVALID_PADDING;
56 }
57
58 if (padding == LTC_PKCS_1_PSS) {
59     /* valid hash ? */
60     if ((err = hash_is_valid(hash_idx)) != CRYPT_OK) {
61         return err;
62     }
63 }
64
65 /* get modulus len in bits */
66 modulus_bitlen = mp_count_bits( (key->N));
67
68 /* outlen must be at least the size of the modulus */
69 modulus_bytelen = mp_unsigned_bin_size( (key->N));
70 if (modulus_bytelen != siglen) {
71     return CRYPT_INVALID_PACKET;
72 }
73
74 /* allocate temp buffer for decoded sig */
75 tmpbuf = XMALLOC(siglen);
76 if (tmpbuf == NULL) {
77     return CRYPT_MEM;
78 }
79
80 /* RSA decode it */
81 x = siglen;
82 if ((err = ltc_mp.rsa_me(sig, siglen, tmpbuf, &x, PK_PUBLIC, key)) != CRYPT_OK) {
83     XFREE(tmpbuf);
84     return err;
85 }
86
87 /* make sure the output is the right size */
88 if (x != siglen) {
89     return CRYPT_INVALID_PACKET;
90 }
91
92 if (padding == LTC_PKCS_1_PSS) {
93     /* PSS decode and verify it */
94     err = pkcs_1_pss_decode(hash, hashlen, tmpbuf, x, saltlen, hash_idx, modulus_bitlen, stat);
95 } else {
96     /* PKCS #1 v1.5 decode it */
97     unsigned char *out;
98     unsigned long outlen, loid[16];
99     int decoded;
100     ltc_asn1_list digestinfo[2], siginfo[2];
101
102     /* not all hashes have OIDs... so sad */
103     if (hash_descriptor[hash_idx].OIDlen == 0) {
104         err = CRYPT_INVALID_ARG;
105         goto bail_2;
106     }
107

```



```

108     /* allocate temp buffer for decoded hash */
109     outlen = ((modulus_bitlen >> 3) + (modulus_bitlen & 7 ? 1 : 0)) - 3;
110     out = XMALLOC(outlen);
111     if (out == NULL) {
112         err = CRYPT_MEM;
113         goto bail_2;
114     }
115
116     if ((err = pkcs_1_v1_5_decode(tmpbuf, x, LTC_PKCS_1_EMSA, modulus_bitlen, out, &outlen, &decoded))
117         XFREE(out);
118         goto bail_2;
119     }
120
121     /* now we must decode out[0...outlen-1] using ASN.1, test the OID and then test the hash */
122     /* construct the SEQUENCE
123     SEQUENCE {
124         SEQUENCE {hashoid OID
125             blah NULL
126         }
127         hash OCTET STRING
128     }
129     */
130     LTC_SET_ASN1(digestinfo, 0, LTC_ASN1_OBJECT_IDENTIFIER, loid, sizeof(loid)/sizeof(loid[0]));
131     LTC_SET_ASN1(digestinfo, 1, LTC_ASN1_NULL, NULL, 0);
132     LTC_SET_ASN1(siginfo, 0, LTC_ASN1_SEQUENCE, digestinfo, 2);
133     LTC_SET_ASN1(siginfo, 1, LTC_ASN1_OCTET_STRING, tmpbuf, siglen);
134
135     if ((err = der_decode_sequence(out, outlen, siginfo, 2)) != CRYPT_OK) {
136         XFREE(out);
137         goto bail_2;
138     }
139
140     /* test OID */
141     if ((digestinfo[0].size == hash_descriptor[hash_idx].OIDlen) &&
142         (XMEMCMP(digestinfo[0].data, hash_descriptor[hash_idx].OID, sizeof(unsigned long) * hash_descr
143         (siginfo[1].size == hashlen) &&
144         (XMEMCMP(siginfo[1].data, hash, hashlen) == 0)) {
145         *stat = 1;
146     }
147
148 #ifdef LTC_CLEAN_STACK
149     zeromem(out, outlen);
150 #endif
151     XFREE(out);
152 }
153
154 bail_2:
155 #ifdef LTC_CLEAN_STACK
156     zeromem(tmpbuf, siglen);
157 #endif
158     XFREE(tmpbuf);
159     return err;
160 }

```

Here is the call graph for this function:

## 5.300 prngs/fortuna.c File Reference

### 5.300.1 Detailed Description

Fortuna PRNG, Tom St Denis.

Definition in file [fortuna.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for fortuna.c:

### Functions

- static void [fortuna\\_update\\_iv](#) ([prng\\_state](#) \*prng)
- static int [fortuna\\_reseed](#) ([prng\\_state](#) \*prng)
- int [fortuna\\_start](#) ([prng\\_state](#) \*prng)  
*Start the PRNG.*
- int [fortuna\\_add\\_entropy](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)  
*Add entropy to the PRNG state.*
- int [fortuna\\_ready](#) ([prng\\_state](#) \*prng)  
*Make the PRNG ready to read from.*
- unsigned long [fortuna\\_read](#) (unsigned char \*out, unsigned long outlen, [prng\\_state](#) \*prng)  
*Read from the PRNG.*
- int [fortuna\\_done](#) ([prng\\_state](#) \*prng)  
*Terminate the PRNG.*
- int [fortuna\\_export](#) (unsigned char \*out, unsigned long \*outlen, [prng\\_state](#) \*prng)  
*Export the PRNG state.*
- int [fortuna\\_import](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)  
*Import a PRNG state.*
- int [fortuna\\_test](#) (void)  
*PRNG self-test.*

### Variables

- const struct [ltc\\_prng\\_descriptor](#) [fortuna\\_desc](#)

### 5.300.2 Function Documentation

#### 5.300.2.1 int fortuna\_add\_entropy (const unsigned char \* in, unsigned long inlen, [prng\\_state](#) \* prng)

Add entropy to the PRNG state.

**Parameters:**

*in* The data to add  
*inlen* Length of the data to add  
*prng* PRNG state to update

**Returns:**

CRYPT\_OK if successful

Definition at line 171 of file fortuna.c.

References CRYPT\_INVALID\_ARG, FORTUNA\_POOLS, LTC\_ARGCHK, LTC\_MUTEX\_LOCK, and LTC\_MUTEX\_UNLOCK.

Referenced by fortuna\_import().

```

172 {
173     unsigned char tmp[2];
174     int          err;
175
176     LTC_ARGCHK(in != NULL);
177     LTC_ARGCHK(prng != NULL);
178
179     LTC_MUTEX_LOCK(&prng->fortuna.prng_lock);
180
181     /* ensure inlen <= 32 */
182     if (inlen > 32) {
183         LTC_MUTEX_UNLOCK(&prng->fortuna.prng_lock);
184         return CRYPT_INVALID_ARG;
185     }
186
187     /* add s || length(in) || in to pool[pool_idx] */
188     tmp[0] = 0;
189     tmp[1] = inlen;
190     if ((err = sha256_process(&prng->fortuna.pool[prng->fortuna.pool_idx], tmp, 2)) != CRYPT_OK) {
191         LTC_MUTEX_UNLOCK(&prng->fortuna.prng_lock);
192         return err;
193     }
194     if ((err = sha256_process(&prng->fortuna.pool[prng->fortuna.pool_idx], in, inlen)) != CRYPT_OK) {
195         LTC_MUTEX_UNLOCK(&prng->fortuna.prng_lock);
196         return err;
197     }
198     if (prng->fortuna.pool_idx == 0) {
199         prng->fortuna.pool0_len += inlen;
200     }
201     if (++(prng->fortuna.pool_idx) == FORTUNA_POOLS) {
202         prng->fortuna.pool_idx = 0;
203     }
204
205     LTC_MUTEX_UNLOCK(&prng->fortuna.prng_lock);
206     return CRYPT_OK;
207 }
```

**5.300.2.2 int fortuna\_done (prng\_state \* prng)**

Terminate the PRNG.

**Parameters:**

*prng* The PRNG to terminate

**Returns:**

CRYPT\_OK if successful

Definition at line 284 of file fortuna.c.

References CRYPT\_OK, FORTUNA\_POOLS, LTC\_ARGCHK, LTC\_Mutex\_LOCK, LTC\_Mutex\_UNLOCK, and sha256\_done().

```

285 {
286     int          err, x;
287     unsigned char tmp[32];
288
289     LTC_ARGCHK(prng != NULL);
290     LTC_Mutex_LOCK(&prng->fortuna.prng_lock);
291
292     /* terminate all the hashes */
293     for (x = 0; x < FORTUNA_POOLS; x++) {
294         if ((err = sha256_done(&(prng->fortuna.pool[x]), tmp)) != CRYPT_OK) {
295             LTC_Mutex_UNLOCK(&prng->fortuna.prng_lock);
296             return err;
297         }
298     }
299     /* call cipher done when we invent one ;- ) */
300
301     #ifdef LTC_CLEAN_STACK
302         zeromem(tmp, sizeof(tmp));
303     #endif
304
305     LTC_Mutex_UNLOCK(&prng->fortuna.prng_lock);
306     return CRYPT_OK;
307 }

```

Here is the call graph for this function:

### 5.300.2.3 int fortuna\_export (unsigned char \* out, unsigned long \* outlen, prng\_state \* prng)

Export the PRNG state.

#### Parameters:

- out* [out] Destination
- outlen* [in/out] Max size and resulting size of the state
- prng* The PRNG to export

#### Returns:

CRYPT\_OK if successful

Definition at line 316 of file fortuna.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_MEM, CRYPT\_OK, FORTUNA\_POOLS, LTC\_ARGCHK, LTC\_Mutex\_LOCK, LTC\_Mutex\_UNLOCK, sha256\_done(), XMALLOC, and XMEMCPY.

```

317 {
318     int          x, err;
319     hash_state *md;
320
321     LTC_ARGCHK(out != NULL);
322     LTC_ARGCHK(outlen != NULL);
323     LTC_ARGCHK(prng != NULL);
324
325     LTC_Mutex_LOCK(&prng->fortuna.prng_lock);
326

```

```

327  /* we'll write bytes for s&g's */
328  if (*outlen < 32*FORTUNA_POOLS) {
329      LTC_Mutex_UNLOCK(&prng->fortuna.prng_lock);
330      *outlen = 32*FORTUNA_POOLS;
331      return CRYPT_BUFFER_OVERFLOW;
332  }
333
334  md = XMALLOC(sizeof(hash_state));
335  if (md == NULL) {
336      LTC_Mutex_UNLOCK(&prng->fortuna.prng_lock);
337      return CRYPT_MEM;
338  }
339
340  /* to emit the state we copy each pool, terminate it then hash it again so
341   * an attacker who sees the state can't determine the current state of the PRNG
342   */
343  for (x = 0; x < FORTUNA_POOLS; x++) {
344      /* copy the PRNG */
345      XMEMCPY(md, &(prng->fortuna.pool[x]), sizeof(*md));
346
347      /* terminate it */
348      if ((err = sha256_done(md, out+x*32)) != CRYPT_OK) {
349          goto LBL_ERR;
350      }
351
352      /* now hash it */
353      if ((err = sha256_init(md)) != CRYPT_OK) {
354          goto LBL_ERR;
355      }
356      if ((err = sha256_process(md, out+x*32, 32)) != CRYPT_OK) {
357          goto LBL_ERR;
358      }
359      if ((err = sha256_done(md, out+x*32)) != CRYPT_OK) {
360          goto LBL_ERR;
361      }
362  }
363  *outlen = 32*FORTUNA_POOLS;
364  err = CRYPT_OK;
365
366 LBL_ERR:
367 #ifdef LTC_CLEAN_STACK
368     zeromem(md, sizeof(*md));
369 #endif
370     XFREE(md);
371     LTC_Mutex_UNLOCK(&prng->fortuna.prng_lock);
372     return err;
373 }

```

Here is the call graph for this function:

#### 5.300.2.4 int fortuna\_import(const unsigned char \*in, unsigned long inlen, prng\_state \*prng)

Import a PRNG state.

##### Parameters:

- in* The PRNG state
- inlen* Size of the state
- prng* The PRNG to import

##### Returns:

CRYPT\_OK if successful

Definition at line 382 of file fortuna.c.

References `CRYPT_INVALID_ARG`, `CRYPT_OK`, `fortuna_add_entropy()`, `FORTUNA_POOLS`, `fortuna_start()`, and `LTC_ARGCHK`.

```

383 {
384     int err, x;
385
386     LTC_ARGCHK(in != NULL);
387     LTC_ARGCHK(prng != NULL);
388
389     if (inlen != 32*FORTUNA_POOLS) {
390         return CRYPT_INVALID_ARG;
391     }
392
393     if ((err = fortuna_start(prng)) != CRYPT_OK) {
394         return err;
395     }
396     for (x = 0; x < FORTUNA_POOLS; x++) {
397         if ((err = fortuna_add_entropy(in+x*32, 32, prng)) != CRYPT_OK) {
398             return err;
399         }
400     }
401     return err;
402 }
```

Here is the call graph for this function:

#### 5.300.2.5 unsigned long fortuna\_read (unsigned char \* out, unsigned long outlen, prng\_state \* prng)

Read from the PRNG.

##### Parameters:

- out* Destination
- outlen* Length of output
- prng* The active PRNG to read from

##### Returns:

- Number of octets read

Definition at line 226 of file fortuna.c.

References `CRYPT_OK`, `fortuna_reseed()`, `fortuna_update_iv()`, `FORTUNA_WD`, `LTC_ARGCHK`, `LTC_MUTEX_LOCK`, `LTC_MUTEX_UNLOCK`, `XMEMCPY`, and `zeromem()`.

```

227 {
228     unsigned char tmp[16];
229     int err;
230     unsigned long tlen;
231
232     LTC_ARGCHK(out != NULL);
233     LTC_ARGCHK(prng != NULL);
234
235     LTC_MUTEX_LOCK(&prng->fortuna.prng_lock);
236
237     /* do we have to reseed? */
238     if (++prng->fortuna.wd == FORTUNA_WD || prng->fortuna.pool0_len >= 64) {
239         if ((err = fortuna_reseed(prng)) != CRYPT_OK) {
```

```

240         LTC_MUTEX_UNLOCK(&prng->fortuna.prng_lock);
241         return 0;
242     }
243 }
244
245 /* now generate the blocks required */
246 tlen = outlen;
247
248 /* handle whole blocks without the extra XMEMCPY */
249 while (outlen >= 16) {
250     /* encrypt the IV and store it */
251     rijndael_ecb_encrypt(prng->fortuna.IV, out, &prng->fortuna.skey);
252     out += 16;
253     outlen -= 16;
254     fortuna_update_iv(prng);
255 }
256
257 /* left over bytes? */
258 if (outlen > 0) {
259     rijndael_ecb_encrypt(prng->fortuna.IV, tmp, &prng->fortuna.skey);
260     XMEMCPY(out, tmp, outlen);
261     fortuna_update_iv(prng);
262 }
263
264 /* generate new key */
265 rijndael_ecb_encrypt(prng->fortuna.IV, prng->fortuna.K, &prng->fortuna.skey); fortuna_update_iv
266 rijndael_ecb_encrypt(prng->fortuna.IV, prng->fortuna.K+16, &prng->fortuna.skey); fortuna_update_iv
267 if ((err = rijndael_setup(prng->fortuna.K, 32, 0, &prng->fortuna.skey)) != CRYPT_OK) {
268     LTC_MUTEX_UNLOCK(&prng->fortuna.prng_lock);
269     return 0;
270 }
271
272 #ifdef LTC_CLEAN_STACK
273     zeromem(tmp, sizeof(tmp));
274 #endif
275     LTC_MUTEX_UNLOCK(&prng->fortuna.prng_lock);
276     return tlen;
277 }

```

Here is the call graph for this function:

### 5.300.2.6 int fortuna\_ready ([prng\\_state](#) \* *prng*)

Make the PRNG ready to read from.

#### Parameters:

*prng* The PRNG to make active

#### Returns:

CRYPT\_OK if successful

Definition at line 214 of file fortuna.c.

References [fortuna\\_reseed\(\)](#).

```

215 {
216     return fortuna_reseed(prng);
217 }

```

Here is the call graph for this function:

**5.300.2.7 static int fortuna\_reseed (prng\_state \*prng) [static]**

Definition at line 66 of file fortuna.c.

References CRYPT\_OK, FORTUNA\_POOLS, MAXBLOCKSIZE, sha256\_done(), and sha256\_init().

Referenced by fortuna\_read(), and fortuna\_ready().

```

67 {
68     unsigned char tmp[MAXBLOCKSIZE];
69     hash_state    md;
70     int           err, x;
71
72     ++prng->fortuna.reset_cnt;
73
74     /* new K == SHA256(K || s) where s == SHA256(P0) || SHA256(P1) ... */
75     sha256_init(&md);
76     if ((err = sha256_process(&md, prng->fortuna.K, 32)) != CRYPT_OK) {
77         sha256_done(&md, tmp);
78         return err;
79     }
80
81     for (x = 0; x < FORTUNA_POOLS; x++) {
82         if (x == 0 || ((prng->fortuna.reset_cnt >> (x-1)) & 1) == 0) {
83             /* terminate this hash */
84             if ((err = sha256_done(&prng->fortuna.pool[x], tmp)) != CRYPT_OK) {
85                 sha256_done(&md, tmp);
86                 return err;
87             }
88             /* add it to the string */
89             if ((err = sha256_process(&md, tmp, 32)) != CRYPT_OK) {
90                 sha256_done(&md, tmp);
91                 return err;
92             }
93             /* reset this pool */
94             if ((err = sha256_init(&prng->fortuna.pool[x])) != CRYPT_OK) {
95                 sha256_done(&md, tmp);
96                 return err;
97             }
98         } else {
99             break;
100         }
101     }
102
103     /* finish key */
104     if ((err = sha256_done(&md, prng->fortuna.K)) != CRYPT_OK) {
105         return err;
106     }
107     if ((err = rijndael_setup(prng->fortuna.K, 32, 0, &prng->fortuna.skey)) != CRYPT_OK) {
108         return err;
109     }
110     fortuna_update_iv(prng);
111
112     /* reset pool len */
113     prng->fortuna.pool0_len = 0;
114     prng->fortuna.wd        = 0;
115
116
117 #ifdef LTC_CLEAN_STACK
118     zeromem(&md, sizeof(md));
119     zeromem(tmp, sizeof(tmp));
120 #endif
121
122     return CRYPT_OK;
123 }

```

Here is the call graph for this function:



**5.300.2.8 int fortuna\_start (prng\_state \* prng)**

Start the PRNG.

**Parameters:**

*prng* [out] The PRNG state to initialize

**Returns:**

CRYPT\_OK if successful

Definition at line 130 of file fortuna.c.

References CRYPT\_OK, FORTUNA\_POOLS, LTC\_ARGCHK, MAXBLOCKSIZE, sha256\_done(), and sha256\_init().

Referenced by fortuna\_import().

```

131 {
132     int err, x, y;
133     unsigned char tmp[MAXBLOCKSIZE];
134
135     LTC_ARGCHK(prng != NULL);
136
137     /* initialize the pools */
138     for (x = 0; x < FORTUNA_POOLS; x++) {
139         if ((err = sha256_init(&prng->fortuna.pool[x])) != CRYPT_OK) {
140             for (y = 0; y < x; y++) {
141                 sha256_done(&prng->fortuna.pool[y], tmp);
142             }
143             return err;
144         }
145     }
146     prng->fortuna.pool_idx = prng->fortuna.pool0_len = prng->fortuna.reset_cnt =
147     prng->fortuna.wd = 0;
148
149     /* reset bufs */
150     zeromem(prng->fortuna.K, 32);
151     if ((err = rijndael_setup(prng->fortuna.K, 32, 0, &prng->fortuna.skey)) != CRYPT_OK) {
152         for (x = 0; x < FORTUNA_POOLS; x++) {
153             sha256_done(&prng->fortuna.pool[x], tmp);
154         }
155         return err;
156     }
157     zeromem(prng->fortuna.IV, 16);
158
159     LTC_MUTEX_INIT(&prng->fortuna.prng_lock)
160
161     return CRYPT_OK;
162 }
```

Here is the call graph for this function:

**5.300.2.9 int fortuna\_test (void)**

PRNG self-test.

**Returns:**

CRYPT\_OK if successful, CRYPT\_NOP if self-testing has been disabled

Definition at line 408 of file fortuna.c.

References CRYPT\_NOP, CRYPT\_OK, and sha256\_test().

```
409 {
410 #ifndef LTC_TEST
411     return CRYPT_NOP;
412 #else
413     int err;
414
415     if ((err = sha256_test()) != CRYPT_OK) {
416         return err;
417     }
418     return rijndael_test();
419 #endif
420 }
```

Here is the call graph for this function:

#### 5.300.2.10 static void fortuna\_update\_iv ([prng\\_state](#) \* *prng*) [static]

Definition at line 53 of file fortuna.c.

Referenced by fortuna\_read().

```
54 {
55     int x;
56     unsigned char *IV;
57     /* update IV */
58     IV = prng->fortuna.IV;
59     for (x = 0; x < 16; x++) {
60         IV[x] = (IV[x] + 1) & 255;
61         if (IV[x] != 0) break;
62     }
63 }
```

### 5.300.3 Variable Documentation

#### 5.300.3.1 const struct [ltc\\_prng\\_descriptor](#) fortuna\_desc

**Initial value:**

```
{
    "fortuna", 1024,
    &fortuna_start,
    &fortuna_add_entropy,
    &fortuna_ready,
    &fortuna_read,
    &fortuna_done,
    &fortuna_export,
    &fortuna_import,
    &fortuna_test
}
```

Definition at line 40 of file fortuna.c.

## 5.301 prngs/rc4.c File Reference

### 5.301.1 Detailed Description

RC4 PRNG, Tom St Denis.

Definition in file [rc4.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for rc4.c:

### Functions

- [int rc4\\_start](#) ([prng\\_state](#) \*prng)  
*Start the PRNG.*
- [int rc4\\_add\\_entropy](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)  
*Add entropy to the PRNG state.*
- [int rc4\\_ready](#) ([prng\\_state](#) \*prng)  
*Make the PRNG ready to read from.*
- [unsigned long rc4\\_read](#) (unsigned char \*out, unsigned long outlen, [prng\\_state](#) \*prng)  
*Read from the PRNG.*
- [int rc4\\_done](#) ([prng\\_state](#) \*prng)  
*Terminate the PRNG.*
- [int rc4\\_export](#) (unsigned char \*out, unsigned long \*outlen, [prng\\_state](#) \*prng)  
*Export the PRNG state.*
- [int rc4\\_import](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)  
*Import a PRNG state.*
- [int rc4\\_test](#) (void)  
*PRNG self-test.*

### Variables

- const struct [ltc\\_prng\\_descriptor](#) [rc4\\_desc](#)

### 5.301.2 Function Documentation

#### 5.301.2.1 [int rc4\\_add\\_entropy](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)

Add entropy to the PRNG state.

#### Parameters:

*in* The data to add

*inlen* Length of the data to add

*prng* PRNG state to update

**Returns:**

CRYPT\_OK if successful

Definition at line 55 of file rc4.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by rc4\_import().

```
56 {
57     LTC_ARGCHK(in != NULL);
58     LTC_ARGCHK(prng != NULL);
59
60     /* trim as required */
61     if (prng->rc4.x + inlen > 256) {
62         if (prng->rc4.x == 256) {
63             /* I can't possibly accept another byte, ok maybe a mint wafer... */
64             return CRYPT_OK;
65         } else {
66             /* only accept part of it */
67             inlen = 256 - prng->rc4.x;
68         }
69     }
70
71     while (inlen-- > 0) {
72         prng->rc4.buf[prng->rc4.x++] = *in++;
73     }
74
75     return CRYPT_OK;
76 }
77 }
```

### 5.301.2.2 int rc4\_done (*prng\_state* \* *prng*)

Terminate the PRNG.

**Parameters:**

*prng* The PRNG to terminate

**Returns:**

CRYPT\_OK if successful

Definition at line 158 of file rc4.c.

References CRYPT\_OK, and LTC\_ARGCHK.

```
159 {
160     LTC_ARGCHK(prng != NULL);
161     return CRYPT_OK;
162 }
```

### 5.301.2.3 int rc4\_export (unsigned char \* out, unsigned long \* outlen, prng\_state \* prng)

Export the PRNG state.

**Parameters:**

*out* [out] Destination  
*outlen* [in/out] Max size and resulting size of the state  
*prng* The PRNG to export

**Returns:**

CRYPT\_OK if successful

Definition at line 171 of file rc4.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_ERROR\_READPRNG, CRYPT\_OK, LTC\_ARGCHK, and rc4\_read().

```
172 {  
173     LTC_ARGCHK(outlen != NULL);  
174     LTC_ARGCHK(out != NULL);  
175     LTC_ARGCHK(prng != NULL);  
176  
177     if (*outlen < 32) {  
178         *outlen = 32;  
179         return CRYPT_BUFFER_OVERFLOW;  
180     }  
181  
182     if (rc4_read(out, 32, prng) != 32) {  
183         return CRYPT_ERROR_READPRNG;  
184     }  
185     *outlen = 32;  
186  
187     return CRYPT_OK;  
188 }
```

Here is the call graph for this function:

### 5.301.2.4 int rc4\_import (const unsigned char \* in, unsigned long inlen, prng\_state \* prng)

Import a PRNG state.

**Parameters:**

*in* The PRNG state  
*inlen* Size of the state  
*prng* The PRNG to import

**Returns:**

CRYPT\_OK if successful

Definition at line 197 of file rc4.c.

References CRYPT\_INVALID\_ARG, CRYPT\_OK, LTC\_ARGCHK, rc4\_add\_entropy(), and rc4\_start().

```
198 {  
199     int err;
```

```

200 LTC_ARGCHK(in != NULL);
201 LTC_ARGCHK(prng != NULL);
202
203 if (inlen != 32) {
204     return CRYPT_INVALID_ARG;
205 }
206
207 if ((err = rc4_start(prng)) != CRYPT_OK) {
208     return err;
209 }
210 return rc4_add_entropy(in, 32, prng);
211 }

```

Here is the call graph for this function:

#### 5.301.2.5 unsigned long rc4\_read (unsigned char \* out, unsigned long outlen, [prng\\_state](#) \* prng)

Read from the PRNG.

##### Parameters:

*out* Destination  
*outlen* Length of output  
*prng* The active PRNG to read from

##### Returns:

Number of octets read

Definition at line 125 of file rc4.c.

References LTC\_ARGCHK, and zeromem().

Referenced by rc4\_export().

```

126 {
127     unsigned char x, y, *s, tmp;
128     unsigned long n;
129
130     LTC_ARGCHK(out != NULL);
131     LTC_ARGCHK(prng != NULL);
132
133 #ifdef LTC_VALGRIND
134     zeromem(out, outlen);
135 #endif
136
137     n = outlen;
138     x = prng->rc4.x;
139     y = prng->rc4.y;
140     s = prng->rc4.buf;
141     while (outlen--) {
142         x = (x + 1) & 255;
143         y = (y + s[x]) & 255;
144         tmp = s[x]; s[x] = s[y]; s[y] = tmp;
145         tmp = (s[x] + s[y]) & 255;
146         *out++ ^= s[tmp];
147     }
148     prng->rc4.x = x;
149     prng->rc4.y = y;
150     return n;
151 }

```

Here is the call graph for this function:

**5.301.2.6 int rc4\_ready ([prng\\_state](#) \* *prng*)**

Make the PRNG ready to read from.

**Parameters:**

*prng* The PRNG to make active

**Returns:**

CRYPT\_OK if successful

Definition at line 84 of file rc4.c.

References LTC\_ARGCHK, and XMEMCPY.

```

85 {
86     unsigned char key[256], tmp, *s;
87     int keylen, x, y, j;
88
89     LTC_ARGCHK(prng != NULL);
90
91     /* extract the key */
92     s = prng->rc4.buf;
93     XMEMCPY(key, s, 256);
94     keylen = prng->rc4.x;
95
96     /* make RC4 perm and shuffle */
97     for (x = 0; x < 256; x++) {
98         s[x] = x;
99     }
100
101     for (j = x = y = 0; x < 256; x++) {
102         y = (y + prng->rc4.buf[x] + key[j++]) & 255;
103         if (j == keylen) {
104             j = 0;
105         }
106         tmp = s[x]; s[x] = s[y]; s[y] = tmp;
107     }
108     prng->rc4.x = 0;
109     prng->rc4.y = 0;
110
111     #ifdef LTC_CLEAN_STACK
112         zeromem(key, sizeof(key));
113     #endif
114
115     return CRYPT_OK;
116 }
```

**5.301.2.7 int rc4\_start ([prng\\_state](#) \* *prng*)**

Start the PRNG.

**Parameters:**

*prng* [out] The PRNG state to initialize

**Returns:**

CRYPT\_OK if successful

Definition at line 38 of file rc4.c.

References CRYPT\_OK, and LTC\_ARGCHK.

Referenced by rc4\_import(), and rc4\_test().

```

39 {
40     LTC_ARGCHK(prng != NULL);
41
42     /* set keysize to zero */
43     prng->rc4.x = 0;
44
45     return CRYPT_OK;
46 }

```

### 5.301.2.8 int rc4\_test (void)

PRNG self-test.

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-testing has been disabled

Definition at line 217 of file rc4.c.

References CRYPT\_NOP, CRYPT\_OK, and rc4\_start().

```

218 {
219 #if !defined(LTC_TEST) || defined(LTC_VALGRIND)
220     return CRYPT_NOP;
221 #else
222     static const struct {
223         unsigned char key[8], pt[8], ct[8];
224     } tests[] = {
225     {
226         { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef },
227         { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef },
228         { 0x75, 0xb7, 0x87, 0x80, 0x99, 0xe0, 0xc5, 0x96 }
229     }
230 };
231     prng_state prng;
232     unsigned char dst[8];
233     int err, x;
234
235     for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
236         if ((err = rc4_start(&prng)) != CRYPT_OK) {
237             return err;
238         }
239         if ((err = rc4_add_entropy(tests[x].key, 8, &prng)) != CRYPT_OK) {
240             return err;
241         }
242         if ((err = rc4_ready(&prng)) != CRYPT_OK) {
243             return err;
244         }
245         XMEMCPY(dst, tests[x].pt, 8);
246         if (rc4_read(dst, 8, &prng) != 8) {
247             return CRYPT_ERROR_READPRNG;
248         }
249         rc4_done(&prng);
250         if (XMEMCMP(dst, tests[x].ct, 8)) {
251 #if 0
252             int y;
253             printf("\n\nRC4 failed, I got:\n");
254             for (y = 0; y < 8; y++) printf("%02x ", dst[y]);
255             printf("\n");
256 #endif
257             return CRYPT_FAIL_TESTVECTOR;
258         }
259     }
260     return CRYPT_OK;

```



```
261 #endif
262 }
```

Here is the call graph for this function:

### 5.301.3 Variable Documentation

#### 5.301.3.1 const struct [ltc\\_prng\\_descriptor](#) rc4\_desc

**Initial value:**

```
{
    "rc4", 32,
    &rc4_start,
    &rc4_add_entropy,
    &rc4_ready,
    &rc4_read,
    &rc4_done,
    &rc4_export,
    &rc4_import,
    &rc4_test
}
```

Definition at line 20 of file rc4.c.

## 5.302 prngs/rng\_get\_bytes.c File Reference

### 5.302.1 Detailed Description

portable way to get secure random bits to feed a PRNG (Tom St Denis)

Definition in file [rng\\_get\\_bytes.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for [rng\\_get\\_bytes.c](#):

### Functions

- static unsigned long [rng\\_nix](#) (unsigned char \*buf, unsigned long len, void(\*callback)(void))
- unsigned long [rng\\_get\\_bytes](#) (unsigned char \*out, unsigned long outlen, void(\*callback)(void))

*Read the system RNG.*

### 5.302.2 Function Documentation

#### 5.302.2.1 unsigned long rng\_get\_bytes (unsigned char \* out, unsigned long outlen, void(\*) (void) callback)

Read the system RNG.

#### Parameters:

*out* Destination

*outlen* Length desired (octets)

*callback* Pointer to void function to act as "callback" when RNG is slow. This can be NULL

#### Returns:

Number of octets read

Definition at line 123 of file [rng\\_get\\_bytes.c](#).

References [LTC\\_ARGCHK](#), and [rng\\_nix\(\)](#).

Referenced by [rng\\_make\\_prng\(\)](#), and [sprng\\_read\(\)](#).

```
125 {
126     unsigned long x;
127
128     LTC_ARGCHK(out != NULL);
129
130     #if defined(DEVRANDOM)
131     x = rng_nix(out, outlen, callback); if (x != 0) { return x; }
132     #endif
133     #ifdef WIN32
134     x = rng_win32(out, outlen, callback); if (x != 0) { return x; }
135     #endif
136     #ifdef ANSI_RNG
137     x = rng_ansi(out, outlen, callback); if (x != 0) { return x; }
138     #endif
139     return 0;
140 }
```

Here is the call graph for this function:

**5.302.2.2 static unsigned long rng\_nix (unsigned char \* *buf*, unsigned long *len*, void(\*) (void) *callback*)** [static]

Definition at line 20 of file rng\_get\_bytes.c.

Referenced by rng\_get\_bytes().

```
22 {
23 #ifdef LTC_NO_FILE
24     return 0;
25 #else
26     FILE *f;
27     unsigned long x;
28 #ifdef TRY_URANDOM_FIRST
29     f = fopen("/dev/urandom", "rb");
30     if (f == NULL)
31 #endif /* TRY_URANDOM_FIRST */
32     f = fopen("/dev/random", "rb");
33
34     if (f == NULL) {
35         return 0;
36     }
37
38     /* disable buffering */
39     if (setvbuf(f, NULL, _IONBF, 0) != 0) {
40         fclose(f);
41         return 0;
42     }
43
44     x = (unsigned long)fread(buf, 1, (size_t)len, f);
45     fclose(f);
46     return x;
47 #endif /* LTC_NO_FILE */
48 }
```

## 5.303 prngs/rng\_make\_prng.c File Reference

### 5.303.1 Detailed Description

portable way to get secure random bits to feed a PRNG (Tom St Denis)

Definition in file [rng\\_make\\_prng.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for rng\_make\_prng.c:

### Functions

- int [rng\\_make\\_prng](#) (int bits, int wprng, [prng\\_state](#) \*prng, void(\*callback)(void))  
*Create a PRNG from a RNG.*

### 5.303.2 Function Documentation

#### 5.303.2.1 int rng\_make\_prng (int bits, int wprng, [prng\\_state](#) \*prng, void(\*) (void) callback)

Create a PRNG from a RNG.

#### Parameters:

- bits* Number of bits of entropy desired (64 ... 1024)
- wprng* Index of which PRNG to setup
- prng* [out] PRNG state to initialize
- callback* A pointer to a void function for when the RNG is slow, this can be NULL

#### Returns:

- CRYPT\_OK if successful

Definition at line 26 of file rng\_make\_prng.c.

References [CRYPT\\_ERROR\\_READPRNG](#), [CRYPT\\_INVALID\\_PRNGSIZE](#), [CRYPT\\_OK](#), [LTC\\_ARGCHK](#), [prng\\_descriptor](#), [prng\\_is\\_valid\(\)](#), [rng\\_get\\_bytes\(\)](#), and [zeromem\(\)](#).

```
28 {
29     unsigned char buf[256];
30     int err;
31
32     LTC_ARGCHK(prng != NULL);
33
34     /* check parameter */
35     if ((err = prng_is_valid(wprng)) != CRYPT_OK) {
36         return err;
37     }
38
39     if (bits < 64 || bits > 1024) {
40         return CRYPT_INVALID_PRNGSIZE;
41     }
42
43     if ((err = prng_descriptor[wprng].start(prng)) != CRYPT_OK) {
44         return err;
45     }
```

```
46
47     bits = ((bits/8)+((bits&7)!=0?1:0)) * 2;
48     if (rng_get_bytes(buf, (unsigned long)bits, callback) != (unsigned long)bits) {
49         return CRYPT_ERROR_READPRNG;
50     }
51
52     if ((err = prng_descriptor[wprng].add_entropy(buf, (unsigned long)bits, prng)) != CRYPT_OK) {
53         return err;
54     }
55
56     if ((err = prng_descriptor[wprng].ready(prng)) != CRYPT_OK) {
57         return err;
58     }
59
60     #ifdef LTC_CLEAN_STACK
61         zeromem(buf, sizeof(buf));
62     #endif
63     return CRYPT_OK;
64 }
```

Here is the call graph for this function:

## 5.304 prngs/sober128.c File Reference

### 5.304.1 Detailed Description

Implementation of SOBER-128 by Tom St Denis.

Based on [s128fast.c](#) reference code supplied by Greg Rose of QUALCOMM.

Definition in file [sober128.c](#).

```
#include "tomcrypt.h"
#include "sober128tab.c"
```

Include dependency graph for sober128.c:

### Defines

- #define [N](#) 17
- #define [FOLD](#) N
- #define [INITKONST](#) 0x6996c53a
- #define [KEYP](#) 15
- #define [FOLDP](#) 4
- #define [B](#)(x, i) ((unsigned char)((((x) >> (8\*i)) & 0xFF))
- #define [WORD2BYTE](#)(w, b) STORE32L(b, w)
- #define [OFF](#)(zero, i) (((zero)+(i)) % N)
- #define [STEP](#)(R, z) R[[OFF](#)(z,0)] = R[[OFF](#)(z,15)] ^ R[[OFF](#)(z,4)] ^ (R[[OFF](#)(z,0)] << 8) ^ [Multab](#)[(R[[OFF](#)(z,0)] >> 24) & 0xFF];
- #define [NLFUNC](#)(c, z)
- #define [ADDKEY](#)(k) c → R[[KEYP](#)] += (k);
- #define [XORN](#)(nl) c → R[[FOLDP](#)] ^= (nl);
- #define [DROUND](#)(z) [STEP](#)(c → R,z); [NLFUNC](#)(c,(z+1)); c → R[[OFF](#)((z+1),[FOLDP](#))] ^= t;
- #define [SROUND](#)(z) [STEP](#)(c → R,z); [NLFUNC](#)(c,(z+1)); XORWORD(t, out+(z\*4));

### Functions

- static [ulong32](#) [BYTE2WORD](#) (unsigned char \*b)
- static void [XORWORD](#) ([ulong32](#) w, unsigned char \*b)
- static void [cycle](#) ([ulong32](#) \*R)
- static [ulong32](#) [nltap](#) (struct sober128\_prng \*c)
- int [sober128\\_start](#) (prng\_state \*prng)  
*Start the PRNG.*
- static void [s128\\_savestate](#) (struct sober128\_prng \*c)
- static void [s128\\_reloadstate](#) (struct sober128\_prng \*c)
- static void [s128\\_genkonst](#) (struct sober128\_prng \*c)
- static void [s128\\_diffuse](#) (struct sober128\_prng \*c)
- int [sober128\\_add\\_entropy](#) (const unsigned char \*in, unsigned long inlen, prng\_state \*prng)  
*Add entropy to the PRNG state.*
- int [sober128\\_ready](#) (prng\_state \*prng)  
*Make the PRNG ready to read from.*

- unsigned long [sober128\\_read](#) (unsigned char \*out, unsigned long outlen, [prng\\_state](#) \*prng)  
*Read from the PRNG.*
- int [sober128\\_done](#) ([prng\\_state](#) \*prng)  
*Terminate the PRNG.*
- int [sober128\\_export](#) (unsigned char \*out, unsigned long \*outlen, [prng\\_state](#) \*prng)  
*Export the PRNG state.*
- int [sober128\\_import](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)  
*Import a PRNG state.*
- int [sober128\\_test](#) (void)  
*PRNG self-test.*

## Variables

- const struct [ltc\\_prng\\_descriptor](#) [sober128\\_desc](#)

### 5.304.2 Define Documentation

#### 5.304.2.1 **#define ADDKEY(k) [c](#) $\rightarrow$ [R](#)[KEYP] += (k);**

Definition at line 169 of file sober128.c.

Referenced by [sober128\\_add\\_entropy\(\)](#).

#### 5.304.2.2 **#define B(x, i) (((unsigned char)(((x) >> (8\*i)) & 0xFF))**

Definition at line 43 of file sober128.c.

Referenced by [blowfish\\_setup\(\)](#), [gcm\\_init\(\)](#), [gf\\_mult\(\)](#), [lrw\\_start\(\)](#), [qsort\\_helper\(\)](#), [rc5\\_ecb\\_decrypt\(\)](#), [rc5\\_ecb\\_encrypt\(\)](#), [rc5\\_setup\(\)](#), [rc6\\_setup\(\)](#), and [twofish\\_setup\(\)](#).

#### 5.304.2.3 **#define DROUND(z) STEP([c](#) $\rightarrow$ [R](#),z); NLFUNC([c](#),z+1); [c](#) $\rightarrow$ [R](#)[OFF((z+1),FOLDP)] ^= t;**

Definition at line 176 of file sober128.c.

Referenced by [s128\\_diffuse\(\)](#).

#### 5.304.2.4 **#define FOLD N**

Definition at line 38 of file sober128.c.

#### 5.304.2.5 **#define FOLDP 4**

Definition at line 41 of file sober128.c.

**5.304.2.6 #define INITKONST 0x6996c53a**

Definition at line 39 of file sober128.c.

**5.304.2.7 #define KEYP 15**

Definition at line 40 of file sober128.c.

**5.304.2.8 #define N 17**

Definition at line 37 of file sober128.c.

Referenced by `anubis_setup()`, `cycle()`, `s128_reloadstate()`, `s128_savestate()`, `sober128_read()`, and `sober128_start()`.

**5.304.2.9 #define NLFUNC(*c*, *z*)**

**Value:**

```
{ \
    t = c->R[OFF(z,0)] + c->R[OFF(z,16)]; \
    t ^= Sbox[(t >> 24) & 0xFF]; \
    t = RORc(t, 8); \
    t = ((t + c->R[OFF(z,1)]) ^ c->konst) + c->R[OFF(z,6)]; \
    t ^= Sbox[(t >> 24) & 0xFF]; \
    t = t + c->R[OFF(z,13)]; \
}
```

Definition at line 87 of file sober128.c.

Referenced by `nltp()`.

**5.304.2.10 #define OFF(zero, i) (((zero)+(i)) % N)**

Definition at line 65 of file sober128.c.

**5.304.2.11 #define SROUND(z) STEP(*c* → *R*,*z*); NLFUNC(*c*,(*z*+1)); XORWORD(*t*, out+(*z*\*4));**

Definition at line 280 of file sober128.c.

Referenced by `sober128_read()`.

**5.304.2.12 #define STEP(*R*, *z*) *R*[OFF(*z*,0)] = *R*[OFF(*z*,15)] ^ *R*[OFF(*z*,4)] ^ (*R*[OFF(*z*,0)] << 8) ^ Multab[*R*[OFF(*z*,0)] >> 24 & 0xFF];**

Definition at line 69 of file sober128.c.

Referenced by `cycle()`.

**5.304.2.13 #define WORD2BYTE(*w*, *b*) STORE32L(*b*, *w*)**

Definition at line 52 of file sober128.c.



**5.304.2.14** `#define XORNL(nl) c → R[FOLDP] ^= (nl);`

Definition at line 172 of file sober128.c.

Referenced by sober128\_add\_entropy().

**5.304.3 Function Documentation****5.304.3.1** `static ulong32 BYTE2WORD (unsigned char * b) [static]`

Definition at line 45 of file sober128.c.

Referenced by sober128\_add\_entropy().

```

46 {
47     ulong32 t;
48     LOAD32L(t, b);
49     return t;
50 }
```

**5.304.3.2** `static void cycle (ulong32 * R) [static]`

Definition at line 72 of file sober128.c.

References N, and STEP.

Referenced by s128\_genkonst(), sober128\_add\_entropy(), and sober128\_read().

```

73 {
74     ulong32 t;
75     int i;
76
77     STEP(R, 0);
78     t = R[0];
79     for (i = 1; i < N; ++i) {
80         R[i-1] = R[i];
81     }
82     R[N-1] = t;
83 }
```

**5.304.3.3** `static ulong32 nltap (struct sober128_prng * c) [static]`

Definition at line 97 of file sober128.c.

References NLFUNC.

Referenced by s128\_genkonst(), sober128\_add\_entropy(), and sober128\_read().

```

98 {
99     ulong32 t;
100     NLFUNC(c, 0);
101     return t;
102 }
```

**5.304.3.4 static void s128\_diffuse (struct sober128\_prng \* c) [static]**

Definition at line 177 of file sober128.c.

References DROUND.

Referenced by sober128\_add\_entropy().

```
178 {
179     ulong32 t;
180     /* relies on FOLD == N == 17! */
181     DROUND(0);
182     DROUND(1);
183     DROUND(2);
184     DROUND(3);
185     DROUND(4);
186     DROUND(5);
187     DROUND(6);
188     DROUND(7);
189     DROUND(8);
190     DROUND(9);
191     DROUND(10);
192     DROUND(11);
193     DROUND(12);
194     DROUND(13);
195     DROUND(14);
196     DROUND(15);
197     DROUND(16);
198 }
```

**5.304.3.5 static void s128\_genkonst (struct sober128\_prng \* c) [static]**

Definition at line 156 of file sober128.c.

References cycle(), and nltap().

Referenced by sober128\_add\_entropy().

```
157 {
158     ulong32 newkonst;
159
160     do {
161         cycle(c->R);
162         newkonst = nltap(c);
163     } while ((newkonst & 0xFF000000) == 0);
164     c->konst = newkonst;
165 }
```

Here is the call graph for this function:

**5.304.3.6 static void s128\_reloadstate (struct sober128\_prng \* c) [static]**

Definition at line 145 of file sober128.c.

References N.

```
146 {
147     int i;
148
149     for (i = 0; i < N; ++i) {
```

```

150         c->R[i] = c->initR[i];
151     }
152 }

```

### 5.304.3.7 static void s128\_savestate (struct sober128\_prng \* c) [static]

Definition at line 135 of file sober128.c.

References N.

Referenced by sober128\_add\_entropy().

```

136 {
137     int i;
138     for (i = 0; i < N; ++i) {
139         c->initR[i] = c->R[i];
140     }
141 }

```

### 5.304.3.8 int sober128\_add\_entropy (const unsigned char \* in, unsigned long inlen, prng\_state \* prng)

Add entropy to the PRNG state.

#### Parameters:

- in* The data to add
- inlen* Length of the data to add
- prng* PRNG state to update

#### Returns:

- CRYPT\_OK if successful

Definition at line 207 of file sober128.c.

References ADDKEY, BYTE2WORD(), c, CRYPT\_INVALID\_KEYSIZE, cycle(), LTC\_ARGCHK, nl-tap(), s128\_diffuse(), s128\_genkonst(), s128\_savestate(), and XORN.L.

Referenced by sober128\_import().

```

208 {
209     struct sober128_prng *c;
210     ulong32 i, k;
211
212     LTC_ARGCHK(in != NULL);
213     LTC_ARGCHK(prng != NULL);
214     c = &(prng->sober128);
215
216     if (c->flag == 1) {
217         /* this is the first call to the add_entropy so this input is the key */
218         /* inlen must be multiple of 4 bytes */
219         if ((inlen & 3) != 0) {
220             return CRYPT_INVALID_KEYSIZE;
221         }
222
223         for (i = 0; i < inlen; i += 4) {
224             k = BYTE2WORD((unsigned char *)&in[i]);
225             ADDKEY(k);

```

```

226         cycle(c->R);
227         XORNL(nltap(c));
228     }
229
230     /* also fold in the length of the key */
231     ADDKEY(inlen);
232
233     /* now diffuse */
234     sl28_diffuse(c);
235
236     sl28_genkonst(c);
237     sl28_savestate(c);
238     c->nbuf = 0;
239     c->flag = 0;
240     c->set = 1;
241 } else {
242     /* ok we are adding an IV then... */
243     sl28_reloadstate(c);
244
245     /* inlen must be multiple of 4 bytes */
246     if ((inlen & 3) != 0) {
247         return CRYPT_INVALID_KEYSIZE;
248     }
249
250     for (i = 0; i < inlen; i += 4) {
251         k = BYTE2WORD((unsigned char *)&in[i]);
252         ADDKEY(k);
253         cycle(c->R);
254         XORNL(nltap(c));
255     }
256
257     /* also fold in the length of the key */
258     ADDKEY(inlen);
259
260     /* now diffuse */
261     sl28_diffuse(c);
262     c->nbuf = 0;
263 }
264
265 return CRYPT_OK;
266 }

```

Here is the call graph for this function:

### 5.304.3.9 int sober128\_done (prng\_state \* prng)

Terminate the PRNG.

#### Parameters:

*prng* The PRNG to terminate

#### Returns:

CRYPT\_OK if successful

Definition at line 368 of file sober128.c.

References CRYPT\_OK, and LTC\_ARGCHK.

```

369 {
370     LTC_ARGCHK(prng != NULL);
371     return CRYPT_OK;
372 }

```

**5.304.3.10 int sober128\_export (unsigned char \* *out*, unsigned long \* *outlen*, prng\_state \* *prng*)**

Export the PRNG state.

**Parameters:**

- out* [out] Destination
- outlen* [in/out] Max size and resulting size of the state
- prng* The PRNG to export

**Returns:**

CRYPT\_OK if successful

Definition at line 381 of file sober128.c.

References CRYPT\_BUFFER\_OVERFLOW, CRYPT\_ERROR\_READPRNG, CRYPT\_OK, LTC\_ARGCHK, and sober128\_read().

```
382 {
383     LTC_ARGCHK(outlen != NULL);
384     LTC_ARGCHK(out != NULL);
385     LTC_ARGCHK(prng != NULL);
386
387     if (*outlen < 64) {
388         *outlen = 64;
389         return CRYPT_BUFFER_OVERFLOW;
390     }
391
392     if (sober128_read(out, 64, prng) != 64) {
393         return CRYPT_ERROR_READPRNG;
394     }
395     *outlen = 64;
396
397     return CRYPT_OK;
398 }
```

Here is the call graph for this function:

**5.304.3.11 int sober128\_import (const unsigned char \* *in*, unsigned long *inlen*, prng\_state \* *prng*)**

Import a PRNG state.

**Parameters:**

- in* The PRNG state
- inlen* Size of the state
- prng* The PRNG to import

**Returns:**

CRYPT\_OK if successful

Definition at line 407 of file sober128.c.

References CRYPT\_INVALID\_ARG, CRYPT\_OK, LTC\_ARGCHK, sober128\_add\_entropy(), sober128\_ready(), and sober128\_start().

```

408 {
409     int err;
410     LTC_ARGCHK(in != NULL);
411     LTC_ARGCHK(prng != NULL);
412
413     if (inlen != 64) {
414         return CRYPT_INVALID_ARG;
415     }
416
417     if ((err = sober128_start(prng)) != CRYPT_OK) {
418         return err;
419     }
420     if ((err = sober128_add_entropy(in, 64, prng)) != CRYPT_OK) {
421         return err;
422     }
423     return sober128_ready(prng);
424 }

```

Here is the call graph for this function:

#### 5.304.3.12 unsigned long sober128\_read (unsigned char \* *out*, unsigned long *outlen*, [prng\\_state](#) \* *prng*)

Read from the PRNG.

##### Parameters:

- out* Destination
- outlen* Length of output
- prng* The active PRNG to read from

##### Returns:

- Number of octets read

Definition at line 289 of file sober128.c.

References `c`, `cycle()`, `LTC_ARGCHK`, `N`, `nltp()`, `SROUND`, `XORWORD()`, and `zeromem()`.

Referenced by `sober128_export()`.

```

290 {
291     struct sober128_prng *c;
292     ulong32 t, tlen;
293
294     LTC_ARGCHK(out != NULL);
295     LTC_ARGCHK(prng != NULL);
296
297 #ifdef LTC_VALGRIND
298     zeromem(out, outlen);
299 #endif
300
301     c = &(prng->sober128);
302     t = 0;
303     tlen = outlen;
304
305     /* handle any previously buffered bytes */
306     while (c->nbuf != 0 && outlen != 0) {
307         *out++ ^= c->sbuf & 0xFF;
308         c->sbuf >>= 8;
309         c->nbuf -= 8;
310         --outlen;

```

```

311     }
312
313 #ifndef LTC_SMALL_CODE
314     /* do lots at a time, if there's enough to do */
315     while (outlen >= N*4) {
316         SROUND(0);
317         SROUND(1);
318         SROUND(2);
319         SROUND(3);
320         SROUND(4);
321         SROUND(5);
322         SROUND(6);
323         SROUND(7);
324         SROUND(8);
325         SROUND(9);
326         SROUND(10);
327         SROUND(11);
328         SROUND(12);
329         SROUND(13);
330         SROUND(14);
331         SROUND(15);
332         SROUND(16);
333         out    += 4*N;
334         outlen -= 4*N;
335     }
336 #endif
337
338     /* do small or odd size buffers the slow way */
339     while (4 <= outlen) {
340         cycle(c->R);
341         t = nltap(c);
342         XORWORD(t, out);
343         out    += 4;
344         outlen -= 4;
345     }
346
347     /* handle any trailing bytes */
348     if (outlen != 0) {
349         cycle(c->R);
350         c->sbuf = nltap(c);
351         c->nbuf = 32;
352         while (c->nbuf != 0 && outlen != 0) {
353             *out++ ^= c->sbuf & 0xFF;
354             c->sbuf >>= 8;
355             c->nbuf -= 8;
356             --outlen;
357         }
358     }
359
360     return tlen;
361 }

```

Here is the call graph for this function:

### 5.304.3.13 int sober128\_ready ([prng\\_state](#) \* *prng*)

Make the PRNG ready to read from.

#### Parameters:

*prng* The PRNG to make active

#### Returns:

CRYPT\_OK if successful

Definition at line 273 of file sober128.c.

References CRYPT\_ERROR, and CRYPT\_OK.

Referenced by sober128\_import().

```
274 {  
275     return prng->sober128.set == 1 ? CRYPT_OK : CRYPT_ERROR;  
276 }
```

#### 5.304.3.14 int sober128\_start (*prng\_state* \* *prng*)

Start the PRNG.

##### Parameters:

*prng* [out] The PRNG state to initialize

##### Returns:

CRYPT\_OK if successful

Definition at line 109 of file sober128.c.

References c, LTC\_ARGCHK, and N.

Referenced by sober128\_import(), and sober128\_test().

```
110 {  
111     int i;  
112     struct sober128_prng *c;  
113  
114     LTC_ARGCHK(prng != NULL);  
115  
116     c = &(prng->sober128);  
117  
118     /* Register initialised to Fibonacci numbers */  
119     c->R[0] = 1;  
120     c->R[1] = 1;  
121     for (i = 2; i < N; ++i) {  
122         c->R[i] = c->R[i-1] + c->R[i-2];  
123     }  
124     c->konst = INITKONST;  
125  
126     /* next add_entropy will be the key */  
127     c->flag = 1;  
128     c->set = 0;  
129  
130     return CRYPT_OK;  
131 }
```

#### 5.304.3.15 int sober128\_test (void)

PRNG self-test.

##### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-testing has been disabled

Definition at line 430 of file sober128.c.

References CRYPT\_NOP, CRYPT\_OK, len, and sober128\_start().



```

431 {
432 #ifndef LTC_TEST
433     return CRYPT_NOP;
434 #else
435     static const struct {
436         int keylen, ivlen, len;
437         unsigned char key[16], iv[4], out[20];
438     } tests[] = {
439
440 {
441     16, 4, 20,
442
443     /* key */
444     { 0x74, 0x65, 0x73, 0x74, 0x20, 0x6b, 0x65, 0x79,
445       0x20, 0x31, 0x32, 0x38, 0x62, 0x69, 0x74, 0x73 },
446
447     /* IV */
448     { 0x00, 0x00, 0x00, 0x00 },
449
450     /* expected output */
451     { 0x43, 0x50, 0x0c, 0xcf, 0x89, 0x91, 0x9f, 0x1d,
452       0xaa, 0x37, 0x74, 0x95, 0xf4, 0xb4, 0x58, 0xc2,
453       0x40, 0x37, 0x8b, 0xbb }
454 }
455 };
456
457 prng_state prng;
458 unsigned char dst[20];
459 int err, x;
460
461 for (x = 0; x < (int)(sizeof(tests)/sizeof(tests[0])); x++) {
462     if ((err = sober128_start(&prng)) != CRYPT_OK) {
463         return err;
464     }
465     if ((err = sober128_add_entropy(tests[x].key, tests[x].keylen, &prng)) != CRYPT_OK) {
466         return err;
467     }
468     /* add IV */
469     if ((err = sober128_add_entropy(tests[x].iv, tests[x].ivlen, &prng)) != CRYPT_OK) {
470         return err;
471     }
472
473     /* ready up */
474     if ((err = sober128_ready(&prng)) != CRYPT_OK) {
475         return err;
476     }
477     XMEMSET(dst, 0, tests[x].len);
478     if (sober128_read(dst, tests[x].len, &prng) != (unsigned long)tests[x].len) {
479         return CRYPT_ERROR_READPRNG;
480     }
481     sober128_done(&prng);
482     if (XMEMCMP(dst, tests[x].out, tests[x].len)) {
483 #if 0
484         printf("\n\nSOBER128 failed, I got:\n");
485         for (y = 0; y < tests[x].len; y++) printf("%02x ", dst[y]);
486         printf("\n");
487 #endif
488         return CRYPT_FAIL_TESTVECTOR;
489     }
490 }
491 return CRYPT_OK;
492 #endif
493 }

```

Here is the call graph for this function:

**5.304.3.16 static void XORWORD ([ulong32](#) *w*, unsigned char \* *b*)** [`static`]

Definition at line 54 of file `sober128.c`.

Referenced by `sober128_read()`.

```
55 {  
56     ulong32 t;  
57     LOAD32L(t, b);  
58     t ^= w;  
59     STORE32L(t, b);  
60 }
```

**5.304.4 Variable Documentation****5.304.4.1 const struct [ltc\\_prng\\_descriptor](#) `sober128_desc`**

**Initial value:**

```
{  
    "sober128", 64,  
    &sober128_start,  
    &sober128_add_entropy,  
    &sober128_ready,  
    &sober128_read,  
    &sober128_done,  
    &sober128_export,  
    &sober128_import,  
    &sober128_test  
}
```

Definition at line 23 of file `sober128.c`.

## 5.305 prngs/sober128tab.c File Reference

### 5.305.1 Detailed Description

SOBER-128 Tables.

Definition in file [sober128tab.c](#).

This graph shows which files directly or indirectly include this file:

### Variables

- static const [ulong32 Multab](#) [256]
- static const [ulong32 Sbox](#) [256]

### 5.305.2 Variable Documentation

#### 5.305.2.1 const [ulong32 Multab](#)[256] [static]

Definition at line 8 of file sober128tab.c.

#### 5.305.2.2 const [ulong32 Sbox](#)[256] [static]

Definition at line 93 of file sober128tab.c.

## 5.306 prngs/sprng.c File Reference

### 5.306.1 Detailed Description

Secure PRNG, Tom St Denis.

Definition in file [sprng.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for sprng.c:

### Functions

- int [sprng\\_start](#) ([prng\\_state](#) \*prng)  
*Start the PRNG.*
- int [sprng\\_add\\_entropy](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)  
*Add entropy to the PRNG state.*
- int [sprng\\_ready](#) ([prng\\_state](#) \*prng)  
*Make the PRNG ready to read from.*
- unsigned long [sprng\\_read](#) (unsigned char \*out, unsigned long outlen, [prng\\_state](#) \*prng)  
*Read from the PRNG.*
- int [sprng\\_done](#) ([prng\\_state](#) \*prng)  
*Terminate the PRNG.*
- int [sprng\\_export](#) (unsigned char \*out, unsigned long \*outlen, [prng\\_state](#) \*prng)  
*Export the PRNG state.*
- int [sprng\\_import](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)  
*Import a PRNG state.*
- int [sprng\\_test](#) (void)  
*PRNG self-test.*

### Variables

- const struct [ltc\\_prng\\_descriptor](#) [sprng\\_desc](#)

### 5.306.2 Function Documentation

#### 5.306.2.1 int [sprng\\_add\\_entropy](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)

Add entropy to the PRNG state.

#### Parameters:

*in* The data to add

*inlen* Length of the data to add  
*prng* PRNG state to update

**Returns:**

CRYPT\_OK if successful

Definition at line 55 of file sprng.c.

References CRYPT\_OK.

```
56 {  
57     return CRYPT_OK;  
58 }
```

**5.306.2.2 int sprng\_done ([prng\\_state](#) \* *prng*)**

Terminate the PRNG.

**Parameters:**

*prng* The PRNG to terminate

**Returns:**

CRYPT\_OK if successful

Definition at line 88 of file sprng.c.

References CRYPT\_OK.

```
89 {  
90     return CRYPT_OK;  
91 }
```

**5.306.2.3 int sprng\_export (unsigned char \* *out*, unsigned long \* *outlen*, [prng\\_state](#) \* *prng*)**

Export the PRNG state.

**Parameters:**

*out* [out] Destination

*outlen* [in/out] Max size and resulting size of the state

*prng* The PRNG to export

**Returns:**

CRYPT\_OK if successful

Definition at line 100 of file sprng.c.

References CRYPT\_OK, and LTC\_ARGCHK.

```
101 {  
102     LTC_ARGCHK(outlen != NULL);  
103  
104     *outlen = 0;  
105     return CRYPT_OK;  
106 }
```

**5.306.2.4 int sprng\_import (const unsigned char \* *in*, unsigned long *inlen*, [prng\\_state](#) \* *prng*)**

Import a PRNG state.

**Parameters:**

*in* The PRNG state  
*inlen* Size of the state  
*prng* The PRNG to import

**Returns:**

CRYPT\_OK if successful

Definition at line 115 of file sprng.c.

References CRYPT\_OK.

```
116 {  
117     return CRYPT_OK;  
118 }
```

**5.306.2.5 unsigned long sprng\_read (unsigned char \* *out*, unsigned long *outlen*, [prng\\_state](#) \* *prng*)**

Read from the PRNG.

**Parameters:**

*out* Destination  
*outlen* Length of output  
*prng* The active PRNG to read from

**Returns:**

Number of octets read

Definition at line 77 of file sprng.c.

References LTC\_ARGCHK, and rng\_get\_bytes().

```
78 {  
79     LTC_ARGCHK(out != NULL);  
80     return rng_get_bytes(out, outlen, NULL);  
81 }
```

Here is the call graph for this function:

**5.306.2.6 int sprng\_ready ([prng\\_state](#) \* *prng*)**

Make the PRNG ready to read from.

**Parameters:**

*prng* The PRNG to make active

**Returns:**

CRYPT\_OK if successful

Definition at line 65 of file sprng.c.

References CRYPT\_OK.

```
66 {  
67     return CRYPT_OK;  
68 }
```

#### 5.306.2.7 int sprng\_start (prng\_state \* prng)

Start the PRNG.

##### Parameters:

*prng* [out] The PRNG state to initialize

##### Returns:

CRYPT\_OK if successful

Definition at line 43 of file sprng.c.

References CRYPT\_OK.

```
44 {  
45     return CRYPT_OK;  
46 }
```

#### 5.306.2.8 int sprng\_test (void)

PRNG self-test.

##### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-testing has been disabled

Definition at line 124 of file sprng.c.

References CRYPT\_OK.

```
125 {  
126     return CRYPT_OK;  
127 }
```

### 5.306.3 Variable Documentation

#### 5.306.3.1 const struct ltc\_prng\_descriptor sprng\_desc

##### Initial value:

```
{  
    "sprng", 0,  
    &sprng_start,  
    &sprng_add_entropy,  
    &sprng_ready,  
    &sprng_read,  
}
```

```
&sprng_done,  
&sprng_export,  
&sprng_import,  
&sprng_test  
}
```

Definition at line 25 of file sprng.c.



## 5.307 prngs/yarrow.c File Reference

### 5.307.1 Detailed Description

Yarrow PRNG, Tom St Denis.

Definition in file [yarrow.c](#).

```
#include "tomcrypt.h"
```

Include dependency graph for yarrow.c:

### Functions

- [int yarrow\\_start](#) ([prng\\_state](#) \*prng)  
*Start the PRNG.*
- [int yarrow\\_add\\_entropy](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)  
*Add entropy to the PRNG state.*
- [int yarrow\\_ready](#) ([prng\\_state](#) \*prng)  
*Make the PRNG ready to read from.*
- unsigned long [yarrow\\_read](#) (unsigned char \*out, unsigned long outlen, [prng\\_state](#) \*prng)  
*Read from the PRNG.*
- [int yarrow\\_done](#) ([prng\\_state](#) \*prng)  
*Terminate the PRNG.*
- [int yarrow\\_export](#) (unsigned char \*out, unsigned long \*outlen, [prng\\_state](#) \*prng)  
*Export the PRNG state.*
- [int yarrow\\_import](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)  
*Import a PRNG state.*
- [int yarrow\\_test](#) (void)  
*PRNG self-test.*

### Variables

- const struct [ltc\\_prng\\_descriptor](#) yarrow\_desc

### 5.307.2 Function Documentation

#### 5.307.2.1 [int yarrow\\_add\\_entropy](#) (const unsigned char \*in, unsigned long inlen, [prng\\_state](#) \*prng)

Add entropy to the PRNG state.

**Parameters:**

*in* The data to add  
*inlen* Length of the data to add  
*prng* PRNG state to update

**Returns:**

CRYPT\_OK if successful

Definition at line 135 of file yarrow.c.

References CRYPT\_OK, hash\_descriptor, hash\_is\_valid(), LTC\_ARGCHK, LTC\_Mutex\_LOCK, and LTC\_Mutex\_UNLOCK.

Referenced by yarrow\_import().

```

136 {
137     hash_state md;
138     int err;
139
140     LTC_ARGCHK(in != NULL);
141     LTC_ARGCHK(prng != NULL);
142
143     LTC_Mutex_LOCK(&prng->yarrow.prng_lock);
144
145     if ((err = hash_is_valid(prng->yarrow.hash)) != CRYPT_OK) {
146         LTC_Mutex_UNLOCK(&prng->yarrow.prng_lock);
147         return err;
148     }
149
150     /* start the hash */
151     if ((err = hash_descriptor[prng->yarrow.hash].init(&md)) != CRYPT_OK) {
152         LTC_Mutex_UNLOCK(&prng->yarrow.prng_lock);
153         return err;
154     }
155
156     /* hash the current pool */
157     if ((err = hash_descriptor[prng->yarrow.hash].process(&md, prng->yarrow.pool,
158                                                         hash_descriptor[prng->yarrow.hash].hashsize))
159         LTC_Mutex_UNLOCK(&prng->yarrow.prng_lock);
160         return err;
161     }
162
163     /* add the new entropy */
164     if ((err = hash_descriptor[prng->yarrow.hash].process(&md, in, inlen)) != CRYPT_OK) {
165         LTC_Mutex_UNLOCK(&prng->yarrow.prng_lock);
166         return err;
167     }
168
169     /* store result */
170     if ((err = hash_descriptor[prng->yarrow.hash].done(&md, prng->yarrow.pool)) != CRYPT_OK) {
171         LTC_Mutex_UNLOCK(&prng->yarrow.prng_lock);
172         return err;
173     }
174
175     LTC_Mutex_UNLOCK(&prng->yarrow.prng_lock);
176     return CRYPT_OK;
177 }
```

Here is the call graph for this function:

### 5.307.2.2 int yarrow\_done (prng\_state \* prng)

Terminate the PRNG.

**Parameters:***prng* The PRNG to terminate**Returns:**

CRYPT\_OK if successful

Definition at line 252 of file yarrow.c.

References `ctr_done()`, `LTC_ARGCHK`, `LTC_Mutex_Lock`, and `LTC_Mutex_Unlock`.

```

253 {
254     int err;
255     LTC_ARGCHK(prng != NULL);
256
257     LTC_Mutex_Lock(&prng->yarrow.prng_lock);
258
259     /* call cipher done when we invent one ;- ) */
260
261     /* we invented one */
262     err = ctr_done(&prng->yarrow.ctr);
263
264     LTC_Mutex_Unlock(&prng->yarrow.prng_lock);
265     return err;
266 }

```

Here is the call graph for this function:

**5.307.2.3 int yarrow\_export (unsigned char \* out, unsigned long \* outlen, [prng\\_state](#) \* prng)**

Export the PRNG state.

**Parameters:***out* [out] Destination*outlen* [in/out] Max size and resulting size of the state*prng* The PRNG to export**Returns:**

CRYPT\_OK if successful

Definition at line 275 of file yarrow.c.

References `CRYPT_BUFFER_OVERFLOW`, `CRYPT_ERROR_READPRNG`, `CRYPT_OK`, `LTC_ARGCHK`, `LTC_Mutex_Lock`, `LTC_Mutex_Unlock`, and `yarrow_read()`.

```

276 {
277     LTC_ARGCHK(out != NULL);
278     LTC_ARGCHK(outlen != NULL);
279     LTC_ARGCHK(prng != NULL);
280
281     LTC_Mutex_Lock(&prng->yarrow.prng_lock);
282
283     /* we'll write 64 bytes for s&g's */
284     if (*outlen < 64) {
285         LTC_Mutex_Unlock(&prng->yarrow.prng_lock);
286         *outlen = 64;
287         return CRYPT_BUFFER_OVERFLOW;
288     }
289

```

```

290     if (yarrow_read(out, 64, prng) != 64) {
291         LTC_MUTEX_UNLOCK(&prng->yarrow.prng_lock);
292         return CRYPT_ERROR_READPRNG;
293     }
294     *outlen = 64;
295
296     return CRYPT_OK;
297 }

```

Here is the call graph for this function:

#### 5.307.2.4 int yarrow\_import (const unsigned char \* *in*, unsigned long *inlen*, *prng\_state* \* *prng*)

Import a PRNG state.

##### Parameters:

- in* The PRNG state
- inlen* Size of the state
- prng* The PRNG to import

##### Returns:

CRYPT\_OK if successful

Definition at line 306 of file yarrow.c.

References CRYPT\_INVALID\_ARG, CRYPT\_OK, LTC\_ARGCHK, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, yarrow\_add\_entropy(), and yarrow\_start().

```

307 {
308     int err;
309
310     LTC_ARGCHK(in != NULL);
311     LTC_ARGCHK(prng != NULL);
312
313     LTC_MUTEX_LOCK(&prng->yarrow.prng_lock);
314
315     if (inlen != 64) {
316         LTC_MUTEX_UNLOCK(&prng->yarrow.prng_lock);
317         return CRYPT_INVALID_ARG;
318     }
319
320     if ((err = yarrow_start(prng)) != CRYPT_OK) {
321         LTC_MUTEX_UNLOCK(&prng->yarrow.prng_lock);
322         return err;
323     }
324     err = yarrow_add_entropy(in, 64, prng);
325     LTC_MUTEX_UNLOCK(&prng->yarrow.prng_lock);
326     return err;
327 }

```

Here is the call graph for this function:

#### 5.307.2.5 unsigned long yarrow\_read (unsigned char \* *out*, unsigned long *outlen*, *prng\_state* \* *prng*)

Read from the PRNG.

**Parameters:**

*out* Destination  
*outlen* Length of output  
*prng* The active PRNG to read from

**Returns:**

Number of octets read

Definition at line 228 of file yarrow.c.

References CRYPT\_OK, ctr\_encrypt(), LTC\_ARGCHK, LTC\_MUTEX\_LOCK, LTC\_MUTEX\_UNLOCK, and zeromem().

Referenced by yarrow\_export().

```

229 {
230     LTC_ARGCHK(out != NULL);
231     LTC_ARGCHK(prng != NULL);
232
233     LTC_MUTEX_LOCK(&prng->yarrow.prng_lock);
234
235     /* put out in predictable state first */
236     zeromem(out, outlen);
237
238     /* now randomize it */
239     if (ctr_encrypt(out, out, outlen, &prng->yarrow.ctr) != CRYPT_OK) {
240         LTC_MUTEX_UNLOCK(&prng->yarrow.prng_lock);
241         return 0;
242     }
243     LTC_MUTEX_UNLOCK(&prng->yarrow.prng_lock);
244     return outlen;
245 }
```

Here is the call graph for this function:

**5.307.2.6 int yarrow\_ready (prng\_state \* prng)**

Make the PRNG ready to read from.

**Parameters:**

*prng* The PRNG to make active

**Returns:**

CRYPT\_OK if successful

Definition at line 184 of file yarrow.c.

References cipher\_descriptor, cipher\_is\_valid(), CRYPT\_OK, ctr\_start(), hash\_descriptor, hash\_is\_valid(), LTC\_ARGCHK, LTC\_MUTEX\_LOCK, and LTC\_MUTEX\_UNLOCK.

```

185 {
186     int ks, err;
187
188     LTC_ARGCHK(prng != NULL);
189     LTC_MUTEX_LOCK(&prng->yarrow.prng_lock);
190
191     if ((err = hash_is_valid(prng->yarrow.hash)) != CRYPT_OK) {
192         LTC_MUTEX_UNLOCK(&prng->yarrow.prng_lock);

```

```

193     return err;
194 }
195
196 if ((err = cipher_is_valid(prng->yarrow.cipher)) != CRYPT_OK) {
197     LTC_Mutex_UNLOCK(&prng->yarrow.prng_lock);
198     return err;
199 }
200
201 /* setup CTR mode using the "pool" as the key */
202 ks = (int)hash_descriptor[prng->yarrow.hash].hashsize;
203 if ((err = cipher_descriptor[prng->yarrow.cipher].keysize(&ks)) != CRYPT_OK) {
204     LTC_Mutex_UNLOCK(&prng->yarrow.prng_lock);
205     return err;
206 }
207
208 if ((err = ctr_start(prng->yarrow.cipher,      /* what cipher to use */
209                     prng->yarrow.pool,        /* IV */
210                     prng->yarrow.pool, ks,    /* KEY and key size */
211                     0,                        /* number of rounds */
212                     CTR_COUNTER_LITTLE_ENDIAN, /* little endian counter */
213                     &prng->yarrow.ctr)) != CRYPT_OK) {
214     LTC_Mutex_UNLOCK(&prng->yarrow.prng_lock);
215     return err;
216 }
217 LTC_Mutex_UNLOCK(&prng->yarrow.prng_lock);
218 return CRYPT_OK;
219 }

```

Here is the call graph for this function:

### 5.307.2.7 int yarrow\_start (prng\_state \* prng)

Start the PRNG.

#### Parameters:

*prng* [out] The PRNG state to initialize

#### Returns:

CRYPT\_OK if successful

Definition at line 38 of file yarrow.c.

References aes\_desc, anubis\_desc, blowfish\_desc, cast5\_desc, cipher\_is\_valid(), CRYPT\_OK, des3\_desc, hash\_is\_valid(), khazad\_desc, kseed\_desc, LTC\_ARGCHK, LTC\_Mutex\_INIT, md2\_desc, md4\_desc, md5\_desc, noekeon\_desc, rc2\_desc, rc5\_desc, rc6\_desc, register\_cipher(), register\_hash(), rijndael\_desc, rmd128\_desc, rmd160\_desc, rmd256\_desc, rmd320\_desc, safer\_sk128\_desc, saferp\_desc, sha1\_desc, sha256\_desc, sha512\_desc, tiger\_desc, twofish\_desc, whirlpool\_desc, xtea\_desc, and zeromem().

Referenced by yarrow\_import(), and yarrow\_test().

```

39 {
40     int err;
41
42     LTC_ARGCHK(prng != NULL);
43
44     /* these are the default hash/cipher combo used */
45 #ifdef RIJNDAEL
46 #if YARROW_AES==0
47     prng->yarrow.cipher = register_cipher(&rijndael_enc_desc);
48 #elif YARROW_AES==1
49     prng->yarrow.cipher = register_cipher(&aes_enc_desc);

```

```
50 #elif YARROW_AES==2
51     prng->yarrow.cipher = register_cipher(&rijndael_desc);
52 #elif YARROW_AES==3
53     prng->yarrow.cipher = register_cipher(&aes_desc);
54 #endif
55 #elif defined(BLOWFISH)
56     prng->yarrow.cipher = register_cipher(&blowfish_desc);
57 #elif defined(TWOFISH)
58     prng->yarrow.cipher = register_cipher(&twofish_desc);
59 #elif defined(RC6)
60     prng->yarrow.cipher = register_cipher(&rc6_desc);
61 #elif defined(RC5)
62     prng->yarrow.cipher = register_cipher(&rc5_desc);
63 #elif defined(SAFERP)
64     prng->yarrow.cipher = register_cipher(&saferp_desc);
65 #elif defined(RC2)
66     prng->yarrow.cipher = register_cipher(&rc2_desc);
67 #elif defined(NOEKEON)
68     prng->yarrow.cipher = register_cipher(&noekeon_desc);
69 #elif defined(ANUBIS)
70     prng->yarrow.cipher = register_cipher(&anubis_desc);
71 #elif defined(KSEED)
72     prng->yarrow.cipher = register_cipher(&kseed_desc);
73 #elif defined(KHAZAD)
74     prng->yarrow.cipher = register_cipher(&khazad_desc);
75 #elif defined(CAST5)
76     prng->yarrow.cipher = register_cipher(&cast5_desc);
77 #elif defined(XTEA)
78     prng->yarrow.cipher = register_cipher(&xtea_desc);
79 #elif defined(SAFER)
80     prng->yarrow.cipher = register_cipher(&safer_sk128_desc);
81 #elif defined(DES)
82     prng->yarrow.cipher = register_cipher(&des3_desc);
83 #else
84     #error YARROW needs at least one CIPHER
85 #endif
86 if ((err = cipher_is_valid(prng->yarrow.cipher)) != CRYPT_OK) {
87     return err;
88 }
89
90 #ifdef SHA256
91     prng->yarrow.hash = register_hash(&sha256_desc);
92 #elif defined(SHA512)
93     prng->yarrow.hash = register_hash(&sha512_desc);
94 #elif defined(TIGER)
95     prng->yarrow.hash = register_hash(&tiger_desc);
96 #elif defined(SHA1)
97     prng->yarrow.hash = register_hash(&sha1_desc);
98 #elif defined(RIPEMD320)
99     prng->yarrow.hash = register_hash(&rmd320_desc);
100 #elif defined(RIPEMD256)
101     prng->yarrow.hash = register_hash(&rmd256_desc);
102 #elif defined(RIPEMD160)
103     prng->yarrow.hash = register_hash(&rmd160_desc);
104 #elif defined(RIPEMD128)
105     prng->yarrow.hash = register_hash(&rmd128_desc);
106 #elif defined(MD5)
107     prng->yarrow.hash = register_hash(&md5_desc);
108 #elif defined(MD4)
109     prng->yarrow.hash = register_hash(&md4_desc);
110 #elif defined(MD2)
111     prng->yarrow.hash = register_hash(&md2_desc);
112 #elif defined(WHIRLPOOL)
113     prng->yarrow.hash = register_hash(&whirlpool_desc);
114 #else
115     #error YARROW needs at least one HASH
116 #endif
```

```

117     if ((err = hash_is_valid(prng->yarrow.hash)) != CRYPT_OK) {
118         return err;
119     }
120
121     /* zero the memory used */
122     zeromem(prng->yarrow.pool, sizeof(prng->yarrow.pool));
123     LTC_MUTEX_INIT(&prng->yarrow.prng_lock)
124
125     return CRYPT_OK;
126 }

```

Here is the call graph for this function:

### 5.307.2.8 int yarrow\_test (void)

PRNG self-test.

#### Returns:

CRYPT\_OK if successful, CRYPT\_NOP if self-testing has been disabled

Definition at line 333 of file yarrow.c.

References cipher\_descriptor, CRYPT\_NOP, CRYPT\_OK, hash\_descriptor, and yarrow\_start().

```

334 {
335 #ifndef LTC_TEST
336     return CRYPT_NOP;
337 #else
338     int err;
339     prng_state prng;
340
341     if ((err = yarrow_start(&prng)) != CRYPT_OK) {
342         return err;
343     }
344
345     /* now let's test the hash/cipher that was chosen */
346     if ((err = cipher_descriptor[prng.yarrow.cipher].test()) != CRYPT_OK) {
347         return err;
348     }
349     if ((err = hash_descriptor[prng.yarrow.hash].test()) != CRYPT_OK) {
350         return err;
351     }
352
353     return CRYPT_OK;
354 #endif
355 }

```

Here is the call graph for this function:

## 5.307.3 Variable Documentation

### 5.307.3.1 const struct ltc\_prng\_descriptor yarrow\_desc

#### Initial value:

```

{
    "yarrow", 64,
    &yarrow_start,
    &yarrow_add_entropy,

```



```
&yarrow_ready,  
&yarrow_read,  
&yarrow_done,  
&yarrow_export,  
&yarrow_import,  
&yarrow_test  
}
```

Definition at line 20 of file yarrow.c.

# Index

- `_chc_process`
      - `chc.c`, 296
  - `accel_cbc_decrypt`
    - `ltc_cipher_descriptor`, 17
  - `accel_cbc_encrypt`
    - `ltc_cipher_descriptor`, 17
  - `accel_ccm_memory`
    - `ltc_cipher_descriptor`, 17
  - `accel_ctr_encrypt`
    - `ltc_cipher_descriptor`, 18
  - `accel_ecb_decrypt`
    - `ltc_cipher_descriptor`, 18
  - `accel_ecb_encrypt`
    - `ltc_cipher_descriptor`, 18
  - `accel_gcm_memory`
    - `ltc_cipher_descriptor`, 19
  - `accel_lrw_decrypt`
    - `ltc_cipher_descriptor`, 19
  - `accel_lrw_encrypt`
    - `ltc_cipher_descriptor`, 20
  - `add`
    - `ltc_math_descriptor`, 31
  - `add_entropy`
    - `ltc_prng_descriptor`, 44
  - `addi`
    - `ltc_math_descriptor`, 31
  - `ADDKEY`
    - `sober128.c`, 899
  - `aes.c`
    - `aes_desc`, 63
    - `ECB_DEC`, 52
    - `ECB_DONE`, 52, 55
    - `ECB_ENC`, 52, 55
    - `ECB_KS`, 52, 58
    - `ECB_TEST`, 52, 58
    - `rijndael_desc`, 63
    - `SETUP`, 52, 60
    - `setup_mix`, 63
  - `aes_desc`
    - `aes.c`, 63
  - `aes_tab.c`
    - `rcon`, 65
    - `TD0`, 66
    - `Td0`, 64
  - `TD1`, 66
  - `Td1`, 64
  - `TD2`, 66
  - `Td2`, 65
  - `TD3`, 66
  - `Td3`, 65
  - `Td4`, 66
  - `TE0`, 66
  - `Te0`, 65
  - `TE1`, 66
  - `Te1`, 65
  - `TE2`, 66
  - `Te2`, 65
  - `TE3`, 66
  - `Te3`, 65
  - `Te4`, 66
  - `Te4_0`, 66
  - `Te4_1`, 67
  - `Te4_2`, 67
  - `Te4_3`, 67
  - `Tks0`, 67
  - `Tks1`, 67
  - `Tks2`, 67
  - `Tks3`, 67
- `ANUBIS`
    - `tomcrypt_custom.h`, 425
  - `anubis.c`
    - `anubis_crypt`, 70
    - `anubis_desc`, 81
    - `anubis_done`, 71
    - `anubis_ecb_decrypt`, 71
    - `anubis_ecb_encrypt`, 72
    - `anubis_keysize`, 72
    - `anubis_setup`, 73
    - `anubis_test`, 75
    - `BLOCKSIZE`, 69
    - `BLOCKSIZEB`, 69
    - `MAX_KEYSIZEB`, 69
    - `MAX_N`, 69
    - `MAX_ROUNDS`, 69
    - `MIN_KEYSIZEB`, 69
    - `MIN_N`, 69
    - `MIN_ROUNDS`, 69
    - `rc`, 81
    - `T0`, 81

- T1, [82](#)
- T2, [82](#)
- T3, [82](#)
- T4, [82](#)
- T5, [82](#)
- anubis\_crypt
  - anubis.c, [70](#)
- anubis\_desc
  - anubis.c, [81](#)
- anubis\_done
  - anubis.c, [71](#)
- anubis\_ecb\_decrypt
  - anubis.c, [71](#)
- anubis\_ecb\_encrypt
  - anubis.c, [72](#)
- anubis\_keysize
  - anubis.c, [72](#)
- anubis\_setup
  - anubis.c, [73](#)
- anubis\_test
  - anubis.c, [75](#)
- ANUBIS\_TWEAK
  - tomcrypt\_custom.h, [425](#)
- ARGTYPE
  - tomcrypt\_cfg.h, [416](#)
- B
  - sober128.c, [899](#)
- BASE64
  - tomcrypt\_custom.h, [425](#)
- base64\_decode
  - base64\_decode.c, [557](#)
- base64\_decode.c
  - base64\_decode, [557](#)
  - map, [558](#)
- base64\_encode
  - base64\_encode.c, [560](#)
- base64\_encode.c
  - base64\_encode, [560](#)
  - codes, [561](#)
- baseten
  - der\_encode\_utctime.c, [754](#)
- bigbyte
  - des.c, [112](#)
- bits\_per\_digit
  - ltc\_math\_descriptor, [31](#)
- block\_length
  - ltc\_cipher\_descriptor, [20](#)
- BLOCKSIZE
  - anubis.c, [69](#)
  - khazad.c, [126](#)
- blocksize
  - ltc\_hash\_descriptor, [26](#)
- BLOCKSIZEB
  - anubis.c, [69](#)
  - khazad.c, [126](#)
- BLOWFISH
  - tomcrypt\_custom.h, [425](#)
- blowfish.c
  - blowfish\_desc, [89](#)
  - blowfish\_done, [84](#)
  - blowfish\_ecb\_decrypt, [84](#)
  - blowfish\_ecb\_encrypt, [85](#)
  - blowfish\_keysize, [86](#)
  - blowfish\_setup, [86](#)
  - blowfish\_test, [87](#)
- F, [83](#)
- ORIG\_P, [89](#)
- ORIG\_S, [89](#)
- blowfish\_desc
  - blowfish.c, [89](#)
- blowfish\_done
  - blowfish.c, [84](#)
- blowfish\_ecb\_decrypt
  - blowfish.c, [84](#)
- blowfish\_ecb\_encrypt
  - blowfish.c, [85](#)
- blowfish\_keysize
  - blowfish.c, [86](#)
- blowfish\_setup
  - blowfish.c, [86](#)
- blowfish\_test
  - blowfish.c, [87](#)
- BSWAP
  - tomcrypt\_macros.h, [446](#)
- burn\_stack
  - burn\_stack.c, [562](#)
  - tomcrypt\_misc.h, [452](#)
- burn\_stack.c
  - burn\_stack, [562](#)
- byte
  - tomcrypt\_macros.h, [446](#)
- BYTE2WORD
  - sober128.c, [901](#)
- bytebit
  - des.c, [112](#)
- c
  - khazad.c, [132](#)
- CAST5
  - tomcrypt\_custom.h, [425](#)
- cast5.c
  - cast5\_desc, [96](#)
  - cast5\_done, [91](#)
  - cast5\_ecb\_decrypt, [91](#)
  - cast5\_ecb\_encrypt, [92](#)
  - cast5\_keysize, [93](#)
  - cast5\_setup, [93](#)

- cast5\_test, 95
- FI, 96
- FII, 96
- FIII, 96
- GB, 91
- INLINE, 91
- S1, 97
- S2, 97
- S3, 97
- S4, 97
- S5, 97
- S6, 97
- S7, 97
- S8, 98
- cast5\_desc
  - cast5.c, 96
- cast5\_done
  - cast5.c, 91
- cast5\_ecb\_decrypt
  - cast5.c, 91
- cast5\_ecb\_encrypt
  - cast5.c, 92
- cast5\_keysize
  - cast5.c, 93
- cast5\_setup
  - cast5.c, 93
- cast5\_test
  - cast5.c, 95
- cbc\_decrypt
  - cbc\_decrypt.c, 600
- cbc\_decrypt.c
  - cbc\_decrypt, 600
- cbc\_done
  - cbc\_done.c, 602
- cbc\_done.c
  - cbc\_done, 602
- cbc\_encrypt
  - cbc\_encrypt.c, 603
- cbc\_encrypt.c
  - cbc\_encrypt, 603
- cbc\_getiv
  - cbc\_getiv.c, 605
- cbc\_getiv.c
  - cbc\_getiv, 605
- cbc\_setiv
  - cbc\_setiv.c, 606
- cbc\_setiv.c
  - cbc\_setiv, 606
- cbc\_start
  - cbc\_start.c, 607
- cbc\_start.c
  - cbc\_start, 607
- ccm\_memory
  - ccm\_memory.c, 219
- ccm\_memory.c
  - ccm\_memory, 219
- CCM\_MODE
  - tomcrypt\_custom.h, 425
- ccm\_test
  - ccm\_test.c, 225
- ccm\_test.c
  - ccm\_test, 225
- cfb\_decrypt
  - cfb\_decrypt.c, 609
- cfb\_decrypt.c
  - cfb\_decrypt, 609
- cfb\_done
  - cfb\_done.c, 611
- cfb\_done.c
  - cfb\_done, 611
- cfb\_encrypt
  - cfb\_encrypt.c, 612
- cfb\_encrypt.c
  - cfb\_encrypt, 612
- cfb\_getiv
  - cfb\_getiv.c, 614
- cfb\_getiv.c
  - cfb\_getiv, 614
- cfb\_setiv
  - cfb\_setiv.c, 615
- cfb\_setiv.c
  - cfb\_setiv, 615
- cfb\_start
  - cfb\_start.c, 616
- cfb\_start.c
  - cfb\_start, 616
- Ch
  - sha256.c, 377
  - sha512.c, 388
- char\_to\_int
  - der\_decode\_utctime.c, 750
- chc.c
  - \_chc\_process, 296
  - chc\_compress, 296
  - chc\_desc, 298
  - chc\_init, 296
  - chc\_register, 297
  - cipher\_blocksize, 298
  - cipher\_idx, 299
  - HASH\_PROCESS, 298
  - in, 299
  - UNDEFED\_HASH, 295
- chc\_compress
  - chc.c, 296
- chc\_desc
  - chc.c, 298
- CHC\_HASH
  - tomcrypt\_custom.h, 425

- chc\_init
  - chc.c, 296
- chc\_register
  - chc.c, 297
- cipher\_blocksize
  - chc.c, 298
- cipher\_descriptor
  - crypt\_cipher\_descriptor.c, 565
  - tomcrypt\_cipher.h, 422
- cipher\_idx
  - chc.c, 299
- cipher\_is\_valid
  - crypt\_cipher\_is\_valid.c, 566
  - tomcrypt\_cipher.h, 418
- ciphers/aes/aes.c, 51
- ciphers/aes/aes\_tab.c, 64
- ciphers/anubis.c, 68
- ciphers/blowfish.c, 83
- ciphers/cast5.c, 90
- ciphers/des.c, 99
- ciphers/kasumi.c, 118
- ciphers/khazad.c, 125
- ciphers/kseed.c, 134
- ciphers/noekeon.c, 141
- ciphers/rc2.c, 149
- ciphers/rc5.c, 156
- ciphers/rc6.c, 163
- ciphers/safer/safer.c, 171
- ciphers/safer/safer\_tab.c, 181
- ciphers/safer/saferp.c, 183
- ciphers/skipjack.c, 194
- ciphers/twofish/twofish.c, 202
- ciphers/twofish/twofish\_tab.c, 213
- ciphers/xtea.c, 214
- code
  - der\_length\_ia5\_string.c, 684
  - der\_length\_printable\_string.c, 709
- codes
  - base64\_encode.c, 561
- compare
  - ltc\_math\_descriptor, 31
- compare\_d
  - ltc\_math\_descriptor, 31
- CONST64
  - tomcrypt\_macros.h, 446
- cont
  - whirltab.c, 408
- cookey
  - des.c, 100
- copy
  - ltc\_math\_descriptor, 32
- count\_bits
  - ltc\_math\_descriptor, 32
- count\_lsb\_bits
  - ltc\_math\_descriptor, 32
- CRYPT
  - tomcrypt.h, 411
- crypt.c
  - crypt\_build\_settings, 563
- crypt\_argchk
  - crypt\_argchk.c, 564
  - tomcrypt\_argchk.h, 415
- crypt\_argchk.c
  - crypt\_argchk, 564
- CRYPT\_BUFFER\_OVERFLOW
  - tomcrypt.h, 412
- crypt\_build\_settings
  - crypt.c, 563
  - tomcrypt\_misc.h, 454
- crypt\_cipher\_descriptor.c
  - cipher\_descriptor, 565
- crypt\_cipher\_is\_valid.c
  - cipher\_is\_valid, 566
- CRYPT\_ERROR
  - tomcrypt.h, 412
- CRYPT\_ERROR\_READPRNG
  - tomcrypt.h, 412
- CRYPT\_FAIL\_TESTVECTOR
  - tomcrypt.h, 412
- CRYPT\_FILE\_NOTFOUND
  - tomcrypt.h, 412
- crypt\_find\_cipher.c
  - find\_cipher, 567
- crypt\_find\_cipher\_any.c
  - find\_cipher\_any, 568
- crypt\_find\_cipher\_id.c
  - find\_cipher\_id, 570
- crypt\_find\_hash.c
  - find\_hash, 571
- crypt\_find\_hash\_any.c
  - find\_hash\_any, 572
- crypt\_find\_hash\_id.c
  - find\_hash\_id, 574
- crypt\_find\_hash\_oid.c
  - find\_hash\_oid, 575
- crypt\_find\_prng.c
  - find\_prng, 576
- crypt\_fsa
  - crypt\_fsa.c, 577
  - tomcrypt\_misc.h, 452
- crypt\_fsa.c
  - crypt\_fsa, 577
- crypt\_hash\_descriptor.c
  - hash\_descriptor, 579
- crypt\_hash\_is\_valid.c
  - hash\_is\_valid, 580
- CRYPT\_INVALID\_ARG
  - tomcrypt.h, 412

- CRYPT\_INVALID\_CIPHER
  - tomcrypt.h, [412](#)
- CRYPT\_INVALID\_HASH
  - tomcrypt.h, [412](#)
- CRYPT\_INVALID\_KEYSIZE
  - tomcrypt.h, [412](#)
- CRYPT\_INVALID\_PACKET
  - tomcrypt.h, [412](#)
- CRYPT\_INVALID\_PRIME\_SIZE
  - tomcrypt.h, [412](#)
- CRYPT\_INVALID\_PRNG
  - tomcrypt.h, [412](#)
- CRYPT\_INVALID\_PRNGSIZE
  - tomcrypt.h, [412](#)
- CRYPT\_INVALID\_ROUNDS
  - tomcrypt.h, [412](#)
- crypt\_ltc\_mp\_descriptor.c
  - ltc\_mp, [581](#)
- CRYPT\_MEM
  - tomcrypt.h, [412](#)
- CRYPT\_NOP
  - tomcrypt.h, [412](#)
- CRYPT\_OK
  - tomcrypt.h, [412](#)
- CRYPT\_PK\_DUP
  - tomcrypt.h, [412](#)
- CRYPT\_PK\_INVALID\_PADDING
  - tomcrypt.h, [412](#)
- CRYPT\_PK\_INVALID\_SIZE
  - tomcrypt.h, [412](#)
- CRYPT\_PK\_INVALID\_SYSTEM
  - tomcrypt.h, [412](#)
- CRYPT\_PK\_INVALID\_TYPE
  - tomcrypt.h, [412](#)
- CRYPT\_PK\_NOT\_FOUND
  - tomcrypt.h, [412](#)
- CRYPT\_PK\_NOT\_PRIVATE
  - tomcrypt.h, [412](#)
- CRYPT\_PK\_TYPE\_MISMATCH
  - tomcrypt.h, [412](#)
- crypt\_prng\_descriptor.c
  - prng\_descriptor, [582](#)
- crypt\_prng\_is\_valid.c
  - prng\_is\_valid, [583](#)
- crypt\_register\_cipher.c
  - register\_cipher, [584](#)
- crypt\_register\_hash.c
  - register\_hash, [586](#)
- crypt\_register\_prng.c
  - register\_prng, [588](#)
- crypt\_unregister\_cipher.c
  - unregister\_cipher, [590](#)
- crypt\_unregister\_hash.c
  - unregister\_hash, [591](#)
- crypt\_unregister\_prng.c
  - unregister\_prng, [592](#)
- ctr\_decrypt
  - ctr\_decrypt.c, [618](#)
- ctr\_decrypt.c
  - ctr\_decrypt, [618](#)
- ctr\_done
  - ctr\_done.c, [619](#)
- ctr\_done.c
  - ctr\_done, [619](#)
- ctr\_encrypt
  - ctr\_encrypt.c, [620](#)
- ctr\_encrypt.c
  - ctr\_encrypt, [620](#)
- ctr\_getiv
  - ctr\_getiv.c, [622](#)
- ctr\_getiv.c
  - ctr\_getiv, [622](#)
- ctr\_setiv
  - ctr\_setiv.c, [623](#)
- ctr\_setiv.c
  - ctr\_setiv, [623](#)
- ctr\_start
  - ctr\_start.c, [625](#)
- ctr\_start.c
  - ctr\_start, [625](#)
- ctr\_test
  - ctr\_test.c, [627](#)
- ctr\_test.c
  - ctr\_test, [627](#)
- cycle
  - sober128.c, [901](#)
- data
  - Hash\_state, [14](#)
  - Symmetric\_key, [49](#)
- DE1
  - des.c, [100](#)
- DECODE\_V
  - der\_decode\_utctime.c, [750](#)
- default\_rounds
  - ltc\_cipher\_descriptor, [20](#)
- deinit
  - ltc\_math\_descriptor, [32](#)
- der\_decode\_bit\_string
  - der\_decode\_bit\_string.c, [667](#)
- der\_decode\_bit\_string.c
  - der\_decode\_bit\_string, [667](#)
- der\_decode\_boolean
  - der\_decode\_boolean.c, [672](#)
- der\_decode\_boolean.c
  - der\_decode\_boolean, [672](#)
- der\_decode\_choice
  - der\_decode\_choice.c, [675](#)

- der\_decode\_choice.c
  - der\_decode\_choice, 675
- der\_decode\_ia5\_string
  - der\_decode\_ia5\_string.c, 678
- der\_decode\_ia5\_string.c
  - der\_decode\_ia5\_string, 678
- der\_decode\_integer
  - der\_decode\_integer.c, 685
- der\_decode\_integer.c
  - der\_decode\_integer, 685
- der\_decode\_object\_identifier
  - der\_decode\_object\_identifier.c, 692
- der\_decode\_object\_identifier.c
  - der\_decode\_object\_identifier, 692
- der\_decode\_octet\_string
  - der\_decode\_octet\_string.c, 698
- der\_decode\_octet\_string.c
  - der\_decode\_octet\_string, 698
- der\_decode\_printable\_string
  - der\_decode\_printable\_string.c, 703
- der\_decode\_printable\_string.c
  - der\_decode\_printable\_string, 703
- der\_decode\_sequence\_ex
  - der\_decode\_sequence\_ex.c, 710
- der\_decode\_sequence\_ex.c
  - der\_decode\_sequence\_ex, 710
- der\_decode\_sequence\_flexi
  - der\_decode\_sequence\_flexi.c, 715
- der\_decode\_sequence\_flexi.c
  - der\_decode\_sequence\_flexi, 715
  - fetch\_length, 720
- der\_decode\_sequence\_multi
  - der\_decode\_sequence\_multi.c, 722
- der\_decode\_sequence\_multi.c
  - der\_decode\_sequence\_multi, 722
- der\_decode\_short\_integer
  - der\_decode\_short\_integer.c, 744
- der\_decode\_short\_integer.c
  - der\_decode\_short\_integer, 744
- der\_decode\_utctime
  - der\_decode\_utctime.c, 751
- der\_decode\_utctime.c
  - char\_to\_int, 750
  - DECODE\_V, 750
  - der\_decode\_utctime, 751
- der\_encode\_bit\_string
  - der\_encode\_bit\_string.c, 669
- der\_encode\_bit\_string.c
  - der\_encode\_bit\_string, 669
- der\_encode\_boolean
  - der\_encode\_boolean.c, 673
- der\_encode\_boolean.c
  - der\_encode\_boolean, 673
- der\_encode\_ia5\_string
  - der\_encode\_ia5\_string.c, 680
- der\_encode\_ia5\_string.c
  - der\_encode\_ia5\_string, 680
- der\_encode\_integer
  - der\_encode\_integer.c, 687
- der\_encode\_integer.c
  - der\_encode\_integer, 687
- der\_encode\_object\_identifier
  - der\_encode\_object\_identifier.c, 694
- der\_encode\_object\_identifier.c
  - der\_encode\_object\_identifier, 694
- der\_encode\_octet\_string
  - der\_encode\_octet\_string.c, 700
- der\_encode\_octet\_string.c
  - der\_encode\_octet\_string, 700
- der\_encode\_printable\_string
  - der\_encode\_printable\_string.c, 705
- der\_encode\_printable\_string.c
  - der\_encode\_printable\_string, 705
- der\_encode\_sequence\_ex
  - der\_encode\_sequence\_ex.c, 725
- der\_encode\_sequence\_ex.c
  - der\_encode\_sequence\_ex, 725
- der\_encode\_sequence\_multi
  - der\_encode\_sequence\_multi.c, 731
- der\_encode\_sequence\_multi.c
  - der\_encode\_sequence\_multi, 731
- der\_encode\_set
  - der\_encode\_set.c, 739
- der\_encode\_set.c
  - der\_encode\_set, 739
  - ltc\_to\_asn1, 739
  - qsort\_helper, 740
- der\_encode\_setof
  - der\_encode\_setof.c, 741
- der\_encode\_setof.c
  - der\_encode\_setof, 741
  - qsort\_helper, 743
- der\_encode\_short\_integer
  - der\_encode\_short\_integer.c, 746
- der\_encode\_short\_integer.c
  - der\_encode\_short\_integer, 746
- der\_encode\_utctime
  - der\_encode\_utctime.c, 753
- der\_encode\_utctime.c
  - baseten, 754
  - der\_encode\_utctime, 753
  - STORE\_V, 753
- der\_ia5\_char\_encode
  - der\_length\_ia5\_string.c, 682
- der\_ia5\_value\_decode
  - der\_length\_ia5\_string.c, 682
- der\_length\_bit\_string
  - der\_length\_bit\_string.c, 671

- der\_length\_bit\_string.c
  - der\_length\_bit\_string, 671
- der\_length\_boolean
  - der\_length\_boolean.c, 674
- der\_length\_boolean.c
  - der\_length\_boolean, 674
- der\_length\_ia5\_string
  - der\_length\_ia5\_string.c, 683
- der\_length\_ia5\_string.c
  - code, 684
  - der\_ia5\_char\_encode, 682
  - der\_ia5\_value\_decode, 682
  - der\_length\_ia5\_string, 683
  - ia5\_table, 684
  - value, 684
- der\_length\_integer
  - der\_length\_integer.c, 690
- der\_length\_integer.c
  - der\_length\_integer, 690
- der\_length\_object\_identifier
  - der\_length\_object\_identifier.c, 696
- der\_length\_object\_identifier.c
  - der\_length\_object\_identifier, 696
  - der\_object\_identifier\_bits, 697
- der\_length\_octet\_string
  - der\_length\_octet\_string.c, 702
- der\_length\_octet\_string.c
  - der\_length\_octet\_string, 702
- der\_length\_printable\_string
  - der\_length\_printable\_string.c, 707
- der\_length\_printable\_string.c
  - code, 709
  - der\_length\_printable\_string, 707
  - der\_printable\_char\_encode, 708
  - der\_printable\_value\_decode, 708
  - printable\_table, 709
  - value, 709
- der\_length\_sequence
  - der\_length\_sequence.c, 734
- der\_length\_sequence.c
  - der\_length\_sequence, 734
- der\_length\_short\_integer
  - der\_length\_short\_integer.c, 748
- der\_length\_short\_integer.c
  - der\_length\_short\_integer, 748
- der\_length\_utctime
  - der\_length\_utctime.c, 755
- der\_length\_utctime.c
  - der\_length\_utctime, 755
- der\_object\_identifier\_bits
  - der\_length\_object\_identifier.c, 697
- der\_printable\_char\_encode
  - der\_length\_printable\_string.c, 708
- der\_printable\_value\_decode
  - der\_length\_printable\_string.c, 708
- der\_sequence\_free
  - der\_sequence\_free.c, 737
- der\_sequence\_free.c
  - der\_sequence\_free, 737
- DES
  - tomcrypt\_custom.h, 425
- des.c
  - bigbyte, 112
  - bytebit, 112
  - cookey, 100
  - DE1, 100
  - des3\_desc, 112
  - des3\_done, 101
  - des3\_ecb\_decrypt, 101
  - des3\_ecb\_encrypt, 102
  - des3\_keysize, 102
  - des3\_setup, 103
  - des3\_test, 103
  - des\_desc, 112
  - des\_done, 104
  - des\_ecb\_decrypt, 104
  - des\_ecb\_encrypt, 105
  - des\_fp, 113
  - des\_ip, 113
  - des\_keysize, 105
  - des\_setup, 106
  - des\_test, 106
  - desfunc, 109
  - deskey, 111
  - EN0, 100
  - pc1, 113
  - pc2, 113
  - SP1, 113
  - SP2, 114
  - SP3, 114
  - SP4, 115
  - SP5, 115
  - SP6, 116
  - SP7, 116
  - SP8, 116
  - totrot, 117
- des3\_desc
  - des.c, 112
- des3\_done
  - des.c, 101
- des3\_ecb\_decrypt
  - des.c, 101
- des3\_ecb\_encrypt
  - des.c, 102
- des3\_keysize
  - des.c, 102
- des3\_setup
  - des.c, 103



- des3\_test
  - des.c, [103](#)
- des\_desc
  - des.c, [112](#)
- des\_done
  - des.c, [104](#)
- des\_ecb\_decrypt
  - des.c, [104](#)
- des\_ecb\_encrypt
  - des.c, [105](#)
- des\_fp
  - des.c, [113](#)
- des\_ip
  - des.c, [113](#)
- des\_keysize
  - des.c, [105](#)
- des\_setup
  - des.c, [106](#)
- des\_test
  - des.c, [106](#)
- DESC\_DEF\_ONLY
  - gmp\_desc.c, [550](#)
  - ltm\_desc.c, [551](#)
  - tfm\_desc.c, [556](#)
- desfunc
  - des.c, [109](#)
- deskey
  - des.c, [111](#)
- DEVRANDOM
  - tomcrypt\_custom.h, [425](#)
- div\_2
  - ltc\_math\_descriptor, [32](#)
- done
  - ltc\_cipher\_descriptor, [20](#)
  - ltc\_hash\_descriptor, [26](#)
  - ltc\_prng\_descriptor, [45](#)
- DROUND
  - sober128.c, [899](#)
- dsa\_decrypt\_key
  - dsa\_decrypt\_key.c, [756](#)
- dsa\_decrypt\_key.c
  - dsa\_decrypt\_key, [756](#)
- dsa\_encrypt\_key
  - dsa\_encrypt\_key.c, [759](#)
- dsa\_encrypt\_key.c
  - dsa\_encrypt\_key, [759](#)
- dsa\_export
  - dsa\_export.c, [762](#)
- dsa\_export.c
  - dsa\_export, [762](#)
- dsa\_free
  - dsa\_free.c, [764](#)
- dsa\_free.c
  - dsa\_free, [764](#)
- dsa\_import
  - dsa\_import.c, [765](#)
- dsa\_import.c
  - dsa\_import, [765](#)
- dsa\_make\_key
  - dsa\_make\_key.c, [767](#)
- dsa\_make\_key.c
  - dsa\_make\_key, [767](#)
- dsa\_shared\_secret
  - dsa\_shared\_secret.c, [770](#)
- dsa\_shared\_secret.c
  - dsa\_shared\_secret, [770](#)
- dsa\_sign\_hash
  - dsa\_sign\_hash.c, [772](#)
- dsa\_sign\_hash.c
  - dsa\_sign\_hash, [772](#)
  - dsa\_sign\_hash\_raw, [773](#)
- dsa\_sign\_hash\_raw
  - dsa\_sign\_hash.c, [773](#)
- dsa\_verify\_hash
  - dsa\_verify\_hash.c, [776](#)
- dsa\_verify\_hash.c
  - dsa\_verify\_hash, [776](#)
  - dsa\_verify\_hash\_raw, [777](#)
- dsa\_verify\_hash\_raw
  - dsa\_verify\_hash.c, [777](#)
- dsa\_verify\_key
  - dsa\_verify\_key.c, [779](#)
- dsa\_verify\_key.c
  - dsa\_verify\_key, [779](#)
- dummy
  - Hash\_state, [14](#)
  - Prng\_state, [48](#)
- eax\_addheader
  - eax\_addheader.c, [228](#)
- eax\_addheader.c
  - eax\_addheader, [228](#)
- eax\_decrypt
  - eax\_decrypt.c, [229](#)
- eax\_decrypt.c
  - eax\_decrypt, [229](#)
- eax\_decrypt\_verify\_memory
  - eax\_decrypt\_verify\_memory.c, [230](#)
- eax\_decrypt\_verify\_memory.c
  - eax\_decrypt\_verify\_memory, [230](#)
- eax\_done
  - eax\_done.c, [232](#)
- eax\_done.c
  - eax\_done, [232](#)
- eax\_encrypt
  - eax\_encrypt.c, [234](#)
- eax\_encrypt.c
  - eax\_encrypt, [234](#)

- eax\_encrypt\_authenticate\_memory
  - eax\_encrypt\_authenticate\_memory.c, 235
- eax\_encrypt\_authenticate\_memory.c
  - eax\_encrypt\_authenticate\_memory, 235
- eax\_init
  - eax\_init.c, 237
- eax\_init.c
  - eax\_init, 237
- EAX\_MODE
  - tomcrypt\_custom.h, 425
- eax\_test
  - eax\_test.c, 240
- eax\_test.c
  - eax\_test, 240
- ECB\_DEC
  - aes.c, 52
- ecb\_decrypt
  - ecb\_decrypt.c, 629
  - ltc\_cipher\_descriptor, 21
- ecb\_decrypt.c
  - ecb\_decrypt, 629
- ECB\_DONE
  - aes.c, 52, 55
- ecb\_done
  - ecb\_done.c, 631
- ecb\_done.c
  - ecb\_done, 631
- ECB\_ENC
  - aes.c, 52, 55
- ecb\_encrypt
  - ecb\_encrypt.c, 632
  - ltc\_cipher\_descriptor, 21
- ecb\_encrypt.c
  - ecb\_encrypt, 632
- ECB\_KS
  - aes.c, 52, 58
- ecb\_start
  - ecb\_start.c, 634
- ecb\_start.c
  - ecb\_start, 634
- ECB\_TEST
  - aes.c, 52, 58
- ecc.c
  - ltc\_ecc\_sets, 781
- ECC112
  - tomcrypt\_custom.h, 425
- ECC128
  - tomcrypt\_custom.h, 426
- ECC160
  - tomcrypt\_custom.h, 426
- ECC192
  - tomcrypt\_custom.h, 426
- ECC224
  - tomcrypt\_custom.h, 426
- ECC256
  - tomcrypt\_custom.h, 426
- ECC384
  - tomcrypt\_custom.h, 426
- ECC521
  - tomcrypt\_custom.h, 426
- ecc\_ansi\_x963\_export
  - ecc\_ansi\_x963\_export.c, 782
- ecc\_ansi\_x963\_export.c
  - ecc\_ansi\_x963\_export, 782
- ecc\_ansi\_x963\_import
  - ecc\_ansi\_x963\_import.c, 784
- ecc\_ansi\_x963\_import.c
  - ecc\_ansi\_x963\_import, 784
- ecc\_decrypt\_key
  - ecc\_decrypt\_key.c, 786
- ecc\_decrypt\_key.c
  - ecc\_decrypt\_key, 786
- ecc\_encrypt\_key
  - ecc\_encrypt\_key.c, 789
- ecc\_encrypt\_key.c
  - ecc\_encrypt\_key, 789
- ecc\_export
  - ecc\_export.c, 792
- ecc\_export.c
  - ecc\_export, 792
- ecc\_free
  - ecc\_free.c, 794
- ecc\_free.c
  - ecc\_free, 794
- ecc\_get\_size
  - ecc\_get\_size.c, 795
- ecc\_get\_size.c
  - ecc\_get\_size, 795
- ecc\_import
  - ecc\_import.c, 796
- ecc\_import.c
  - ecc\_import, 796
  - is\_point, 797
- ecc\_make\_key
  - ecc\_make\_key.c, 799
- ecc\_make\_key.c
  - ecc\_make\_key, 799
- ecc\_map
  - ltc\_math\_descriptor, 33
- ecc\_point
  - tomcrypt\_math.h, 450
- ecc\_ptadd
  - ltc\_math\_descriptor, 33
- ecc\_ptdbl
  - ltc\_math\_descriptor, 33
- ecc\_ptmul
  - ltc\_math\_descriptor, 34
- ecc\_shared\_secret

- ecc\_shared\_secret.c, [801](#)
- ecc\_shared\_secret.c
  - ecc\_shared\_secret, [801](#)
- ecc\_sign\_hash
  - ecc\_sign\_hash.c, [803](#)
- ecc\_sign\_hash.c
  - ecc\_sign\_hash, [803](#)
- ecc\_sizes
  - ecc\_sizes.c, [805](#)
- ecc\_sizes.c
  - ecc\_sizes, [805](#)
- ecc\_test
  - ecc\_test.c, [806](#)
- ecc\_test.c
  - ecc\_test, [806](#)
- ecc\_verify\_hash
  - ecc\_verify\_hash.c, [808](#)
- ecc\_verify\_hash.c
  - ecc\_verify\_hash, [808](#)
- edge, [13](#)
  - size, [13](#)
  - start, [13](#)
- ENO
  - des.c, [100](#)
- encauth/ccm/ccm\_memory.c, [219](#)
- encauth/ccm/ccm\_test.c, [225](#)
- encauth/eax/eax\_addheader.c, [228](#)
- encauth/eax/eax\_decrypt.c, [229](#)
- encauth/eax/eax\_decrypt\_verify\_memory.c, [230](#)
- encauth/eax/eax\_done.c, [232](#)
- encauth/eax/eax\_encrypt.c, [234](#)
- encauth/eax/eax\_encrypt\_authenticate\_memory.c, [235](#)
- encauth/eax/eax\_init.c, [237](#)
- encauth/eax/eax\_test.c, [240](#)
- encauth/gcm/gcm\_add\_aad.c, [245](#)
- encauth/gcm/gcm\_add\_iv.c, [248](#)
- encauth/gcm/gcm\_done.c, [250](#)
- encauth/gcm/gcm\_gf\_mult.c, [252](#)
- encauth/gcm/gcm\_init.c, [254](#)
- encauth/gcm/gcm\_memory.c, [256](#)
- encauth/gcm/gcm\_mult\_h.c, [259](#)
- encauth/gcm/gcm\_process.c, [261](#)
- encauth/gcm/gcm\_reset.c, [264](#)
- encauth/gcm/gcm\_test.c, [265](#)
- encauth/ocb/ocb\_decrypt.c, [272](#)
- encauth/ocb/ocb\_decrypt\_verify\_memory.c, [274](#)
- encauth/ocb/ocb\_done\_decrypt.c, [276](#)
- encauth/ocb/ocb\_done\_encrypt.c, [278](#)
- encauth/ocb/ocb\_encrypt.c, [279](#)
- encauth/ocb/ocb\_encrypt\_authenticate\_memory.c, [281](#)
- encauth/ocb/ocb\_init.c, [283](#)
- encauth/ocb/ocb\_ntz.c, [286](#)
- encauth/ocb/ocb\_shift\_xor.c, [287](#)
- encauth/ocb/ocb\_test.c, [288](#)
- encauth/ocb/s\_ocb\_done.c, [292](#)
- ENCRYPT\_ONLY
  - pelican.c, [509](#)
- ENDIAN\_NEUTRAL
  - tomcrypt\_cfg.h, [416](#)
- err\_2\_str
  - error\_to\_string.c, [593](#)
- error\_to\_string
  - error\_to\_string.c, [593](#)
  - tomcrypt\_misc.h, [453](#)
- error\_to\_string.c
  - err\_2\_str, [593](#)
  - error\_to\_string, [593](#)
- EXP
  - safer.c, [171](#)
- export\_size
  - ltc\_prng\_descriptor, [45](#)
- exptmod
  - ltc\_math\_descriptor, [34](#)
- F
  - blowfish.c, [83](#)
  - kseed.c, [135](#)
  - md4.c, [314](#)
  - md5.c, [322](#)
  - rmd128.c, [329](#)
  - rmd160.c, [339](#)
  - rmd256.c, [348](#)
  - rmd320.c, [358](#)
- F0
  - sha1.c, [368](#)
- F1
  - sha1.c, [368](#)
- F2
  - sha1.c, [369](#)
- F3
  - sha1.c, [369](#)
- f8\_decrypt
  - f8\_decrypt.c, [635](#)
- f8\_decrypt.c
  - f8\_decrypt, [635](#)
- f8\_done
  - f8\_done.c, [636](#)
- f8\_done.c
  - f8\_done, [636](#)
- f8\_encrypt
  - f8\_encrypt.c, [637](#)
- f8\_encrypt.c
  - f8\_encrypt, [637](#)
- f8\_getiv
  - f8\_getiv.c, [639](#)
- f8\_getiv.c

- f8\_getiv, [639](#)
- f8\_setiv
  - f8\_setiv.c, [640](#)
- f8\_setiv.c
  - f8\_setiv, [640](#)
- f8\_start
  - f8\_start.c, [641](#)
- f8\_start.c
  - f8\_start, [641](#)
- f8\_test\_mode
  - f8\_test\_mode.c, [643](#)
- f8\_test\_mode.c
  - f8\_test\_mode, [643](#)
- f9\_done
  - f9\_done.c, [463](#)
- f9\_done.c
  - f9\_done, [463](#)
- f9\_file
  - f9\_file.c, [465](#)
- f9\_file.c
  - f9\_file, [465](#)
- f9\_init
  - f9\_init.c, [467](#)
- f9\_init.c
  - f9\_init, [467](#)
- f9\_memory
  - f9\_memory.c, [469](#)
  - ltc\_cipher\_descriptor, [21](#)
- f9\_memory.c
  - f9\_memory, [469](#)
- f9\_memory\_multi
  - f9\_memory\_multi.c, [471](#)
- f9\_memory\_multi.c
  - f9\_memory\_multi, [471](#)
- f9\_process
  - f9\_process.c, [473](#)
- f9\_process.c
  - f9\_process, [473](#)
- f9\_test
  - f9\_test.c, [475](#)
- f9\_test.c
  - f9\_test, [475](#)
- fetch\_length
  - der\_decode\_sequence\_flexi.c, [720](#)
- FF
  - md4.c, [314](#)
  - md5.c, [322](#)
  - rmd128.c, [329](#)
  - rmd160.c, [339](#)
  - rmd256.c, [348](#)
  - rmd320.c, [358](#)
- FF0
  - sha1.c, [369](#)
- FF1
  - sha1.c, [369](#)
- FF2
  - sha1.c, [369](#)
- FF3
  - sha1.c, [369](#)
- FFF
  - rmd128.c, [330](#)
  - rmd160.c, [339](#)
  - rmd256.c, [349](#)
  - rmd320.c, [358](#)
- FI
  - cast5.c, [96](#)
  - kasumi.c, [119](#)
- FII
  - cast5.c, [96](#)
- FIII
  - cast5.c, [96](#)
- find\_cipher
  - crypt\_find\_cipher.c, [567](#)
  - tomcrypt\_cipher.h, [419](#)
- find\_cipher\_any
  - crypt\_find\_cipher\_any.c, [568](#)
  - tomcrypt\_cipher.h, [419](#)
- find\_cipher\_id
  - crypt\_find\_cipher\_id.c, [570](#)
  - tomcrypt\_cipher.h, [420](#)
- find\_hash
  - crypt\_find\_hash.c, [571](#)
  - tomcrypt\_hash.h, [436](#)
- find\_hash\_any
  - crypt\_find\_hash\_any.c, [572](#)
  - tomcrypt\_hash.h, [436](#)
- find\_hash\_id
  - crypt\_find\_hash\_id.c, [574](#)
  - tomcrypt\_hash.h, [437](#)
- find\_hash\_oid
  - crypt\_find\_hash\_oid.c, [575](#)
  - tomcrypt\_hash.h, [438](#)
- find\_prng
  - crypt\_find\_prng.c, [576](#)
  - tomcrypt\_prng.h, [458](#)
- FL
  - kasumi.c, [120](#)
- FO
  - kasumi.c, [120](#)
- FOLD
  - sober128.c, [899](#)
- FOLDP
  - sober128.c, [899](#)
- FORTUNA
  - tomcrypt\_custom.h, [426](#)
- fortuna.c
  - fortuna\_add\_entropy, [878](#)
  - fortuna\_desc, [886](#)

- fortuna\_done, [879](#)
- fortuna\_export, [880](#)
- fortuna\_import, [881](#)
- fortuna\_read, [882](#)
- fortuna\_ready, [883](#)
- fortuna\_reseed, [883](#)
- fortuna\_start, [884](#)
- fortuna\_test, [885](#)
- fortuna\_update\_iv, [886](#)
- fortuna\_add\_entropy
  - fortuna.c, [878](#)
- fortuna\_desc
  - fortuna.c, [886](#)
- fortuna\_done
  - fortuna.c, [879](#)
- fortuna\_export
  - fortuna.c, [880](#)
- fortuna\_import
  - fortuna.c, [881](#)
- FORTUNA\_POOLS
  - tomcrypt\_custom.h, [426](#)
- fortuna\_read
  - fortuna.c, [882](#)
- fortuna\_ready
  - fortuna.c, [883](#)
- fortuna\_reseed
  - fortuna.c, [883](#)
- fortuna\_start
  - fortuna.c, [884](#)
- fortuna\_test
  - fortuna.c, [885](#)
- fortuna\_update\_iv
  - fortuna.c, [886](#)
- FORTUNA\_WD
  - tomcrypt\_custom.h, [426](#)
- four\_rounds
  - pelican.c, [509](#)
- G
  - kseed.c, [135](#)
  - md4.c, [315](#)
  - md5.c, [322](#)
  - rmd128.c, [330](#)
  - rmd160.c, [339](#)
  - rmd256.c, [349](#)
  - rmd320.c, [358](#)
- g1\_func
  - twofish.c, [203](#)
- g\_func
  - skipjack.c, [195](#)
  - twofish.c, [203](#)
- GAMMA
  - noekeon.c, [142](#)
- Gamma0
  - sha256.c, [377](#)
  - sha512.c, [388](#)
- Gamma1
  - sha256.c, [378](#)
  - sha512.c, [388](#)
- GB
  - cast5.c, [91](#)
  - whirl.c, [401](#)
- gcd
  - ltc\_math\_descriptor, [34](#)
- gcm\_add\_aad
  - gcm\_add\_aad.c, [245](#)
- gcm\_add\_aad.c
  - gcm\_add\_aad, [245](#)
- gcm\_add\_iv
  - gcm\_add\_iv.c, [248](#)
- gcm\_add\_iv.c
  - gcm\_add\_iv, [248](#)
- gcm\_done
  - gcm\_done.c, [250](#)
- gcm\_done.c
  - gcm\_done, [250](#)
- gcm\_gf\_mult
  - gcm\_gf\_mult.c, [252](#)
- gcm\_gf\_mult.c
  - gcm\_gf\_mult, [252](#)
  - gcm\_rightshift, [253](#)
  - gcm\_shift\_table, [253](#)
  - mask, [253](#)
  - poly, [253](#)
- gcm\_init
  - gcm\_init.c, [254](#)
- gcm\_init.c
  - gcm\_init, [254](#)
- gcm\_memory
  - gcm\_memory.c, [256](#)
- gcm\_memory.c
  - gcm\_memory, [256](#)
- GCM\_MODE
  - tomcrypt\_custom.h, [426](#)
- gcm\_mult\_h
  - gcm\_mult\_h.c, [259](#)
- gcm\_mult\_h.c
  - gcm\_mult\_h, [259](#)
- gcm\_process
  - gcm\_process.c, [261](#)
- gcm\_process.c
  - gcm\_process, [261](#)
- gcm\_reset
  - gcm\_reset.c, [264](#)
- gcm\_reset.c
  - gcm\_reset, [264](#)
- gcm\_rightshift
  - gcm\_gf\_mult.c, [253](#)

- gcm\_shift\_table
  - gcm\_gf\_mult.c, 253
- GCM\_TABLES
  - tomcrypt\_custom.h, 427
- gcm\_test
  - gcm\_test.c, 265
- gcm\_test.c
  - gcm\_test, 265
- get\_digit
  - ltc\_math\_descriptor, 35
- get\_digit\_count
  - ltc\_math\_descriptor, 35
- get\_int
  - ltc\_math\_descriptor, 35
- gf\_mult
  - twofish.c, 203
- GG
  - md4.c, 315
  - md5.c, 323
  - rmd128.c, 330
  - rmd160.c, 339
  - rmd256.c, 349
  - rmd320.c, 358
- GGG
  - rmd128.c, 330
  - rmd160.c, 339
  - rmd256.c, 349
  - rmd320.c, 358
- gmp\_desc.c
  - DESC\_DEF\_ONLY, 550
- H
  - md4.c, 315
  - md5.c, 323
  - rmd128.c, 330
  - rmd160.c, 339
  - rmd256.c, 349
  - rmd320.c, 358
- h\_func
  - twofish.c, 204
- hash\_descriptor
  - crypt\_hash\_descriptor.c, 579
  - tomcrypt\_hash.h, 444
- hash\_file
  - hash\_file.c, 300
  - tomcrypt\_hash.h, 438
- hash\_file.c
  - hash\_file, 300
- hash\_filehandle
  - hash\_filehandle.c, 302
  - tomcrypt\_hash.h, 439
- hash\_filehandle.c
  - hash\_filehandle, 302
- hash\_is\_valid
  - crypt\_hash\_is\_valid.c, 580
  - tomcrypt\_hash.h, 440
- hash\_memory
  - hash\_memory.c, 304
  - tomcrypt\_hash.h, 440
- hash\_memory.c
  - hash\_memory, 304
- hash\_memory\_multi
  - hash\_memory\_multi.c, 306
  - tomcrypt\_hash.h, 441
- hash\_memory\_multi.c
  - hash\_memory\_multi, 306
- HASH\_PROCESS
  - chc.c, 298
  - tomcrypt\_hash.h, 436
- Hash\_state, 14
  - data, 14
  - dummy, 14
- hash\_state
  - tomcrypt\_hash.h, 436
- hashes/chc/chc.c, 295
- hashes/helper/hash\_file.c, 300
- hashes/helper/hash\_filehandle.c, 302
- hashes/helper/hash\_memory.c, 304
- hashes/helper/hash\_memory\_multi.c, 306
- hashes/md2.c, 308
- hashes/md4.c, 314
- hashes/md5.c, 322
- hashes/rmd128.c, 329
- hashes/rmd160.c, 338
- hashes/rmd256.c, 348
- hashes/rmd320.c, 357
- hashes/sha1.c, 368
- hashes/sha2/sha224.c, 374
- hashes/sha2/sha256.c, 377
- hashes/sha2/sha384.c, 384
- hashes/sha2/sha512.c, 388
- hashes/tiger.c, 394
- hashes/whirl/whirl.c, 401
- hashes/whirl/whirltab.c, 407
- hashsize
  - ltc\_hash\_descriptor, 26
- headers/tomcrypt.h, 410
- headers/tomcrypt\_argchk.h, 414
- headers/tomcrypt\_cfg.h, 416
- headers/tomcrypt\_cipher.h, 418
- headers/tomcrypt\_custom.h, 423
- headers/tomcrypt\_hash.h, 435
- headers/tomcrypt\_mac.h, 445
- headers/tomcrypt\_macros.h, 446
- headers/tomcrypt\_math.h, 449
- headers/tomcrypt\_misc.h, 452
- headers/tomcrypt\_pk.h, 455
- headers/tomcrypt\_pkcs.h, 457

headers/tomcrypt\_prng.h, 458

## HH

- md4.c, 315
- md5.c, 323
- rmd128.c, 330
- rmd160.c, 340
- rmd256.c, 349
- rmd320.c, 359

## HHH

- rmd128.c, 330
- rmd160.c, 340
- rmd256.c, 349
- rmd320.c, 359

## hmac\_block

- ltc\_hash\_descriptor, 26

## HMAC\_BLOCKSIZE

- hmac\_done.c, 477
- hmac\_init.c, 482
- hmac\_test.c, 490

## hmac\_done

- hmac\_done.c, 477

## hmac\_done.c

- HMAC\_BLOCKSIZE, 477
- hmac\_done, 477

## hmac\_file

- hmac\_file.c, 480

## hmac\_file.c

- hmac\_file, 480

## hmac\_init

- hmac\_init.c, 482

## hmac\_init.c

- HMAC\_BLOCKSIZE, 482
- hmac\_init, 482

## hmac\_memory

- hmac\_memory.c, 485

## hmac\_memory.c

- hmac\_memory, 485

## hmac\_memory\_multi

- hmac\_memory\_multi.c, 487

## hmac\_memory\_multi.c

- hmac\_memory\_multi, 487

## hmac\_process

- hmac\_process.c, 489

## hmac\_process.c

- hmac\_process, 489

## hmac\_test

- hmac\_test.c, 490

## hmac\_test.c

- HMAC\_BLOCKSIZE, 490
- hmac\_test, 490

## I

- md5.c, 323
- rmd128.c, 331

- rmd160.c, 340

- rmd256.c, 350

- rmd320.c, 359

## ia5\_table

- der\_length\_ia5\_string.c, 684

## ID

- ltc\_cipher\_descriptor, 22
- ltc\_hash\_descriptor, 26

## ig\_func

- skipjack.c, 196

## II

- md5.c, 323
- rmd128.c, 331
- rmd160.c, 340
- rmd256.c, 350
- rmd320.c, 359

## III

- rmd128.c, 331
- rmd160.c, 340
- rmd256.c, 350
- rmd320.c, 359

## ikestep

- skipjack.c, 200

## iLT

- saferp.c, 184

## in

- chc.c, 299

## init

- ltc\_hash\_descriptor, 26
- ltc\_math\_descriptor, 35

## init\_copy

- ltc\_math\_descriptor, 36

## INITKONST

- sober128.c, 899

## INLINE

- cast5.c, 91
- tiger.c, 394

## invmod

- ltc\_math\_descriptor, 36

## IPHT

- safer.c, 171

## iPHT

- saferp.c, 184

## iROUND

- saferp.c, 184

## is\_point

- ecc\_import.c, 797

## iSHUF

- saferp.c, 184

## isprime

- ltc\_math\_descriptor, 36

## J

- rmd160.c, 340

- rmd320.c, 359
- JJ
  - rmd160.c, 340
  - rmd320.c, 359
- JJJ
  - rmd160.c, 341
  - rmd320.c, 360
- K
  - sha512.c, 393
- kasumi.c
  - FI, 119
  - FL, 120
  - FO, 120
  - kasumi\_desc, 124
  - kasumi\_done, 120
  - kasumi\_ecb\_decrypt, 121
  - kasumi\_ecb\_encrypt, 121
  - kasumi\_keysize, 122
  - kasumi\_setup, 122
  - kasumi\_test, 123
  - ROL16, 118
  - u16, 118
- kasumi\_desc
  - kasumi.c, 124
- kasumi\_done
  - kasumi.c, 120
- kasumi\_ecb\_decrypt
  - kasumi.c, 121
- kasumi\_ecb\_encrypt
  - kasumi.c, 121
- kasumi\_keysize
  - kasumi.c, 122
- kasumi\_setup
  - kasumi.c, 122
- kasumi\_test
  - kasumi.c, 123
- KCi
  - kseed.c, 139
- key\_schedule
  - tiger.c, 395
- KEYP
  - sober128.c, 900
- KEYSIZE
  - khazad.c, 126
- keysize
  - ltc\_cipher\_descriptor, 22
- KEYSIZEB
  - khazad.c, 126
- keystep
  - skipjack.c, 200
- KHAZAD
  - tomcrypt\_custom.h, 427
- khazad.c
  - BLOCKSIZE, 126
  - BLOCKSIZEB, 126
  - c, 132
  - KEYSIZE, 126
  - KEYSIZEB, 126
  - khazad\_crypt, 126
  - khazad\_desc, 132
  - khazad\_done, 127
  - khazad\_ecb\_decrypt, 127
  - khazad\_ecb\_encrypt, 128
  - khazad\_keysize, 128
  - khazad\_setup, 129
  - khazad\_test, 131
  - R, 126
  - T0, 132
  - T1, 132
  - T2, 133
  - T3, 133
  - T4, 133
  - T5, 133
  - T6, 133
  - T7, 133
  - khazad\_crypt
    - khazad.c, 126
  - khazad\_desc
    - khazad.c, 132
  - khazad\_done
    - khazad.c, 127
  - khazad\_ecb\_decrypt
    - khazad.c, 127
  - khazad\_ecb\_encrypt
    - khazad.c, 128
  - khazad\_keysize
    - khazad.c, 128
  - khazad\_setup
    - khazad.c, 129
  - khazad\_test
    - khazad.c, 131
- KSEED
  - tomcrypt\_custom.h, 427
- kseed.c
  - F, 135
  - G, 135
  - KCi, 139
  - kseed\_desc, 139
  - kseed\_done, 135
  - kseed\_ecb\_decrypt, 135
  - kseed\_ecb\_encrypt, 136
  - kseed\_keysize, 136
  - kseed\_setup, 137
  - kseed\_test, 138
  - rounds, 139
  - SS0, 140
  - SS1, 140



- SS2, [140](#)
- SS3, [140](#)
- kseed\_desc
  - kseed.c, [139](#)
- kseed\_done
  - kseed.c, [135](#)
- kseed\_ecb\_decrypt
  - kseed.c, [135](#)
- kseed\_ecb\_encrypt
  - kseed.c, [136](#)
- kseed\_keysize
  - kseed.c, [136](#)
- kseed\_setup
  - kseed.c, [137](#)
- kseed\_test
  - kseed.c, [138](#)
- kTHETA
  - noekeon.c, [142](#)
- lcm
  - ltc\_math\_descriptor, [36](#)
- len
  - ocb\_init.c, [285](#)
  - pmac\_init.c, [523](#)
- LOG
  - safer.c, [172](#)
- lrw\_decrypt
  - lrw\_decrypt.c, [645](#)
- lrw\_decrypt.c
  - lrw\_decrypt, [645](#)
- lrw\_done
  - lrw\_done.c, [646](#)
- lrw\_done.c
  - lrw\_done, [646](#)
- lrw\_encrypt
  - lrw\_encrypt.c, [647](#)
- lrw\_encrypt.c
  - lrw\_encrypt, [647](#)
- lrw\_getiv
  - lrw\_getiv.c, [648](#)
- lrw\_getiv.c
  - lrw\_getiv, [648](#)
- lrw\_process
  - lrw\_process.c, [649](#)
- lrw\_process.c
  - lrw\_process, [649](#)
- lrw\_setiv
  - lrw\_setiv.c, [652](#)
- lrw\_setiv.c
  - lrw\_setiv, [652](#)
- lrw\_start
  - lrw\_start.c, [654](#)
- lrw\_start.c
  - lrw\_start, [654](#)
- LRW\_TABLES
  - tomcrypt\_custom.h, [427](#)
- lrw\_test
  - lrw\_test.c, [656](#)
- lrw\_test.c
  - lrw\_test, [656](#)
- LT
  - saferp.c, [185](#)
- LTC\_ARGCHK
  - tomcrypt\_argchk.h, [414](#)
- LTC\_ARGCHKVD
  - tomcrypt\_argchk.h, [415](#)
- LTC\_CBC\_MODE
  - tomcrypt\_custom.h, [427](#)
- LTC\_CFB\_MODE
  - tomcrypt\_custom.h, [427](#)
- ltc\_cipher\_descriptor, [15](#)
  - accel\_cbc\_decrypt, [17](#)
  - accel\_cbc\_encrypt, [17](#)
  - accel\_ccm\_memory, [17](#)
  - accel\_ctr\_encrypt, [18](#)
  - accel\_ecb\_decrypt, [18](#)
  - accel\_ecb\_encrypt, [18](#)
  - accel\_gcm\_memory, [19](#)
  - accel\_lrw\_decrypt, [19](#)
  - accel\_lrw\_encrypt, [20](#)
  - block\_length, [20](#)
  - default\_rounds, [20](#)
  - done, [20](#)
  - ecb\_decrypt, [21](#)
  - ecb\_encrypt, [21](#)
  - f9\_memory, [21](#)
  - ID, [22](#)
  - keysize, [22](#)
  - max\_key\_length, [22](#)
  - min\_key\_length, [22](#)
  - name, [22](#)
  - omac\_memory, [22](#)
  - setup, [23](#)
  - test, [23](#)
  - xcbc\_memory, [23](#)
- LTC\_CTR\_MODE
  - tomcrypt\_custom.h, [427](#)
- ltc\_deinit\_multi
  - multi.c, [552](#)
  - tomcrypt\_math.h, [450](#)
- LTC\_DER
  - tomcrypt\_custom.h, [427](#)
- LTC\_ECB\_MODE
  - tomcrypt\_custom.h, [427](#)
- ltc\_ecc\_del\_point
  - ltc\_ecc\_points.c, [819](#)
- ltc\_ecc\_is\_valid\_idx
  - ltc\_ecc\_is\_valid\_idx.c, [811](#)

- ltc\_ecc\_is\_valid\_idx.c
  - ltc\_ecc\_is\_valid\_idx, 811
- ltc\_ecc\_map
  - ltc\_ecc\_map.c, 812
- ltc\_ecc\_map.c
  - ltc\_ecc\_map, 812
- ltc\_ecc\_mulmod
  - ltc\_ecc\_mulmod.c, 814
- ltc\_ecc\_mulmod.c
  - ltc\_ecc\_mulmod, 814
  - WINSIZE, 814
- ltc\_ecc\_new\_point
  - ltc\_ecc\_points.c, 819
- ltc\_ecc\_points.c
  - ltc\_ecc\_del\_point, 819
  - ltc\_ecc\_new\_point, 819
- ltc\_ecc\_projective\_add\_point
  - ltc\_ecc\_projective\_add\_point.c, 821
- ltc\_ecc\_projective\_add\_point.c
  - ltc\_ecc\_projective\_add\_point, 821
- ltc\_ecc\_projective\_dbl\_point
  - ltc\_ecc\_projective\_dbl\_point.c, 825
- ltc\_ecc\_projective\_dbl\_point.c
  - ltc\_ecc\_projective\_dbl\_point, 825
- ltc\_ecc\_sets
  - ecc.c, 781
- LTC\_F8\_MODE
  - tomcrypt\_custom.h, 427
- LTC\_F9\_MODE
  - tomcrypt\_custom.h, 427
- ltc\_hash\_descriptor, 25
  - blocksize, 26
  - done, 26
  - hashsize, 26
  - hmac\_block, 26
  - ID, 26
  - init, 26
  - name, 26
  - OID, 27
  - OIDlen, 27
  - process, 27
  - test, 27
- LTC\_HMAC
  - tomcrypt\_custom.h, 428
- ltc\_init\_multi
  - multi.c, 552
  - tomcrypt\_math.h, 450
- LTC\_KASUMI
  - tomcrypt\_custom.h, 428
- LTC\_LRW\_MODE
  - tomcrypt\_custom.h, 428
- ltc\_math\_descriptor, 28
  - add, 31
  - addi, 31
  - bits\_per\_digit, 31
  - compare, 31
  - compare\_d, 31
  - copy, 32
  - count\_bits, 32
  - count\_lsb\_bits, 32
  - deinit, 32
  - div\_2, 32
  - ecc\_map, 33
  - ecc\_ptadd, 33
  - ecc\_ptdbl, 33
  - ecc\_ptmul, 34
  - exptmod, 34
  - gcd, 34
  - get\_digit, 35
  - get\_digit\_count, 35
  - get\_int, 35
  - init, 35
  - init\_copy, 36
  - invmod, 36
  - isprime, 36
  - lcm, 36
  - modi, 37
  - montgomery\_deinit, 37
  - montgomery\_normalization, 37
  - montgomery\_reduce, 37
  - montgomery\_setup, 38
  - mpdiv, 38
  - mul, 38
  - muli, 38
  - mulmod, 39
  - name, 39
  - neg, 39
  - read\_radix, 39
  - rsa\_keygen, 40
  - rsa\_me, 40
  - set\_int, 40
  - sqr, 41
  - sqrmod, 41
  - sub, 41
  - subi, 41
  - twoexpt, 42
  - unsigned\_read, 42
  - unsigned\_size, 42
  - unsigned\_write, 42
  - write\_radix, 43
- ltc\_mp
  - crypt\_ltc\_mp\_descriptor.c, 581
  - tomcrypt\_math.h, 451
- LTC\_MP\_EQ
  - tomcrypt\_math.h, 449
- LTC\_MP\_GT
  - tomcrypt\_math.h, 449
- LTC\_MP\_LT

- tomcrypt\_math.h, 449
- LTC\_MP\_NO
  - tomcrypt\_math.h, 450
- LTC\_MP\_YES
  - tomcrypt\_math.h, 450
- LTC\_MUTEX\_GLOBAL
  - tomcrypt\_custom.h, 428
- LTC\_MUTEX\_INIT
  - tomcrypt\_custom.h, 428
- LTC\_MUTEX\_LOCK
  - tomcrypt\_custom.h, 428
- LTC\_MUTEX\_PROTO
  - tomcrypt\_custom.h, 428
- LTC\_MUTEX\_TYPE
  - tomcrypt\_custom.h, 428
- LTC\_MUTEX\_UNLOCK
  - tomcrypt\_custom.h, 428
- LTC\_OFB\_MODE
  - tomcrypt\_custom.h, 429
- LTC\_OMAC
  - tomcrypt\_custom.h, 429
- LTC\_PMAC
  - tomcrypt\_custom.h, 429
- ltc\_prng\_descriptor, 44
  - add\_entropy, 44
  - done, 45
  - export\_size, 45
  - name, 45
  - pexport, 45
  - pimport, 45
  - read, 46
  - ready, 46
  - start, 46
  - test, 46
- LTC\_TEST
  - tomcrypt\_custom.h, 429
- ltc\_to\_asn1
  - der\_encode\_set.c, 739
- LTC\_XCBC
  - tomcrypt\_custom.h, 429
- ltm\_desc.c
  - DESC\_DEF\_ONLY, 551
- mac/f9/f9\_done.c, 463
- mac/f9/f9\_file.c, 465
- mac/f9/f9\_init.c, 467
- mac/f9/f9\_memory.c, 469
- mac/f9/f9\_memory\_multi.c, 471
- mac/f9/f9\_process.c, 473
- mac/f9/f9\_test.c, 475
- mac/hmac/hmac\_done.c, 477
- mac/hmac/hmac\_file.c, 480
- mac/hmac/hmac\_init.c, 482
- mac/hmac/hmac\_memory.c, 485
- mac/hmac/hmac\_memory\_multi.c, 487
- mac/hmac/hmac\_process.c, 489
- mac/hmac/hmac\_test.c, 490
- mac/omac/omac\_done.c, 495
- mac/omac/omac\_file.c, 497
- mac/omac/omac\_init.c, 499
- mac/omac/omac\_memory.c, 501
- mac/omac/omac\_memory\_multi.c, 503
- mac/omac/omac\_process.c, 505
- mac/omac/omac\_test.c, 507
- mac/pelican/pelican.c, 509
- mac/pelican/pelican\_memory.c, 513
- mac/pelican/pelican\_test.c, 515
- mac/pmac/pmac\_done.c, 517
- mac/pmac/pmac\_file.c, 519
- mac/pmac/pmac\_init.c, 521
- mac/pmac/pmac\_memory.c, 524
- mac/pmac/pmac\_memory\_multi.c, 526
- mac/pmac/pmac\_ntz.c, 528
- mac/pmac/pmac\_process.c, 529
- mac/pmac/pmac\_shift\_xor.c, 531
- mac/pmac/pmac\_test.c, 532
- mac/xcbc/xcbc\_done.c, 535
- mac/xcbc/xcbc\_file.c, 537
- mac/xcbc/xcbc\_init.c, 539
- mac/xcbc/xcbc\_memory.c, 541
- mac/xcbc/xcbc\_memory\_multi.c, 543
- mac/xcbc/xcbc\_process.c, 545
- mac/xcbc/xcbc\_test.c, 547
- Maj
  - sha256.c, 378
  - sha512.c, 389
- map
  - base64\_decode.c, 558
- mask
  - gcm\_gf\_mult.c, 253
- math/fp/ltc\_ecc\_fp\_mulmod.c, 549
- math/gmp\_desc.c, 550
- math/ltm\_desc.c, 551
- math/multi.c, 552
- math/rand\_prime.c, 554
- math/tfm\_desc.c, 556
- MAX
  - tomcrypt\_macros.h, 447
- max\_key\_length
  - ltc\_cipher\_descriptor, 22
- MAX\_KEYSIZEB
  - anubis.c, 69
- MAX\_N
  - anubis.c, 69
- MAX\_ROUNDS
  - anubis.c, 69
- MAXBLOCKSIZE
  - tomcrypt.h, 411

- MD2
  - tomcrypt\_custom.h, 429
- md2.c
  - md2\_compress, 308
  - md2\_desc, 312
  - md2\_done, 308
  - md2\_init, 309
  - md2\_process, 310
  - md2\_test, 311
  - md2\_update\_chksum, 312
  - PI\_SUBST, 313
- md2\_compress
  - md2.c, 308
- md2\_desc
  - md2.c, 312
- md2\_done
  - md2.c, 308
- md2\_init
  - md2.c, 309
- md2\_process
  - md2.c, 310
- md2\_test
  - md2.c, 311
- md2\_update\_chksum
  - md2.c, 312
- MD4
  - tomcrypt\_custom.h, 429
- md4.c
  - F, 314
  - FF, 314
  - G, 315
  - GG, 315
  - H, 315
  - HH, 315
  - md4\_compress, 317
  - md4\_desc, 320
  - md4\_done, 318
  - md4\_init, 319
  - md4\_test, 319
  - ROTATE\_LEFT, 315
  - S11, 315
  - S12, 315
  - S13, 316
  - S14, 316
  - S21, 316
  - S22, 316
  - S23, 316
  - S24, 316
  - S31, 316
  - S32, 316
  - S33, 316
  - S34, 316
- md4\_compress
  - md4.c, 317
- md4\_desc
  - md4.c, 320
- md4\_done
  - md4.c, 318
- md4\_init
  - md4.c, 319
- md4\_test
  - md4.c, 319
- MD5
  - tomcrypt\_custom.h, 429
- md5.c
  - F, 322
  - FF, 322
  - G, 322
  - GG, 323
  - H, 323
  - HH, 323
  - I, 323
  - II, 323
  - md5\_compress, 323
  - md5\_desc, 328
  - md5\_done, 325
  - md5\_init, 326
  - md5\_test, 326
- md5\_compress
  - md5.c, 323
- md5\_desc
  - md5.c, 328
- md5\_done
  - md5.c, 325
- md5\_init
  - md5.c, 326
- md5\_test
  - md5.c, 326
- MDS
  - twofish.c, 211
- mds\_column\_mult
  - twofish.c, 203
- mds\_mult
  - twofish.c, 204
- MDS\_POLY
  - twofish.c, 203
- MDSA
  - tomcrypt\_custom.h, 429
- MECC
  - tomcrypt\_custom.h, 429
- MIN
  - tomcrypt\_macros.h, 447
- min\_key\_length
  - ltc\_cipher\_descriptor, 22
- MIN\_KEYSIZEB
  - anubis.c, 69
- MIN\_N
  - anubis.c, 69

## MIN\_ROUNDS

- anubis.c, 69
- misc/base64/base64\_decode.c, 557
- misc/base64/base64\_encode.c, 560
- misc/burn\_stack.c, 562
- misc/crypt/crypt.c, 563
- misc/crypt/crypt\_argchk.c, 564
- misc/crypt/crypt\_cipher\_descriptor.c, 565
- misc/crypt/crypt\_cipher\_is\_valid.c, 566
- misc/crypt/crypt\_find\_cipher.c, 567
- misc/crypt/crypt\_find\_cipher\_any.c, 568
- misc/crypt/crypt\_find\_cipher\_id.c, 570
- misc/crypt/crypt\_find\_hash.c, 571
- misc/crypt/crypt\_find\_hash\_any.c, 572
- misc/crypt/crypt\_find\_hash\_id.c, 574
- misc/crypt/crypt\_find\_hash\_oid.c, 575
- misc/crypt/crypt\_find\_prng.c, 576
- misc/crypt/crypt\_fsa.c, 577
- misc/crypt/crypt\_hash\_descriptor.c, 579
- misc/crypt/crypt\_hash\_is\_valid.c, 580
- misc/crypt/crypt\_ltc\_mp\_descriptor.c, 581
- misc/crypt/crypt\_prng\_descriptor.c, 582
- misc/crypt/crypt\_prng\_is\_valid.c, 583
- misc/crypt/crypt\_register\_cipher.c, 584
- misc/crypt/crypt\_register\_hash.c, 586
- misc/crypt/crypt\_register\_prng.c, 588
- misc/crypt/crypt\_unregister\_cipher.c, 590
- misc/crypt/crypt\_unregister\_hash.c, 591
- misc/crypt/crypt\_unregister\_prng.c, 592
- misc/error\_to\_string.c, 593
- misc/pkcs5/pkcs\_5\_1.c, 594
- misc/pkcs5/pkcs\_5\_2.c, 596
- misc/zeromem.c, 599
- modes/cbc/cbc\_decrypt.c, 600
- modes/cbc/cbc\_done.c, 602
- modes/cbc/cbc\_encrypt.c, 603
- modes/cbc/cbc\_getiv.c, 605
- modes/cbc/cbc\_setiv.c, 606
- modes/cbc/cbc\_start.c, 607
- modes/cfb/cfb\_decrypt.c, 609
- modes/cfb/cfb\_done.c, 611
- modes/cfb/cfb\_encrypt.c, 612
- modes/cfb/cfb\_getiv.c, 614
- modes/cfb/cfb\_setiv.c, 615
- modes/cfb/cfb\_start.c, 616
- modes/ctr/ctr\_decrypt.c, 618
- modes/ctr/ctr\_done.c, 619
- modes/ctr/ctr\_encrypt.c, 620
- modes/ctr/ctr\_getiv.c, 622
- modes/ctr/ctr\_setiv.c, 623
- modes/ctr/ctr\_start.c, 625
- modes/ctr/ctr\_test.c, 627
- modes/ecb/ecb\_decrypt.c, 629
- modes/ecb/ecb\_done.c, 631
- modes/ecb/ecb\_encrypt.c, 632
- modes/ecb/ecb\_start.c, 634
- modes/f8/f8\_decrypt.c, 635
- modes/f8/f8\_done.c, 636
- modes/f8/f8\_encrypt.c, 637
- modes/f8/f8\_getiv.c, 639
- modes/f8/f8\_setiv.c, 640
- modes/f8/f8\_start.c, 641
- modes/f8/f8\_test\_mode.c, 643
- modes/lrw/lrw\_decrypt.c, 645
- modes/lrw/lrw\_done.c, 646
- modes/lrw/lrw\_encrypt.c, 647
- modes/lrw/lrw\_getiv.c, 648
- modes/lrw/lrw\_process.c, 649
- modes/lrw/lrw\_setiv.c, 652
- modes/lrw/lrw\_start.c, 654
- modes/lrw/lrw\_test.c, 656
- modes/ofb/ofb\_decrypt.c, 659
- modes/ofb/ofb\_done.c, 660
- modes/ofb/ofb\_encrypt.c, 661
- modes/ofb/ofb\_getiv.c, 663
- modes/ofb/ofb\_setiv.c, 664
- modes/ofb/ofb\_start.c, 665
- modi
  - ltc\_math\_descriptor, 37
- montgomery\_deinit
  - ltc\_math\_descriptor, 37
- montgomery\_normalization
  - ltc\_math\_descriptor, 37
- montgomery\_reduce
  - ltc\_math\_descriptor, 37
- montgomery\_setup
  - ltc\_math\_descriptor, 38
- mpdiv
  - ltc\_math\_descriptor, 38
- MPI
  - tomcrypt\_custom.h, 429
- MRSA
  - tomcrypt\_custom.h, 430
- mul
  - ltc\_math\_descriptor, 38
- muli
  - ltc\_math\_descriptor, 38
- mulmod
  - ltc\_math\_descriptor, 39
- Multab
  - sober128tab.c, 911
- multi.c
  - ltc\_deinit\_multi, 552
  - ltc\_init\_multi, 552
- N
  - sober128.c, 900
- name

- ltc\_cipher\_descriptor, 22
- ltc\_hash\_descriptor, 26
- ltc\_math\_descriptor, 39
- ltc\_prng\_descriptor, 45
- neg
  - ltc\_math\_descriptor, 39
- NLFUNC
  - sober128.c, 900
- nltap
  - sober128.c, 901
- NOEKEON
  - tomcrypt\_custom.h, 430
- noekeon.c
  - GAMMA, 142
  - kTHETA, 142
  - noekeon\_desc, 147
  - noekeon\_done, 143
  - noekeon\_ecb\_decrypt, 143
  - noekeon\_ecb\_encrypt, 144
  - noekeon\_keysize, 145
  - noekeon\_setup, 145
  - noekeon\_test, 146
  - PI1, 142
  - PI2, 142
  - RC, 148
  - ROUND, 142
  - THETA, 143
- noekeon\_desc
  - noekeon.c, 147
- noekeon\_done
  - noekeon.c, 143
- noekeon\_ecb\_decrypt
  - noekeon.c, 143
- noekeon\_ecb\_encrypt
  - noekeon.c, 144
- noekeon\_keysize
  - noekeon.c, 145
- noekeon\_setup
  - noekeon.c, 145
- noekeon\_test
  - noekeon.c, 146
- ocb\_decrypt
  - ocb\_decrypt.c, 272
- ocb\_decrypt.c
  - ocb\_decrypt, 272
- ocb\_decrypt\_verify\_memory
  - ocb\_decrypt\_verify\_memory.c, 274
- ocb\_decrypt\_verify\_memory.c
  - ocb\_decrypt\_verify\_memory, 274
- ocb\_done\_decrypt
  - ocb\_done\_decrypt.c, 276
- ocb\_done\_decrypt.c
  - ocb\_done\_decrypt, 276
- ocb\_done\_encrypt
  - ocb\_done\_encrypt.c, 278
- ocb\_done\_encrypt.c, 278
- ocb\_encrypt
  - ocb\_encrypt.c, 279
- ocb\_encrypt.c
  - ocb\_encrypt, 279
- ocb\_encrypt\_authenticate\_memory
  - ocb\_encrypt\_authenticate\_memory.c, 281
- ocb\_encrypt\_authenticate\_memory.c
  - ocb\_encrypt\_authenticate\_memory, 281
- ocb\_init
  - ocb\_init.c, 283
- ocb\_init.c
  - len, 285
  - ocb\_init, 283
  - poly\_div, 285
  - poly\_mul, 285
  - polys, 285
- OCB\_MODE
  - tomcrypt\_custom.h, 430
- ocb\_ntz
  - ocb\_ntz.c, 286
- ocb\_ntz.c
  - ocb\_ntz, 286
- ocb\_shift\_xor
  - ocb\_shift\_xor.c, 287
- ocb\_shift\_xor.c
  - ocb\_shift\_xor, 287
- ocb\_test
  - ocb\_test.c, 288
- ocb\_test.c
  - ocb\_test, 288
- ofb\_decrypt
  - ofb\_decrypt.c, 659
- ofb\_decrypt.c
  - ofb\_decrypt, 659
- ofb\_done
  - ofb\_done.c, 660
- ofb\_done.c
  - ofb\_done, 660
- ofb\_encrypt
  - ofb\_encrypt.c, 661
- ofb\_encrypt.c
  - ofb\_encrypt, 661
- ofb\_getiv
  - ofb\_getiv.c, 663
- ofb\_getiv.c
  - ofb\_getiv, 663
- ofb\_setiv
  - ofb\_setiv.c, 664
- ofb\_setiv.c
  - ofb\_setiv, 664

- ofb\_start
  - ofb\_start.c, 665
- ofb\_start.c
  - ofb\_start, 665
- OFF
  - sober128.c, 900
- OID
  - ltc\_hash\_descriptor, 27
- OIDlen
  - ltc\_hash\_descriptor, 27
- omac\_done
  - omac\_done.c, 495
- omac\_done.c
  - omac\_done, 495
- omac\_file
  - omac\_file.c, 497
- omac\_file.c
  - omac\_file, 497
- omac\_init
  - omac\_init.c, 499
- omac\_init.c
  - omac\_init, 499
- omac\_memory
  - ltc\_cipher\_descriptor, 22
  - omac\_memory.c, 501
- omac\_memory.c
  - omac\_memory, 501
- omac\_memory\_multi
  - omac\_memory\_multi.c, 503
- omac\_memory\_multi.c
  - omac\_memory\_multi, 503
- omac\_process
  - omac\_process.c, 505
- omac\_process.c
  - omac\_process, 505
- omac\_test
  - omac\_test.c, 507
- omac\_test.c
  - omac\_test, 507
- ORIG\_P
  - blowfish.c, 89
- ORIG\_S
  - blowfish.c, 89
- pass
  - tiger.c, 395
- pc1
  - des.c, 113
- pc2
  - des.c, 113
- PELI\_TAB
  - pelican.c, 509
- PELICAN
  - tomcrypt\_custom.h, 430
- pelican.c
  - ENCRYPT\_ONLY, 509
  - four\_rounds, 509
  - PELI\_TAB, 509
  - pelican\_done, 510
  - pelican\_init, 511
  - pelican\_process, 511
- pelican\_done
  - pelican.c, 510
- pelican\_init
  - pelican.c, 511
- pelican\_memory
  - pelican\_memory.c, 513
- pelican\_memory.c
  - pelican\_memory, 513
- pelican\_process
  - pelican.c, 511
- pelican\_test
  - pelican\_test.c, 515
- pelican\_test.c
  - pelican\_test, 515
- permute
  - rc2.c, 154
- pexport
  - ltc\_prng\_descriptor, 45
- PHT
  - safer.c, 172
  - saferp.c, 185
- PI1
  - noekeon.c, 142
- PI2
  - noekeon.c, 142
- PI\_SUBST
  - md2.c, 313
- pimport
  - ltc\_prng\_descriptor, 45
- pk/asn1/der/bit/der\_decode\_bit\_string.c, 667
- pk/asn1/der/bit/der\_encode\_bit\_string.c, 669
- pk/asn1/der/bit/der\_length\_bit\_string.c, 671
- pk/asn1/der/boolean/der\_decode\_boolean.c, 672
- pk/asn1/der/boolean/der\_encode\_boolean.c, 673
- pk/asn1/der/boolean/der\_length\_boolean.c, 674
- pk/asn1/der/choice/der\_decode\_choice.c, 675
- pk/asn1/der/ia5/der\_decode\_ia5\_string.c, 678
- pk/asn1/der/ia5/der\_encode\_ia5\_string.c, 680
- pk/asn1/der/ia5/der\_length\_ia5\_string.c, 682
- pk/asn1/der/integer/der\_decode\_integer.c, 685
- pk/asn1/der/integer/der\_encode\_integer.c, 687
- pk/asn1/der/integer/der\_length\_integer.c, 690
- pk/asn1/der/object\_identifier/der\_decode\_object\_identifier.c, 692
- pk/asn1/der/object\_identifier/der\_encode\_object\_identifier.c, 694

- pk/asn1/der/object\_identifier/der\_length\_object\_Identifier.c, 696
- pk/asn1/der/octet/der\_decode\_octet\_string.c, 698
- pk/asn1/der/octet/der\_encode\_octet\_string.c, 700
- pk/asn1/der/octet/der\_length\_octet\_string.c, 702
- pk/asn1/der/printable\_string/der\_decode\_printable\_string.c, 703
- pk/asn1/der/printable\_string/der\_encode\_printable\_string.c, 705
- pk/asn1/der/printable\_string/der\_length\_printable\_string.c, 707
- pk/asn1/der/sequence/der\_decode\_sequence\_ex.c, 710
- pk/asn1/der/sequence/der\_decode\_sequence\_flexi.c, 715
- pk/asn1/der/sequence/der\_decode\_sequence\_multi.c, 722
- pk/asn1/der/sequence/der\_encode\_sequence\_ex.c, 725
- pk/asn1/der/sequence/der\_encode\_sequence\_multi.c, 731
- pk/asn1/der/sequence/der\_length\_sequence.c, 734
- pk/asn1/der/sequence/der\_sequence\_free.c, 737
- pk/asn1/der/set/der\_encode\_set.c, 739
- pk/asn1/der/set/der\_encode\_setof.c, 741
- pk/asn1/der/short\_integer/der\_decode\_short\_integer.c, 744
- pk/asn1/der/short\_integer/der\_encode\_short\_integer.c, 746
- pk/asn1/der/short\_integer/der\_length\_short\_integer.c, 748
- pk/asn1/der/utctime/der\_decode\_utctime.c, 750
- pk/asn1/der/utctime/der\_encode\_utctime.c, 753
- pk/asn1/der/utctime/der\_length\_utctime.c, 755
- pk/dsa/dsa\_decrypt\_key.c, 756
- pk/dsa/dsa\_encrypt\_key.c, 759
- pk/dsa/dsa\_export.c, 762
- pk/dsa/dsa\_free.c, 764
- pk/dsa/dsa\_import.c, 765
- pk/dsa/dsa\_make\_key.c, 767
- pk/dsa/dsa\_shared\_secret.c, 770
- pk/dsa/dsa\_sign\_hash.c, 772
- pk/dsa/dsa\_verify\_hash.c, 776
- pk/dsa/dsa\_verify\_key.c, 779
- pk/ecc/ecc.c, 781
- pk/ecc/ecc\_ansi\_x963\_export.c, 782
- pk/ecc/ecc\_ansi\_x963\_import.c, 784
- pk/ecc/ecc\_decrypt\_key.c, 786
- pk/ecc/ecc\_encrypt\_key.c, 789
- pk/ecc/ecc\_export.c, 792
- pk/ecc/ecc\_free.c, 794
- pk/ecc/ecc\_get\_size.c, 795
- pk/ecc/ecc\_import.c, 796
- pk/ecc/ecc\_make\_key.c, 799
- pk/ecc/ecc\_shared\_secret.c, 801
- pk/ecc/ecc\_sign\_hash.c, 803
- pk/ecc/ecc\_sizes.c, 805
- pk/ecc/ecc\_test.c, 806
- pk/ecc/ecc\_verify\_hash.c, 808
- pk/ecc/lte\_ecc\_is\_valid\_idx.c, 811
- pk/ecc/lte\_ecc\_map.c, 812
- pk/ecc/lte\_ecc\_mulmod.c, 814
- pk/ecc/lte\_ecc\_mulmod\_timing.c, 818
- pk/ecc/lte\_ecc\_points.c, 819
- pk/ecc/lte\_ecc\_projective\_add\_point.c, 821
- pk/ecc/lte\_ecc\_projective\_dbl\_point.c, 825
- pk/katja/katja\_decrypt\_key.c, 828
- pk/katja/katja\_encrypt\_key.c, 829
- pk/katja/katja\_export.c, 830
- pk/katja/katja\_exptmod.c, 831
- pk/katja/katja\_free.c, 832
- pk/katja/katja\_import.c, 833
- pk/katja/katja\_make\_key.c, 834
- pk/pkcs1/pkcs\_1\_i2osp.c, 835
- pk/pkcs1/pkcs\_1\_mgf1.c, 836
- pk/pkcs1/pkcs\_1\_oaep\_decode.c, 838
- pk/pkcs1/pkcs\_1\_oaep\_encode.c, 842
- pk/pkcs1/pkcs\_1\_os2ip.c, 845
- pk/pkcs1/pkcs\_1\_pss\_decode.c, 846
- pk/pkcs1/pkcs\_1\_pss\_encode.c, 850
- pk/pkcs1/pkcs\_1\_v1\_5\_decode.c, 853
- pk/pkcs1/pkcs\_1\_v1\_5\_encode.c, 855
- pk/rsa/rsa\_decrypt\_key.c, 857
- pk/rsa/rsa\_encrypt\_key.c, 859
- pk/rsa/rsa\_export.c, 861
- pk/rsa/rsa\_exptmod.c, 863
- pk/rsa/rsa\_free.c, 866
- pk/rsa/rsa\_import.c, 867
- pk/rsa/rsa\_make\_key.c, 870
- pk/rsa/rsa\_sign\_hash.c, 872
- pk/rsa/rsa\_verify\_hash.c, 875
- PK\_PRIVATE
  - tomcrypt\_pk.h, 455
- PK\_PUBLIC
  - tomcrypt\_pk.h, 455
- PKCS\_1
  - tomcrypt\_custom.h, 430
- pkcs\_1\_i2osp
  - pkcs\_1\_i2osp.c, 835
- pkcs\_1\_i2osp.c
  - pkcs\_1\_i2osp, 835
- pkcs\_1\_mgf1
  - pkcs\_1\_mgf1.c, 836
- pkcs\_1\_mgf1.c
  - pkcs\_1\_mgf1, 836
- pkcs\_1\_oaep\_decode
  - pkcs\_1\_oaep\_decode.c, 838
- pkcs\_1\_oaep\_decode.c



- pkcs\_1\_oaep\_decode, 838
- pkcs\_1\_oaep\_encode
  - pkcs\_1\_oaep\_encode.c, 842
- pkcs\_1\_oaep\_encode.c
  - pkcs\_1\_oaep\_encode, 842
- pkcs\_1\_os2ip
  - pkcs\_1\_os2ip.c, 845
- pkcs\_1\_os2ip.c
  - pkcs\_1\_os2ip, 845
- pkcs\_1\_pss\_decode
  - pkcs\_1\_pss\_decode.c, 846
- pkcs\_1\_pss\_decode.c
  - pkcs\_1\_pss\_decode, 846
- pkcs\_1\_pss\_encode
  - pkcs\_1\_pss\_encode.c, 850
- pkcs\_1\_pss\_encode.c
  - pkcs\_1\_pss\_encode, 850
- pkcs\_1\_v1\_5\_decode
  - pkcs\_1\_v1\_5\_decode.c, 853
- pkcs\_1\_v1\_5\_decode.c
  - pkcs\_1\_v1\_5\_decode, 853
- pkcs\_1\_v1\_5\_encode
  - pkcs\_1\_v1\_5\_encode.c, 855
- pkcs\_1\_v1\_5\_encode.c
  - pkcs\_1\_v1\_5\_encode, 855
- PKCS\_5
  - tomcrypt\_custom.h, 430
- pkcs\_5\_1.c
  - pkcs\_5\_alg1, 594
- pkcs\_5\_2.c
  - pkcs\_5\_alg2, 596
- pkcs\_5\_alg1
  - pkcs\_5\_1.c, 594
- pkcs\_5\_alg2
  - pkcs\_5\_2.c, 596
- pmac\_done
  - pmac\_done.c, 517
- pmac\_done.c
  - pmac\_done, 517
- pmac\_file
  - pmac\_file.c, 519
- pmac\_file.c
  - pmac\_file, 519
- pmac\_init
  - pmac\_init.c, 521
- pmac\_init.c
  - len, 523
  - pmac\_init, 521
  - poly\_div, 523
  - poly\_mul, 523
  - polys, 523
- pmac\_memory
  - pmac\_memory.c, 524
- pmac\_memory.c
  - pmac\_memory, 524
- pmac\_memory\_multi
  - pmac\_memory\_multi.c, 526
- pmac\_memory\_multi.c
  - pmac\_memory\_multi, 526
- pmac\_ntz
  - pmac\_ntz.c, 528
- pmac\_ntz.c
  - pmac\_ntz, 528
- pmac\_process
  - pmac\_process.c, 529
- pmac\_process.c
  - pmac\_process, 529
- pmac\_shift\_xor
  - pmac\_shift\_xor.c, 531
- pmac\_shift\_xor.c
  - pmac\_shift\_xor, 531
- pmac\_test
  - pmac\_test.c, 532
- pmac\_test.c
  - pmac\_test, 532
- poly
  - gcm\_gf\_mult.c, 253
- poly\_div
  - ocb\_init.c, 285
  - pmac\_init.c, 523
- poly\_mul
  - ocb\_init.c, 285
  - pmac\_init.c, 523
- polys
  - ocb\_init.c, 285
  - pmac\_init.c, 523
- printable\_table
  - der\_length\_printable\_string.c, 709
- prng\_descriptor
  - crypt\_prng\_descriptor.c, 582
  - tomcrypt\_prng.h, 462
- prng\_is\_valid
  - crypt\_prng\_is\_valid.c, 583
  - tomcrypt\_prng.h, 459
- Prng\_state, 48
  - dummy, 48
- prng\_state
  - tomcrypt\_prng.h, 458
- prngs/fortuna.c, 878
- prngs/rc4.c, 887
- prngs/rng\_get\_bytes.c, 894
- prngs/rng\_make\_prng.c, 896
- prngs/sober128.c, 898
- prngs/sober128tab.c, 911
- prngs/sprng.c, 912
- prngs/yarrow.c, 917
- process
  - ltc\_hash\_descriptor, 27

- qord
  - twofish.c, 211
- qsort\_helper
  - der\_encode\_set.c, 740
  - der\_encode\_setof.c, 743
- R
  - khazad.c, 126
  - sha256.c, 378
  - sha512.c, 389
- rand\_prime
  - rand\_prime.c, 554
  - tomcrypt\_pk.h, 455
- rand\_prime.c
  - rand\_prime, 554
  - USE\_BBS, 554
- RC
  - noekeon.c, 148
- rc
  - anubis.c, 81
- RC2
  - tomcrypt\_custom.h, 430
- rc2.c
  - permute, 154
  - rc2\_desc, 154
  - rc2\_done, 149
  - rc2\_ecb\_decrypt, 149
  - rc2\_ecb\_encrypt, 150
  - rc2\_keysize, 151
  - rc2\_setup, 152
  - rc2\_test, 153
- rc2\_desc
  - rc2.c, 154
- rc2\_done
  - rc2.c, 149
- rc2\_ecb\_decrypt
  - rc2.c, 149
- rc2\_ecb\_encrypt
  - rc2.c, 150
- rc2\_keysize
  - rc2.c, 151
- rc2\_setup
  - rc2.c, 152
- rc2\_test
  - rc2.c, 153
- RC4
  - tomcrypt\_custom.h, 430
- rc4.c
  - rc4\_add\_entropy, 887
  - rc4\_desc, 893
  - rc4\_done, 888
  - rc4\_export, 888
  - rc4\_import, 889
  - rc4\_read, 890
  - rc4\_ready, 890
  - rc4\_start, 891
  - rc4\_test, 892
- RC5
  - tomcrypt\_custom.h, 430
- rc5.c
  - rc5\_desc, 161
  - rc5\_done, 156
  - rc5\_ecb\_decrypt, 156
  - rc5\_ecb\_encrypt, 157
  - rc5\_keysize, 158
  - rc5\_setup, 159
  - rc5\_test, 160
  - stab, 161
- rc5\_desc
  - rc5.c, 161
- rc5\_done
  - rc5.c, 156
- rc5\_ecb\_decrypt
  - rc5.c, 156
- rc5\_ecb\_encrypt
  - rc5.c, 157
- rc5\_keysize
  - rc5.c, 158
- rc5\_setup
  - rc5.c, 159
- rc5\_test
  - rc5.c, 160
- RC6
  - tomcrypt\_custom.h, 430
- rc6.c
  - rc6\_desc, 169
  - rc6\_done, 164
  - rc6\_ecb\_decrypt, 164
  - rc6\_ecb\_encrypt, 165
  - rc6\_keysize, 166

- rc6\_setup, 166
- rc6\_test, 167
- RND, 163, 164
- stab, 169
- rc6\_desc
  - rc6.c, 169
- rc6\_done
  - rc6.c, 164
- rc6\_ecb\_decrypt
  - rc6.c, 164
- rc6\_ecb\_encrypt
  - rc6.c, 165
- rc6\_keysize
  - rc6.c, 166
- rc6\_setup
  - rc6.c, 166
- rc6\_test
  - rc6.c, 167
- rcon
  - aes\_tab.c, 65
- read
  - ltc\_prng\_descriptor, 46
- read\_radix
  - ltc\_math\_descriptor, 39
- ready
  - ltc\_prng\_descriptor, 46
- register\_cipher
  - crypt\_register\_cipher.c, 584
  - tomcrypt\_cipher.h, 421
- register\_hash
  - crypt\_register\_hash.c, 586
  - tomcrypt\_hash.h, 443
- register\_prng
  - crypt\_register\_prng.c, 588
  - tomcrypt\_prng.h, 459
- RIJNDAEL
  - tomcrypt\_custom.h, 431
- rijndael\_desc
  - aes.c, 63
- RIPEMD128
  - tomcrypt\_custom.h, 431
- RIPEMD160
  - tomcrypt\_custom.h, 431
- RIPEMD256
  - tomcrypt\_custom.h, 431
- RIPEMD320
  - tomcrypt\_custom.h, 431
- rmd128.c
  - F, 329
  - FF, 329
  - FFF, 330
  - G, 330
  - GG, 330
  - GGG, 330
  - H, 330
  - HH, 330
  - HHH, 330
  - I, 331
  - II, 331
  - III, 331
  - rmd128\_compress, 331
  - rmd128\_desc, 336
  - rmd128\_done, 334
  - rmd128\_init, 335
  - rmd128\_test, 335
- rmd128\_compress
  - rmd128.c, 331
- rmd128\_desc
  - rmd128.c, 336
- rmd128\_done
  - rmd128.c, 334
- rmd128\_init
  - rmd128.c, 335
- rmd128\_test
  - rmd128.c, 335
- rmd160.c
  - F, 339
  - FF, 339
  - FFF, 339
  - G, 339
  - GG, 339
  - GGG, 339
  - H, 339
  - HH, 340
  - HHH, 340
  - I, 340
  - II, 340
  - III, 340
  - J, 340
  - JJ, 340
  - JJJ, 341
  - rmd160\_compress, 341
  - rmd160\_desc, 347
  - rmd160\_done, 344
  - rmd160\_init, 345
  - rmd160\_test, 346
- rmd160\_compress
  - rmd160.c, 341
- rmd160\_desc
  - rmd160.c, 347
- rmd160\_done
  - rmd160.c, 344
- rmd160\_init
  - rmd160.c, 345
- rmd160\_test
  - rmd160.c, 346
- rmd256.c
  - F, 348

- FF, 348
- FFF, 349
- G, 349
- GG, 349
- GGG, 349
- H, 349
- HH, 349
- HHH, 349
- I, 350
- II, 350
- III, 350
- rmd256\_compress, 350
- rmd256\_desc, 356
- rmd256\_done, 353
- rmd256\_init, 354
- rmd256\_test, 355
- rmd256\_compress
  - rmd256.c, 350
- rmd256\_desc
  - rmd256.c, 356
- rmd256\_done
  - rmd256.c, 353
- rmd256\_init
  - rmd256.c, 354
- rmd256\_test
  - rmd256.c, 355
- rmd320.c
  - F, 358
  - FF, 358
  - FFF, 358
  - G, 358
  - GG, 358
  - GGG, 358
  - H, 358
  - HH, 359
  - HHH, 359
  - I, 359
  - II, 359
  - III, 359
  - J, 359
  - JJ, 359
  - JJJ, 360
  - rmd320\_compress, 360
  - rmd320\_desc, 366
  - rmd320\_done, 363
  - rmd320\_init, 364
  - rmd320\_test, 365
- rmd320\_compress
  - rmd320.c, 360
- rmd320\_desc
  - rmd320.c, 366
- rmd320\_done
  - rmd320.c, 363
- rmd320\_init
  - rmd320.c, 364
- rmd320\_test
  - rmd320.c, 365
- RND
  - rc6.c, 163, 164
  - sha256.c, 378
  - sha512.c, 389
- rng\_get\_bytes
  - rng\_get\_bytes.c, 894
  - tomcrypt\_prng.h, 460
- rng\_get\_bytes.c
  - rng\_get\_bytes, 894
  - rng\_nix, 894
- rng\_make\_prng
  - rng\_make\_prng.c, 896
  - tomcrypt\_prng.h, 461
- rng\_make\_prng.c
  - rng\_make\_prng, 896
- rng\_nix
  - rng\_get\_bytes.c, 894
- ROL
  - tomcrypt\_macros.h, 447
- ROL16
  - kasumi.c, 118
- ROL64
  - tomcrypt\_macros.h, 447
- ROL64c
  - tomcrypt\_macros.h, 447
- ROL8
  - safer.c, 172
- ROLc
  - tomcrypt\_macros.h, 447
- ROR
  - tomcrypt\_macros.h, 447
- ROR64
  - tomcrypt\_macros.h, 448
- ROR64c
  - tomcrypt\_macros.h, 448
- RORc
  - tomcrypt\_macros.h, 448
- ROTATE\_LEFT
  - md4.c, 315
- ROUND
  - noekeon.c, 142
  - saferp.c, 185
- rounds
  - kseed.c, 139
- RS
  - twofish.c, 212
- rs\_mult
  - twofish.c, 205
- RS\_POLY
  - twofish.c, 203
- rsa\_decrypt\_key.c

- rsa\_decrypt\_key\_ex, 857
- rsa\_decrypt\_key\_ex
  - rsa\_decrypt\_key.c, 857
- rsa\_encrypt\_key.c
  - rsa\_encrypt\_key\_ex, 859
- rsa\_encrypt\_key\_ex
  - rsa\_encrypt\_key.c, 859
- rsa\_export
  - rsa\_export.c, 861
- rsa\_export.c
  - rsa\_export, 861
- rsa\_exptmod
  - rsa\_exptmod.c, 863
- rsa\_exptmod.c
  - rsa\_exptmod, 863
- rsa\_free
  - rsa\_free.c, 866
- rsa\_free.c
  - rsa\_free, 866
- rsa\_import
  - rsa\_import.c, 867
- rsa\_import.c
  - rsa\_import, 867
- rsa\_key
  - tomcrypt\_math.h, 450
- rsa\_keygen
  - ltc\_math\_descriptor, 40
- rsa\_make\_key
  - rsa\_make\_key.c, 870
- rsa\_make\_key.c
  - rsa\_make\_key, 870
- rsa\_me
  - ltc\_math\_descriptor, 40
- rsa\_sign\_hash.c
  - rsa\_sign\_hash\_ex, 872
- rsa\_sign\_hash\_ex
  - rsa\_sign\_hash.c, 872
- rsa\_verify\_hash.c
  - rsa\_verify\_hash\_ex, 875
- rsa\_verify\_hash\_ex
  - rsa\_verify\_hash.c, 875
- RULE\_A
  - skipjack.c, 195
- RULE\_A1
  - skipjack.c, 195
- RULE\_B
  - skipjack.c, 195
- RULE\_B1
  - skipjack.c, 195
- S
  - sha256.c, 378
  - sha512.c, 389
- S1
  - cast5.c, 97
- S11
  - md4.c, 315
- S12
  - md4.c, 315
- s128\_diffuse
  - sober128.c, 901
- s128\_genkonst
  - sober128.c, 902
- s128\_reloadstate
  - sober128.c, 902
- s128\_savestate
  - sober128.c, 903
- S13
  - md4.c, 316
- S14
  - md4.c, 316
- S2
  - cast5.c, 97
- S21
  - md4.c, 316
- S22
  - md4.c, 316
- S23
  - md4.c, 316
- S24
  - md4.c, 316
- S3
  - cast5.c, 97
- S31
  - md4.c, 316
- S32
  - md4.c, 316
- S33
  - md4.c, 316
- S34
  - md4.c, 316
- S4
  - cast5.c, 97
- S5
  - cast5.c, 97
- S6
  - cast5.c, 97
- S7
  - cast5.c, 97
- S8
  - cast5.c, 98
- s\_ocb\_done
  - s\_ocb\_done.c, 292
- s\_ocb\_done.c
  - s\_ocb\_done, 292
- SAFER
  - tomcrypt\_custom.h, 431
- safer.c

- EXP, 171
- IPHT, 171
- LOG, 172
- PHT, 172
- ROL8, 172
- safer\_128\_keysize, 172
- safer\_64\_keysize, 172
- safer\_done, 173
- safer\_ebox, 179
- safer\_ecb\_decrypt, 173
- safer\_ecb\_encrypt, 174
- Safer\_Expand\_Userkey, 174
- safer\_k128\_desc, 179
- safer\_k128\_setup, 175
- safer\_k64\_desc, 179
- safer\_k64\_setup, 176
- safer\_k64\_test, 176
- safer\_lbox, 180
- safer\_sk128\_desc, 180
- safer\_sk128\_setup, 177
- safer\_sk128\_test, 177
- safer\_sk64\_desc, 180
- safer\_sk64\_setup, 178
- safer\_sk64\_test, 178
- safer\_128\_keysize
  - safer.c, 172
- safer\_64\_keysize
  - safer.c, 172
- safer\_bias
  - saferp.c, 192
- safer\_done
  - safer.c, 173
- safer\_ebox
  - safer.c, 179
  - safer\_tab.c, 181
  - saferp.c, 192
- safer\_ecb\_decrypt
  - safer.c, 173
- safer\_ecb\_encrypt
  - safer.c, 174
- Safer\_Expand\_Userkey
  - safer.c, 174
- safer\_k128\_desc
  - safer.c, 179
- safer\_k128\_setup
  - safer.c, 175
- safer\_k64\_desc
  - safer.c, 179
- safer\_k64\_setup
  - safer.c, 176
- safer\_k64\_test
  - safer.c, 176
- safer\_lbox
  - safer.c, 180
  - safer\_tab.c, 181
  - saferp.c, 192
- safer\_sk128\_desc
  - safer.c, 180
- safer\_sk128\_setup
  - safer.c, 177
- safer\_sk128\_test
  - safer.c, 177
- safer\_sk64\_desc
  - safer.c, 180
- safer\_sk64\_setup
  - safer.c, 178
- safer\_sk64\_test
  - safer.c, 178
- safer\_tab.c
  - safer\_ebox, 181
  - safer\_lbox, 181
- SAFERP
  - tomcrypt\_custom.h, 431
- saferp.c
  - iLT, 184
  - iPHT, 184
  - iROUND, 184
  - iSHUF, 184
  - LT, 185
  - PHT, 185
  - ROUND, 185
  - safer\_bias, 192
  - safer\_ebox, 192
  - safer\_lbox, 192
  - saferp\_desc, 192
  - saferp\_done, 186
  - saferp\_ecb\_decrypt, 186
  - saferp\_ecb\_encrypt, 187
  - saferp\_keysize, 188
  - saferp\_setup, 189
  - saferp\_test, 191
  - SHUF, 185
- saferp\_desc
  - saferp.c, 192
- saferp\_done
  - saferp.c, 186
- saferp\_ecb\_decrypt
  - saferp.c, 186
- saferp\_ecb\_encrypt
  - saferp.c, 187
- saferp\_keysize
  - saferp.c, 188
- saferp\_setup
  - saferp.c, 189
- saferp\_test
  - saferp.c, 191
- SBO
  - whirltab.c, 407

- SB1
  - whirltab.c, [407](#)
- SB2
  - whirltab.c, [407](#)
- SB3
  - whirltab.c, [407](#)
- SB4
  - whirltab.c, [408](#)
- SB5
  - whirltab.c, [408](#)
- SB6
  - whirltab.c, [408](#)
- SB7
  - whirltab.c, [408](#)
- Sbox
  - sober128tab.c, [911](#)
- sbox
  - skipjack.c, [201](#)
  - twofish.c, [203](#)
- sbox0
  - whirltab.c, [408](#)
- sbox1
  - whirltab.c, [408](#)
- sbox2
  - whirltab.c, [408](#)
- sbox3
  - whirltab.c, [409](#)
- sbox4
  - whirltab.c, [409](#)
- sbox5
  - whirltab.c, [409](#)
- sbox6
  - whirltab.c, [409](#)
- sbox7
  - whirltab.c, [409](#)
- SCRYPT
  - tomcrypt.h, [411](#)
- set\_int
  - ltc\_math\_descriptor, [40](#)
- SETUP
  - aes.c, [52](#), [60](#)
- setup
  - ltc\_cipher\_descriptor, [23](#)
- setup\_mix
  - aes.c, [63](#)
- SHA1
  - tomcrypt\_custom.h, [431](#)
- sha1.c
  - F0, [368](#)
  - F1, [368](#)
  - F2, [369](#)
  - F3, [369](#)
  - FF0, [369](#)
  - FF1, [369](#)
  - FF2, [369](#)
  - FF3, [369](#)
  - sha1\_compress, [369](#)
  - sha1\_desc, [373](#)
  - sha1\_done, [371](#)
  - sha1\_init, [372](#)
  - sha1\_test, [372](#)
- sha1\_compress
  - sha1.c, [369](#)
- sha1\_desc
  - sha1.c, [373](#)
- sha1\_done
  - sha1.c, [371](#)
- sha1\_init
  - sha1.c, [372](#)
- sha1\_test
  - sha1.c, [372](#)
- SHA224
  - tomcrypt\_custom.h, [431](#)
- sha224.c
  - sha224\_desc, [376](#)
  - sha224\_done, [374](#)
  - sha224\_init, [374](#)
  - sha224\_test, [375](#)
- sha224\_desc
  - sha224.c, [376](#)
- sha224\_done
  - sha224.c, [374](#)
- sha224\_init
  - sha224.c, [374](#)
- sha224\_test
  - sha224.c, [375](#)
- SHA256
  - tomcrypt\_custom.h, [431](#)
- sha256.c
  - Ch, [377](#)
  - Gamma0, [377](#)
  - Gamma1, [378](#)
  - Maj, [378](#)
  - R, [378](#)
  - RND, [378](#)
  - S, [378](#)
  - sha256\_compress, [378](#)
  - sha256\_desc, [383](#)
  - sha256\_done, [380](#)
  - sha256\_init, [381](#)
  - sha256\_test, [382](#)
  - Sigma0, [378](#)
  - Sigma1, [378](#)
- sha256\_compress
  - sha256.c, [378](#)
- sha256\_desc
  - sha256.c, [383](#)
- sha256\_done

- sha256.c, 380
- sha256\_init
  - sha256.c, 381
- sha256\_test
  - sha256.c, 382
- SHA384
  - tomcrypt\_custom.h, 431
- sha384.c
  - sha384\_desc, 386
  - sha384\_done, 384
  - sha384\_init, 385
  - sha384\_test, 385
- sha384\_desc
  - sha384.c, 386
- sha384\_done
  - sha384.c, 384
- sha384\_init
  - sha384.c, 385
- sha384\_test
  - sha384.c, 385
- SHA512
  - tomcrypt\_custom.h, 432
- sha512.c
  - Ch, 388
  - Gamma0, 388
  - Gamma1, 388
  - K, 393
  - Maj, 389
  - R, 389
  - RND, 389
  - S, 389
  - sha512\_compress, 389
  - sha512\_desc, 393
  - sha512\_done, 390
  - sha512\_init, 391
  - sha512\_test, 392
  - Sigma0, 389
  - Sigma1, 389
- sha512\_compress
  - sha512.c, 389
- sha512\_desc
  - sha512.c, 393
- sha512\_done
  - sha512.c, 390
- sha512\_init
  - sha512.c, 391
- sha512\_test
  - sha512.c, 392
- SHUF
  - saferp.c, 185
- Sigma0
  - sha256.c, 378
  - sha512.c, 389
- Sigma1
  - sha256.c, 378
  - sha512.c, 389
- size
  - edge, 13
- SKIPJACK
  - tomcrypt\_custom.h, 432
- skipjack.c
  - g\_func, 195
  - ig\_func, 196
  - ikeystep, 200
  - keystep, 200
  - RULE\_A, 195
  - RULE\_A1, 195
  - RULE\_B, 195
  - RULE\_B1, 195
  - sbox, 201
  - skipjack\_desc, 201
  - skipjack\_done, 196
  - skipjack\_ecb\_decrypt, 196
  - skipjack\_ecb\_encrypt, 197
  - skipjack\_keysize, 198
  - skipjack\_setup, 199
  - skipjack\_test, 199
- skipjack\_desc
  - skipjack.c, 201
- skipjack\_done
  - skipjack.c, 196
- skipjack\_ecb\_decrypt
  - skipjack.c, 196
- skipjack\_ecb\_encrypt
  - skipjack.c, 197
- skipjack\_keysize
  - skipjack.c, 198
- skipjack\_setup
  - skipjack.c, 199
- skipjack\_test
  - skipjack.c, 199
- SOBER128
  - tomcrypt\_custom.h, 432
- sober128.c
  - ADDKEY, 899
  - B, 899
  - BYTE2WORD, 901
  - cycle, 901
  - DROUND, 899
  - FOLD, 899
  - FOLDP, 899
  - INITKONST, 899
  - KEYP, 900
  - N, 900
  - NLFUNC, 900
  - nltp, 901
  - OFF, 900
  - s128\_diffuse, 901



- s128\_genkonst, [902](#)
- s128\_reloadstate, [902](#)
- s128\_savestate, [903](#)
- sober128\_add\_entropy, [903](#)
- sober128\_desc, [910](#)
- sober128\_done, [904](#)
- sober128\_export, [904](#)
- sober128\_import, [905](#)
- sober128\_read, [906](#)
- sober128\_ready, [907](#)
- sober128\_start, [908](#)
- sober128\_test, [908](#)
- SROUND, [900](#)
- STEP, [900](#)
- WORD2BYTE, [900](#)
- XORNL, [900](#)
- XORWORD, [909](#)
- sober128\_add\_entropy
  - sober128.c, [903](#)
- sober128\_desc
  - sober128.c, [910](#)
- sober128\_done
  - sober128.c, [904](#)
- sober128\_export
  - sober128.c, [904](#)
- sober128\_import
  - sober128.c, [905](#)
- sober128\_read
  - sober128.c, [906](#)
- sober128\_ready
  - sober128.c, [907](#)
- sober128\_start
  - sober128.c, [908](#)
- sober128\_test
  - sober128.c, [908](#)
- sober128tab.c
  - Multab, [911](#)
  - Sbox, [911](#)
- SP1
  - des.c, [113](#)
- SP2
  - des.c, [114](#)
- SP3
  - des.c, [114](#)
- SP4
  - des.c, [115](#)
- SP5
  - des.c, [115](#)
- SP6
  - des.c, [116](#)
- SP7
  - des.c, [116](#)
- SP8
  - des.c, [116](#)
- SPRNG
  - tomcrypt\_custom.h, [432](#)
- sprng.c
  - sprng\_add\_entropy, [912](#)
  - sprng\_desc, [915](#)
  - sprng\_done, [913](#)
  - sprng\_export, [913](#)
  - sprng\_import, [913](#)
  - sprng\_read, [914](#)
  - sprng\_ready, [914](#)
  - sprng\_start, [915](#)
  - sprng\_test, [915](#)
- sprng\_add\_entropy
  - sprng.c, [912](#)
- sprng\_desc
  - sprng.c, [915](#)
- sprng\_done
  - sprng.c, [913](#)
- sprng\_export
  - sprng.c, [913](#)
- sprng\_import
  - sprng.c, [913](#)
- sprng\_read
  - sprng.c, [914](#)
- sprng\_ready
  - sprng.c, [914](#)
- sprng\_start
  - sprng.c, [915](#)
- sprng\_test
  - sprng.c, [915](#)
- sqr
  - ltc\_math\_descriptor, [41](#)
- sqrmod
  - ltc\_math\_descriptor, [41](#)
- SROUND
  - sober128.c, [900](#)
- SS0
  - kseed.c, [140](#)
- SS1
  - kseed.c, [140](#)
- SS2
  - kseed.c, [140](#)
- SS3
  - kseed.c, [140](#)
- stab
  - rc5.c, [161](#)
  - rc6.c, [169](#)
- start
  - edge, [13](#)
  - ltc\_prng\_descriptor, [46](#)
- STEP
  - sober128.c, [900](#)
- STORE\_V
  - der\_encode\_utctime.c, [753](#)

- sub
  - ltc\_math\_descriptor, [41](#)
- subi
  - ltc\_math\_descriptor, [41](#)
- Symmetric\_key, [49](#)
  - data, [49](#)
- symmetric\_key
  - tomcrypt\_cipher.h, [418](#)
- T0
  - anubis.c, [81](#)
  - khazad.c, [132](#)
- T1
  - anubis.c, [82](#)
  - khazad.c, [132](#)
- t1
  - tiger.c, [394](#)
- T2
  - anubis.c, [82](#)
  - khazad.c, [133](#)
- t2
  - tiger.c, [395](#)
- T3
  - anubis.c, [82](#)
  - khazad.c, [133](#)
- t3
  - tiger.c, [395](#)
- T4
  - anubis.c, [82](#)
  - khazad.c, [133](#)
- t4
  - tiger.c, [395](#)
- T5
  - anubis.c, [82](#)
  - khazad.c, [133](#)
- T6
  - khazad.c, [133](#)
- T7
  - khazad.c, [133](#)
- TAB\_SIZE
  - tomcrypt.h, [411](#)
- table
  - tiger.c, [399](#)
- TD0
  - aes\_tab.c, [66](#)
- Td0
  - aes\_tab.c, [64](#)
- TD1
  - aes\_tab.c, [66](#)
- Td1
  - aes\_tab.c, [64](#)
- TD2
  - aes\_tab.c, [66](#)
- Td2
  - aes\_tab.c, [65](#)
- TD3
  - aes\_tab.c, [66](#)
- Td3
  - aes\_tab.c, [65](#)
- Td4
  - aes\_tab.c, [66](#)
- TE0
  - aes\_tab.c, [66](#)
- Te0
  - aes\_tab.c, [65](#)
- TE1
  - aes\_tab.c, [66](#)
- Te1
  - aes\_tab.c, [65](#)
- TE2
  - aes\_tab.c, [66](#)
- Te2
  - aes\_tab.c, [65](#)
- TE3
  - aes\_tab.c, [66](#)
- Te3
  - aes\_tab.c, [65](#)
- Te4
  - aes\_tab.c, [66](#)
- Te4\_0
  - aes\_tab.c, [66](#)
- Te4\_1
  - aes\_tab.c, [67](#)
- Te4\_2
  - aes\_tab.c, [67](#)
- Te4\_3
  - aes\_tab.c, [67](#)
- test
  - ltc\_cipher\_descriptor, [23](#)
  - ltc\_hash\_descriptor, [27](#)
  - ltc\_prng\_descriptor, [46](#)
- tfm\_desc.c
  - DESC\_DEF\_ONLY, [556](#)
- THETA
  - noekeon.c, [143](#)
- theta\_pi\_gamma
  - whirl.c, [401](#)
- TIGER
  - tomcrypt\_custom.h, [432](#)
- tiger.c
  - INLINE, [394](#)
  - key\_schedule, [395](#)
  - pass, [395](#)
  - t1, [394](#)
  - t2, [395](#)
  - t3, [395](#)
  - t4, [395](#)
  - table, [399](#)

- tiger\_compress, 396
- tiger\_desc, 399
- tiger\_done, 396
- tiger\_init, 397
- tiger\_round, 398
- tiger\_test, 398
- tiger\_compress
  - tiger.c, 396
- tiger\_desc
  - tiger.c, 399
- tiger\_done
  - tiger.c, 396
- tiger\_init
  - tiger.c, 397
- tiger\_round
  - tiger.c, 398
- tiger\_test
  - tiger.c, 398
- Tks0
  - aes\_tab.c, 67
- Tks1
  - aes\_tab.c, 67
- Tks2
  - aes\_tab.c, 67
- Tks3
  - aes\_tab.c, 67
- tomcrypt.h
  - CRYPT, 411
  - CRYPT\_BUFFER\_OVERFLOW, 412
  - CRYPT\_ERROR, 412
  - CRYPT\_ERROR\_READPRNG, 412
  - CRYPT\_FAIL\_TESTVECTOR, 412
  - CRYPT\_FILE\_NOTFOUND, 412
  - CRYPT\_INVALID\_ARG, 412
  - CRYPT\_INVALID\_CIPHER, 412
  - CRYPT\_INVALID\_HASH, 412
  - CRYPT\_INVALID\_KEYSIZE, 412
  - CRYPT\_INVALID\_PACKET, 412
  - CRYPT\_INVALID\_PRIME\_SIZE, 412
  - CRYPT\_INVALID\_PRNG, 412
  - CRYPT\_INVALID\_PRNGSIZE, 412
  - CRYPT\_INVALID\_ROUNDS, 412
  - CRYPT\_MEM, 412
  - CRYPT\_NOP, 412
  - CRYPT\_OK, 412
  - CRYPT\_PK\_DUP, 412
  - CRYPT\_PK\_INVALID\_PADDING, 412
  - CRYPT\_PK\_INVALID\_SIZE, 412
  - CRYPT\_PK\_INVALID\_SYSTEM, 412
  - CRYPT\_PK\_INVALID\_TYPE, 412
  - CRYPT\_PK\_NOT\_FOUND, 412
  - CRYPT\_PK\_NOT\_PRIVATE, 412
  - CRYPT\_PK\_TYPE\_MISMATCH, 412
  - MAXBLOCKSIZE, 411
  - SCRYPT, 411
  - TAB\_SIZE, 411
- tomcrypt\_argchk.h
  - crypt\_argchk, 415
  - LTC\_ARGCHK, 414
  - LTC\_ARGCHKVD, 415
- tomcrypt\_cfg.h
  - ARGTYPE, 416
  - ENDIAN\_NEUTRAL, 416
  - XCALLOC, 417
  - XCLOCK, 417
  - XFREE, 417
  - XMALLOC, 417
  - XMEMCMP, 417
  - XMEMCPY, 417
  - XMEMSET, 417
  - XQSORT, 417
  - XREALLOC, 417
- tomcrypt\_cipher.h
  - cipher\_descriptor, 422
  - cipher\_is\_valid, 418
  - find\_cipher, 419
  - find\_cipher\_any, 419
  - find\_cipher\_id, 420
  - register\_cipher, 421
  - symmetric\_key, 418
  - unregister\_cipher, 421
- tomcrypt\_custom.h
  - ANUBIS, 425
  - ANUBIS\_TWEAK, 425
  - BASE64, 425
  - BLOWFISH, 425
  - CAST5, 425
  - CCM\_MODE, 425
  - CHC\_HASH, 425
  - DES, 425
  - DEVRANDOM, 425
  - EAX\_MODE, 425
  - ECC112, 425
  - ECC128, 426
  - ECC160, 426
  - ECC192, 426
  - ECC224, 426
  - ECC256, 426
  - ECC384, 426
  - ECC521, 426
  - FORTUNA, 426
  - FORTUNA\_POOLS, 426
  - FORTUNA\_WD, 426
  - GCM\_MODE, 426
  - GCM\_TABLES, 427
  - KHAZAD, 427
  - KSEED, 427
  - LRW\_TABLES, 427

- LTC\_CBC\_MODE, [427](#)
- LTC\_CFB\_MODE, [427](#)
- LTC\_CTR\_MODE, [427](#)
- LTC\_DER, [427](#)
- LTC\_ECB\_MODE, [427](#)
- LTC\_F8\_MODE, [427](#)
- LTC\_F9\_MODE, [427](#)
- LTC\_HMAC, [428](#)
- LTC\_KASUMI, [428](#)
- LTC\_LRW\_MODE, [428](#)
- LTC\_MUTEX\_GLOBAL, [428](#)
- LTC\_MUTEX\_INIT, [428](#)
- LTC\_MUTEX\_LOCK, [428](#)
- LTC\_MUTEX\_PROTO, [428](#)
- LTC\_MUTEX\_TYPE, [428](#)
- LTC\_MUTEX\_UNLOCK, [428](#)
- LTC\_OFB\_MODE, [429](#)
- LTC\_OMAC, [429](#)
- LTC\_PMAC, [429](#)
- LTC\_TEST, [429](#)
- LTC\_XCBC, [429](#)
- MD2, [429](#)
- MD4, [429](#)
- MD5, [429](#)
- MDSA, [429](#)
- MECC, [429](#)
- MPI, [429](#)
- MRSA, [430](#)
- NOEKEON, [430](#)
- OCB\_MODE, [430](#)
- PELICAN, [430](#)
- PKCS\_1, [430](#)
- PKCS\_5, [430](#)
- RC2, [430](#)
- RC4, [430](#)
- RC5, [430](#)
- RC6, [430](#)
- RIJNDAEL, [431](#)
- RIPEMD128, [431](#)
- RIPEMD160, [431](#)
- RIPEMD256, [431](#)
- RIPEMD320, [431](#)
- SAFER, [431](#)
- SAFERP, [431](#)
- SHA1, [431](#)
- SHA224, [431](#)
- SHA256, [431](#)
- SHA384, [431](#)
- SHA512, [432](#)
- SKIPJACK, [432](#)
- SOBER128, [432](#)
- SPRNG, [432](#)
- TIGER, [432](#)
- TRY\_URANDOM\_FIRST, [432](#)
- TWOFISH, [432](#)
- TWOFISH\_TABLES, [432](#)
- WHIRLPOOL, [432](#)
- XCALLOC, [432](#)
- XCLOCK, [432](#)
- XCLOCKS\_PER\_SEC, [433](#)
- XFREE, [433](#)
- XMALLOC, [433](#)
- XMEMCMP, [433](#)
- XMEMCPY, [433](#)
- XMEMSET, [434](#)
- XQSORT, [434](#)
- XREALLOC, [434](#)
- XTEA, [434](#)
- YARROW, [434](#)
- YARROW\_AES, [434](#)
- tomcrypt\_hash.h
  - find\_hash, [436](#)
  - find\_hash\_any, [436](#)
  - find\_hash\_id, [437](#)
  - find\_hash\_oid, [438](#)
  - hash\_descriptor, [444](#)
  - hash\_file, [438](#)
  - hash\_filehandle, [439](#)
  - hash\_is\_valid, [440](#)
  - hash\_memory, [440](#)
  - hash\_memory\_multi, [441](#)
  - HASH\_PROCESS, [436](#)
  - hash\_state, [436](#)
  - register\_hash, [443](#)
  - unregister\_hash, [443](#)
- tomcrypt\_macros.h
  - BSWAP, [446](#)
  - byte, [446](#)
  - CONST64, [446](#)
  - MAX, [447](#)
  - MIN, [447](#)
  - ROL, [447](#)
  - ROL64, [447](#)
  - ROL64c, [447](#)
  - ROLc, [447](#)
  - ROR, [447](#)
  - ROR64, [448](#)
  - ROR64c, [448](#)
  - RORc, [448](#)
  - ulong32, [448](#)
  - ulong64, [448](#)
- tomcrypt\_math.h
  - ecc\_point, [450](#)
  - ltc\_deinit\_multi, [450](#)
  - ltc\_init\_multi, [450](#)
  - ltc\_mp, [451](#)
  - LTC\_MP\_EQ, [449](#)
  - LTC\_MP\_GT, [449](#)

- LTC\_MP\_LT, 449
- LTC\_MP\_NO, 450
- LTC\_MP\_YES, 450
- rsa\_key, 450
- tomcrypt\_misc.h
  - burn\_stack, 452
  - crypt\_build\_settings, 454
  - crypt\_fsa, 452
  - error\_to\_string, 453
  - zeromem, 453
- tomcrypt\_pk.h
  - PK\_PRIVATE, 455
  - PK\_PUBLIC, 455
- tomcrypt\_pk.h
  - rand\_prime, 455
- tomcrypt\_prng.h
  - find\_prng, 458
  - prng\_descriptor, 462
  - prng\_is\_valid, 459
  - prng\_state, 458
  - register\_prng, 459
  - rng\_get\_bytes, 460
  - rng\_make\_prng, 461
  - unregister\_prng, 462
- totrot
  - des.c, 117
- TRY\_URANDOM\_FIRST
  - tomcrypt\_custom.h, 432
- twoexpt
  - ltc\_math\_descriptor, 42
- TWOFISH
  - tomcrypt\_custom.h, 432
- twofish.c
  - gl\_func, 203
  - g\_func, 203
  - gf\_mult, 203
  - h\_func, 204
  - MDS, 211
  - mds\_column\_mult, 203
  - mds\_mult, 204
  - MDS\_POLY, 203
  - qord, 211
  - RS, 212
  - rs\_mult, 205
  - RS\_POLY, 203
  - sbox, 203
  - twofish\_desc, 212
  - twofish\_done, 205
  - twofish\_ecb\_decrypt, 205
  - twofish\_ecb\_encrypt, 206
  - twofish\_keysize, 207
  - twofish\_setup, 208
  - twofish\_test, 210
- twofish\_desc
  - twofish.c, 212
- twofish\_done
  - twofish.c, 205
- twofish\_ecb\_decrypt
  - twofish.c, 205
- twofish\_ecb\_encrypt
  - twofish.c, 206
- twofish\_keysize
  - twofish.c, 207
- twofish\_setup
  - twofish.c, 208
- TWOFISH\_TABLES
  - tomcrypt\_custom.h, 432
- twofish\_test
  - twofish.c, 210
- u16
  - kasumi.c, 118
- ulong32
  - tomcrypt\_macros.h, 448
- ulong64
  - tomcrypt\_macros.h, 448
- UNDEFED\_HASH
  - chc.c, 295
- unregister\_cipher
  - crypt\_unregister\_cipher.c, 590
  - tomcrypt\_cipher.h, 421
- unregister\_hash
  - crypt\_unregister\_hash.c, 591
  - tomcrypt\_hash.h, 443
- unregister\_prng
  - crypt\_unregister\_prng.c, 592
  - tomcrypt\_prng.h, 462
- unsigned\_read
  - ltc\_math\_descriptor, 42
- unsigned\_size
  - ltc\_math\_descriptor, 42
- unsigned\_write
  - ltc\_math\_descriptor, 42
- USE\_BBS
  - rand\_prime.c, 554
- value
  - der\_length\_ia5\_string.c, 684
  - der\_length\_printable\_string.c, 709
- whirl.c
  - GB, 401
  - theta\_pi\_gamma, 401
  - whirlpool\_compress, 402
  - whirlpool\_desc, 406
  - whirlpool\_done, 402
  - whirlpool\_init, 403
  - whirlpool\_test, 404

- WHIRLPOOL
  - tomcrypt\_custom.h, 432
- whirlpool\_compress
  - whirl.c, 402
- whirlpool\_desc
  - whirl.c, 406
- whirlpool\_done
  - whirl.c, 402
- whirlpool\_init
  - whirl.c, 403
- whirlpool\_test
  - whirl.c, 404
- whirltab.c
  - cont, 408
  - SB0, 407
  - SB1, 407
  - SB2, 407
  - SB3, 407
  - SB4, 408
  - SB5, 408
  - SB6, 408
  - SB7, 408
  - sbox0, 408
  - sbox1, 408
  - sbox2, 408
  - sbox3, 409
  - sbox4, 409
  - sbox5, 409
  - sbox6, 409
  - sbox7, 409
- WINSIZE
  - ltc\_ecc\_mulmod.c, 814
- WORD2BYTE
  - sober128.c, 900
- write\_radix
  - ltc\_math\_descriptor, 43
- XCALLOC
  - tomcrypt\_cfg.h, 417
  - tomcrypt\_custom.h, 432
- xcbc\_done
  - xcbc\_done.c, 535
- xcbc\_done.c
  - xcbc\_done, 535
- xcbc\_file
  - xcbc\_file.c, 537
- xcbc\_file.c
  - xcbc\_file, 537
- xcbc\_init
  - xcbc\_init.c, 539
- xcbc\_init.c
  - xcbc\_init, 539
- xcbc\_memory
  - ltc\_cipher\_descriptor, 23
  - xcbc\_memory.c, 541
- xcbc\_memory.c
  - xcbc\_memory, 541
- xcbc\_memory\_multi
  - xcbc\_memory\_multi.c, 543
- xcbc\_memory\_multi.c
  - xcbc\_memory\_multi, 543
- xcbc\_process
  - xcbc\_process.c, 545
- xcbc\_process.c
  - xcbc\_process, 545
- xcbc\_test
  - xcbc\_test.c, 547
- xcbc\_test.c
  - xcbc\_test, 547
- XCLOCK
  - tomcrypt\_cfg.h, 417
  - tomcrypt\_custom.h, 432
- XCLOCKS\_PER\_SEC
  - tomcrypt\_custom.h, 433
- XFREE
  - tomcrypt\_cfg.h, 417
  - tomcrypt\_custom.h, 433
- XMALLOC
  - tomcrypt\_cfg.h, 417
  - tomcrypt\_custom.h, 433
- XMEMCMP
  - tomcrypt\_cfg.h, 417
  - tomcrypt\_custom.h, 433
- XMEMCPY
  - tomcrypt\_cfg.h, 417
  - tomcrypt\_custom.h, 433
- XMEMSET
  - tomcrypt\_cfg.h, 417
  - tomcrypt\_custom.h, 434
- XORNL
  - sober128.c, 900
- XORWORD
  - sober128.c, 909
- XQSORT
  - tomcrypt\_cfg.h, 417
  - tomcrypt\_custom.h, 434
- XREALLOC
  - tomcrypt\_cfg.h, 417
  - tomcrypt\_custom.h, 434
- XTEA
  - tomcrypt\_custom.h, 434
- xtea.c
  - xtea\_desc, 218
  - xtea\_done, 214
  - xtea\_ecb\_decrypt, 214
  - xtea\_ecb\_encrypt, 215
  - xtea\_keysize, 216
  - xtea\_setup, 216

- xtea\_test, [217](#)
- xtea\_desc
  - xtea.c, [218](#)
- xtea\_done
  - xtea.c, [214](#)
- xtea\_ecb\_decrypt
  - xtea.c, [214](#)
- xtea\_ecb\_encrypt
  - xtea.c, [215](#)
- xtea\_keysize
  - xtea.c, [216](#)
- xtea\_setup
  - xtea.c, [216](#)
- xtea\_test
  - xtea.c, [217](#)
- YARROW
  - tomcrypt\_custom.h, [434](#)
- yarrow.c
  - yarrow\_add\_entropy, [917](#)
  - yarrow\_desc, [924](#)
  - yarrow\_done, [918](#)
  - yarrow\_export, [919](#)
  - yarrow\_import, [920](#)
  - yarrow\_read, [920](#)
  - yarrow\_ready, [921](#)
  - yarrow\_start, [922](#)
  - yarrow\_test, [924](#)
- yarrow\_add\_entropy
  - yarrow.c, [917](#)
- YARROW\_AES
  - tomcrypt\_custom.h, [434](#)
- yarrow\_desc
  - yarrow.c, [924](#)
- yarrow\_done
  - yarrow.c, [918](#)
- yarrow\_export
  - yarrow.c, [919](#)
- yarrow\_import
  - yarrow.c, [920](#)
- yarrow\_read
  - yarrow.c, [920](#)
- yarrow\_ready
  - yarrow.c, [921](#)
- yarrow\_start
  - yarrow.c, [922](#)
- yarrow\_test
  - yarrow.c, [924](#)
- zeromem
  - tomcrypt\_misc.h, [453](#)
  - zeromem.c, [599](#)
- zeromem.c
  - zeromem, [599](#)