CrossWorks for ARM Reference Manual

Version: 2.2.0.2012022100.14129

Copyright © 1997-2012 Rowley Associates Ltd.

Contents

Introduction ... 29
What is CrossWorks? ... 30
What we don't tell you ... 31
Activating your product ... 32
Text conventions ... 34
Additional resources ... 36
Release notes ... 37
CrossStudio Tutorial ... 49
Activating CrossWorks ... 50
Managing support packages ... 52
Creating a project ... 55
Managing files in a project ... 61
Setting project options ... 65
Building projects ... 67
Exploring projects ... 70
Using the debugger ... 82
Low-level debugging ... 88
Debugging externally built applications ... 92
CrossStudio User Guide ... 97
CrossStudio standard layout ... 98
Menu bar ... 99
Title bar ... 100
Status bar ... 101
Editing workspace ... 103
Docking windows ... 104
Dashboard ... 105
CrossStudio help and assistance ... 106
Creating and managing projects ... 108
Solutions and projects ... 109
Creating a project ... 112
Adding existing files to a project ... 113
Adding new files to a project ... 114
Removing a file, folder, project, or project link ... 115
Project macros ... 116
Building your application ... 118
Creating variants using configurations ... 120
Project properties ... 122
Unique properties ... 123

Aggregate properties ... 124
Configurations and property values ... 125
Dependencies and build order ... 127
Linking and section placement ... 128
Using source control ... 131
Source control capabilities ... 132
Choosing your source-control provider ... 133
Connecting to the source-control system ... 134
Opening a project from source control ... 135
Files source-control status ... 136
Source-control operations ... 137
Adding files to source control ... 138
Checking files out ... 139
Checking files in ... 140
Undoing a check out ... 141
Getting the latest version ... 142
Showing differences between files ... 143
Merging files ... 144
Source-control explorer ... 145
Source-control properties ... 146
Visual SourceSafe provider ... 147
SourceOffSite provider ... 148
Subversion provider ... 149
CVS provider ... 151
Package management ... 153
Exploring your application ... 157
Project explorer ... 158
Source navigator window ... 163
Symbol browser ... 165
Memory usage window ... 170
Editing your code ... 173
Basic editing ... 174
Moving the insertion point ... 175
Selecting text ... 176
Adding text ... 177
Deleting text ... 178
Using the clipboard ... 179
Undo and redo ... 180
Drag and drop ... 181
Searching ... 182
Advanced editing ... 183

Indenting source code ... 184
Commenting out sections of code ... 185
Changing letter case ... 186
Indenting ... 187
Bookmarks ... 188
Find and Replace window ... 190
Clipboard-ring window ... 192
Regular expressions ... 194
Debugging windows ... 197
Locals window ... 198
Globals window ... 200
Watch window ... 202
Register window ... 206
Memory window ... 209
Breakpoints window ... 212
Call Stack window ... 216
Threads window ... 219
Execution Counts window ... 223
Trace window ... 224
Debug file search editor ... 225
Breakpoint expressions ... 227
Debug expressions ... 228
Utility windows ... 229
Output window ... 230
Properties window ... 231
Targets window ... 232
Terminal emulator window ... 237
Script Console window ... 238
Debug Immediate window ... 239
Downloads window ... 240
Latest News window ... 241
Memory-map editor ... 242
Environment options dialog ... 245
Building Environment Options ... 246
Debugging Environment Options ... 248
IDE Environment Options ... 251
Programming Language Environment Options ... 256
Source Control Environment Options ... 259
Text Editor Environment Options ... 261
Windows Environment Options ... 265
ARM target support ... 268

Target startup code ... 269
Startup code ... 271
memory-map files ... 274
Project configurations ... 276
Target script file ... 279
Program loading ... 283
Debug Capabilities ... 284
Target interfaces ... 287
ARM Simulator target interface ... 289
CrossConnect Target Interface ... 290
Segger J-Link Target Interface ... 292
Olimex ARM-USB-OCD Target Interface ... 294
Amontec JTAGkey Target Interface ... 296
PandE UNIT Interface DLL Target Interface ... 298
ST-LINK Target Interface ... 299
ST-LINK/V2 Target Interface ... 301
Kinetis OSJTAG Target Interface ... 303
Stellaris ICDI Target Interface ... 304
Macraigor Wiggler (20 and 14 pin) Target Interface ... 305
Generic FT2232 Target Interface ... 307
Generic Target Interface ... 310
C Library User Guide ... 311
Floating point ... 312
Single and double precision ... 313
Multithreading ... 315
Thread safety in the CrossWorks library ... 316
Implementing mutual exclusion in the C library ... 317
Input and output ... 318
Customizing putchar ... 319
Locales ... 324
Unicode, ISO 10646, and wide characters ... 325
Multi-byte characters ... 326
The standard C and POSIX locales ... 327
Additional locales in source form ... 328
Installing a locale ... 329
Complete API reference ... 330
<assert.h> ... 332
__assert ... 333
assert ... 334
<cross_studio_io.h> ... 335
debug_abort ... 338

debug_break ... 339
debug_clearerr ... 340
debug_enabled ... 341
debug_exit ... 342
debug_fclose ... 343
debug_feof ... 344
debug_ferror ... 345
debug_fflush ... 346
debug_fgetc ... 347
debug_fgetpos ... 348
debug_fgets ... 349
debug_filesize ... 350
debug_fopen ... 351
debug_fprintf ... 352
debug_fprintf_c ... 353
debug_fputc ... 354
debug_fputs ... 355
debug_fread ... 356
debug_freopen ... 357
debug_fscanf ... 358
debug_fscanf_c ... 359
debug_fseek ... 360
debug_fsetpos ... 361
debug_ftell ... 362
debug_fwrite ... 363
debug_getargs ... 364
debug_getch ... 365
debug_getchar ... 366
debug_getd ... 367
debug_getenv ... 368
debug_getf ... 369
debug_geti ... 370
debug_getl ... 371
debug_getll ... 372
debug_gets ... 373
debug_getu ... 374
debug_getul ... 375
debug_getull ... 376
debug_kbhit ... 377
debug_loadsymbols ... 378
debug_perror ... 379

debug_printf ... 380
debug_printf_c ... 381
debug_putchar ... 382
debug_puts ... 383
debug_remove ... 384
debug_rename ... 385
debug_rewind ... 386
debug_runtime_error ... 387
debug_scanf ... 388
debug_scanf_c ... 389
debug_system ... 390
debug_time ... 391
debug_tmpfile ... 392
debug_tmpnam ... 393
debug_ungetc ... 394
debug_unloadsymbols ... 395
debug_vfprintf ... 396
debug_vfscanf ... 397
debug_vprintf ... 398
debug_vscanf ... 399
<ctype.h> ... 400
isalnum ... 401
isalpha ... 402
isblank ... 403
iscntrl ... 404
isdigit ... 405
isgraph ... 406
islower ... 407
isprint ... 408
ispunct ... 409
isspace ... 410
isupper ... 411
isxdigit ... 412
tolower ... 413
toupper ... 414
<errno.h> ... 415
EDOM ... 416
EILSEQ ... 417
ERANGE ... 418
__errno ... 419
errno ... 420

<float.h> 421
DBL_DIG 422
DBL_EPSILON 423
DBL_MANT_DIG 424
DBL_MAX 425
DBL_MAX_10_EXP 426
DBL_MAX_EXP 427
DBL_MIN 428
DBL_MIN_10_EXP 429
DBL_MIN_EXP 430
DECIMAL_DIG 431
FLT_DIG 432
FLT_EPSILON 433
FLT_EVAL_METHOD 434
FLT_MANT_DIG 435
FLT_MAX 436
FLT_MAX_10_EXP 437
FLT_MAX_EXP 438
FLT_MIN 439
FLT_MIN_10_EXP 440
FLT_MIN_EXP 441
FLT_RADIX 442
FLT_ROUNDS 443
<intrinsics.h> 444
__breakpoint 449
__cdp 450
__cdp2 451
__clrex 452
__clz 453
__dbg 454
__disable_fiq 455
__disable_interrupt 456
__disable_irq 457
__dmb 458
__dsb 459
__enable_fiq 460
__enable_interrupt 461
__enable_irq 462
__fabs 463
__fabsf 464
__get_APSR 465

__get_BASEPRI 466
__get_CONTROL 467
__get_CPSR 468
__get_FAULTMASK 469
__get_PRIMASK 470
__isb 471
__ldc 472
__ldc2 473
__ldc2_noidx 474
__ldc2l 475
__ldc2l_noidx 476
__ldc_noidx 477
__ldcl 478
__ldcl_noidx 479
__ldrbt 480
__ldrex 481
__ldrexb 482
__ldrexd 483
__ldrexh 484
__ldrht 485
__ldrsbt 486
__ldrsht 487
__ldrt 488
__mcr 489
__mcr2 490
__mcrr 491
__mcrr2 492
__mrc 493
__mrc2 494
__mrrc 495
__mrrc2 496
__nop 497
__pld 498
__pli 499
__qadd 500
__qadd16 501
__qadd8 502
__qasx 503
__qdadd 504
__qdbl 505
__qdsub 506

__qflag 507
__qsax 508
__qsub 509
__qsub16 510
__qsub8 511
__rbit 512
__rev 513
__rev16 514
__revsh 515
__sadd16 516
__sadd8 517
__sasx 518
__sel 519
__set_APSR 520
__set_BASEPRI 521
__set_CONTROL 522
__set_CPSR 523
__set_FAULTMASK 524
__set_PRIMASK 525
__sev 526
__shadd16 527
__shadd8 528
__shasx 529
__shsax 530
__shsub16 531
__shsub8 532
__smlabb 533
__smlabt 534
__smlad 535
__smladx 536
__smlalbb 537
__smlalbt 538
__smlald 539
__smlaldx 540
__smlaltb 541
__smlaltt 542
__smlatb 543
__smlatt 544
__smlawb 545
__smlawt 546
__smlsd 547

__smlsdx 548
__smlsld 549
__smlsldx 550
__smuad 551
__smuadx 552
__smulbb 553
__smulbt 554
__smultb 555
__smultt 556
__smulwb 557
__smulwt 558
__smusd 559
__smusdx 560
__sqrt 561
__sqrtf 562
__ssat 563
__ssat16 564
__ssax 565
__ssub16 566
__ssub8 567
__stc 568
__stc2 569
__stc2l 570
__stc_noidx 571
__stcl 572
__strbt 573
__strex 574
__strexb 575
__strexd 576
__strexh 577
__strht 578
__strt 579
__swp 580
__swpb 581
__sxtab16 582
__sxtb16 583
__uadd16 584
__uadd8 585
__uasx 586
__uhadd16 587
__uhadd8 588

__uhasx ... 589
__uhsax ... 590
__uhsub16 ... 591
__uhsub8 ... 592
__uqadd16 ... 593
__uqadd8 ... 594
__uqasx ... 595
__uqsax ... 596
__uqsub16 ... 597
__uqsub8 ... 598
__usad8 ... 599
__usad8a ... 600
__usat ... 601
__usat16 ... 602
__usax ... 603
__usub8 ... 604
__uxtab16 ... 605
__uxtb16 ... 606
__wfe ... 607
__wfi ... 608
__yield ... 609
<iso646.h> ... 610
and ... 611
and_eq ... 612
bitand ... 613
bitor ... 614
compl ... 615
not ... 616
not_eq ... 617
or ... 618
or_eq ... 619
xor ... 620
xor_eq ... 621
<libarm.h> ... 622
libarm_dcc_read ... 623
libarm_dcc_write ... 624
libarm_disable_fiq ... 625
libarm_disable_irq ... 626
libarm_disable_irq_fiq ... 627
libarm_enable_fiq ... 628
libarm_enable_irq ... 629

libarm_enable_irq_fiq ... 630
libarm_get_cpsr ... 631
libarm_isr_disable_irq ... 632
libarm_isr_enable_irq ... 633
libarm_mmu_flat_initialise_level_1_table ... 634
libarm_mmu_flat_initialise_level_2_small_page_table ... 635
libarm_mmu_flat_set_level_1_cacheable_region ... 636
libarm_mmu_flat_set_level_2_small_page_cacheable_region ... 637
libarm_restore_irq_fiq ... 638
libarm_run_dcc_port_server ... 639
libarm_set_cpsr ... 640
libarm_set_fiq ... 641
libarm_set_irq ... 642
<limits.h> ... 643
CHAR_BIT ... 644
CHAR_MAX ... 645
CHAR_MIN ... 646
INT_MAX ... 647
INT_MIN ... 648
LLONG_MAX ... 649
LLONG_MIN ... 650
LONG_MAX ... 651
LONG_MIN ... 652
SCHAR_MAX ... 653
SCHAR_MIN ... 654
SHRT_MAX ... 655
SHRT_MIN ... 656
UCHAR_MAX ... 657
UINT_MAX ... 658
ULLONG_MAX ... 659
ULONG_MAX ... 660
USHRT_MAX ... 661
<locale.h> ... 662
lconv ... 663
localeconv ... 665
setlocale ... 666
<math.h> ... 667
acos ... 670
acosf ... 671
acosh ... 672
acoshf ... 673

asin 674
asinf 675
asinh 676
asinhf 677
atan 678
atan2 679
atan2f 680
atanf 681
atanh 682
atanhf 683
cbrt 684
cbrtf 685
ceil 686
ceilf 687
cos 688
cosf 689
cosh 690
coshf 691
exp 692
expf 693
fabs 694
fabsf 695
floor 696
floorf 697
fmax 698
fmaxf 699
fmin 700
fminf 701
fmod 702
fmodf 703
fpclassify 704
frexp 705
frexpf 706
hypot 707
hypotf 708
isfinite 709
isinf 710
isnan 711
isnormal 712
ldexp 713
ldexpf 714

log ... 715
log10 ... 716
log10f ... 717
logf ... 718
modf ... 719
modff ... 720
pow ... 721
powf ... 722
scalbn ... 723
scalbnf ... 724
signbit ... 725
sin ... 726
sinf ... 727
sinh ... 728
sinhf ... 729
sqrt ... 730
sqrtf ... 731
tan ... 732
tanf ... 733
tanh ... 734
tanhf ... 735
<setjmp.h> ... 736
longjmp ... 737
setjmp ... 738
<stdarg.h> ... 739
va_arg ... 740
va_copy ... 741
va_end ... 742
va_start ... 743
<stddef.h> ... 744
NULL ... 745
offsetof ... 746
ptrdiff_t ... 747
size_t ... 748
wchar_t ... 749
<stdio.h> ... 750
getchar ... 751
gets ... 752
printf ... 753
putchar ... 758
puts ... 759

scanf ... 760
snprintf ... 764
sprintf ... 765
sscanf ... 766
vprintf ... 767
vscanf ... 768
vsnprintf ... 769
vsprintf ... 770
vsscanf ... 771
<stdlib.h> ... 772
EXIT_FAILURE ... 774
EXIT_SUCCESS ... 775
RAND_MAX ... 776
abs ... 777
atexit ... 778
atof ... 779
atoi ... 780
atol ... 781
atoll ... 782
bsearch ... 783
calloc ... 784
div ... 785
div_t ... 786
exit ... 787
free ... 788
itoa ... 789
labs ... 790
ldiv ... 791
ldiv_t ... 792
llabs ... 793
lldiv ... 794
lldiv_t ... 795
lltoa ... 796
ltoa ... 797
malloc ... 798
qsort ... 799
rand ... 800
realloc ... 801
srand ... 802
strtod ... 803
strtof ... 804

strtol ... 805
strtoll ... 807
strtoul ... 808
strtoull ... 809
ulltoa ... 810
ultoa ... 811
utoa ... 812
<string.h> ... 813
memccpy ... 815
memchr ... 816
memcmp ... 817
memcpy ... 818
memmove ... 819
mempcpy ... 820
memset ... 821
strcasecmp ... 822
strcasestr ... 823
strcat ... 824
strchr ... 825
strcmp ... 826
strcpy ... 827
strcspn ... 828
strdup ... 829
strerror ... 830
strlcat ... 831
strlcpy ... 832
strlen ... 833
strncasecmp ... 834
strncasestr ... 835
strncat ... 836
strnchr ... 837
strncmp ... 838
strncpy ... 839
strndup ... 840
strnlen ... 841
strnstr ... 842
strpbrk ... 843
strrchr ... 844
strsep ... 845
strspn ... 846
strstr ... 847

strtok ... 848
strtok_r ... 849
<time.h> ... 850
asctime ... 851
asctime_r ... 852
clock_t ... 853
ctime ... 854
ctime_r ... 855
difftime ... 856
gmtime ... 857
gmtime_r ... 858
localtime ... 859
localtime_r ... 860
mktime ... 861
strftime ... 862
time_t ... 864
tm ... 865
<wchar.h> ... 866
WCHAR_MAX ... 868
WCHAR_MIN ... 869
wcscat ... 870
wcschr ... 871
wcscmp ... 872
wcscpy ... 873
wcscspn ... 874
wcsdup ... 875
wcslen ... 876
wcsncat ... 877
wcsnchr ... 878
wcsncmp ... 879
wcsncpy ... 880
wcsnlen ... 881
wcsnstr ... 882
wcspbrk ... 883
wcsrchr ... 884
wcsspn ... 885
wcsstr ... 886
wcstok ... 887
wcstok_r ... 888
wint_t ... 889
wmemccpy ... 890

wmemchr ... 891
wmemcmp ... 892
wmemcpy ... 893
wmemmove ... 894
wmempcpy ... 895
wmemset ... 896
wstrsep ... 897
C++ Library User Guide ... 899
Standard template library ... 900
Subset API reference ... 901
<new> - memory allocation ... 902
operator delete ... 903
operator new ... 904
set_new_handler ... 905
LIBMEM User Guide ... 907
Using the LIBMEM library ... 908
Light version of LIBMEM ... 911
Writing LIBMEM drivers ... 912
LIBMEM loader library ... 916
Complete API reference ... 918
<libmem.h> ... 919
LIBMEM_ADDRESS_IN_RANGE ... 924
LIBMEM_ADDRESS_IS_ALIGNED ... 925
LIBMEM_ALIGNED_ADDRESS ... 926
LIBMEM_CFI_CMDSET_AMD_EXTENDED ... 927
LIBMEM_CFI_CMDSET_AMD_STANDARD ... 928
LIBMEM_CFI_CMDSET_INTEL_EXTENDED ... 929
LIBMEM_CFI_CMDSET_INTEL_STANDARD ... 930
LIBMEM_CFI_CMDSET_MITSUBISHI_EXTENDED ... 931
LIBMEM_CFI_CMDSET_MITSUBISHI_STANDARD ... 932
LIBMEM_CFI_CMDSET_NONE ... 933
LIBMEM_CFI_CMDSET_RESERVED ... 934
LIBMEM_CFI_CMDSET_SST_PAGE_WRITE ... 935
LIBMEM_CFI_CMDSET_WINBOND_STANDARD ... 936
LIBMEM_INLINE ... 937
LIBMEM_KB ... 938
LIBMEM_MB ... 939
LIBMEM_RANGE_OCCLUDES_RANGE ... 940
LIBMEM_RANGE_OVERLAPS_RANGE ... 941
LIBMEM_RANGE_WITHIN_RANGE ... 942
LIBMEM_STATUS_CFI_ERROR ... 943

LIBMEM_STATUS_ERROR ... 944
LIBMEM_STATUS_GEOMETRY_REGION_OVERFLOW ... 945
LIBMEM_STATUS_INVALID_DEVICE ... 946
LIBMEM_STATUS_INVALID_PARAMETER ... 947
LIBMEM_STATUS_INVALID_RANGE ... 948
LIBMEM_STATUS_INVALID_WIDTH ... 949
LIBMEM_STATUS_LOCKED ... 950
LIBMEM_STATUS_NOT_IMPLEMENTED ... 951
LIBMEM_STATUS_NO_DRIVER ... 952
LIBMEM_STATUS_SUCCESS ... 953
LIBMEM_STATUS_TIMEOUT ... 954
LIBMEM_VERSION_NUMBER ... 955
_libmem_driver_functions_t ... 956
_libmem_driver_handle_t ... 957
_libmem_ext_driver_functions_t ... 958
_libmem_flash_info_t ... 959
_libmem_geometry_t ... 960
_libmem_sector_info_t ... 961
libmem_busy_handler_fn ... 962
libmem_busy_handler_fn_t ... 963
libmem_cfi_get_info ... 964
libmem_crc32 ... 966
libmem_crc32_direct ... 967
libmem_driver_crc32_fn_t ... 968
libmem_driver_erase_fn_t ... 969
libmem_driver_fill_fn_t ... 970
libmem_driver_flush_fn_t ... 971
libmem_driver_inrange_fn_t ... 972
libmem_driver_lock_fn_t ... 973
libmem_driver_read_fn_t ... 974
libmem_driver_unlock_fn_t ... 975
libmem_driver_write_fn_t ... 976
libmem_drivers ... 977
libmem_enable_timeouts ... 978
libmem_erase ... 979
libmem_erase_all ... 980
libmem_fill ... 981
libmem_flush ... 982
libmem_foreach_driver ... 983
libmem_foreach_driver_fn_t ... 984
libmem_foreach_sector ... 985

libmem_foreach_sector_fn_t 986
libmem_foreach_sector_in_range 987
libmem_foreach_sector_in_range_ex 988
libmem_get_driver 989
libmem_get_driver_sector_size 990
libmem_get_geometry_size 991
libmem_get_number_of_regions 992
libmem_get_number_of_sectors 993
libmem_get_sector_info 994
libmem_get_sector_number 995
libmem_get_sector_size 996
libmem_get_ticks 997
libmem_get_ticks_fn 998
libmem_get_ticks_fn_t 999
libmem_lock 1000
libmem_lock_all 1001
libmem_read 1002
libmem_register_am29f200b_driver 1003
libmem_register_am29f200t_driver 1004
libmem_register_am29f400bb_driver 1005
libmem_register_am29f400bt_driver 1006
libmem_register_am29fxxx_driver 1007
libmem_register_am29lv010b_driver 1008
libmem_register_cfi_0001_16_driver 1009
libmem_register_cfi_0001_8_driver 1010
libmem_register_cfi_0002_16_driver 1011
libmem_register_cfi_0002_8_driver 1012
libmem_register_cfi_0003_16_driver 1013
libmem_register_cfi_0003_8_driver 1014
libmem_register_cfi_amd_driver 1015
libmem_register_cfi_driver 1017
libmem_register_cfi_intel_driver 1019
libmem_register_driver 1021
libmem_register_ram_driver 1022
libmem_register_sst39xFx00A_16_driver 1023
libmem_register_st_m28w320cb_driver 1024
libmem_register_st_m28w320ct_driver 1025
libmem_set_busy_handler 1026
libmem_ticks_per_second 1027
libmem_unlock 1028
libmem_unlock_all 1029

libmem_write ... 1030
<libmem_loader.h> ... 1031
LIBMEM_RPC_LOADER_FLAG_PARAM ... 1032
LIBMEM_RPC_LOADER_FLAG_PRESERVE_STATE ... 1033
LIBMEM_RPC_LOADER_MAGIC_NUMBER ... 1034
libmem_dcc_loader_set_param_fn_t ... 1035
libmem_dcc_loader_start ... 1036
libmem_dcc_rpc_loader_exit ... 1037
libmem_dcc_rpc_loader_start ... 1039
libmem_rpc_loader_exit ... 1041
libmem_rpc_loader_start ... 1043
Tasking Library User Guide ... 1045
Overview ... 1046
Tasks ... 1048
Event sets ... 1051
Semaphores ... 1055
Mutexes ... 1058
Message queues ... 1060
Byte queues ... 1064
Timers and interrupts ... 1068
Global interrupts control ... 1069
Timer support ... 1070
Interrupt service routines ... 1071
Memory areas ... 1072
Task scheduling example ... 1074
ARM implementation details ... 1076
Cortex-M3 implementation details ... 1078
CTL Revisions ... 1080
Complete API reference ... 1083
<ctl.h> ... 1084
CTL_BYTE_QUEUE_t ... 1087
CTL_ERROR_CODE_t ... 1088
CTL_EVENT_SET_t ... 1089
CTL_EVENT_WAIT_TYPE_t ... 1090
CTL_MEMORY_AREA_t ... 1091
CTL_MESSAGE_QUEUE_t ... 1092
CTL_MUTEX_t ... 1093
CTL_SEMAPHORE_t ... 1094
CTL_STATE_t ... 1095
CTL_TASK_t ... 1096
CTL_TIMEOUT_t ... 1098

CTL_TIME_t 1099
ctl_byte_queue_init 1100
ctl_byte_queue_num_free 1101
ctl_byte_queue_num_used 1102
ctl_byte_queue_post 1103
ctl_byte_queue_post_multi 1104
ctl_byte_queue_post_multi_nb 1105
ctl_byte_queue_post_nb 1106
ctl_byte_queue_receive 1107
ctl_byte_queue_receive_multi 1108
ctl_byte_queue_receive_multi_nb 1109
ctl_byte_queue_receive_nb 1110
ctl_byte_queue_setup_events 1111
ctl_current_time 1112
ctl_events_init 1113
ctl_events_pulse 1114
ctl_events_set_clear 1115
ctl_events_wait 1116
ctl_get_current_time 1117
ctl_global_interrupts_disable 1118
ctl_global_interrupts_enable 1119
ctl_global_interrupts_set 1120
ctl_handle_error 1121
ctl_increment_tick_from_isr 1122
ctl_interrupt_count 1123
ctl_last_schedule_time 1124
ctl_memory_area_allocate 1125
ctl_memory_area_free 1126
ctl_memory_area_init 1127
ctl_memory_area_setup_events 1128
ctl_message_queue_init 1129
ctl_message_queue_num_free 1130
ctl_message_queue_num_used 1131
ctl_message_queue_post 1132
ctl_message_queue_post_multi 1133
ctl_message_queue_post_multi_nb 1134
ctl_message_queue_post_nb 1135
ctl_message_queue_receive 1136
ctl_message_queue_receive_multi 1137
ctl_message_queue_receive_multi_nb 1138
ctl_message_queue_receive_nb 1139

ctl_message_queue_setup_events ... 1140
ctl_mutex_init ... 1141
ctl_mutex_lock ... 1142
ctl_mutex_unlock ... 1143
ctl_reschedule_on_last_isr_exit ... 1144
ctl_semaphore_init ... 1145
ctl_semaphore_signal ... 1146
ctl_semaphore_wait ... 1147
ctl_task_die ... 1148
ctl_task_executing ... 1149
ctl_task_init ... 1150
ctl_task_list ... 1151
ctl_task_remove ... 1152
ctl_task_reschedule ... 1153
ctl_task_restore ... 1154
ctl_task_run ... 1155
ctl_task_set_priority ... 1156
ctl_task_switch_callout ... 1157
ctl_time_increment ... 1158
ctl_timeout_wait ... 1159
ctl_timeslice_period ... 1160
Utilities Reference ... 1161
Compiler driver ... 1161
File naming conventions ... 1162
Command-line options ... 1163
-ansi (Warn about potential ANSI problems) ... 1164
-ar (Archive output) ... 1165
-arch (Set ARM architecture) ... 1166
-be (Big Endian) ... 1167
-c (Compile to object code, do not link) ... 1168
-d (Define linker symbol) ... 1169
-D (Define macro symbol) ... 1170
-e (Set entry point symbol) ... 1171
-E (Preprocess) ... 1172
-exceptions (Enable C++ Exception Support) ... 1173
-fabi (Floating Point Code Generation) ... 1174
-fpu (Set ARM FPU) ... 1175
-F (Set output format) ... 1176
-g (Generate debugging information) ... 1177
-g1 (Generate minimal debugging information) ... 1178
-help (Display help information) ... 1179

-io (Select I/O library implementation) ... 1180
-I (Define user include directories) ... 1181
-I- (Exclude standard include directories) ... 1182
-J (Define system include directories) ... 1183
-K (Keep linker symbol) ... 1184
-L (Set library directory path) ... 1185
-l- (Do not link standard libraries) ... 1186
-make (Make-style build) ... 1187
-M (Display linkage map) ... 1188
-n (Dry run, no execution) ... 1189
-nostderr (No stderr output) ... 1190
-o (Set output file name) ... 1191
-oabi (Use oabi compiler) ... 1192
-O (Optimize output) ... 1193
-printf (Select printf capability) ... 1194
-rtti (Enable C++ RTTI Support) ... 1195
-R (Set section name) ... 1196
-scanf (Select scanf capability) ... 1197
-sd (Treat double as float) ... 1198
-Thumb (Generate Thumb code) ... 1199
-v (Verbose execution) ... 1200
-w (Suppress warnings) ... 1201
-we (Treat warnings as errors) ... 1202
-Wa (Pass option to tool) ... 1203
-x (Specify file types) ... 1204
-y (Use project template) ... 1205
-z (Set project property) ... 1206
CrossBuild ... 1207
Building with a CrossStudio project file ... 1208
Building without a CrossStudio project file ... 1209
Command-line options ... 1210
-batch (Batch build) ... 1211
-config (Select build configuration) ... 1212
-clean (Remove output files) ... 1213
-define (Define macro) ... 1214
-echo (Show command lines) ... 1215
-file (Build a named file) ... 1216
-packagesdir (Specify packages directory) ... 1217
-project (Specify project to build) ... 1218
-property (Set project property) ... 1219
-rebuild (Always rebuild) ... 1220

-show (Dry run, don't execute) ... 1221
-solution (Specify solution to build) ... 1222
-studiodir (Specify CrossStudio directory) ... 1223
-template (Specify project template) ... 1224
-type (Specify project type) ... 1225
-verbose (Show build information) ... 1226
CrossLoad ... 1227
Command line debugging ... 1229
Managing breakpoints ... 1230
Displaying state ... 1233
Locating the current context ... 1235
Controlling execution ... 1237
Command-line options ... 1238
-break (Stop execution at symbol) ... 1239
-config (Specify build configuration) ... 1240
-debug (Enter command line debugging) ... 1241
-filetype (Specify load file type) ... 1242
-help (Display help) ... 1243
-listfiletypes (Display supported load file types) ... 1244
-listprops (Display target properties) ... 1245
-listtargets (Display supported target interfaces) ... 1246
-loadaddress (Set load address) ... 1247
-nodisconnect (Inhibit target disconnection) ... 1248
-nodownload (Inhibit download) ... 1249
-noverify (Inhibit verification) ... 1250
-packagesdir (Specify package directory) ... 1251
-project (Specify project name) ... 1252
-quiet (Be silent) ... 1253
-script (Execute debug script) ... 1254
-serve (Run semihosting server) ... 1255
-setprop (Set target interface property) ... 1256
-solution (Specify solution file) ... 1257
-studiodir (Specify Studio directory) ... 1258
-target (Specify target interface) ... 1259
-verbose (Display additional status) ... 1260
CrossScript ... 1261
Command-line options ... 1262
-define (Define global variable) ... 1263
-help (Show usage) ... 1264
-load (Load script file) ... 1265
-define (Verbose output) ... 1266

CrossScript classes ... 1267
Example uses ... 1268
Header file generator ... 1269
Using the header generator ... 1270
Command line options ... 1272
-regbaseoffsets (Use offsets from peripheral base) ... 1273
-nobitfields (Inhibit bitfield macros) ... 1274
Package generator ... 1275
Appendices ... 1277
Technical ... 1277
File formats ... 1277
Memory Map file format ... 1278
Section Placement file format ... 1280
Project file format ... 1281
Project Templates file format ... 1283
Property Groups file format ... 1285
Package Description file format ... 1287
Property categories ... 1292
General Build Properties ... 1292
Combining Project Properties ... 1294
Compilation Properties ... 1295
Debugging Properties ... 1300
Externally Built Executable Project Properties ... 1305
File and Folder Properties ... 1306
Library Project Properties ... 1308
Executable Project Properties ... 1309
Staging Project Properties ... 1313
Macros ... 1314
System Macros ... 1314
Build Macros ... 1316
Script classes ... 1318
CWSys ... 1318
Debug ... 1319
ElfFile ... 1321
TargetInterface ... 1322
WScript ... 1326

Introduction

This guide is divided into a number of sections:

Introduction

Covers installing CrossWorks on your machine and verifying that it operates correctly, followed by a brief guide to the operation of the CrossStudio integrated development environment, debugger, and other software supplied in the CrossWorks package.

CrossStudio Tutorial

Describes how to get started with CrossStudio and runs through all the steps from creating a project to debugging it on hardware.

CrossStudio User Guide

Contains information on how to use the CrossStudio development environment to manage your projects, build, and debug your applications.

Tasking Library User Guide

Contains documentation on using the CrossWorks tasking library to write multi-threaded applications.

C Library User Guide

Contains documentation for the functions in the standard C library supplied in CrossWorks.

ARM target support

Contains a description of system files used for startup and debugging of ARM applications.

Target interfaces

Contains a description of the support for programming ARM microcontrollers.

What is CrossWorks?

CrossWorks for ARM is a complete C/C++ development system for Cortex, ARM, and XScale microprocessors that runs on Windows, Mac OS, Linux, and Solaris. It comprises the ARM GCC C/C++ compiler, the CrossWorks C Library and the CrossStudio integrated development environment.

GNU Compiler Collection

CrossWorks for ARM comes with a pre-built version of the GNU Compiler Collection (GCC) C and C++ compiler, assembler, linker and other tools to enable you to immediately begin developing applications for ARM.

CrossWorks C Library

CrossWorks for ARM has its own royalty-free ANSI and ISO C compliant C library that has been specifically designed for use within embedded systems.

CrossStudio IDE

CrossWorks for ARM is a streamlined integrated development environment for building, testing, and deploying your applications. CrossStudio provides:

- *Source Code Editor*: A powerful source code editor with multi-level undo and redo, makes editing your code a breeze.
- *Project System*: A complete project system organizes your source code and build rules.
- *Build System*: With a single key press you can build all your applications in a solution, ready for them to be loaded onto a target microcontroller.
- *Debugger and Flash Programming*: You can download your programs directly into Flash and debug them seamlessly from within the IDE using a wide range of target interfaces.
- *Help system*: The built-in help system provides context-sensitive help and a complete reference to the CrossStudio IDE and tools.
- *Core Simulator*: As well as providing cross-compilation technology, CrossWorks provides a PC-based fully functional simulation of the target microcontroller core so you can debug parts of your application without waiting for hardware.
- *Utilities*: A set of tools for generating output files in multiple formats and a command line facility for flashing your applications onto the target board provide the final stage of the software development system.

What we don't tell you

This documentation does not attempt to teach the C or assembly language programming; rather, you should seek out one of the many introductory texts available. And similarly the documentation doesn't cover the ARM architecture or microcontroller application development in any great depth.

We also assume that you're fairly familiar with the operating system of the host computer being used.

C programming guides

These are must-have books for any C programmer:

- Kernighan, B.W. and Ritchie, D.M., *The C Programming Language* (2nd edition, 1988). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362-8.
  The original C bible, updated to cover the essentials of ANCI C (1990 version).
- Harbison, S.P. and Steele, G.L., *C: A Reference Manual* (second edition, 1987). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-109802-0.
  A nice reference guide to C, including a useful amount of information on ANSI C. Co-authored by Guy Steele, a noted language expert.

ANSI C reference

If you're serious about C programming, you may want to have the ISO standard on hand:

- ISO/IEC 9899:1990, C Standard and ISO/IEC 9899:1999, C Standard. The standard is available from your national standards body or directly from ISO at **http://www.iso.ch/**.

ARM microcontrollers

For ARM technical reference manuals, specifications, user guides and white papers, go to:

- **http://www.arm.com/Documentation**.

GNU compiler collection

For the latest GCC documentation, news and downloads, go to:

- **http://gcc.gnu.org/**.

Activating your product

Each copy of CrossWorks must be licensed and registered before it can be used. Each time you purchase a CrossWorks license, you, as a single user, can use CrossWorks on the computers you need to develop and deploy your application. This covers the usual scenario of using both a laptop and desktop and, optionally, a laboratory computer.

Evaluating CrossWorks

If you are evaluating CrossWorks on your computer, you must activate it. To activate your software for evaluation, follow these instructions:

- Install CrossWorks on your computer using the CrossWorks installer and accept the license agreement.
- Run the CrossStudio application.
- Choose **Tools** > **License Manager**.
- Click "Evaluate CrossWorks". If you have a default mailer, click the **By Mail** button.
- Using e-mail, send the registration key to the e-mail address **license@rowley.co.uk**.
- If you don't have a default mailer, select the text underneath "Activation request".
- Send the registration key to the e-mail address **license@rowley.co.uk**.

By return you will receive an **activation key**. To activate CrossWorks for evaluation, do the following:

- Run the CrossStudio application.
- Choose **Tools** > **License Manager**.
- Click **Activate CrossWorks**.
- Type in or paste the returned activation key into the dialog and click **Install License**.

If you need more time to evaluate CrossWorks, simply request a new evaluation key when the issued one expires or is about to expire.

After purchasing CrossWorks

When you purchase CrossStudio, either directly from ourselves or through a distributor, you will be issued a Product Key which uniquely identifies your purchase

To permanently activate your software:

- Install CrossWorks on your computer using the CrossWorks installer and accept the license agreement.
- Run the CrossStudio application.
- Choose **Tools** > **License Manager**.
- If you have a default mailer, click the **By Mail** button.

- Using e-mail, send the registration key to the e-mail address **license@rowley.co.uk**.
- If you don't have a default mailer, select the text underneath "Activation request".
- Send the registration key to the e-mail address **license@rowley.co.uk**.

By return you will receive an **activation key**. Then, complete the activation process:

- Run the CrossStudio application.
- Choose **Tools** > **License Manager**.
- Click **Activate CrossWorks**.
- Type in or paste the returned activation key into the dialog and click **Install License**.

As CrossWorks is licensed per developer, you can install the software on any computer that you use such as a desktop, laptop, and laboratory computer, but on each of these you must go through activation using your issued product key.

Text conventions

Menus and user interface elements

When this document refers to any user interface element, it will do so in **bold font**. For instance, you will often see reference to the **Project Explorer**, which is taken to mean the project explorer window. Similarly, you'll see references to the **Standard** toolbar which is positioned at the top of the CrossStudio window, just below the menu bar on Windows and Linux.

When you are directed to select an item from a menu in CrossStudio, we use the form *menu-name* > *item-name*. For instance, **File** > **Save** means that you need to click the **File** menu in the menu bar and then select the **Save** item. This form extends to items in sub-menus, so **File** > **Open With Binary Editor** has the obvious meaning.

Keyboard accelerators

Frequently-used commands are assigned keyboard *accelerators* to speed up common tasks. CrossStudio uses standard Windows and Mac OS keyboard accelerators wherever possible.

Windows, Linux, and Solaris have three key modifiers which are **Ctrl**, **Alt**, and **Shift**. For instance, **Ctrl+Alt+P** means that you should hold down the **Ctrl** and **Alt** buttons whilst pressing the **P** key; and **Shift+F5** means that you should hold down the **Shift** key whilst pressing **F5**.

Mac OS has four key modifiers which are ? (command), ? (option), ? (control), and ? (shift). Generally there is a one-to-one correspondence between the Windows modifiers and the Mac OS modifiers: **Ctrl** is ?, **Alt** is ?, and **Shift** is ?. CrossStudio on Mac OS has its own set of unique key sequences using ? (control) that have no direct Windows equivalent.

CrossStudio on Windows, Solaris, and Linux also uses *key chords* to expand the set of accelerators. Key chords are key sequences composed of two or more key presses. For instance, the key chord **Ctrl+T, D** means that you should type **Ctrl+T** followed by **D**; and **Ctrl+K, Ctrl+Z** means that you should type **Ctrl+T** followed by **Ctrl+Z**. Mac OS does not support accelerator key chords.

Code examples and human interaction

Throughout the documentation, text printed **in this typeface** represents verbatim communication with the computer: for example, pieces of C text, commands to the operating system, or responses from the computer. In examples, text printed *in this typeface* is not to be used verbatim: it represents a class of items, one of which should be used. For example, this is the format of one kind of compilation command:

hcl *source-file*

This means that the command consists of:

- The word **hcl**, typed exactly like that.
- A *source-file*: not the text **source-file**, but an item of the *source-file* class, for example **myprog.c**.

Whenever commands to and responses from the computer are mixed in the same example, the commands (i.e. the items which you enter) will be presented `in this typeface`. For example, here is a dialog with the computer using the format of the compilation command given above:

```
c:\crossworks\examples>hcl -v myprog.c
```

The user types the text **hcl -v myprog.c** and then presses the enter key (which is assumed and is not shown); the computer responds with the rest.

Additional resources

With software as complex as CrossWorks, it's almost inevitable that you will need assistance at some point. Along with the documentation that comes with CrossWorks for ARM, there are a variety of other resources you can use to find out more.

CrossWorks for ARM website

- **http://www.rowley.co.uk/arm/index.htm**

Support

If you need some help working with CrossWorks, or if something you consider a bug, go to:

- **http://rowley.zendesk.com/**

You can subscribe to our RSS newsfeed here:

- **http://www.rowley.co.uk/rss.xml**

Suggestions

If you have any comments or suggestions regarding the software or documentation, you can make suggestions on our UserVoice forum:

- **http://rowley.uservoice.com/**

Finding your way around

CrossStudio is a complex program in many ways, but we have tried to simplify it so that it's easy to use. It's very easy to get started and CrossStudio scales well to complex multi-programmer projects that need to manage large code bases and the inevitable software variants.

In the tutorial you were presented with a whistle-stop tour of CrossStudio to get you up and running. Here we dig deeper into the corners of CrossStudio so you can get the best from it.

Release notes

Release 2.2.0

- Release of new drag-and-dock system; HUDs are now removed.
- Drag windows by the title bar to new dock sites or to tear them off; hold **Ctrl** down when dragging to prevent docking.
- Configurable title bar with option of wrapping dock title buttons.
- Addition of a dock site in the central area for side-by-side editing.
- Capability to drop bookmarks in a selection of colors.
- Alt-Left-Click in the margin of a code editor brings up a bookmark drop selector.
- **Ctrl+F** brings up incremental find; a second **Ctrl+F** brings up the standard **Find** dialog.
- Refined the look of the Dashboard.
- Better property editing dialogs.
- Added **Error List** window (**View** > **Error List**, **Ctrl+Alt+E**) which is a summary of the last set of build diagnostics.
- Files in the **Project Explorer** can be tagged to provide a visual highlight.
- Added editable **Recent Files**, **Recent Projects** lists.
- Added **Tasks** and **Favorites** lists for users to customize with links.
- Bookmarks from the **Bookmarks**, **Recent Files**, **Recent Projects**, **Favorites**, and **Tasks** windows can be dragged to any other bookmark window.
- Clicking the header fields of the **Recent Files**, **Recent Projects**, **Favorites**, **Tasks**, and **Error List** window will sort and group the items in the list. Clicking the leftmost header will cycle through a number of useful sorting/grouping arrangements.
- Any dock site can host editors, including torn-off dock sites.
- Package documentation automatically appears in the **Help** > **Contents** window.
- Fixed **Tools** > **Options** so that search box works.
- Experimental addition of an Office-like ribbon.
- Current execution context is displayed in the editor when debugging.
- Addition of external tools; use **File** > **Open CrossStudio** > **External Tools Configuration**.
- External tools can run over any file, folder, project, or solution from the **Open With** meuu.
- **Search in Project** and **Search in Solution** use Search Dependencies setting.
- Sentinel USB tokens now work under Linux (Ubuntu 8.04 and 10.04 tested with Sentinel USB Daemon v7.5.2).
- Fixed bug in CrossScript ElfFile class peek and CRC32 functions.
- Release notes and license agreement now appear in Mac OS X Lion installer.
- Fixed **inttypes.h** to align with the standard.
- Added support for CodeRed Red Probe+.
- Faster memcpy and memset.

- Added CrossStudio launcher.
- Changed TI text file output to use 4-nibble addresses if they can.
- Hovering over a warning or error icon in the margin brings up diagnostic tooltip.
- Fixed DNS bug causing downloads to hang.

Build

- Supplied build of GCC 4.6.2 based on GCC ARM Embedded source code.
- Default weak __getchar and __putchar symbols are now provided in *_crt0.s to avoid linker problems when 'Enabled Unused Symbol Removal' is set to 'No'.
- Added 'v6', 'v7A' and 'v7R' option to 'ARM Architecture' project property.
- Added 'ARM1136J-S', 'ARM1136JF-S', 'Cortex-R4', 'Cortex-R4F', 'Cortex-A8' and 'Cortex-A9' options to 'ARM Core Type' project property.
- Added 'VFPv3-D32' and VFPv3-D16' options to 'ARM FPU Type' project property.
- Added 'ARM Advanced SIMD Type' code generation project property.
- Added library variants for new architectures/FPU combinations.
- debug_putchar now returns -1 on error.
- Added 'Supply Absolute File Path' project property.
- The 'Clean' and 'Build' command lines for "Externally Built Executable' project types can contain spaces.

Debug

- Added 'Get Part Name Script' project property to enable the target script to supply a part name for the connected device.
- Added 'Match Part Name Script' project property to enable the target script to check the connected device matches the 'Target' property group of the project.
- Fixed usage of 'Debug.wait()' in crossload -debug -script.
- Fixed crossload -debug when executable filename is provided on the command line.
- Fixed exit status returned by crossload when crossload -debug -script is used and the script calls debug.quit().
- Added 'Debug.stopped()' method to crossload debug object.
- 'Kinetis OSJTAG' target interface no longer requires osbdm-jm60.dll.
- 'Kinetis OSJTAG' target interface now supported on MacOS.
- Improved memory read/write performance of 'ST-LINK' target interface.
- New 'ST-LINK/V2' target interface that supports ST-LINK/V2 on Linux/MacOS/Win and doesn't require STLinkUSBDriver.dll.
- Fixed debugger setting initial stack pointer value on 'ST-LINK' target interfaces.
- Added ROM table detection for Cortex-M4.
- Added CONTROL bit[2] to CFBP register display on v7EM processors.
- Generic target interfaces now have a host property to stop the error messages that occurred trying to load nonexistent .dll files.
- J-Link target interface now looks in Windows registry for DLL if path has not been specified.

- Added support for FT232H devices.
- New 'Stellaris ICDI' target interface for the ICDI on the LM4F232H board.
- Fixed debugIO implementation detection when project has been built with 'Enabled Unused Symbol Removal' set to 'No'.
- Added 'ARM11' option to 'ARM Debug Interface' project propety.
- Added ARM11, Cortex-R and Cortex-A debug support to Wiggler, FTDI2232 and CrossConnect target interfaces.
- Added VFPv3 capabilities to the ARM simulator.
- Better support for displaying variables with incomplete types (i.e. types defined in a different compilation unit). C++ classes with virtual functions manifested this problem.
- Support for expanding arrays/structs in the watch window when the processor is running.
- Added debug support for big endian Cortex-M as implemented in the TI TMS470M i.e. code and debug registers are little endian, data is big endian.

CTL

- Added 'ctl_enter_isr_fp()' and 'ctl_exit_isr_fp()' which can be used to save the ABI specified VFP registers on entry/exit of an exception that uses the VFP registers.
- Added ctl sample and support code to show how to save/restore the VFP registers on context switch.

CrossStudio

- Fixed crash using escape key to select default dialog button.
- Added 'Reload Dynamic Folder' to project window context menu.
- Fixed crash when 'Remove Dynamic Folder' was used with one of it's files open in the editor.

Command Line Tools

- Added a tool 'cc' that can be used for command line building.

Release 2.1.2

- Didn't happen.

Release 2.1.1

Build

- Fixed build of (undocumented) memcpy_fast function.
- Added word based memcpy to crt0.s.

Debug

- Fixed SWD with J-Link.

- Renamed J-Link target interface to stop it disappearing when v2.0 has been run.
- Added debugging project property 'Do Not Use bkpt Instruction'.
- Fixed Size enumeration of Memory_region CP15 registers of ARM946E-S in registers window.
- Fixed VFP register groups appearing in registers window on targets without VFP.
- Fixed disassembly of Cortex-M msr/mrs instructions
- Fixed simulator reset (lack of) after loader has run.
- Unused global variables that have been removed are not displayed.
- Added Refresh option to disassembly window - use this if your program copies code.

CrossStudio

- Target now disconnected when solution is closed.

Release 2.1.0

Build

- Now using GCC version 4.6.0.
- Changed the default value of the project property 'Enable Unused Symbol Removal' to be 'Yes'. This is now required if you are using the CrossWorks supplied libraries.
- Added build project property 'GCC Target' to specify that the ARM EABI toolchain and library are to be used. The default value of this project property can be modified by the build environment option 'Default GCC Target' so you don't need to set it for each project.
- Added 'v7EM' option to 'ARM Architecture' project property.
- Added 'ARM946E-S', 'ARM968E-S' and 'Cortex-M4' options to 'ARM Core Type' project property.
- Added 'FPv4-SP-D16' option to 'ARM FPU Type' project property. Set this if your Cortex-M4 has a hardware floating point unit.
- Added 'Hard' option to 'ARM FP ABI Type'.
- Added linker option 'Library Optimization' to enable you to select 'Fast' or 'Small' library variant builds.
- Added compiler project property 'No Common' to specify that global data isn't put into the COMMON section, this is set by default.
- Added build project property 'Treat double as float'. Use this if you use floats rather than doubles to avoid floating point literals defaulting to double precision.
- Added linker project property 'Allow Multiple Symbol Definition' to allow a link to complete if multiply defined symbols are detected. The reorganization of the CrossWorks libraries may require this to be set on an existing projects.
- Added linker project property 'Printf Supported' that specifies if printf should be linked in. The default value of this project property can be modified by the build environment option 'Link In Printf Default'.
- Added linker project property 'Scanf Supported' that specifies if scanf should be linked in. The default value of this project property can be modified by the build environment option 'Link In Scanf Default'.

- Added linker project property 'DebugIO Supported' that specifies if debugIO should be linked in. The default value of this project property can be modified by the build environment option 'Link In debugIO Default'.
- Added linker project property 'Emit Relocations' which enables the debugger to load the executable to a different load address.
- Library variants are now built for hard floating point (ARM EABI only).
- Library variants are now built for double as float (ARM EABI only).
- Library variants are now built with both Fast and Small optimisation levels.
- Added new intrinsics.h file to enable access to ARM/Cortex-M instruction set from C code.
- Added 'strcasecmp', 'strncasecmp' and 'strndup' to <string.h>.
- Added flash_tcm_placement.xml and tcm_placement.xml section placement files.
- Added support for 'end_symbol' and 'address_symbol' properties to ProgramSection nodes in section placement and memory map files.
- Added '.ARM.exidx' ProgramSection node to standard placement files.
- thumb_crt0.s no longer initialises heap if size is less than 8 bytes.
- For EABI builds crt0.s and thumb_crt0.s align stackpointers to 8 bytes.
- Fixed ldexp for all big-endian targets.
- Fixed big-endian v6m double precision multiply, divide and conversion to/from single precision.
- Fixed v6m long long shift helper functions.

Debug

- Added 'Download File' and 'Verify File' to Target menu.
- Debug expression parser now supports 'addressof(filename, linenumber)' operator.
- Added entry into targets window context menu to select the particular target to connect to. Supported on CrossConnect and J-Link target interfaces only.
- Added 'isLittleEndian', 'getEntryPoint', 'findProgram' and 'getSection' to ElfFile JavaScript class.
- Added debugger project property 'TAP Reset Script' that can be called to reset the TAP.
- Added JavaScript setDebugInterfaceProperty 'be32_swap' to support big endian (BE-32) Cortex-M devices.
- ROM table is now decoded when connecting to Cortex-M device.
- Fixed setting/clearing software breakpoints on ARM920T with caches enabled.
- Segger J-Link now supported as a generic debug interface.
- J-Link and ST-Link target interfaces now use the "Target Control Options > Target Interface Type" target property to determine whether to use JTAG or SWD.
- Improved error message when loader detects an unsupported FLASH device.
- Fix erase, lock and unlock operations.
- Added 'No Oscillator' option to frequencies property group.
- The 'Olimex ARM-USB-OCD' target interface now supports ARM-USB-OCD-H devices.
- Added debugger project property 'Load Offset' to change the load address the debugger loads the executable to. The executable should be built with 'Emit Relocations' for this to work.

- Added debugger project property 'Load Offset Symbol Limit' to limit the relocations that are patched.
- Added debugger support for accessing FP registers on Cortex-M4 devices.
- Added separate 'Kinetis OSJTAG' and 'P&E UNIT Interface DLL' target interfaces.
- Fixed displaying arrays bigger than the size of the array.

CTL

- ctl_timeslice_period is now always used and is set to the maximum by default.
- ctl_global_interrupts_disable returns the enabled state.
- ctl_global_interrupts_disable/ctl_global_interrupts_enable now implemented using compiler intrinsics.
- ctl threads.js now shows the number of bytes of stack left.
- For EABI builds stackpointers are 8 byte aligned.

CrossStudio

- Improved new breakpoint dialogs.
- Improved dynamic folder dialog.
- Added 'Show File Count on Folder' to project explorer environment options.
- Added 'Keep Going On Error' to build environment options.
- Added 'Default Text File Encoding' to text editor environment options.
- Added 'Search Dependencies' to file search environment options.
- Added environment options to specify the file extensions for object and executable files.
- New project wizard now works correctly when PackagesDir contains Unicode characters.
- Fixed package manager prompting for installation directory unnecessarily when installing packages.
- Removed documentation references to the memory map editor.

Command Line Tools

- CrossBuild now has the -keepgoing option to keep building after error.
- CrossLoad: Fixed the calling of connect scripts. Private configurations can now be specified. The -config option is no longer required if project has only one configuration.
- The -D command line option to CrossBuild doesn't modify the user environment, the global macro is defined for the execution of the program only.

Release 2.0.11

- Added support for P&E Microcomputer Systems Kinetis OSJTAG target interface.
- Added TargetInterface.getRegister and TargetInterface.setRegister JavaScript functions to get and set registers in Cortex debug access points.
- Fixed reschedule problem using ctl_message_queue_post_multi_nb, ctl_message_queue_receive_multi_nb, ctl_byte_queue_post_multi_nb, ctl_byte_queue_receive_multi_nb from an ISR.

- Fixed CTL_TIMEOUT_DELAY timeouts when used with ctl_message_queue_post_multi, ctl_message_queue_receive_multi, ctl_byte_queue_post_multi and ctl_byte_queue_receive_multi.
- Pre-compile and pre-link commands now stop build immediately if an error exit code is returned.
- Added KrisTech KT-LINK target interface
- Fixed Linux/Mac OS X/Solaris channel B support on FT2232 target interfaces
- Register and bitfield definitions in the memory map can now include descriptions.
- stdbool.h can now be included by C++ source code.

Release 2.0.10

- Fixed 'SVC is not permitted on this architecture' error when assembling Cortex-M3 'svc' instruction.

Release 2.0.9

- Fixed problem setting stack pointer to non word aligned value with ST-LINK.
- Fixed CrossLoad downloading a file twice when the file has been specified on the command line.
- Improved display of linker error messages.
- CTL - fixed priority restore on mutex unlock when higher priority waiting task has timed out.
- Debugger - support for 0b numbers on expressions.
- Fixed hyperlink to LIBMEM loader library in target support documentation.
- Now using BINUTILS version 2.21.

Release 2.0.8

- Save Solution As.. now reloads the saved solution.
- CrossScript ElfFile.peekBytes and ElfFile.crc32 functions now work across program sections.
- Project window context menu target operations (Program, Erase, Verify) support Debugger Options | Additional Load File and External Build Options | Load Address properties.
- Added support for breakpoint chaining on ARM7/ARM9 targets by setting the action in the breakpoint edit dialog when connected to the target.
- Removed breakpoint chaining documentation from breakpoint window.
- Added a Debug Capabilities section to the ARM target support documentation.
- Fixed problem with timeout when ctl_task_reschedule is called from an isr.
- Linker script generator now supplies default inputsections for well known sections enabling simplification of section placement files.
- The first section in Cortex_M section placement files now have a macro settable start address to enable memory to be reserved at the beginning of memory segments.
- Added Cortex_M flash_to_ram_placement file which enables a program to be loaded into flash and then copied into RAM for execution.

- Added tbss and tdata sections to standard section placement files to implement thread local storage.
- Removed unused section placement files.
- Added debugger support for displaying thread local storage allocated variables.
- Added environment option (set by default) to disassemble to UAL assembler mnemonics.
- Support for SEGGER J-Link on Linux (requires the appropriate jlinkarm.so from SEGGER).
- Support for ST-LINK on Windows (requires the appropriate STLinkUSBDriver.dll from STMicroelectronics).
- Added support for assembly code files with .arm file extension.
- Added Remove Dynamic Folder on the project explorer right click context menu.
- Now using GCC version 4.4.5.
- Added -load and -define options to CrossLoad.
- Fixed loader timeout when running lpc2000 project on simulator.
- New file dialog no longer appends extension if it already exists.
- Fixed data valued breakpoints with value > 255.
- Right click on the project window shows the target and placement menus when the property groups file is specific to a configuration.
- Linux version of license manager now uses Ethernet addresses of adapters that are not activated.
- Fixed simulator execution of Thumb2 PC relative load when PC is not word aligned.
- Fixed additional options from file project properties not releasing file when build is complete.
- Status bar messages can now be copied to clipboard.
- CrossLoad can now use CrossStudio target interfaces.
- Added -x option to mkpkg.
- Fixed memory map file caching when downloading and not debugging.
- Fixed mis-optimization of ECMAScript IR which caused '<' to fail in some cases.
- Added new filter feature to register window.
- Fixed 'Go To Included File' so that it ignores comments.

Release 2.0.7

- Fixed executing ARM7/ARM9 loaders compiled in Thumb mode.
- Fixed loader erase operations timing out too early.
- Fixed header files that caused STL compilation problems.

Release 2.0.6

- Fixed debug attach when using debug handler.
- Added breakpoint debugIO implementation for v4(t) architecture. This extends and uses the v4(t) software breakpoint implementation. The v4(t) ARM software breakpoint instructions are now 0xdfffdfff and 0xdffedffe. The v4(t) Thumb software breakpoint instructions are now 0xdfff and 0xdffe.
- Fixed bugs in simulator Thumb sbc and pop {pc} instructions.

- Debugger support for displaying anonymous structs/unions.
- Support for data-valued breakpoints on Cortex-M3 e.g x==3.
- The Olimex ARM-USB-TINY target interface now supports ARM-USB-TINY-H as well.
- Support for ignore count on breakpoints.
- CrossBuild implements 1.5/1.6/1.7 build compatibility environment settings.
- Fixed crash when setting a breakpoint after breakpoint group has been deleted from breakpoint window.
- Fixed debugger support for setting software breakpoints when MMU/caches are enabled on ARM926EJS parts.
- Memory usage window now copes with overlapped sections when computing the amount of memory used.
- Added debugger support for accessing coprocessor registers without debug handler.
- Fixed ctl_mutex documentation examples.
- Improved ARM920T/ARM926EJS debug support when using Segger JLINK.
- Fixed crash when threads.js file is missing.
- Fixed lock up on Mac OS version when file is modified externally.
- Fixed code editor jumping to start of file when Outline window is active.
- CrossConnect and Wiggler target interfaces now support SWD (with adaptor).
- Fixed bugs when using CrossLoad -serve option.
- Improved Cortex-M verify performance.
- Improved V6M floating point performance.
- Now using GCC version 4.4.4.
- File > Open now opens file browser in a sensible place.
- Reworked some alert notification dialogs.
- Documentation for a subset of header files now generated automatically by internal tools.
- Improvements in License Manager dialog on Mac OS.

Release 2.0.5

- Added support for Cortex-M0.
- Now using GCC version 4.4.3.
- Fixed ARM7/ARM9 debug handler overwriting supervisor mode registers.
- Fixed debugger displaying references to class members.
- FT2232 target interfaces now support SWD.
- Added debug_enabled() function that returns true if the application is being debugged.
- Added environment option to specify the default debugIO implementation.
- Modified breakpoint implementation of debugIO so that it doesn't use memory locations.
- Fixed J-Link executing ARM7/ARM9/XScale RPC loaders built in Thumb mode.
- Improved Thumb-2 disassembly.
- Added a "No Load Sections" target property.
- Added a "Gap Fill Value" linker property.

- Enhancements to the threads window
  - The filename of the threads script need not be fixed as threads.js it can now be specified using the Threads Script file type property.
  - The threads script can specify the set of displayed columns.
  - Can be sorted by clicking on a column.
  - Gives a better indication of the selected thread.
  - The threads script can be written so that the register state of a thread is retrieved when the thread is selected.
  - The threads script has access to the TargetInterface object.
- The Target Script has access to the CWSys object.
- Improved variable display of C++ classes.
- Fixed display of const void * and volatile void * parameters in call stack window.

Release 2.0.4

- Added $(StudioDir)/targets/Cortex_M/flash_placement2.xml file for Cortex-M3 parts with external/multiple RAM memories.
- Added INITIALIZE_SECONDARY_SECTIONS to $(StudioDir)/src/thumb_crt0.s for Cortex-M parts with external/multiple memories.
- Fixed start debugging using Debug | Step Into on Cortex-M targets.
- libmem_cfi_get_info() now correctly returns to read array mode on Cortex-M3.
- Fixed the StrataFlash CFI LIBMEM driver so that it will write correctly to Flash with a write buffer greater than 255 words.
- Support for selection of Ethernet or USB interface of J-Link Pro
- The generic FT2232 target interface will now allow multiple PIDs to be specified.
- The Luminary USB Debug interface will now work with devices with a USB PID of 0xBCDA.
- Amontec JTAGkey and Olimex ARM_USB_OCD nTRST signal now default to push-pull.
- Fixed decimal display in registers window.
- On Cortex-M parts hardware breakpoints are now used on read-only addresses that aren't in code space e.g external flash.
- Modified debug_load_symbols to take an extra parameter that is the name of a function to break on.
- The filename passed to debug_load_symbols is now crossworks macro expanded.
- The debugger fixes up data addresses on executables that have been dynamically loaded.
- The symbol browser now updates on start/stop debugging and when debug_load_symbols/debug_unload_symbols is used for dynamic code loading.
- Fixed a problem with breakpoints when using the linked in debug handler.
- Fixed the debugger locating to files opened using the symbols browser window when debugging externally built executables.
- The ARM FP ABI property can have the None value in which case the -mfloat-abi and -mfpu options are not supplied to the compiler.

- The environment option to not supply the -mpfu option to the compiler has been removed.
- Now using GCC version 4.4.2 and BINUTILS version 2.20.

Release 2.0.3

- File names in projects can now have embedded periods, e.g. "foo.bar.c".
- New File dialog now supports embedded periods in file names.
- Fix stack and heap sections occupying space in ELF file.
- Fix Thumb-2 step out and backtrace.
- Fixed editor crash when editing doxygen style comments.
- Debug actions are now available on the disassembly window toolbar.
- Additional assembler/compiler/linker options properties can now be held in a file referenced by new project properties.
- Added debug_load_symbols/debug_unload_symbols functions to enable debugging of dynamically loaded applications.
- Disassembler now displays data sections as hex (rather than trying to disassemble).
- Terminal emulator can (and does) set DTR signal on connect.
- Terminal emulator supports ansi clear display sequence and \r.
- On windows hosts terminal emulator port property can be right clicked to select the list of available COM ports.
- Added -docgen option to cpphtml tool to enable html to be generated from structured comments.
- Added XScale exception vector synchronization - see XScale CPU support package documentation for more information.

Release 2.0.2

- Added Hitex LPC-Stick target interface.
- Cortex-M C runtime startup code now sets up SP to point to __stack_end__ in FLASH configuration allowing FLASH applications to be re-located away from 0x00000000.
- Fixed J-Link Cortex-M3 support.
- Fixed CrossConnect timeout when using adaptive clocking detection on Cortex-M3 devices that partially support RTCK.

Release 2.0.1

- A customer produced executable generated with ARM RVCT 3.1 compiler appears to have an incomplete debug_aranges section which caused symbolic debugging to not work. For executables that are

produced by ARM RVCT 3.1 the debugger has been modified to generate the equivalent of the debug_aranges section at startup. Depending on the size of the application this is not a quick operation and as such slows down the debugger startup.

- Decoding of the Cortex-M3 control[0] bit in the registers window fixed.
- The **Restrict Memory Accesses** target property now defaults to **No**.
- Fixed Replace inactive replace button in Replace dialog.
- Fixed thumb_crt0.s so that it builds for Cortex-M1 projects.
- Added ctl_isr function prototypes to $(StudioDir)/include/ctl_api.h and $(StudioDir)/ctl/include/ctl_api.h files.

Release 2.0

Start over.

CrossStudio Tutorial

In this tutorial, we will take you through activating your copy of CrossWorks; installing support packages; and creating, compiling, and debugging a simple application using the built-in simulator.

Note

If you're viewing this tutorial from within the CrossWorks help **Browser** window, you may find it more convenient to view using an external web browser so you can still see the entire CrossWorks window. To do so, simply right-click on the help content in the CrossWorks **Browser** and choose **Open in External Browser**.

In this section

Activating CrossWorks
Describes how to activate your copy of CrossWorks by obtaining and installing an activation key for evaluation.

Managing support packages
Describes how to download, install, and view CPU-support and board-support packages.

Creating a project
Describes how to start a project, select your target processor, and other common options.

Managing files in a project
Describes how to add existing and new files to a project and how to remove items from a project.

Setting project options
Describes how to set options on project items and how inheritance works for project settings.

Building projects
Describes how to build a project, correct compilation and linkage errors, and find out how big your applications are.

Exploring projects
Describes how to use the **Project Explorer** and **Symbol Browser** to learn how much memory your project takes and how to navigate among the files that make up the project.

Using the debugger
Describes the debugger and how to find and fix problems at a high level when executing your application.

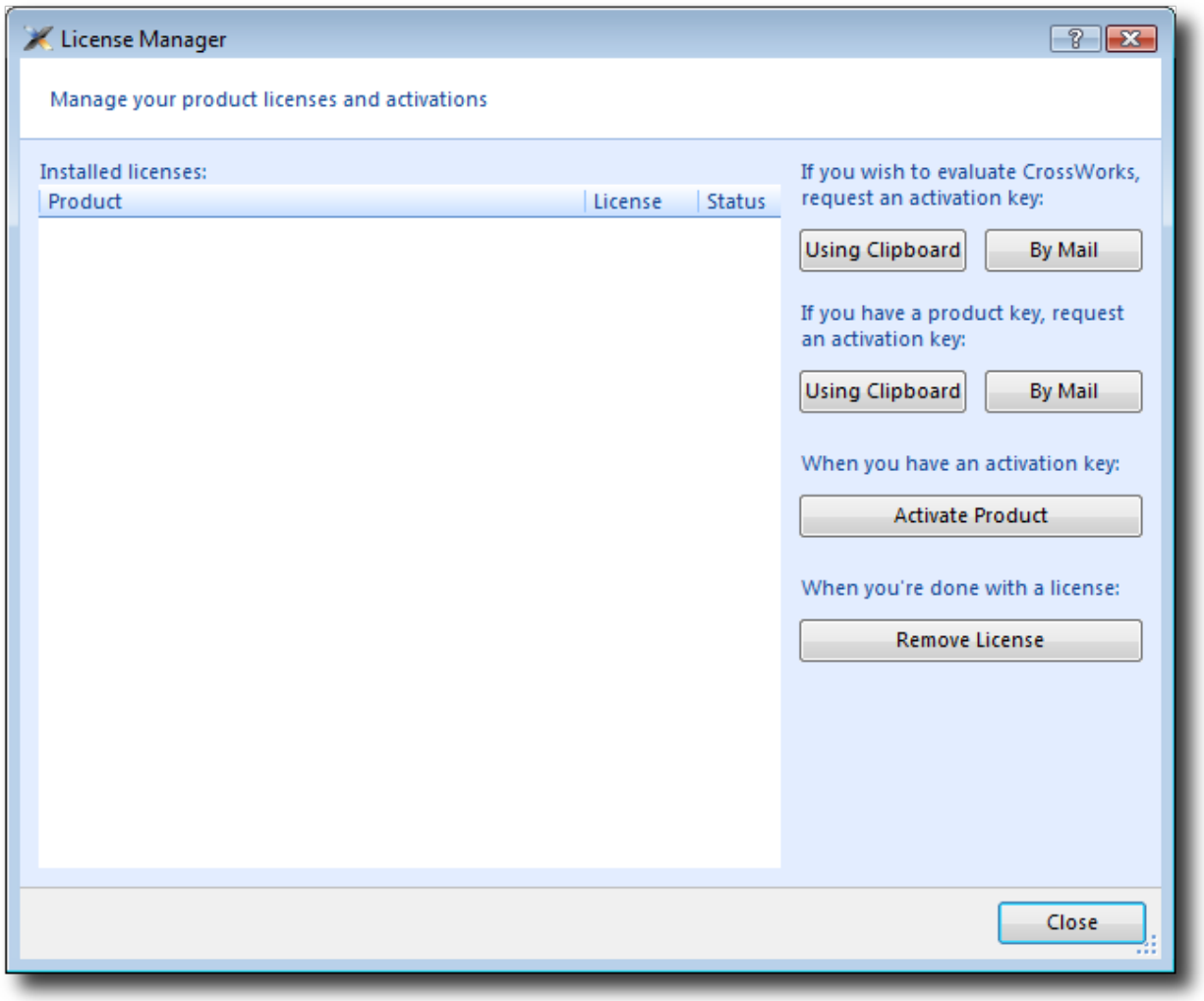
Low-level debugging
Describes how to use debugger features to debug your program at the machine level by watching registers and tracing instructions.

Debugging externally built applications
Describes how to use the debugger to debug externally built applications.

Activating CrossWorks

Each copy of CrossWorks must be registered and activated before it will build projects or download and debug applications. In this tutorial, we are going to use CrossStudio's **License Manager** dialog to request an evaluation activation key and, after the key is received, to activate CrossWorks.

If you have already activated your copy of CrossWorks, you can skip this page.



Requesting an evaluation activation key (with a default e-mail client)

To receive an evaluation activation key that is valid for 30 days:

- Choose **Tools** > **License Manager**.
- Click the **Evaluate CrossWorks** option.
- Choose whether to lock the license to your computer's MAC address or to your system's primary disk.

- Send the e-mail containing the registration key to **license@rowley.co.uk**. If your development system does not have a default e-mail client, copy the activation request and paste it into an e-mail to this address.

Choosing which hardware to lock to is a matter of personal choice. If you lock to your primary disk and then replace that disk drive, reformat it, or upgrade the operating system, CrossWorks may need to be reactivated. If you lock to a network adapter and the network adapter fails and is replaced, then CrossWorks will require reactivation.

When we receive your registration key we will send an activation key back to your e-mail's reply address. You then will use the activation key to unlock and activate CrossWorks.

Activating CrossWorks

When you receive your activation key from us, you can activate CrossWorks as follows:

- Choose **Tools** > **License Manager**.
- Click the **Activate CrossWorks** option.
- Enter the activation key you have received from us.
- Click **Install License**.
- The new activation should now be visible in the list of **Installed licenses**. Click **Close** to close the **License Manager** window.

Note

If you request an activation key outside office hours, there may be a delay processing the registration. If this is the case, you can continue the tutorial until you reach the **Building projects** section—you will need to activate CrossWorks before you can build.

Managing support packages

Before a project can be created, a CPU-support or board-support package suitable for the device you are targeting must be installed. A support package is a single, compressed file that can contain project templates, system files, example projects, and documentation for a particular target.

In this tutorial, we are going to use the **Generic ARM CPU Support Package** to create our project. This will allow us to create a project that will run on CrossWorks' ARM simulator. To create a project that would run on hardware, you would need to install and use support packages suitable for that target hardware but, for the purposes of this tutorial, we'll just target the simulator.

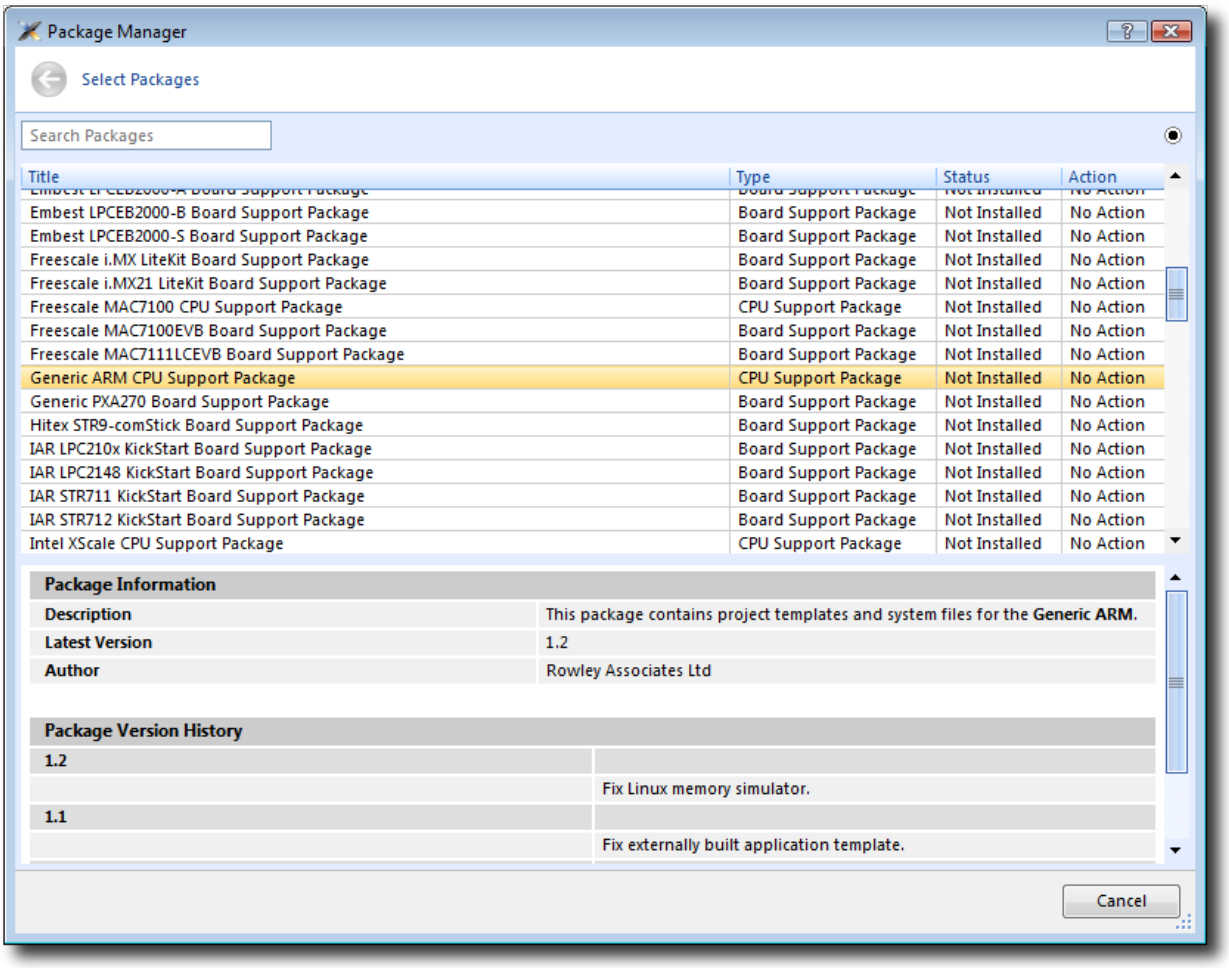
Note that the **Generic ARM CPU Support Package** project templates can be used to target real hardware for devices that don't currently have a suitable support package; however, it is highly likely that you will need to modify memory map files, startup code, reset scripts, and the loader program in order to support the target. This is outside the scope of this tutorial but, should you wish to do this, see the documentation included in the **Generic ARM CPU Support Package** for more information.

If you have already installed this support package, you can skip this page.

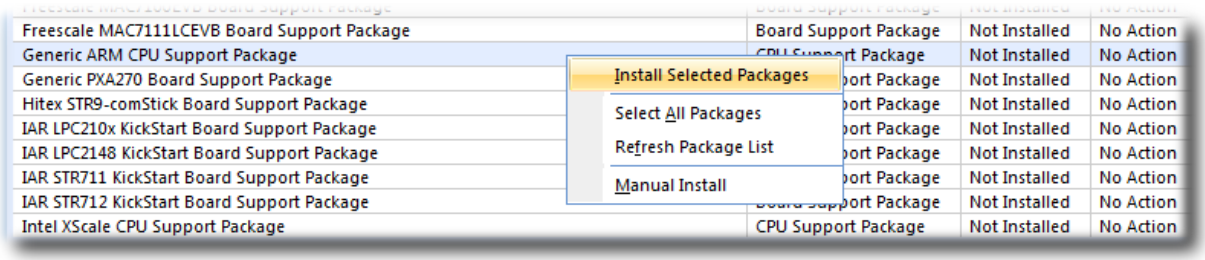
Downloading and installing a support package

To download and install a support package:

- Choose **Tools** > **Manage Packages**.
- Select the Generic ARM CPU Support Package entry.
- (To select more packages to download and install at the same time, you can control-click the additional packages.)



- Right-click the selected package and choose to **Install Selected Packages**.



- Click the **Next** button and you will be presented with a list of actions the package manager is going to carry out.
- Click **Next** again to download and install the support package.
- Upon successful completion, you will see a list of the newly installed packages. Click **Finish**.

Viewing installed support packages

To view the installed support packages:

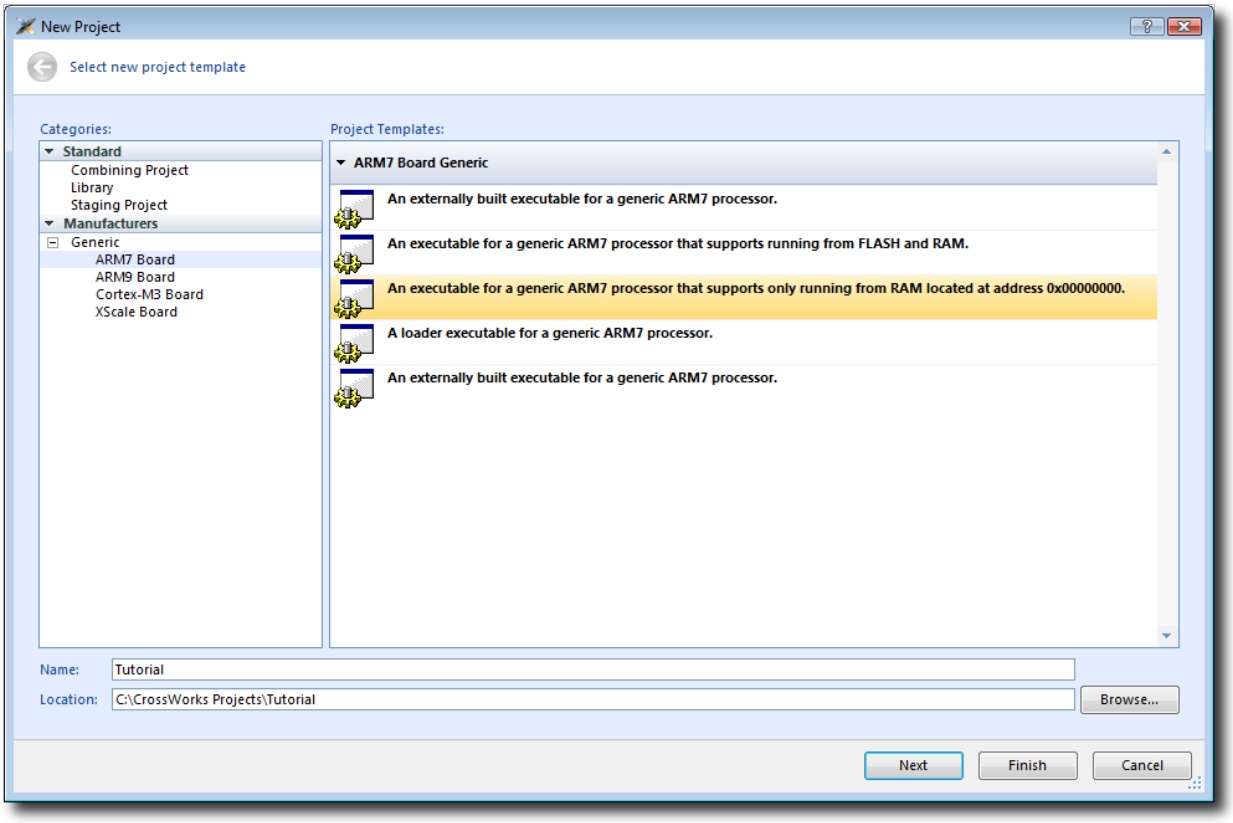
- Choose **Tools > Show Installed Packages** to list the support packages you have installed on your system. You should see the name of the **Generic ARM CPU Support Package** you just installed.
- Click **Generic ARM CPU Support Package** to view the support package page in the CrossStudio **Browser** window. This page provides more information about the support package and links to any documentation, example projects, and system files that may be included in the package.

Creating a project

To start developing an application, first create a new project. To create a new project:

- Choose **File** > **New Project** or press **Ctrl+Shift+N**

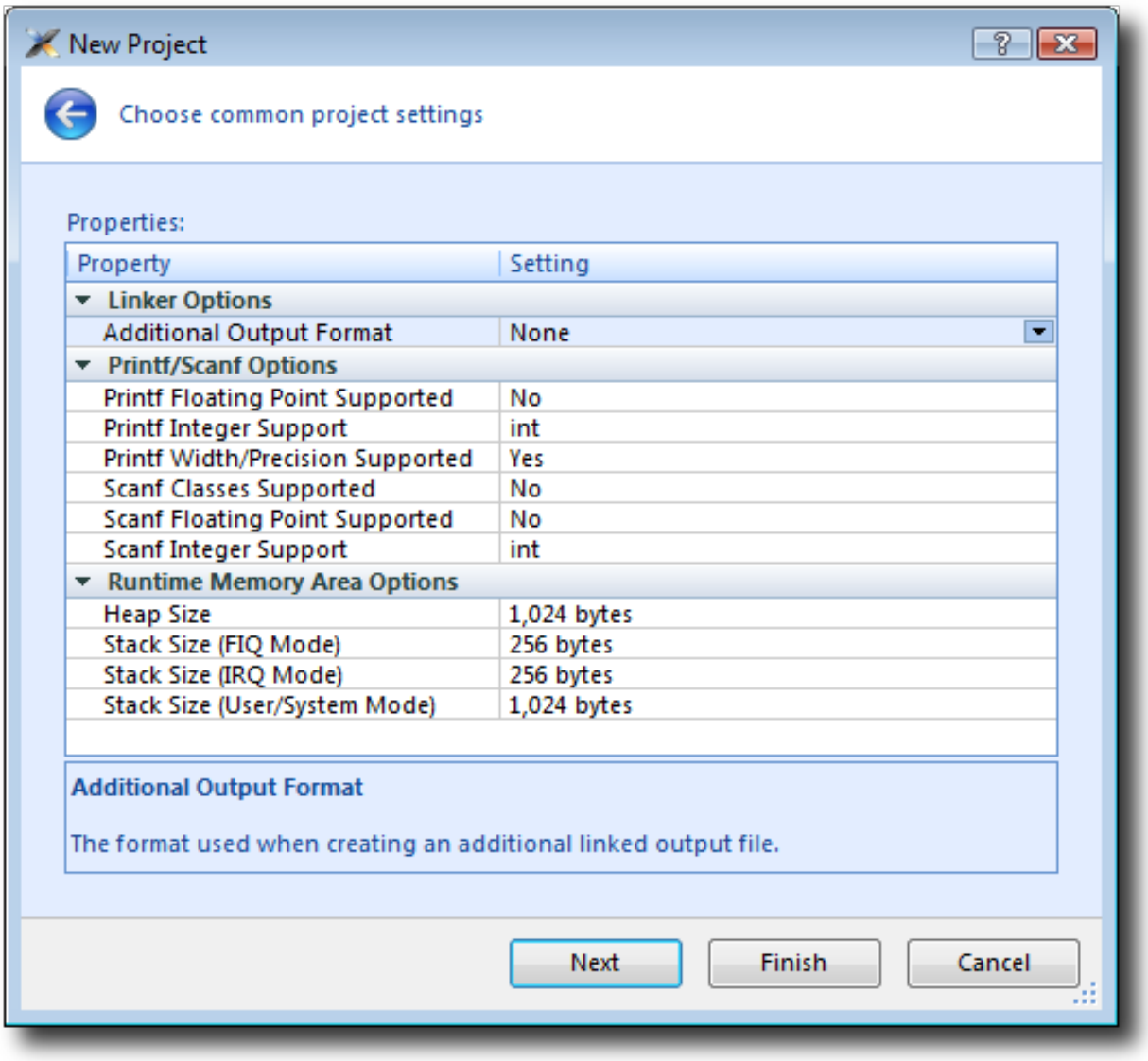
The **New Project** dialog appears. This dialog displays the set of project types (**Categories**) and project templates.



)

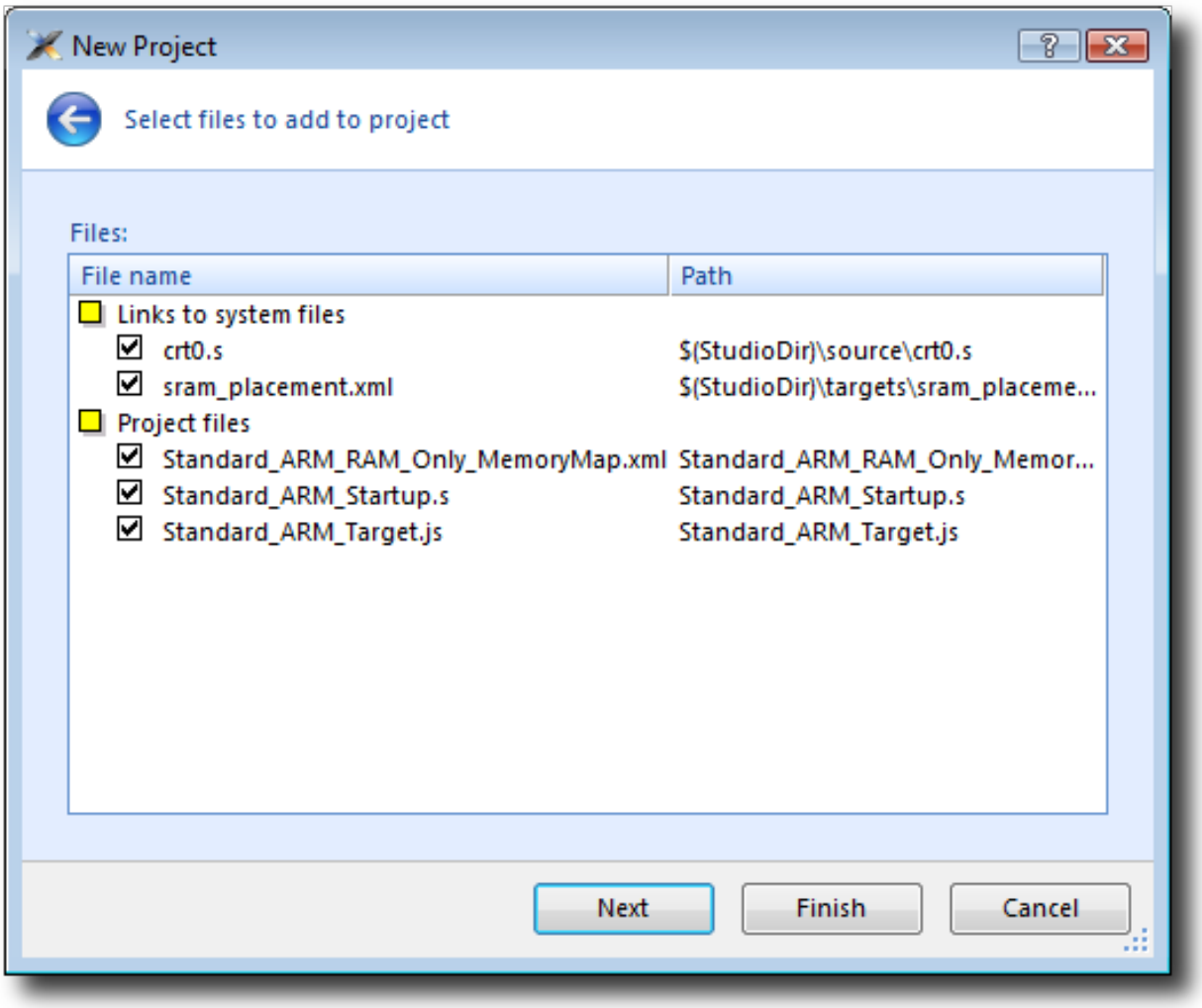
We'll create a project to develop our application in C:

1. In the **Categories** pane, select the **Generic** > **ARM7 Board**
2. From the list in the **Project Templates** pane, select the **An executable for a generic ARM7 processor that supports only running from RAM located at address 0x00000000**
3. In the **Name** text field, type `Tutorial` to assign that as the new project's name.
4. You can use the **Location** text field or the **Browse** button to locate where you want to save the project in your local file system.
5. Click **Next**.



Here you can customize the project by altering a number of common project properties, such as an additional file format to be output when the application is linked and what library support to include if you use **printf** and **scanf**. After the project is created, you can change these settings in the Project Explorer as needed.

1. You can double-click a project property or its value to display either a drop-down menu of potential, valid values or a text field in which you can type arbitrary values. For our tutorial, the default values are fine.
2. Click **Next** to display a list of the files CrossStudio will add to this project be default. You can uncheck any file you plan to add manually or that you know will not be needed.

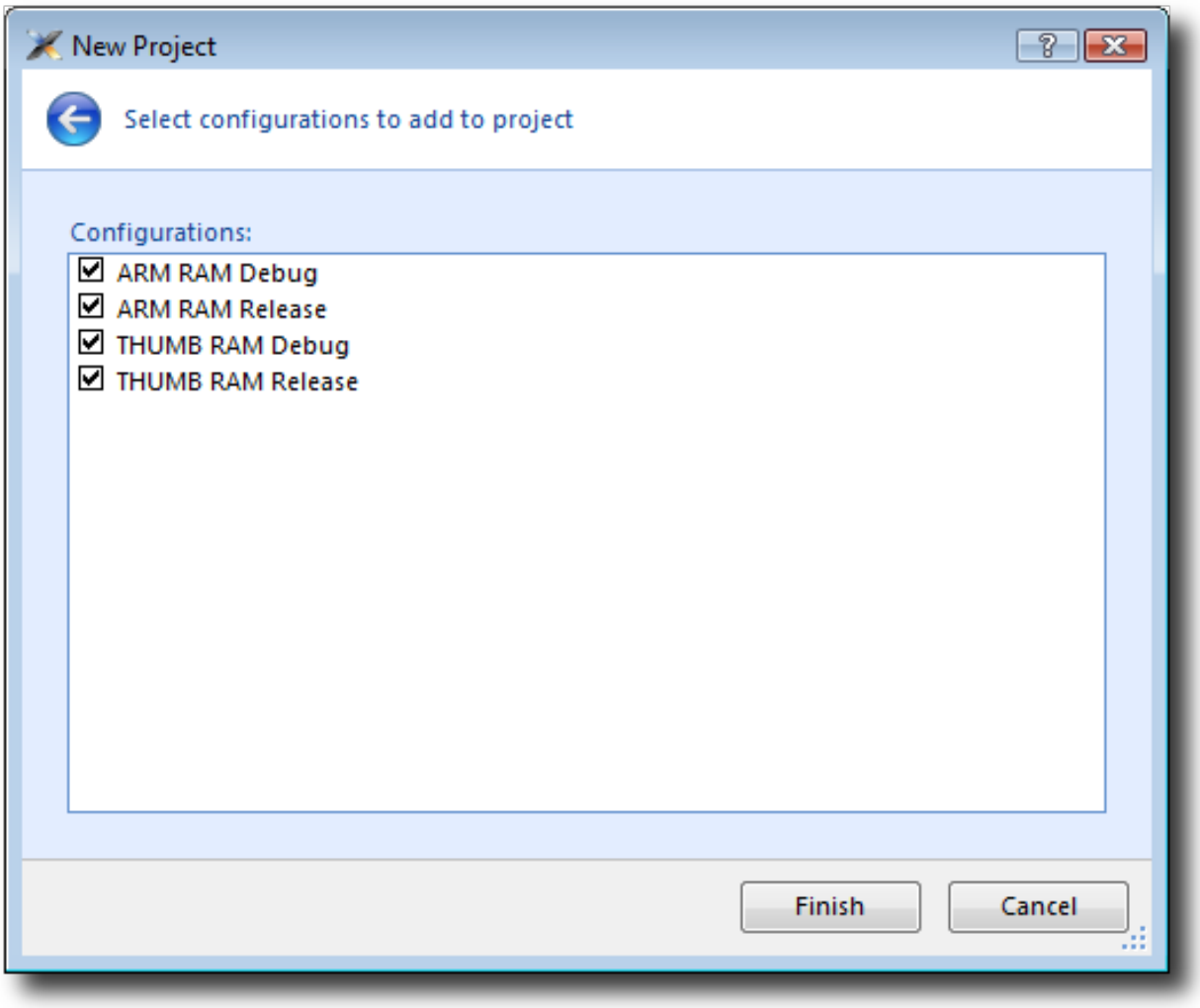


The **Links to system files** group shows the links to CrossStudio system files that will be created in the project. Because these files are links, the default behavior is that they will be shared with other projects—so modifying one will affect all projects containing similar links. To prevent accidental modification, these files are created as read-only. Should you wish to modify a shared file without affecting other projects, first import it into the project. (Importing a shared file will be demonstrated later in this tutorial.) See **Creating and managing projects** for more information on project links.

The **Project files** pane shows the files that will be copied into the project. Because these files are *copied* to the project directory, they can be modified without affecting any other project.

If you uncheck an item, that file is not linked to, or created in, the project. We will leave all items checked for the moment.

1. Click **Next** to view the default configurations that will be added to the project. Again, you can uncheck any you know will not be needed but, for this tutorial, we will leave the defaults unchanged.



Here you can specify the default configurations that will be added to the project. See **Creating and managing projects** for more information on project configurations.

1. Click **Finish** to complete the new project's creation.

This will create a project for a generic ARM 7 device with RAM located at address 0x00000000. This is fine, because we are going to run this example on the simulator. ARM hardware, however, is rarely so accommodating because memory will be mapped at different addresses, target-specific startup code may be required to initialize peripherals, different techniques need to be employed to reset the target, and target-specific loader applications are required to program flash memory. To create a project to run on hardware, you should instead select a template from the project type matching your target—that will create a project with the memory maps, startup code, reset script, and a flash loader for your target.

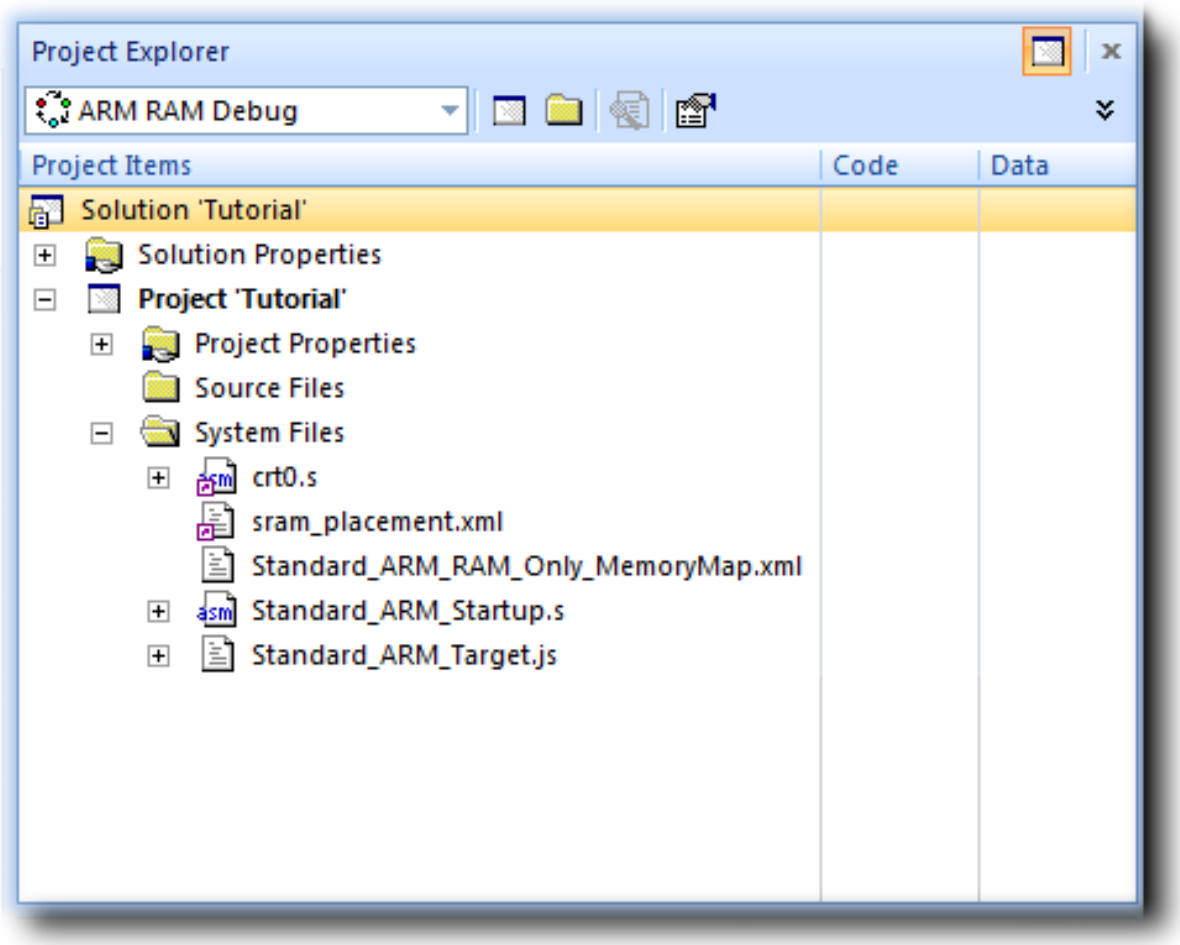
The **Project Explorer** shows the overall structure of your project. To invoke it, do one of the following:

- Choose **View** > **Project Explorer**.

—or—

- Type **Ctrl+Alt+P**.

This is what our project looks like in the **Project Explorer**:



The project name is shown in bold to indicate it is the active project (and, in our case, the only project). If you have more than one project, you can set the active project by using the drop-down box on the **Build** tool bar or by right-clicking the desired project's name in the **Project Explorer** to display the shortcut menu with the **Set as Active Project** command.

The files are arranged into two groups; click the + symbol next to the project name to reveal them:

- **Source Files** contains the main source files for your application, typically header files, C files, and assembly code files. You may want to add files with other extensions or documentation files in HTML format, for instance.
- **System Files** contains links to source files that are not part of the project but are required when the project is built and run. In this case, the system files are: `crt0.s` — the C run-time startup, written in assembly code

`sram_placement.xml` placement file — describes how program sections should be placed in memory segments

`Standard_ARM_RAM_Only_MemoryMap.xml` — a memory map file that describes a target's memory segments

`Standard_ARM_Startup.s` — contains the target-specific start code and exception vectors

`Standard_ARM_Target.js` — contains the target-specific target script that tells the debugger how to reset the target and what to do when the processor stops or starts

Files stored outside the project's home directory (with a small purple shortcut indicator at the bottom left of the icon, as above.

These folders have nothing to do with directories on disk, they are simply a means to group related files in the **Project Explorer**. You can create new folders and specify filters for them based on the project files' extensions; thereafter, when you add a new file to the project, it will be shown in the **Project Explorer** folder whose filter matches the new file's extension.

Managing files in a project

We'll now set up the example project with some files that demonstrate features of the CrossStudio IDE. For this, we will add one pre-prepared file and one new file to the project.

Adding an existing file to a project

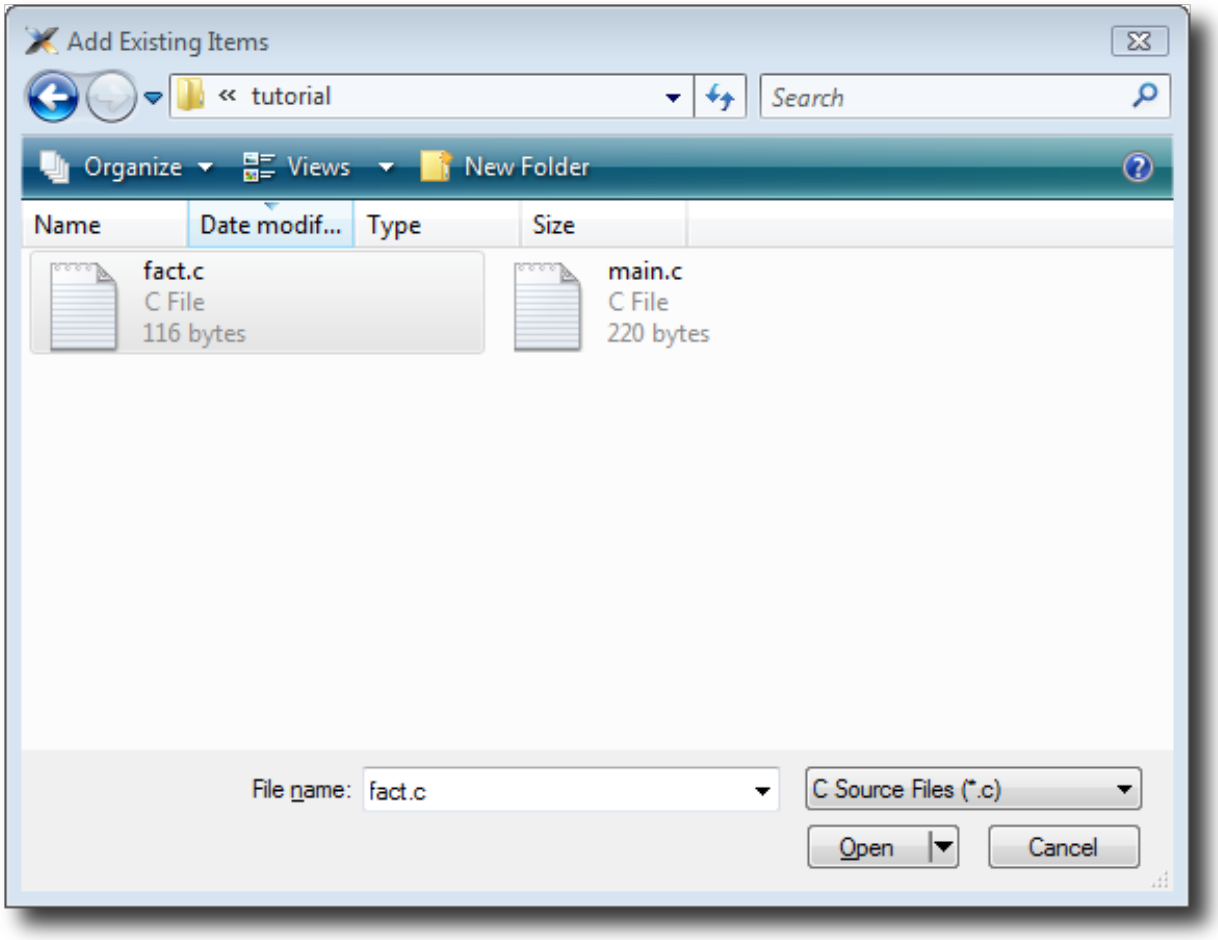
To add one of the existing tutorial files to the project:

- Choose **Project** > **Add Existing File**.

—or—

- In the **Project Explorer**, right-click the Tutorial project node.
- Choose **Add Existing File** from the shortcut menu.

In response, CrossStudio displays a standard file-locator dialog. Use it to navigate to the CrossStudio installation directory, then to the `tutorial` folder, where you should select the `fact.c` file.

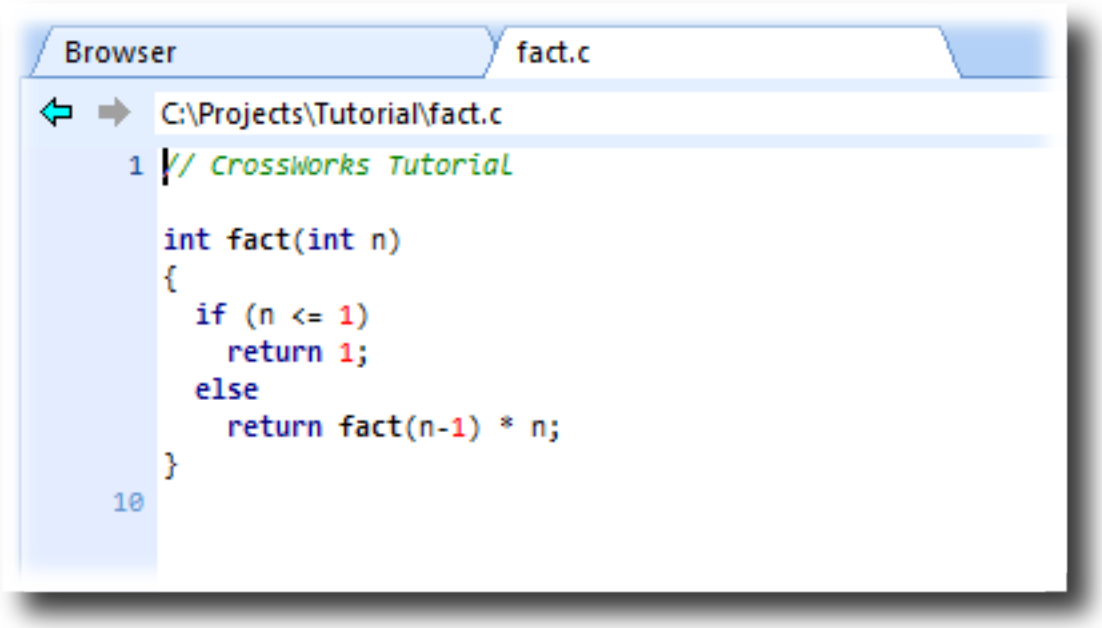


Click **Open** to add the file to the project. The **Project Explorer** will list `fact.c` in the **Project Items**' **Source Files** folder, with a shortcut arrow because the file is not in the project's home directory. Rather than edit the file in the tutorial directory, we'll put a copy of it into the project's home directory:

- In the **Project Explorer**, right-click the `fact.c` node.
- From the pop-up menu, click **Import**.

The shortcut arrow disappears from the `fact.c` node, indicating that our working version of that file is now in our Tutorial project's home directory.

We can open a file for editing by double-clicking the node in the **Project Explorer**. For example, double-clicking `fact.c` opens it in the code editor:



Adding a new file to a project

Our project isn't complete, because `fact.c` is only part of an application. To our project we'll add a new C file that will contain the **main()** function. To add a new file to the project, do the following:

- Choose **File** > **New** to open the **New File** dialog.

—or—

- On the **Project Explorer** tool bar, click the **Add New File** button.

—or—

- In the **Project Explorer**, right-click the Tutorial node.
- Choose **Add New File** from the shortcut menu.

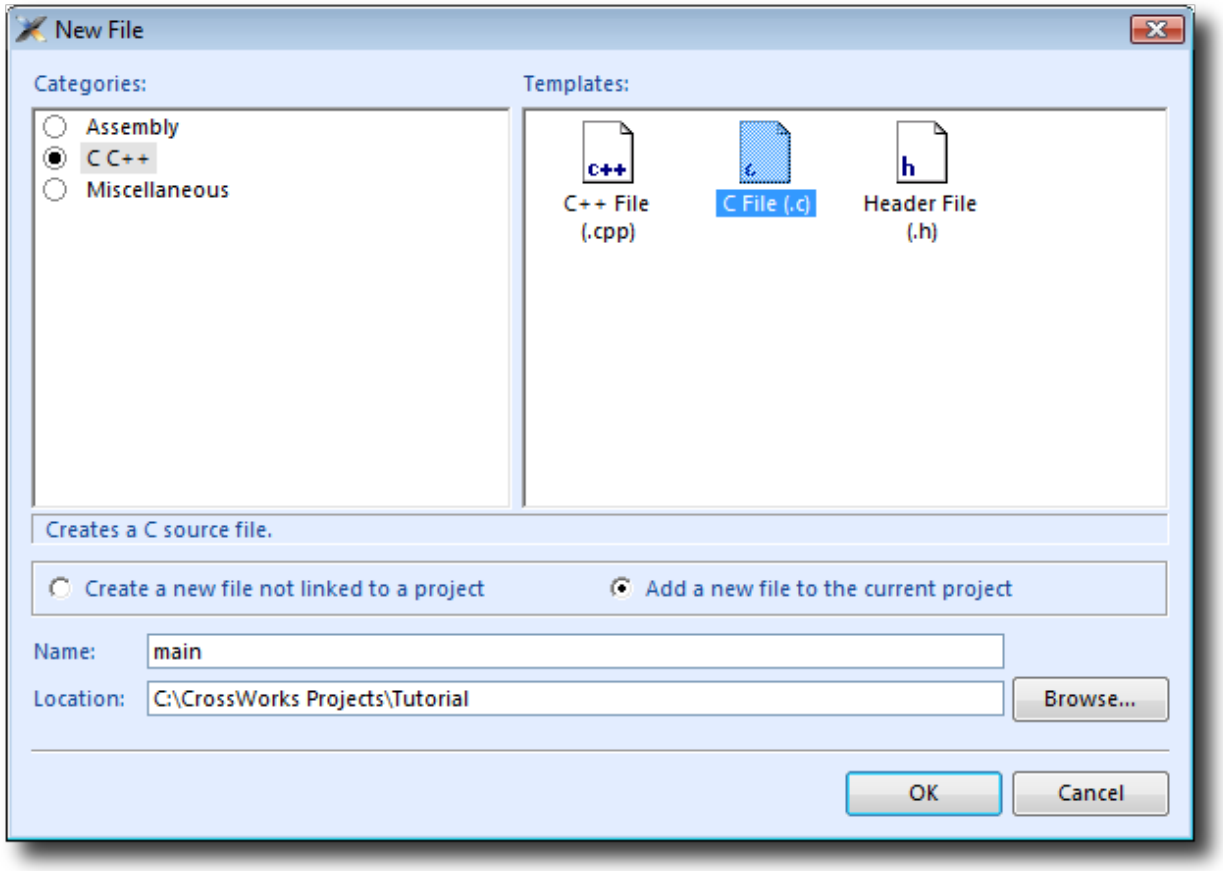
—or—

- Type **Ctrl+N**.

The **New File** dialog appears.

- In the **Categories** pane, select **C C++** to indicate the general type of file.
- In the **Templates** pane, select the **C File (.c)** option to further specify the kind of file we will be adding.
- In the **Name** edit box, type `main`.

The dialog box will now look like this:

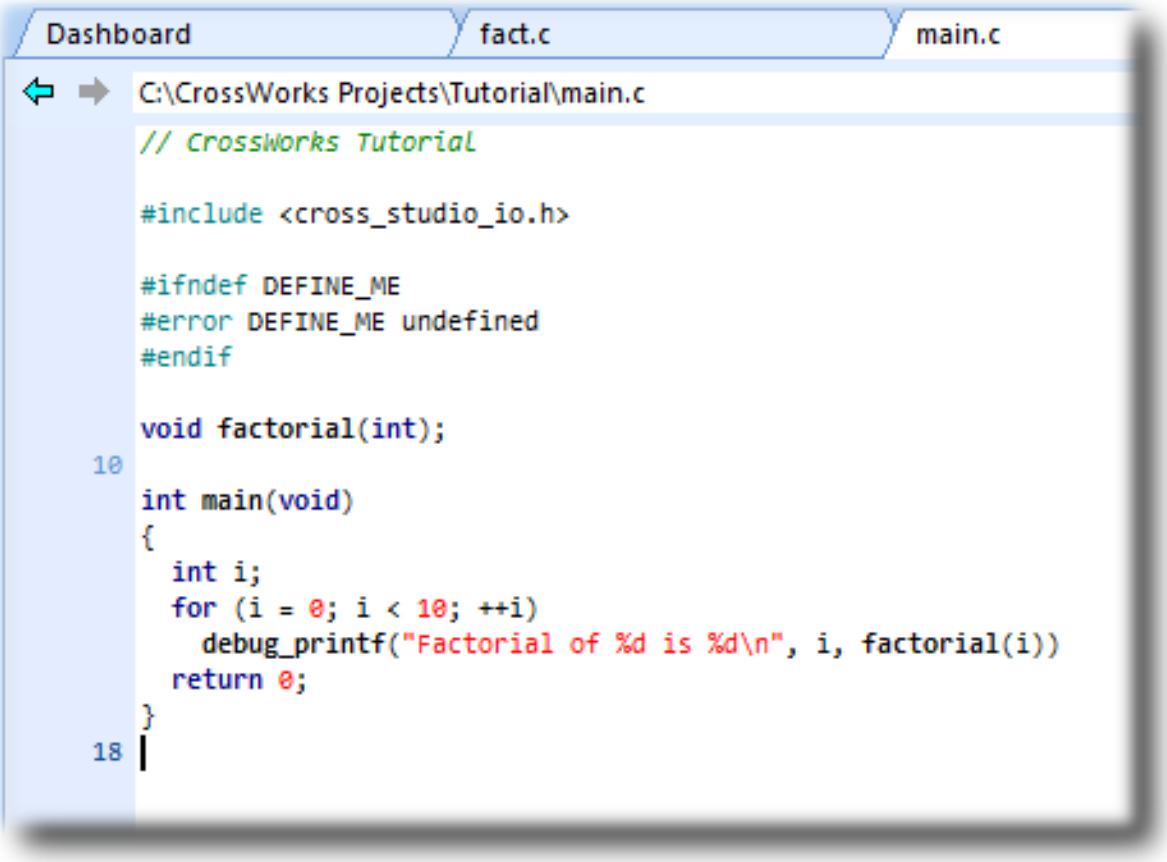


Click **OK** to add the new file.

CrossStudio opens the new file in the code editor. Rather than type the program from scratch, we'll add it from a file stored on disk. With the new, empty `main.c` in the foreground:

- Choose **Edit** > **Insert File** or press **Ctrl+K, Ctrl+I**.
- Using the resulting file-selection dialog, navigate to the `tutorial` directory.
- Select the `main.c` file.
- Click **OK**.

Your `main.c` file should now look like this:



Next, we'll set up some project options.

Setting project options

Up to this point, you have created a simple project. In this section, we will set some options for that project.

You can set project options on any node of a solution. That is, you can set options on a solution-wide basis, on a project-wide basis, on a project-group basis, or on an individual-file basis. For instance, options you set on a solution are inherited by all projects in that solution, by all groups in each of those projects, and by all files in each of those groups. If you set an option further down in the hierarchy, that setting will be inherited by nodes that are children of (or grandchildren of, etc.) that node. This provides a powerful way to customize and manage your projects.

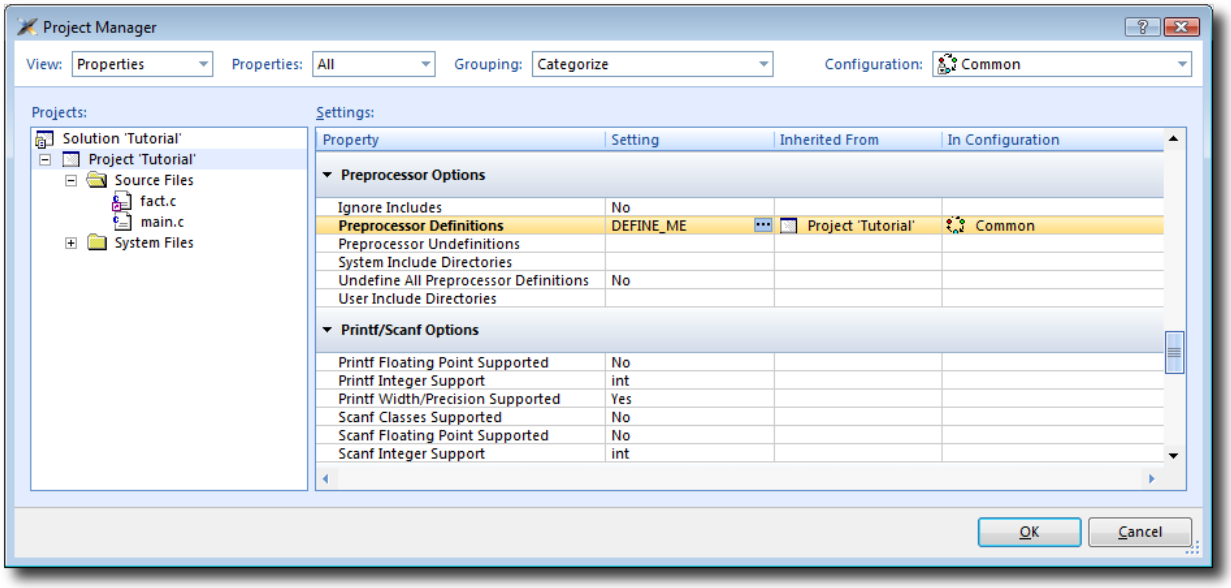
Adding a C preprocessor definition

In this instance, we will define a C preprocessor definition that will apply to the entire Tutorial *project*. This means every file in the project will inherit our new definition. If, however, we were to later add other projects to the solution, they would not inherit the definition; if we wanted that, we could set the property on the solution node rather than the project node.

To set a C preprocessor definition on the project node:

- Right-click the Tutorial project in the **Project Explorer** and select **Properties** from the menu—the **Project Manager** dialog appears.
- Click the **Configuration** drop-down and change to the **Common** configuration (it is one of the "Private Configurations").
- Scroll down the list as necessary to click the **Preprocessor Options** > **Preprocessor Definitions** property. Double-click the property name or value field, or click the `...` symbol to display the empty **Preprocessor Definitions** window, and in that window type the definition `DEFINE_ME`.

The dialog box will now look like this:



Notice that, when you change between **Debug** and **Release** configurations, the code generation options change. This dialog shows the options used when building a project (or anything in a project) in a given configuration. Because we put the above, new definition in the **Common** configuration, both **Debug** and **Release** configurations will use this setting. We could, however, set the definition to be different in **Debug** and **Release** configurations if we wanted to pass different definitions into debug and release builds.

Now click **OK** to accept the changes made to the project.

Using the Properties Window

If you click on the project node, the **Properties Window** will show the properties of the project—all were inherited from the solution. If you modify a property when the project node is selected, you'll find that its value is highlighted because you have overridden the property value inherited from the solution. To restore the inherited value of a property that was changed, right-click the property and select **Use Inherited Value**.

Next, we'll build the project.

Building projects

Now that the project is created and set up, it's time to build it. There are some deliberate errors in the program that we need to correct; doing that is the next step in this tutorial.

Setting the build configuration

The first thing to do is set the active build configuration you want to use:

- Select **ARM RAM Debug** from the Active Configuration .

This means we are going to use a build configuration that generates ARM code, will run from RAM, and generates code with debug information and no optimization, so it can be debugged. If we wanted to produce production code with no debug information and optimization enabled, we could use the **ARM RAM Release** configuration. However, because we are going to use the debugger, we shall use the **ARM RAM Debug** configuration.

Building the project

To build the project:

- Choose **Build** > **Build Tutorial**.

—or—

- On the **Build** tool bar, click the **Build Active Project** button.

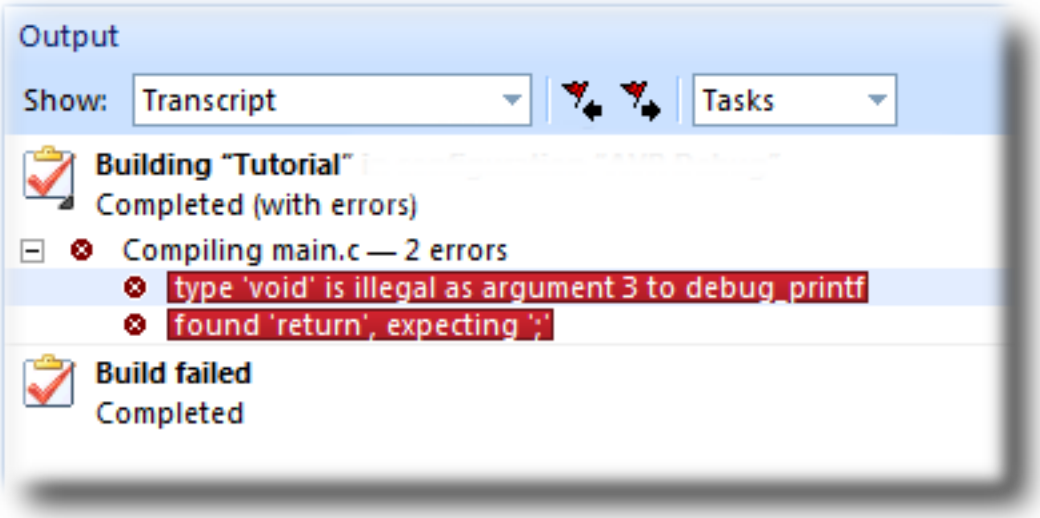
—or—

- Type **F7**.

Alternatively, to build the Tutorial project using a shortcut menu:

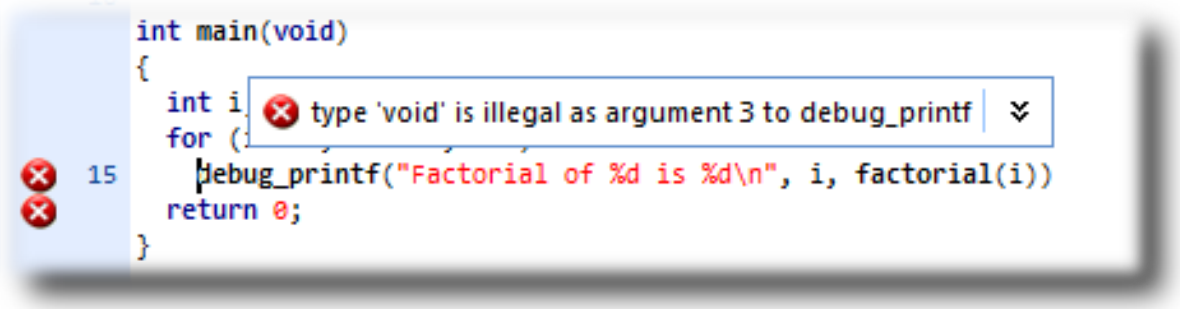
- In the **Project Explorer**, right-click the Tutorial project node.
- Select **Build** from the shortcut menu.

CrossStudio starts compiling the project files, but stops after detecting an error. The **Output** window shows the Transcript, which contains the errors found in the project:



Correcting compilation and linkage errors

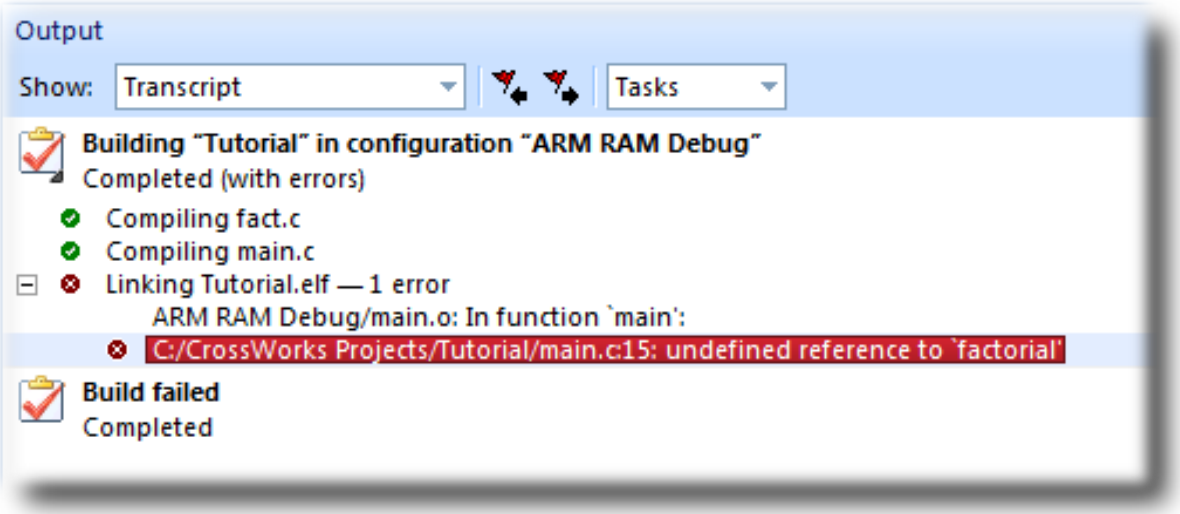
The file `main.c` contains two errors. After compilation, CrossStudio moves the cursor to the line containing the first reported error and displays an error message in the **Output** window. (You can change this behavior by modifying the **Text Editor** > **Editing Options** > **Enable Popup Diagnostics** environment option using the **Tools** > **Options** dialog.)



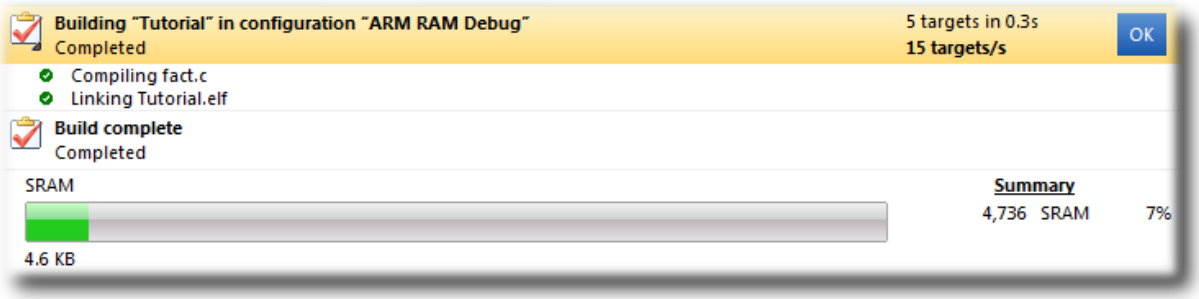
To correct the error, change the return type of `factorial` from `void` to `int` in its prototype.

To move the cursor to the line containing the next error, type **F4** or choose **Search** > **Next Location**. The cursor is now positioned at the **debug_printf** statement, which is missing a terminating semicolon—add the semicolon to the end of the line. Using **F4** again reveals that we have corrected all errors.

Pressing **F4** again wraps around and moves the cursor to the first error, and you can use **Shift+F4** or **Search** > **Previous Location** to move back through errors. Now that the errors are corrected, build the project again by pressing **F7**. The Transcript shows there still is a problem.



The remaining error is a linkage error. Double-click `fact.c` in the **Project Explorer** to open it for editing and change the two occurrences of `fact` to `factorial`. Rebuild the project—this time, the project compiles correctly:



A summary of the memory used by the project is displayed at the end of the build log. The results for your application may be different, so don't worry if they don't match.

In the next sections, we'll explore the characteristics of the newly built project.

Exploring projects

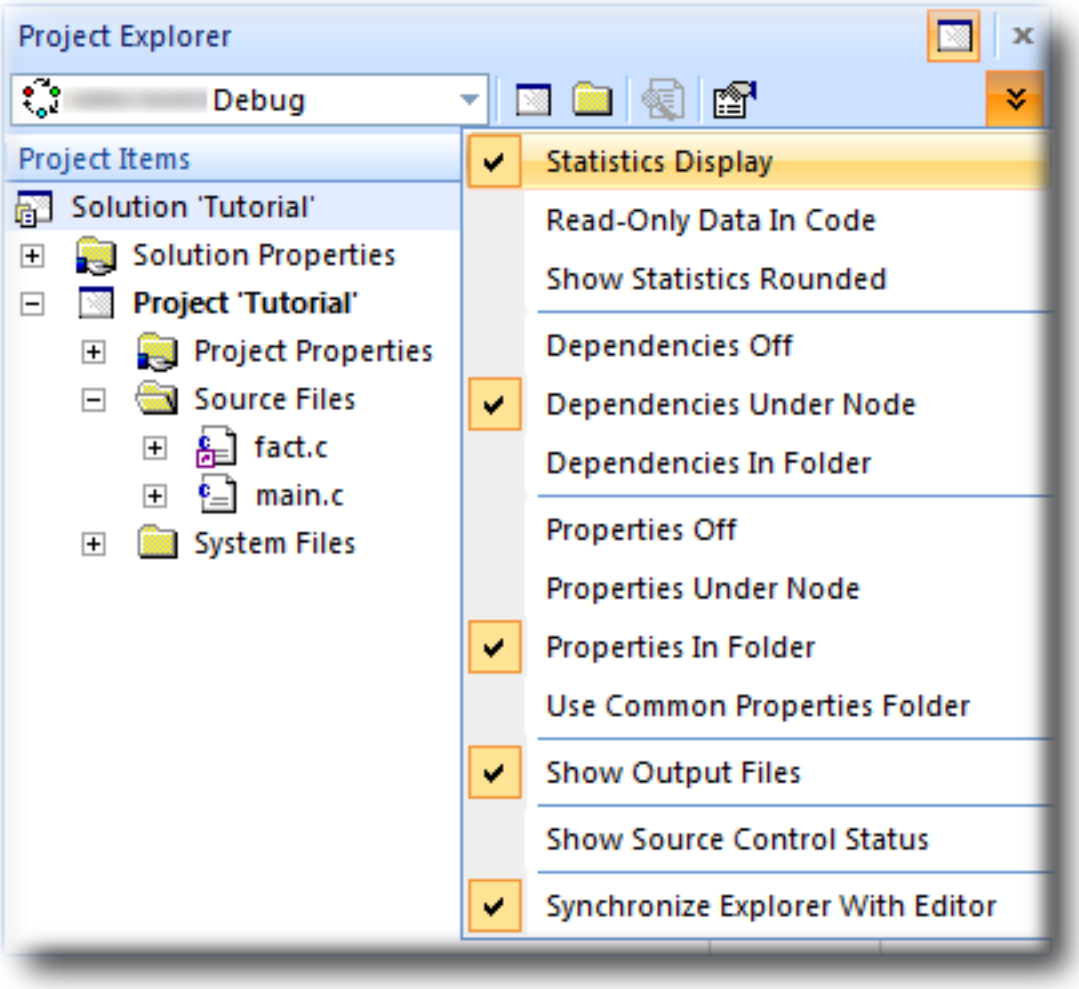
Now that the project has no errors and builds correctly, we can turn our attention to uncovering exactly how our application fits in memory and how to navigate around it.

Using Project Explorer features

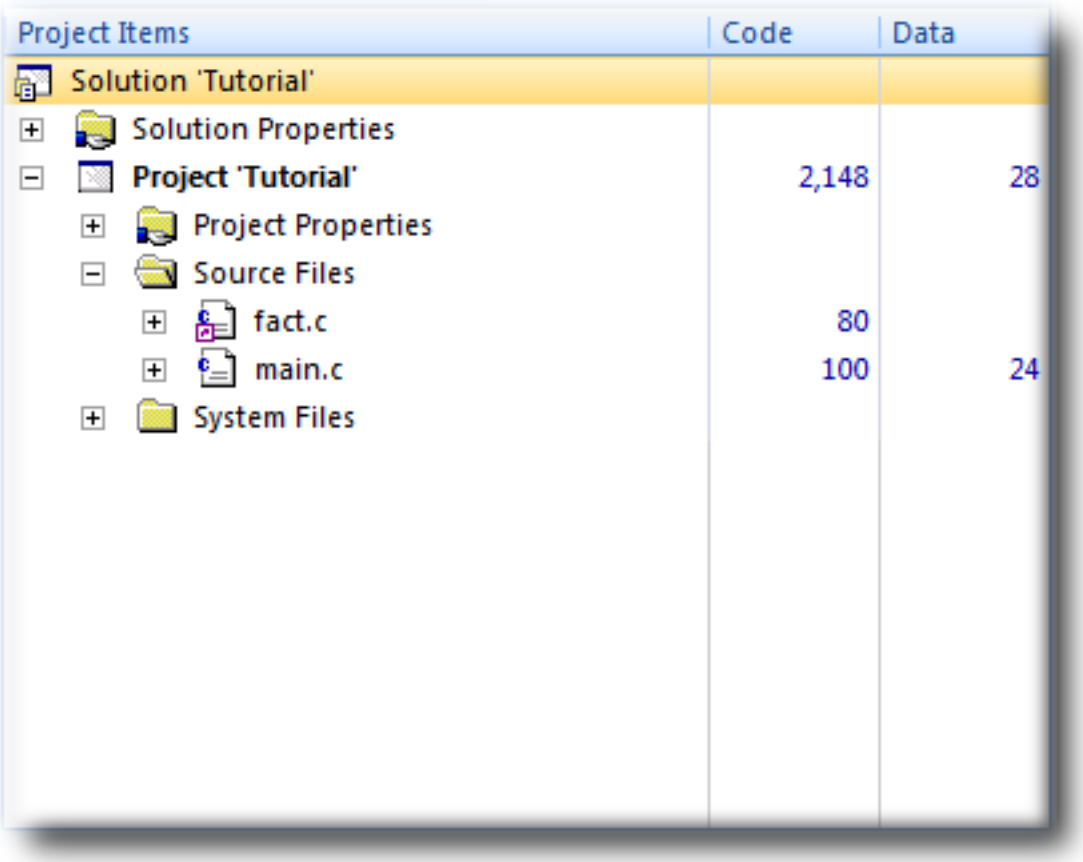
The **Project Explorer** is the central focus for arranging your source code into projects, and it's a good place to show ancillary information gathered when CrossStudio builds your applications. This section will cover features the **Project Explorer** offers to give you an overview of your project.

Project code and data sizes

Developers are always interested in how much memory their applications use, especially when they are working with small, embedded microcontrollers. The **Project Explorer** can display the code and data sizes for each project and individual source file that successfully compiled. To view this information, use the **Options** pop-up menu on the **Project Explorer** tool bar to ensure that **Statistics Column** is checked.



When the **Statistics Column** option is checked, the **Project Explorer** displays two additional columns, **Code** and **Data**.

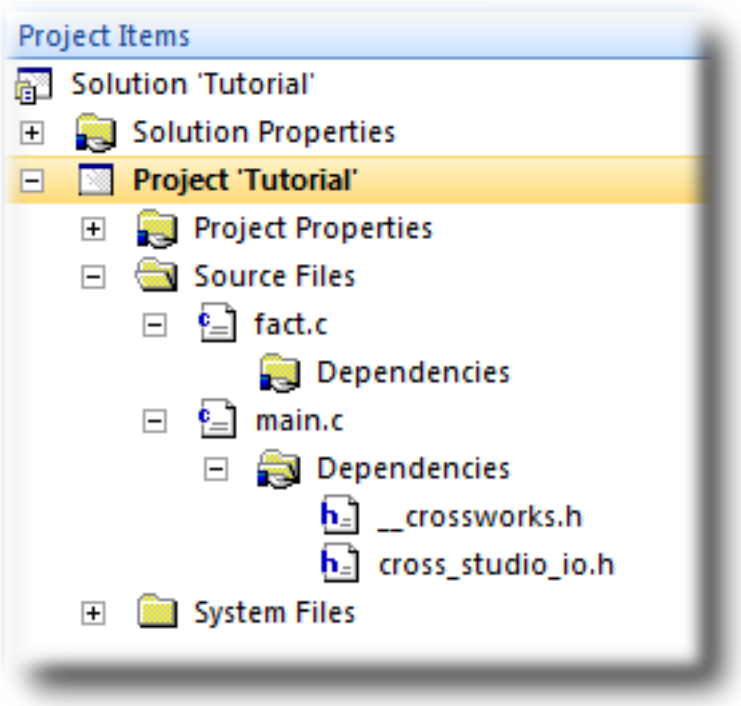


The **Code** column displays the total code space required for the project. The **Data** column displays the total data space required. The code and data sizes shown for each C and assembly source file are *estimates*, but good ones. Because the linker removes any unreferenced code and data, and performs a number of optimizations, the sizes for the linked project may not be the sum of the sizes of each individual file. The code and data sizes for the project, however, *are* accurate. As already mentioned, your numbers may not match these exactly.

Dependencies

The **Project Explorer** is very versatile: not only can you display the code and data sizes for each element of a project and for the project as a whole, you can also configure it to show the *dependencies* for a file. As part of the compilation process, CrossStudio finds and records the relationships between files—that is, it finds which files depend upon other files. CrossStudio uses these known relationships when it builds the project again, to minimize the amount of work required to bring the project up to date.

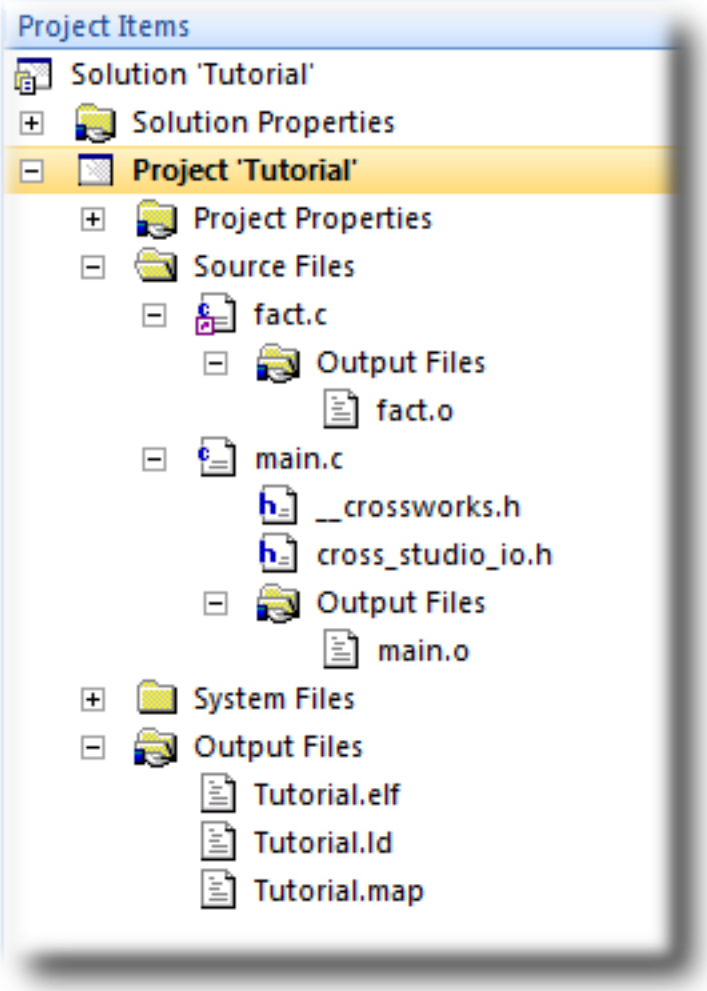
To show the dependencies for a project, use the **Options** button on the **Project Explorer** tool bar to ensure that either **Dependencies Under Node** or **Dependencies In Folder** is checked. Once checked, dependent files are shown as sub-nodes of the file that depends on them.



In this case, `main.c` is dependent upon `cross_studio_io.h` because it includes it with an `#include` directive. It is also dependent on `__crossworks.h` because that is included by `cross_studio_io.h`. You can open the files in an editor by double-clicking them, so having dependencies turned on is an effective way of navigating to and summarizing the files a source file includes.

Output files

It is useful to know the output files when compiling and linking the application, and CrossStudio can display this information, too. To turn on output-file display, click the **Project Explorer** tool bar's **Options** button and verify that **Output Files Folder** option is checked in the menu. Once checked, output files are shown in an **Output Files** folder under the node that generates them. Click that folder's + symbol to expand the view of the output files.



In the above figure, we can see that the files `fact.o` and `main.o` are object files, produced by compiling their corresponding source files. The linker script `Tutorial.ld`, the map file `Tutorial.map`, and the linked executable `Tutorial.elf` are produced by the linker. As a convenience, double-clicking an object file or a linked executable file in the **Project Explorer** will open an editor showing the disassembled contents of the file.

Disassembling a project or file

You can disassemble a project either by double-clicking the corresponding file in the **Project Explorer**, as described above, or by using the **Disassemble** tool.

To disassemble a project or file:

- Right-click the appropriate project or file in the **Project Explorer**.
- From the shortcut menu, choose **Disassemble**.

CrossStudio then opens a new read-only editor showing the disassembled listing. If you change your project and rebuild it, thereby causing a change in the object or executable file, the disassembly updates to keep the display's contents synchronized with the file on disk.

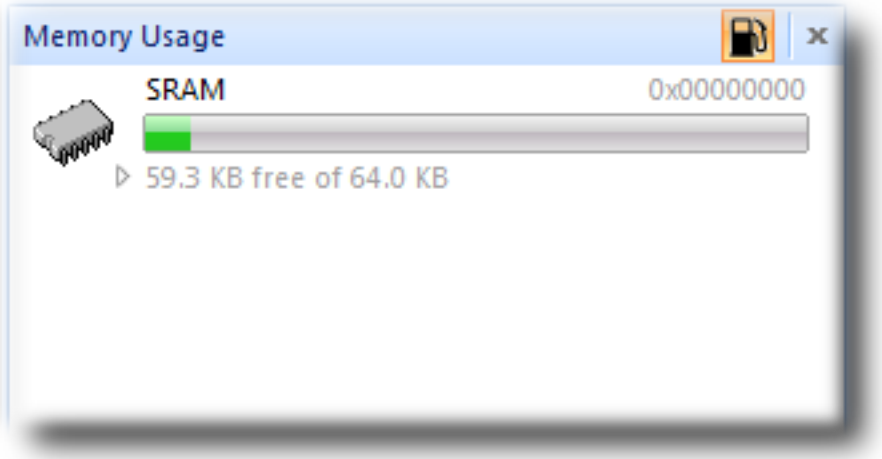
Using Memory Usage Window features

The **Memory Usage** window can be used to view a graphical summary of how memory was used in each memory segment of a linked application.

To display the memory usage:

- Choose **View** > **Memory Usage** or press **Ctrl+Alt+Z**.

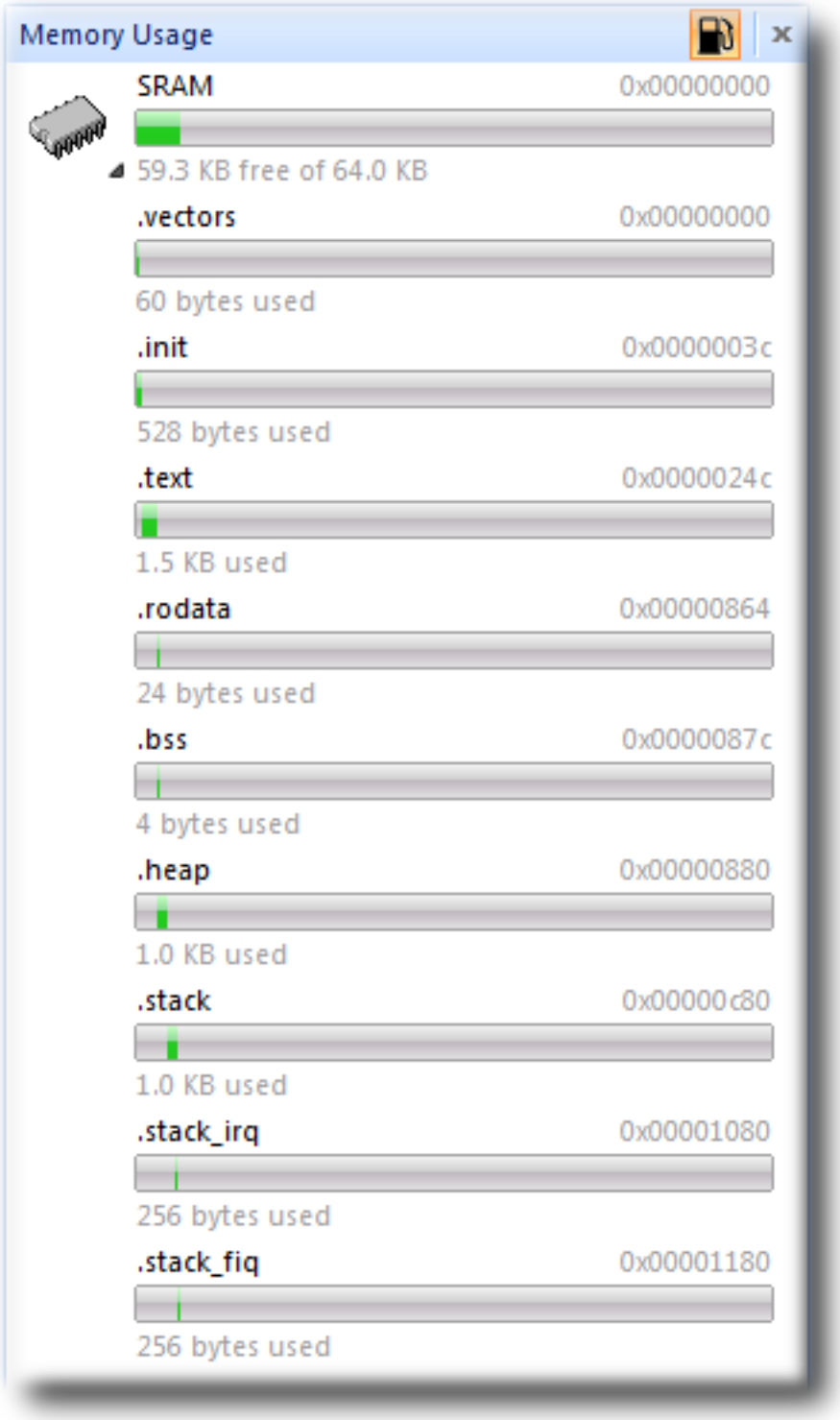
For the Tutorial project, the **Memory Usage** window shows this:



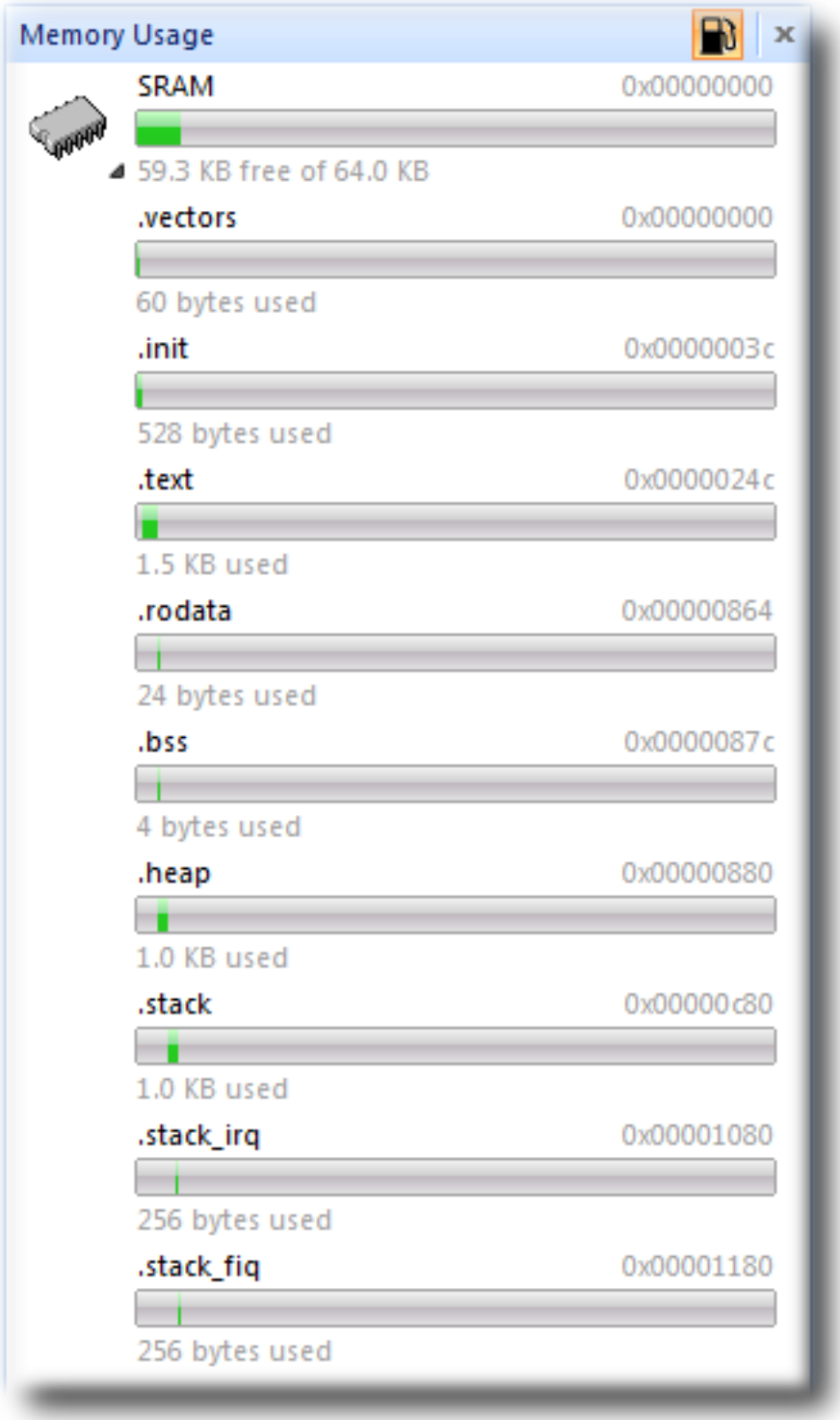
From this, you can see:

- The **SRAM** segment is located at 0x00000000.
- The **SRAM** segment is 64KB in length.
- There is 59.3KB of unused memory in the **SRAM** segment.

If you expand the **SRAM** segment by clicking it, CrossStudio will display all the program sections contained within the segment:



If you expand the **FLASH** segment, CrossStudio will display the program sections contained within the segment:



From this, you can see that the the **CODE** section is located at @(__DOCGEN_CODE_START) and is @(__DOCGEN_CODE_SIZE) bytes in length.

Using Symbol Browser features

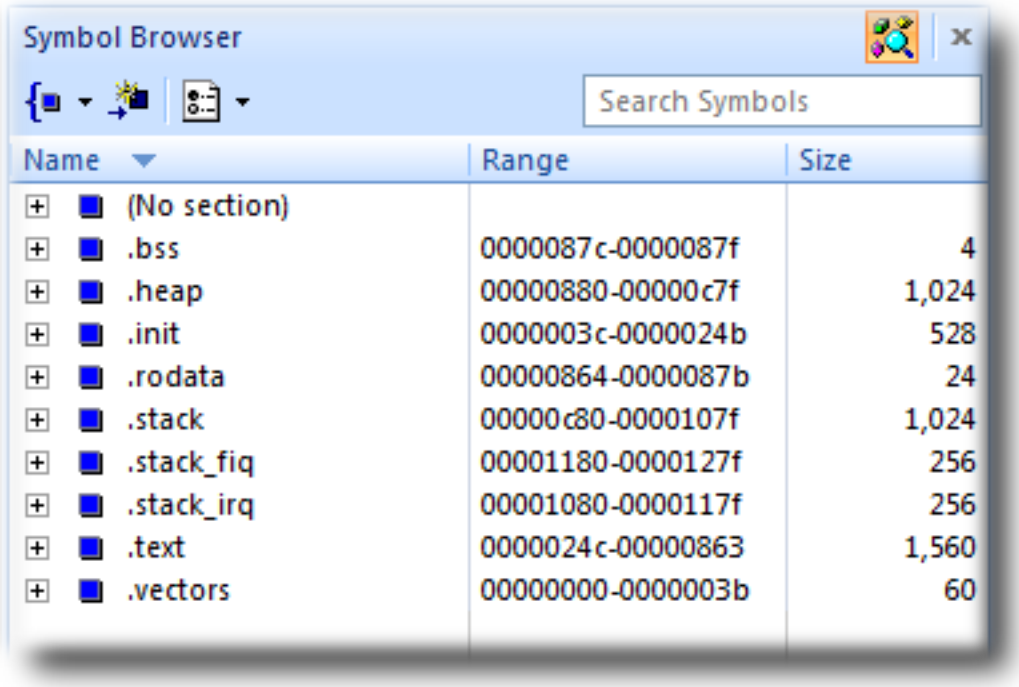
For a more-detailed view of how your application is laid out in memory than the **Memory Usage** window provides, you can use the **Symbol Browser**. It allows you to navigate your application, see which data objects and functions have been linked into your application, what their sizes are, which section they are in, and where they are placed in memory.

To activate the Symbol Browser:

- Choose **Tools** > **Symbol Browser** or press **Ctrl+Alt+Y**.

Drilling down into the application

The Tutorial project shows this in the **Symbol Browser**:

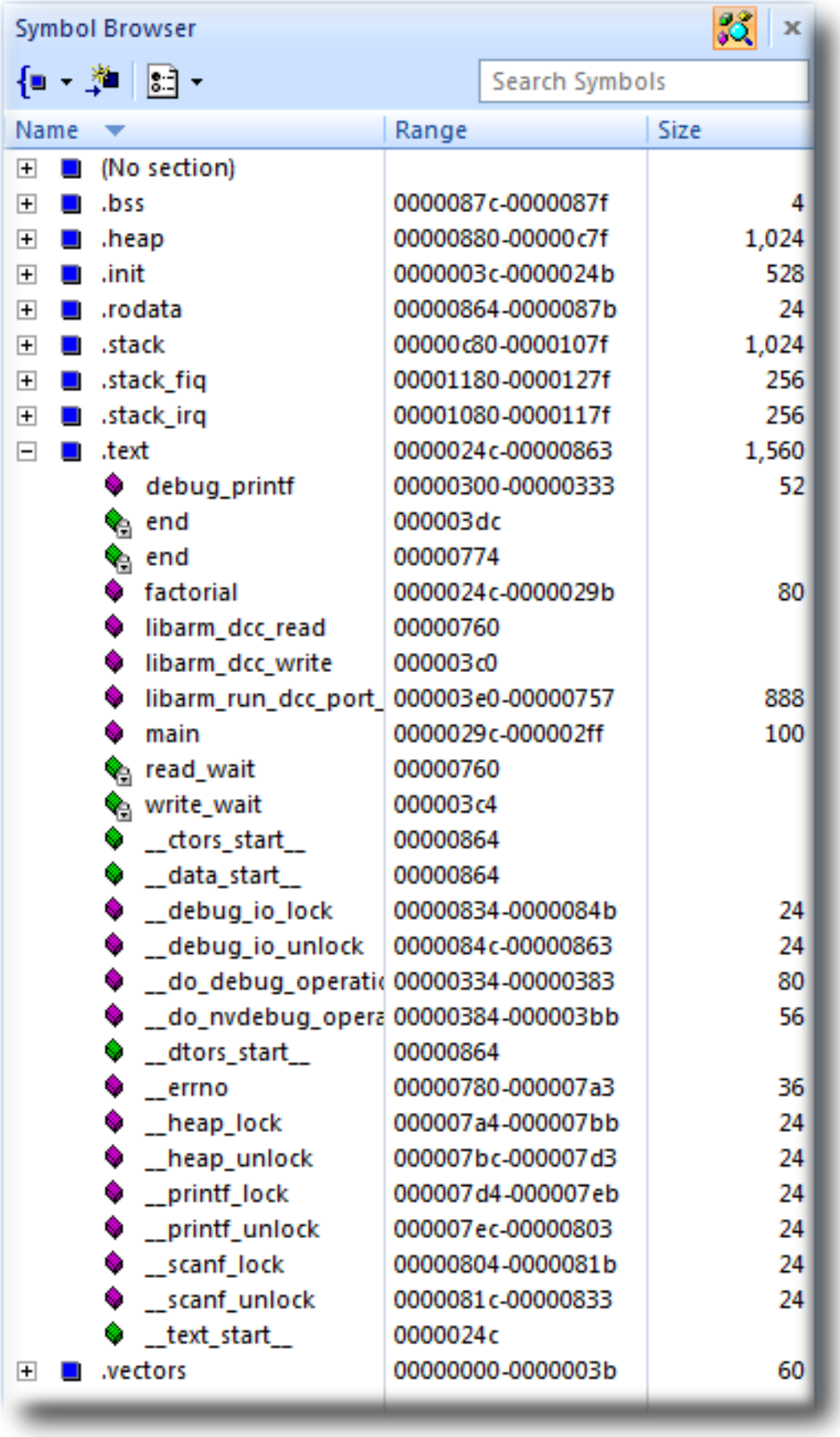


From this, you can see sections and their sizes. For example, the **.vectors** section containing the ARM exception vectors is placed in memory between address 0x00000000 and 0x0000003B.

- The **.init** section containing the system startup code is placed in memory
- The **.text** section containing the program code is placed in memory
- The **.rodata** section containing read-only data is placed in memory
- The **.heap** section is 1024 bytes in length and is located at 0x00000880. **Linker** > **Heap Size** project property.

- The **.stack** section which contains the User/System mode stack is 1024 **Linker** > **Stack Size** properties.
- The **.stack_irq** section which contains the IRQ mode stack is 256 bytes in
- The **.stack_fiq** section which contains the FIQ mode stack is 256 bytes in

To drill down, open the **CODE** node by double-clicking it: CrossStudio displays the individual functions that have been placed in memory and their sizes:


Symbol Browser
Search Symbols
Name
Range
Size
(No section)
.bss 0000087c-0000087f 4
.heap 00000880-00000c7f 1,024
.init 0000003c-0000024b 528
.rodata 00000864-0000087b 24
.stack 00000c80-0000107f 1,024
.stack_fiq 00001180-0000127f 256
.stack_irq 00001080-0000117f 256
.text 0000024c-00000863 1,560
debug_printf 00000300-00000333 52
end 000003dc
end 00000774
factorial 0000024c-0000029b 80
libarm_dcc_read 00000760
libarm_dcc_write 000003c0
libarm_run_dcc_port_ 000003e0-00000757 888
main 0000029c-000002ff 100
read_wait 00000760
write_wait 000003c4
__ctors_start__ 00000864
__data_start__ 00000864
__debug_io_lock 00000834-0000084b 24
__debug_io_unlock 0000084c-00000863 24
__do_debug_operati 00000334-00000383 80
__do_nvdebug_opera 00000384-000003bb 56
__dtors_start__ 00000864
__errno 00000780-000007a3 36
__heap_lock 000007a4-000007bb 24
__heap_unlock 000007bc-000007d3 24
__printf_lock 000007d4-000007eb 24
__printf_unlock 000007ec-00000803 24
__scanf_lock 00000804-0000081b 24
__scanf_unlock 0000081c-00000833 24
__text_start__ 0000024c
.vectors 00000000-0000003b 60

Here, we can see that **main** is 100 bytes in size and is placed in memory between addresses 0000029C and 000002FF, inclusive, and that **factorial** is 80 bytes and occupies addresses 0000024C through 0000029B. Just as in the **Project Explorer**, if you double-click a function, CrossStudio moves the cursor to the line containing the definition of that function, so you can easily use the **Symbol Browser** to navigate around your application.

Printing Symbol Browser contents

You can print the contents of the **Symbol Browser** by selecting its window and choosing **Print** from the **File** menu, or **Print Preview** if you want to see what it will look like before printing. CrossStudio prints only the columns you have selected for display, and prints items in the order displayed in the **Symbol Browser**, so you can choose which columns to print and how to print symbols by configuring the **Symbol Browser** display.

We have touched on only some of the features the Symbol Browser offers; to learn more, refer to **Symbol Browser**, where it is described in detail.

Using the debugger

Our sample application, which we have just compiled and linked, is now built and ready to run. In this section, we'll concentrate on downloading and debugging this application, and on using the features of CrossStudio to see how it performs.

Getting set up

Before running your application, you need to select the target to run it on. Choose **Target** > **Targets** to list in the **Targets** window each target interface that is defined. You will use these to connect CrossStudio to a target. For this tutorial, you'll be debugging on the simulator, not hardware, to simplify matters.

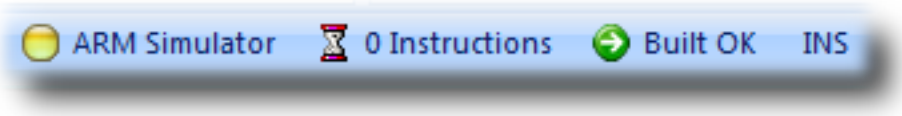
To connect to the simulator:

- Choose **Target** > **Connect** > **ARM Simulator**.

—or—

- Choose **View** > **Targets** to activate the **Targets** window.
- In the **Targets** window, double-click **ARM Simulator**.

After connecting, the ARM Simulator target is shown in the status bar:



The color of the target-status LED in the status bar changes according to what CrossStudio and the target are doing:

- **White** — No target is connected.
- **Yellow** — Target is connected.
- **Solid green** — Target is free running, not under control of CrossStudio or the debugger.
- **Flashing green** — Target is running under control of the debugger.
- **Solid red** — Target is stopped at a breakpoint or because execution is paused.
- **Flashing red** — CrossStudio is programming the application into the target.

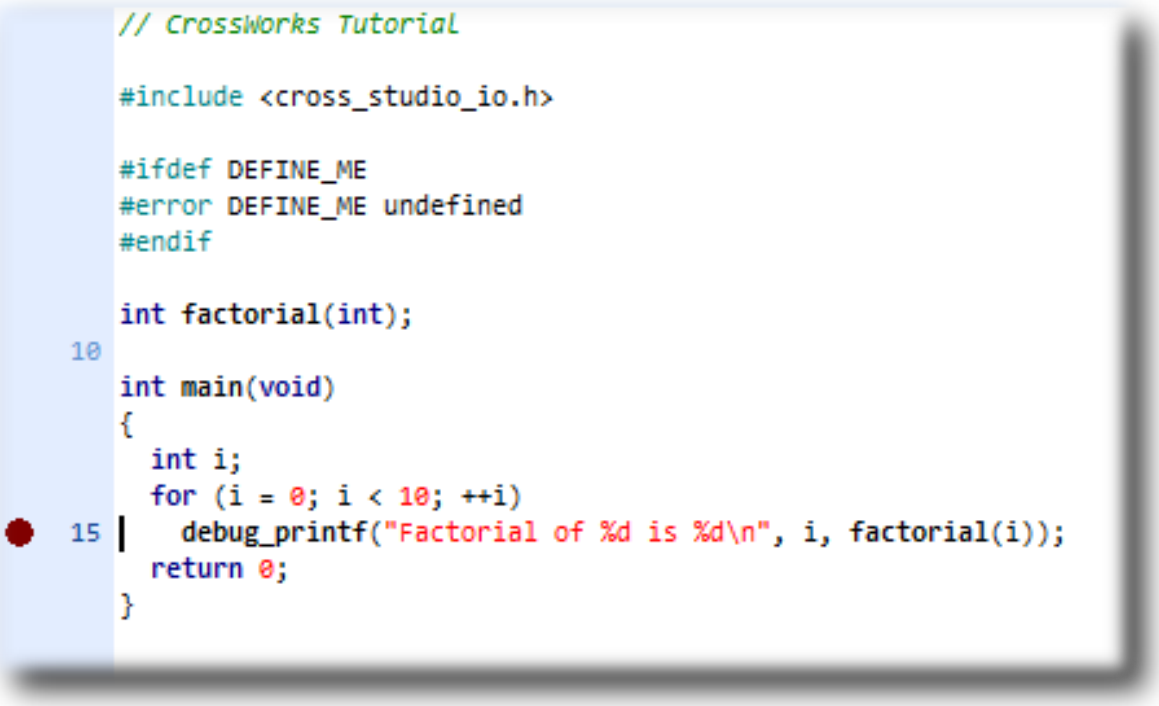
Double-clicking the **Target Status** will show the **Targets** window, if it is not already visible.

The core simulator target can accurately count the cycles spent executing your application, so the status bar shows a cycle counter. If you connect a target that cannot provide performance information, the cycle counter panel is hidden. Double-clicking the **Cycle Counter** panel will reset the cycle counter to zero.

Setting a breakpoint

CrossStudio will run a program until it hits a breakpoint. We'll place a breakpoint on the call to `debug_printf` in `main.c`. To set the breakpoint, move the cursor to the line containing `debug_printf` and Choose **Debug** > **Toggle Breakpoint** or press **F9**.

Alternately, you can set a breakpoint without changing the cursor's position by clicking in the gutter of the line to set the breakpoint on.

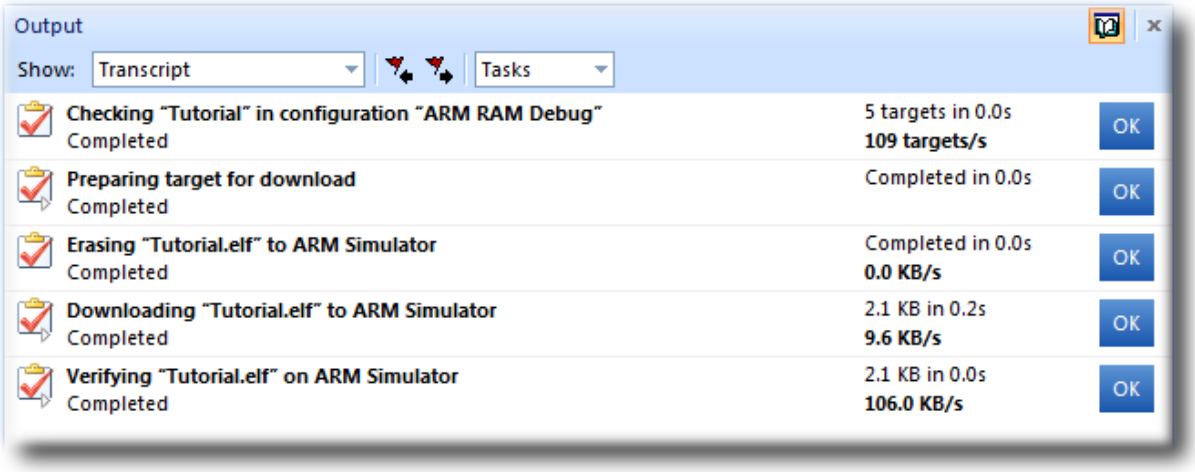


The gutter displays an icon on lines where breakpoints are set. The **Breakpoints** window updates to show where each breakpoint is set and whether it's set, disabled, or invalid—you can find more detailed information in the Breakpoints window section. The breakpoints you set are stored in a session file associated with the project, so your breakpoints are remembered if you exit and re-run CrossStudio.

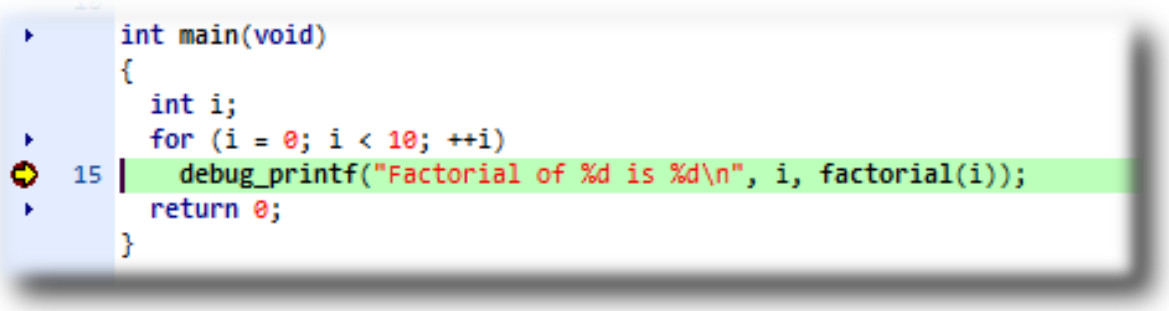
Starting the application

To start the application, Choose **Debug** > **Start** or press **F5**.

The workspace will change from the standard Editing workspace to the Debugging workspace. You can choose which windows to display in each of these workspaces and manage them independently. CrossStudio loads the active project into the target and places the breakpoints you have set. During loading, the **Target Log** in the **Output Window** shows its progress and any problems:

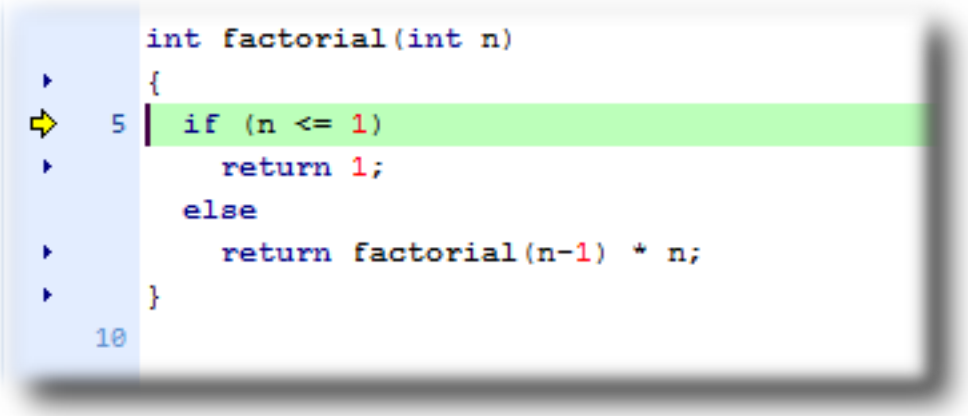


The program stops at our breakpoint and a yellow arrow in the gutter indicates where the program is paused.



Step into the `factorial` function by selecting **Debug** > **Step Into**, by typing **F11**, or by clicking the **Step Into** button on the **Debug** tool bar.

Now step to the first statement in the function by selecting **Debug** > **Step Over**, by typing **F10**, or by clicking the **Step Over** button on the **Debug** tool bar.

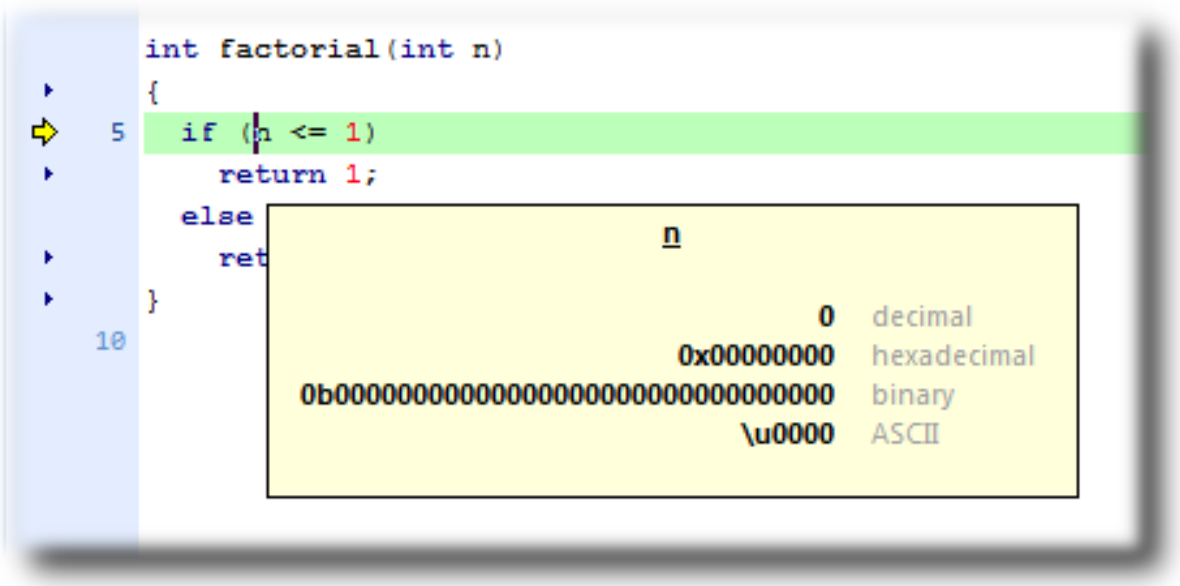


You can step out of a function by choosing **Debug** > **Step Out**, by typing **Shift+F11**, or by clicking the **Step Out** button on the **Debug** tool bar. You can also step to a specific statement by choosing **Debug** > **Run To Cursor**. To allow your application to run to the next breakpoint, choose **Debug** > **Go**.

Note that, when single-stepping, you may step into a function whose source code the debugger cannot locate. In such cases, the debugger will display the instructions of the application; you can step out to get back to source code or continue to debug at the instruction-code level. There may be cases in which the debugger cannot display the instructions; in such cases, you will be informed of this with a dialog and you should step out.

Inspecting data

Being able to control execution isn't very helpful if you can't look at the values of variables, registers, and peripherals. Hovering the mouse cursor over a variable will show its value as a *data tip*:

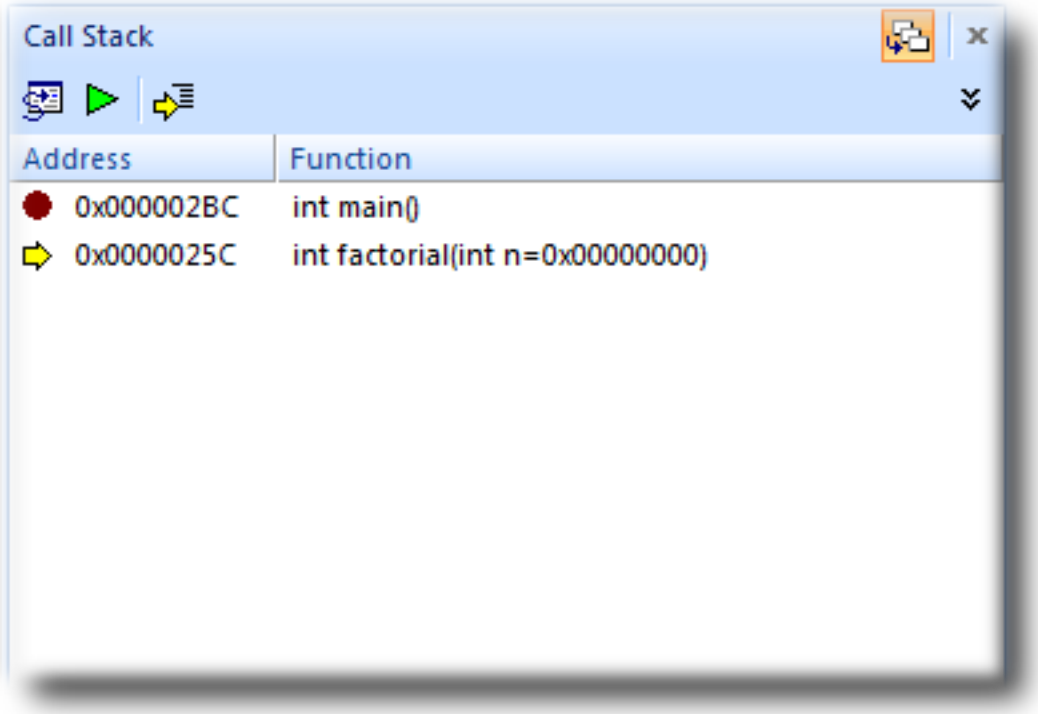


You can configure CrossStudio to display data tips in a variety of formats at the same time using the **Environment Options** dialog. You can also use the **Autos**, **Locals**, **Globals**, **Watch**, and **Memory** windows to view variables and memory. These windows are described in CrossStudio User Guide.

The **Call Stack** window shows the function calls that have been made but have not yet finished executing, that is the list of active functions.

To display the call stack:

item Choose **Debug** > **Call Stack** or press **Ctrl+Alt+S**.



You can learn more about this in the Call Stack window section.

Program output

The Tutorial application uses the function `debug_printf` to output a string to the **Debug Terminal** in the **Output** window. The **Debug Terminal** appears automatically whenever something is written to it—press **F5** to continue program execution and you will notice that the **Debug Terminal** appears. In fact, the program runs forever, writing the same messages over and over again. To pause the program, select **Debug** > **Break** or type **Ctrl+.** (control-period).

In the next section, we'll cover low-level debugging at the machine level.

Low-level debugging

This section describes how to debug your application at the register and instruction level. Debugging at a high level is fine, but sometimes you need to look more closely into the way your program executes to track down the causes of difficult-to-find bugs. CrossStudio provides the tools you need to do so.

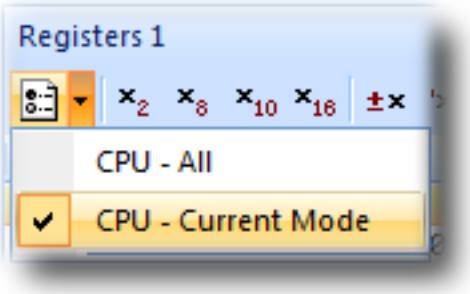
Setting up again

Next, we'll run the sample application again and look at how it executes at the machine level. If you haven't done so already, stop the program executing by typing **Shift+F5**, by selecting **Debug** > **Stop**, or by clicking the **Stop Debugging** button on the **Debug** tool bar. Now, run the program until it stops at the first breakpoint again.

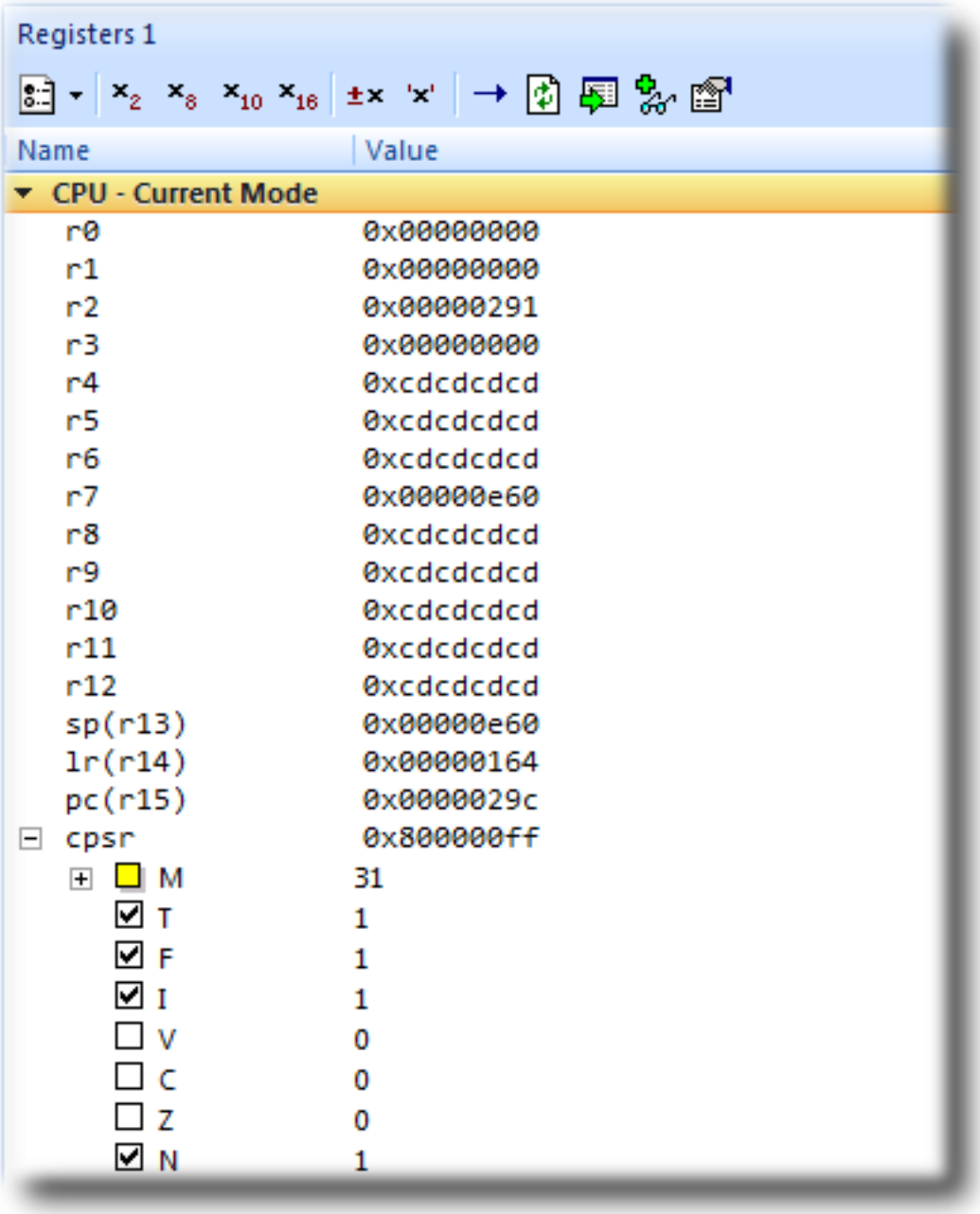
You can see the current processor state in the **Register** windows. To show the first **Registers** window:

- Choose **Debug** > **Other Windows** > **Registers** > **Registers 1** or press **Ctrl+T, R, 1**.

The **Registers** window can be used to view CPU and peripheral registers. To display the state of the registers for the active processor mode, use the **Registers 1** window's **Register Groups** menu to select **CPU - Current Mode**.



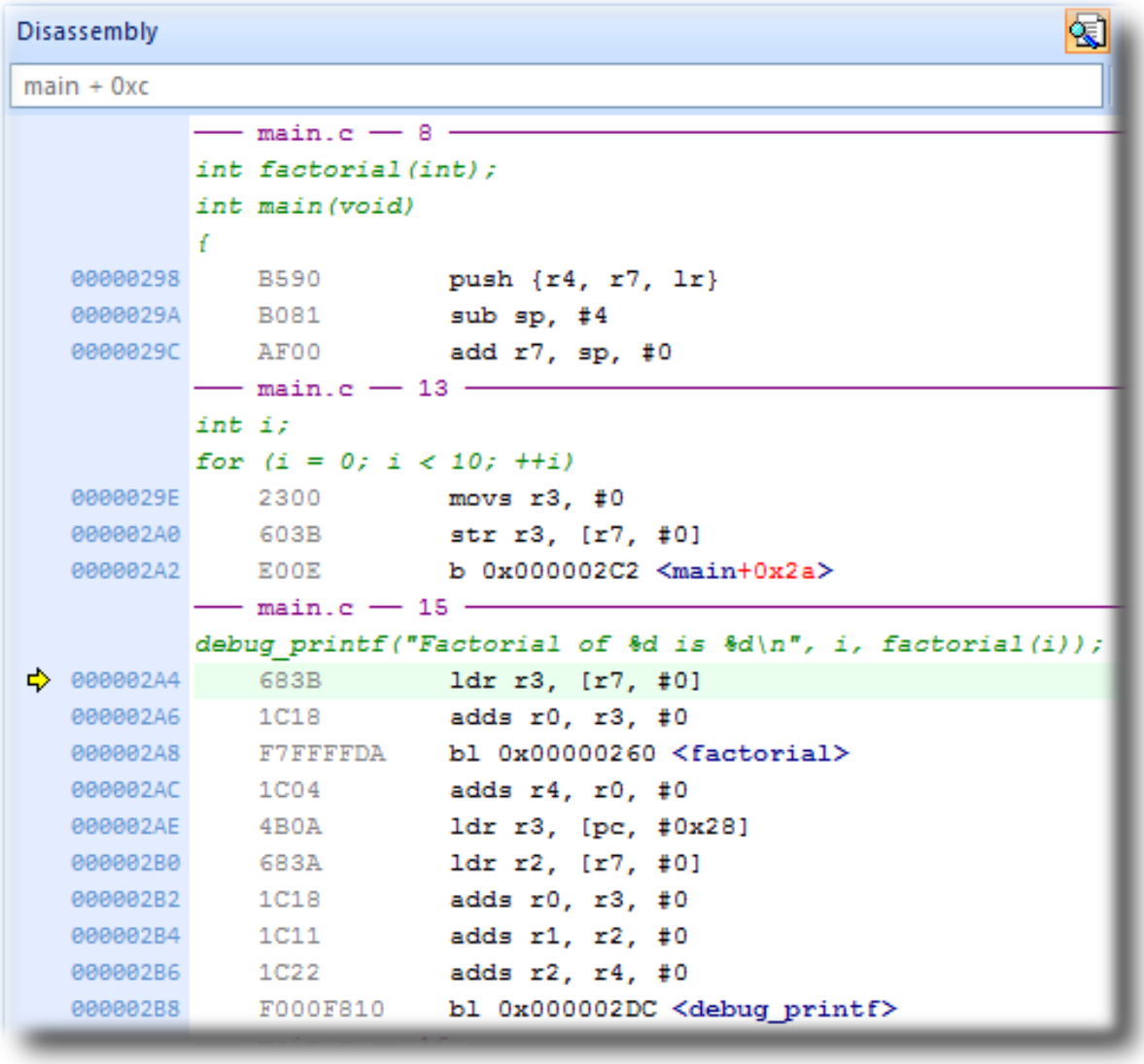
This view is displaying the registers for the active processor mode. You can also display the entire set of ARM registers: to do this, select **CPU - All** from the **Register Groups** menu. Your registers window will look something like this:



There are four register windows, so you can open and display four sets of CPU and peripheral registers at the same time. You can configure which registers and peripherals to display in the **Registers** windows individually. As you single-step the program, the contents of the **Registers** window updates and any change in a register value is highlighted in red.

Disassembly

The **Disassembly** window can be used to debug your program at the instruction level. It displays a disassembly of the instructions around the currently located instruction, interleaved with the source code of the program, if the source is available. When the **Disassembly** window has focus, all single-stepping is done one instruction at a time. This window also allows you to set breakpoints by clicking in the gutter of lines containing instructions on which you want to set a breakpoint.



Stopping and starting debugging

- You can stop debugging using **Debug** > **Stop** or `Shift+F5`.

- To restart debugging without reloading the program, you can use **Debug** > **Debug From Reset**. Note that, when you debug from reset, no loading takes place; it is expected that your program resets any data values as necessary as part of its startup.
- You can attach the debugger to a running target, other than a simulator, using **Target** > **Attach Debugger**.

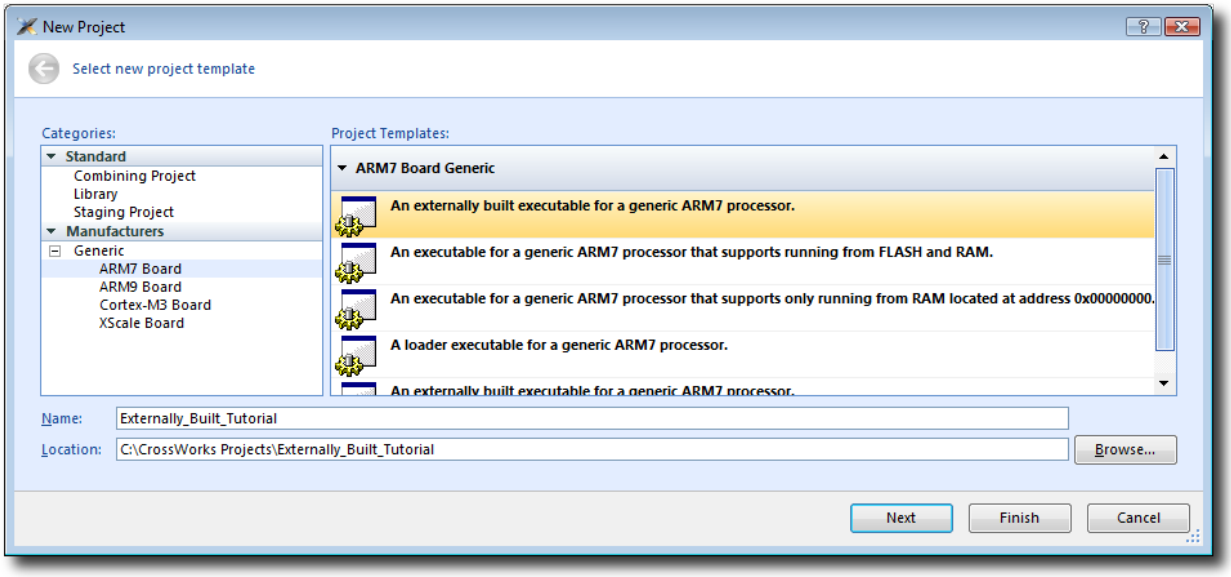
Debugging externally built applications

This section describes how to debug applications that were not built by CrossStudio. To keep things simple, we shall use the application we just built as our externally built application.

Start by creating a new, externally built executable project:

- Choose **File** > **New Project** or press **Ctrl+Shift+N**.

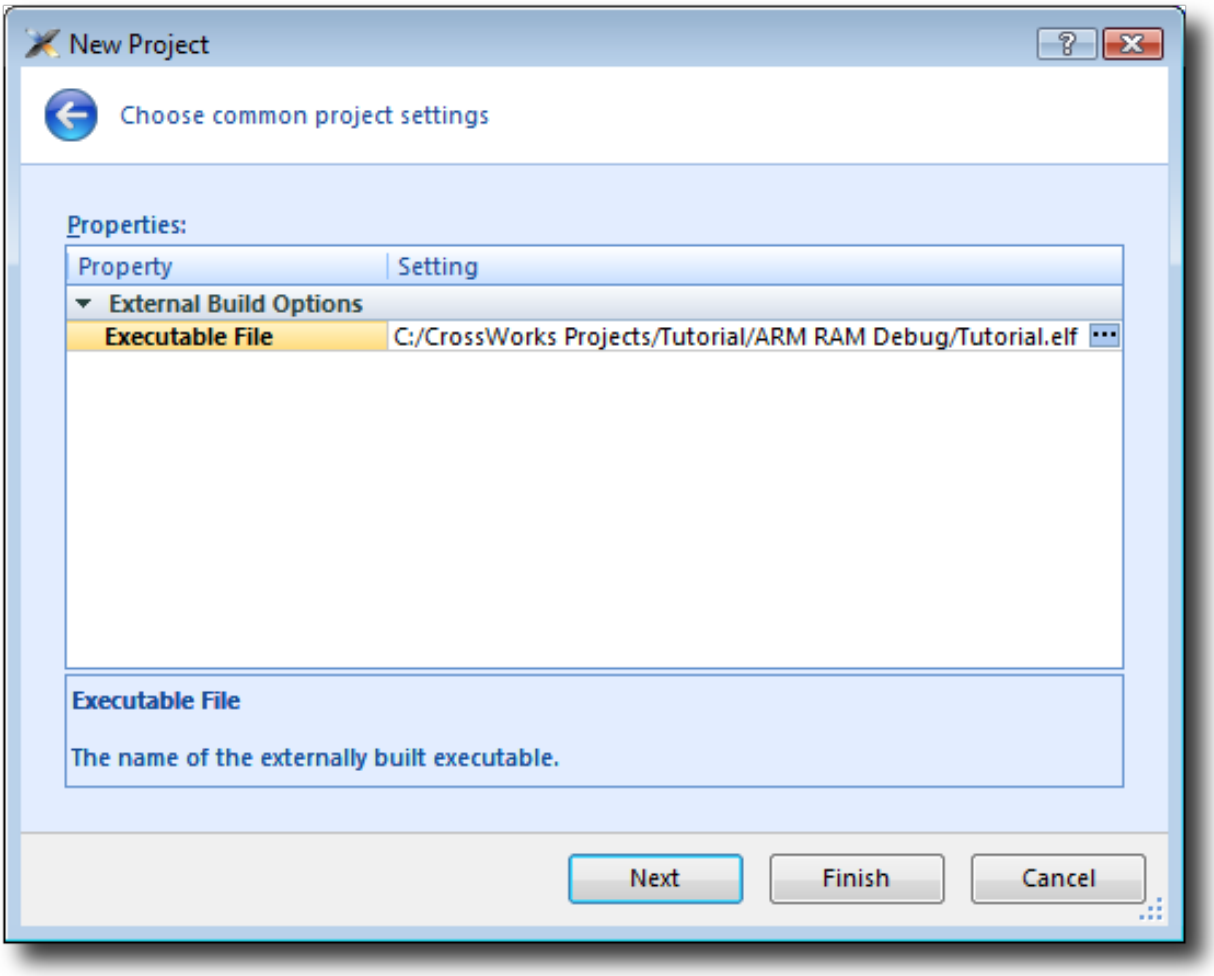
The **New Project** dialog appears. It displays the set of project types and project templates.



We'll create an externally built executable project:

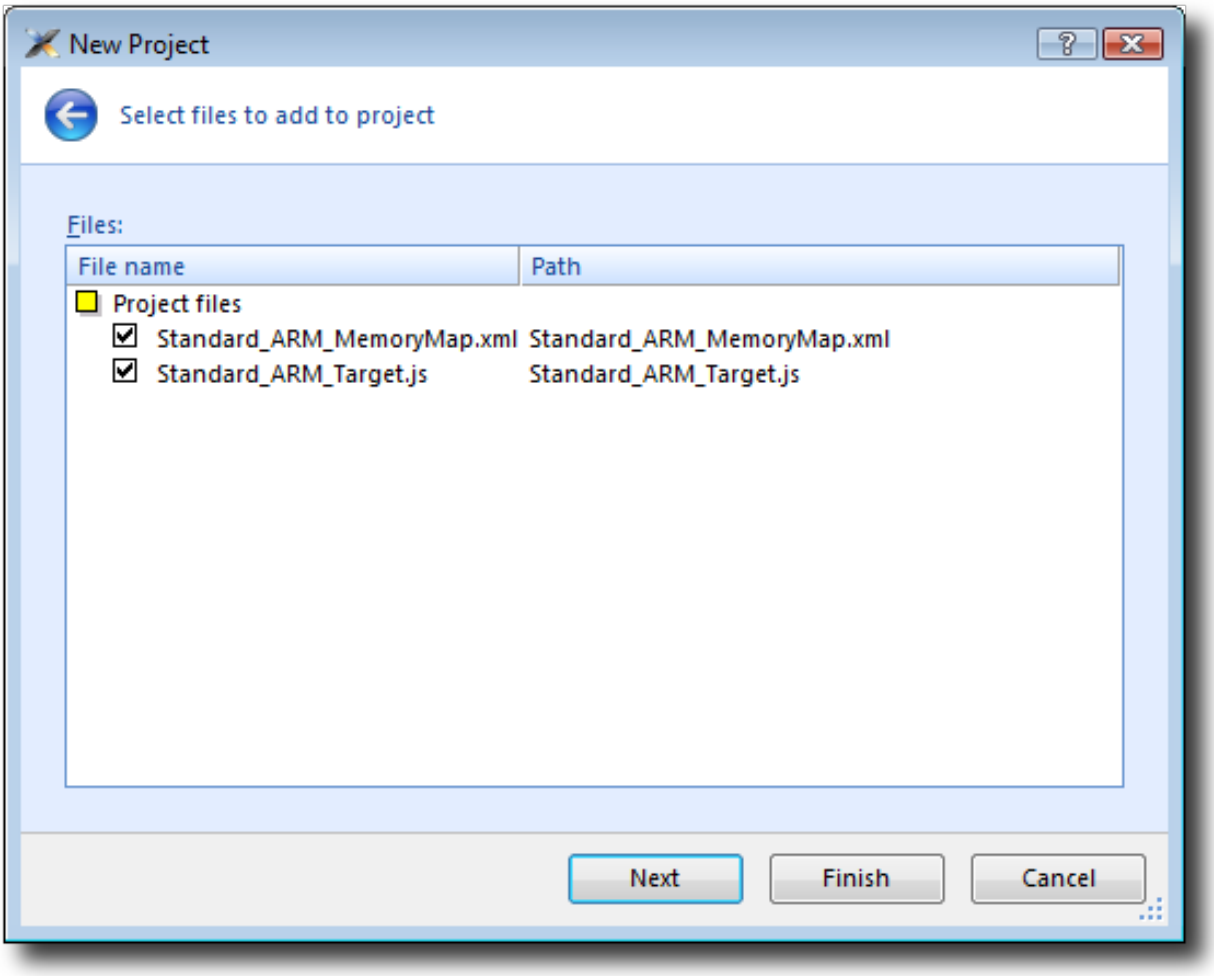
- In the **Categories** pane, select the **Generic** > **ARM7 Board** project type.
- In the **Project Templates** pane, select the **An externally built executable for a generic ARM7 processor** icon, which selects the type of project to add.
- Type `Externally_Built_Tutorial` in the **Name** field, which names the project.
- You can use the **Location** field or the **Browse** button to locate where you want the project to be created.
- Click **OK**.

Once created, the project-setup wizard prompts you for the executable file you want to use.



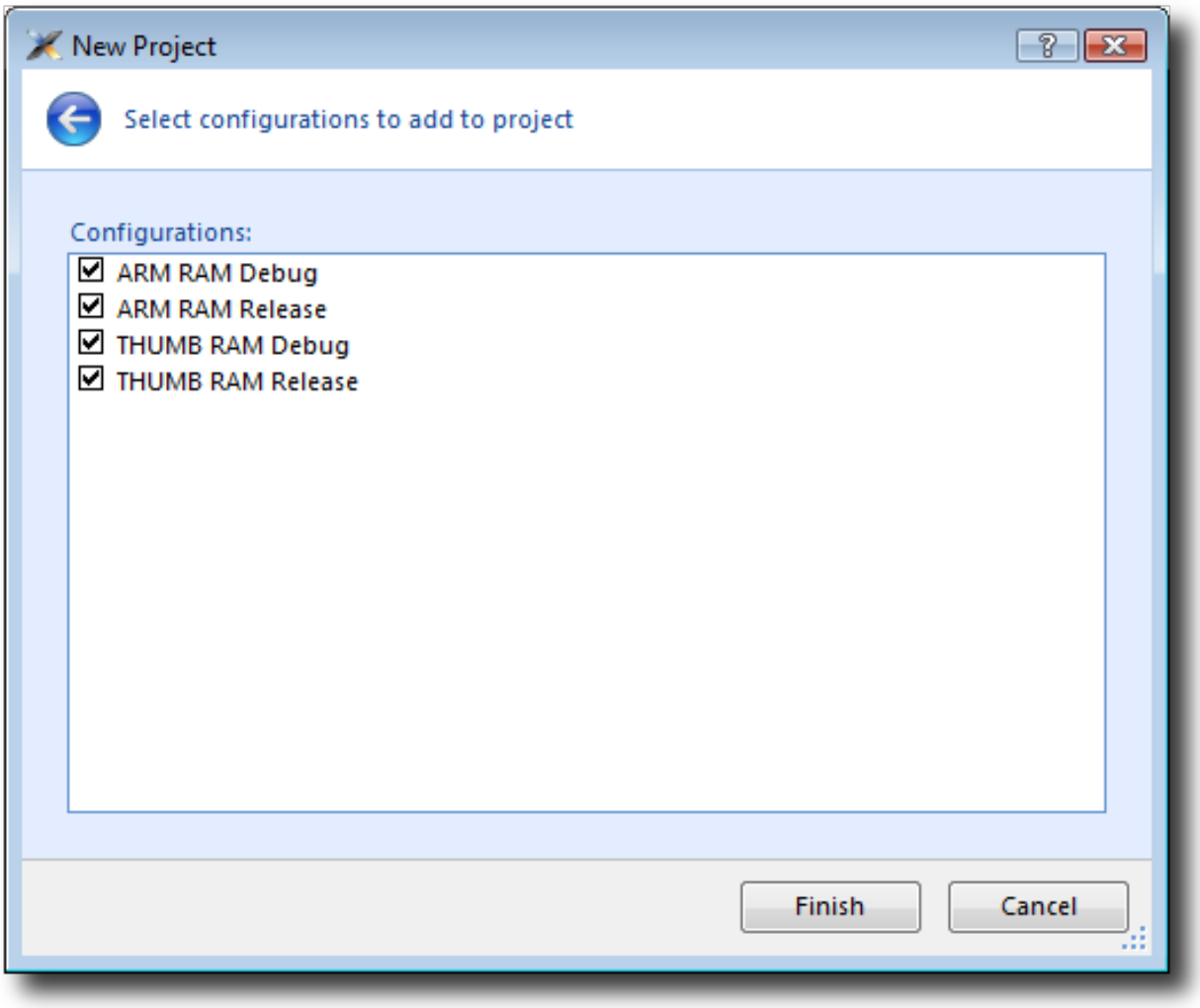
In the **Executable File** field, type the path to the `Tutorial.elf` executable file we generated earlier. For example, if the project was created in the `C:/CrossWorks Projects/Tutorial` directory and was built using the **ARM RAM Debug** configuration, the path to the executable file will be `C:/CrossWorks Projects/Tutorial/ARM RAM Debug/Tutorial.elf`.

Clicking **Next** displays the files that will be added to the project.



The **Project files** group shows the files that will be copied into the project. The only files used are the memory map file, which describes the memory layout used by the application, and the script used to reset and control the target. For the debugging session to work correctly, each of these files must match and be appropriate for the application you are debugging.

Clicking **Next** displays the configurations that will be added to the project.



Complete the project creation by clicking **Finish**.

You will be prompted as to whether you want to overwrite the existing memory map and target script. Click **No** to keep the existing files.

Now you have created the externally built executable project. You should be able to use the debugger just as we did earlier in the tutorial.

CrossStudio User Guide

This is the user guide for the CrossStudio integrated development environment (IDE). The CrossStudio IDE consists of:

- a project system to organize your source files
- a build system to build your applications
- programmer aids to navigate and work effectively
- a target programmer to download applications into RAM or flash
- a debugger to pinpoint bugs

CrossStudio standard layout

CrossStudio's main window is divided into the following areas:

- *Title bar*: Displays the name of the current solution.
- *Menu bar*: Menus for editing, building, and debugging your program.
- *Toolbars*: Frequently used actions are quickly accessible on toolbars below the menu bar.
- *Editing area*: A tabbed view of any open editor windows and the HTML viewer.
- *Docked windows*: CrossStudio has many windows that dock to the left, right, or below the editing area. You can configure which windows will be visible, and their placement, when editing and debugging.
- *Status bar* At the bottom of the main window, the status bar contains useful information about the current editor, build status, and debugging environment.

Menu bar

The menu bar contains menus for editing, building, and debugging your program. You can navigate menus using the keyboard or the mouse.

Navigating menus using the mouse

To navigate menus using the mouse:

1. Click a menu title in the menu bar to show the related menu.
2. Click the desired command in the menu to execute that command.

—or—

1. Click and hold the mouse on a menu title in the menu bar to show the related menu.
2. Drag the mouse to the desired command in the menu.
3. Release the mouse while it is over the command to execute that command.

Navigating menus with the keyboard

To navigate menus using the keyboard:

1. Tap the **Alt** key activate the menu bar.
2. Use the **Left** and **Right** keys to display the required menu.
3. Use the **Up** or **Down** key to select the required command or submenu. Press Enter to execute a selected command.
4. Press **Alt** or **Esc** at any time to cancel menu selection.

After you press the **Alt** key once, each menu on the menu bar has one letter underlined—its shortcut key. So, to activate a menu using the keyboard:

- While holding down the **Alt** key, type the desired menu's shortcut key.

After the menu appears, you can navigate it using the cursor keys:

- Use **Up** and **Down** to move up and down the list of menu items.
- Use **Esc** to cancel a menu.
- Use **Right** or **Enter** to open a submenu.
- Use **Left** or **Esc** to close a submenu and return to the parent menu.
- Type the underlined letter in a command's name to execute that command.

Title bar

The first item shown in the title bar is CrossStudio's name. Because CrossStudio can be used to target different processors, the name of the target processor family is also shown, to help you distinguish between instances of CrossStudio when debugging multi-processor or multi-core systems.

The filename of the active editor follows CrossStudio's name; you can configure the presentation of this filename as described below.

After the filename, the title bar displays status information on CrossStudio's state:

- **[building]** — CrossStudio is building a solution, building a project, or compiling a file.
- **[run]** — An application is running under control of CrossStudio's debugger.
- **[break]** — The debugger is stopped at a breakpoint.
- **[autostep]** — The debugger is single stepping the application without user interaction (*autostepping*).

Status bar

At the bottom of the window, the status bar contains useful information about the current editor, build status, and debugging environment. The status bar is divided into two regions: one contains a set of fixed panels and the other is used for messages.

The message area

The leftmost part of the status bar is a message area used for things such as status tips, progress information, warnings, errors, and other notifications.

Status bar panels

You can show or hide the following panels on the status bar:

Panel	Description
Target device status	Displays the connected target interface. When connected, this panel contains the selected target interface's name and, if applicable, the processor to which the target interface is connected. The LED icon flashes green when a program is running, is solid red when stopped at a breakpoint, and is yellow when connected to a target but not running a program. Double-clicking this panel displays the **Targets** pane, and right-clicking it invokes the **Target** shortcut menu.
Cycle count panel	Displays the number of processor cycles used by the executing program. This panel is only visible if the connected target supports performance counters that can report the total number of cycles executed. Double-clicking this panel resets the cycle counter to zero, and right-clicking it brings up the **Cycle Count** shortcut menu.
Insert/overwrite status	Indicates whether the current editor is in insert or overwrite mode. In overwrite mode, the panel displays "OVR"; in insert mode, the panel displays "INS".
Read-only status	Indicates whether the editor is in read-only mode. If the editor is editing a read-only file or is in read-only mode, the panel display "R/O"; if the editor is in read-write mode, the panel displays "R/W".

Build status	Indicates the success or failure of the last build. If the last build completed without errors or warnings, the build status pane contains 'Built OK'; otherwise, it contains the number of errors and warnings reported. If there were errors, double-clicking this panel displays the **Build Log** in the **Output** pane.
Caret position	Indicates the cursor position in the editor window. For text files, the **caret position** pane displays the line number and column number of the cursor in the active window; when editing binary files, it displays the address being edited.
Time panel	Displays the current time.

Configuring the status bar panels

To configure which panels are shown on the status bar:

- Choose **View** > **Status Bar**.
- From the status bar menu, select the panels to display and deselect the ones you want hidden.

—or—

- Right-click the status bar.
- From the status bar menu, select the panels to display and deselect the ones you want to hide.

To show or hide the status bar:

- Choose **View** > **Status Bar**.
- From the status bar menu, select or deselect the **Status Bar** item.

You can choose to hide or display the *size grip* when CrossStudio's main window is not maximized. (The size grip is never shown in full-screen mode or when maximized.)

To show or hide the size grip

- Choose **View** > **Status Bar**.
- From the status bar menu, select or deselect the **Size Grip** item.

Editing workspace

The main area of CrossStudio is the editing workspace. It contains any files being edited, the on-line help system's HTML browser, and the Dashboard.

Docking windows

CrossStudio has a flexible docking system you can use to position windows as you like them. You can dock windows in the CrossStudio window or in the four *head-up display* windows. CrossStudio will remember the position of the windows when you leave the IDE and will restore them when you return.

Window groups

You can organize CrossStudio windows into *window groups*. A window group has multiple windows docked in it, only one of which is *active* at a time. The window group displays the active window's title for each of the windows docked in the group.

Clicking on the window icons in the window group's header changes the active window. Hovering over a docked window's icon in the header will display that window's title in a *tooltip*.

To dock a window to a different window group:

- Press and hold the left mouse button over the title of the window you wish to move.
- Drag the window over the window group to dock in.
- Release the mouse button.

Holding **Ctrl** when moving the window will prevent the window from being docked. If you do not dock a window on a window group, the window will float in a new window group.

Perspectives

CrossStudio remembers the dock position and visibility of each window in each *perspective*. The most common use for this is to lay your windows out in the **Standard** perspective, which is the perspective used when you are editing and not debugging. When CrossStudio starts to debug a program, it switches to the **Debug** perspective. You can now lay out your windows in this perspective and CrossStudio will remember how you laid them them out. When you stop debugging, CrossStudio will revert to the **Standard** perspective and that window layout; when you return to **Debug** perspective on the next debug session, the windows will be restored to how you laid them out in that perspective.

CrossStudio remembers the layout of windows, in all perspectives, such that they can be restored when you run CrossStudio again. However, you may wish to revert back to the standard docking positions; to do this:

- Choose **Window** > **Reset Window Layout**.

Some customers are accustomed to having the Project Explorer on the left or the right, depending upon which version of Microsoft Visual Studio they commonly use. To quickly switch the CrossStudio layout to match your preferred Visual Studio setup:

- Choose **Window** > **Reverse Workspace Layout**.

Dashboard

When CrossStudio starts, it presents the **Dashboard**, a collection of panels that provide useful information, one-click loading of recent projects, and at-a-glance summaries of activity relevant to you.

Tasks

The **Tasks** panel indicates tasks you need to carry out before CrossWorks is fully functional—for instance, whether you need to activate CrossWorks, install packages, and so on.

Updates

The **Updates** panel indicates whether any packages you have installed are now out of date because a newer version is available. You can install each new package individually by clicking the **Install** button under each notification, or install all packages by clicking the **Install all updates** link at the bottom of the panel.

Projects

The **Projects** panel contains links to projects you have worked on recently. You can load a project by clicking the appropriate link, or clear the project history by clicking the **Clear List** button. To manage the contents of the list, click the **Manage Projects** link and edit the list of projects in the **Recent Projects** window.

News

The **News** panel summarizes the activity of any RSS and Atom feeds to which you have subscribed. Clicking a link will display the published article in an external web browser. You can manage your feed subscriptions to by clicking the **Manage Feeds** link at the end of the **News** panel and *pinning* the feeds in the **Favorites** window—you are only subscrbed to the pinned feeds.

Links

The **Links** panel is a handy set of links to your favorite websites. If you pin a link in the **Favorites** window, it appears in the **Links** panel.

CrossStudio help and assistance

CrossStudio provides context-sensitive help in increasing detail:

Tooltips

When you position the pointer over a button and keep it still, a small window displays a brief description of the button and its keyboard shortcut, if it has one.

Status tips

In addition to tooltips, CrossStudio provides a longer description in the status bar when you hover over a button or menu item.

Online manual

CrossStudio has links from all windows to the online help system.

The browser

Documentation pages are shown in the **Browser**.

Help using CrossStudio

CrossStudio provides an extensive, HTML-based help system that is available at all times.

To view the help text for a particular window or other user-interface element:

- Click to select the item with which you want assistance.
- Choose **Help** > **Help** or press **F1**.

Help within the text editor

The text editor is linked to the help system in a special way. If you place the cursor over a word and press **F1**, the help-system page most likely to be useful is displayed in the HTML browser. This a great way to quickly find the help text for functions provided in the library.

Browsing the documentation

The **Contents** window lists all the topics in the CrossWorks documentation and gives a way to search through them.

The highlighted entry indicates the current help topic. When you click a topic, the corresponding page appears in the **Browser** window.

The **Next Topic** and **Previous Topic** items in the **Help** menu, or the buttons on the **Contents** window toolbar, help navigate through topics.

To search the online documentation, type a search phrase into the **Search** box on the **Contents** window toolbar.

To search the online documentation:

- Choose **Help** > **Search**.
- Enter your search phrase in the **Search** box and press **Enter** (or **Return** on Macs).

The search commences and the table of contents is replaced by links to pages matching your query, listed in order of relevance. To clear the search and return to the table of contents, click the clear icon in the **Search** box.

Creating and managing projects

A CrossStudio *project* is a container for everything required to build your applications. It contains all the assorted resources and maintains the relationships between them.

A project is a convenient place to find every file and piece of information associated with your work. You place projects into a *solution*, which can contain one or more projects.

This chapter introduces the various parts of a project, shows how to create projects, and describes how to organize the contents of a project. It describes how to use the **Project Explorer** and **Project Manager** for project-management tasks.

Solutions and projects

To develop a product using CrossStudio, you must understand the concepts of *projects* and *solutions*.

- A *project* contains and organizes everything you need to create a single application or a library.
- A *solution* is a collection of projects and configurations.

Organizing your projects into a solution allows you to build all the projects in a solution with a single keystroke, and to load them onto the target ready for debugging.

In your CrossWorks project, you…

- …organize build-system inputs for building a product.
- …add information about items in the project, and their relationships, to assist you in the development process.

Projects in a solution can reside in the same or different directories. Project directories are always relative to the directory of the solution file, which enables you to more-easily move or share project-file hierarchies.

The **Project Explorer** organizes your projects and files, and provides quick access to the commands that operate on them. A toolbar at the top of the window offers quick access to commonly used commands.

Solutions

When you have created a solution, it is stored in a project file. Project files are text files, with the file extension **hzp**, that contain an XML description of your project. See **Project file format** for a description of the project-file format.

Projects

The projects you create within a solution have a *project type* CrossStudio uses to determine how to build the project. The project type is selected when you use the **New Project** dialog. The available project types depend on the CrossWorks variant you are using, but the following are present in most CrossWorks variants:

- *Executable*: — a program that can be loaded and executed.
- *Externally Built Executable*: — an executable that was not built by CrossWorks.
- *Library*: — a group of object files collected into a single file (sometimes called an *archive*).
- *Object File*: — the result of a single compilation.
- *Staging*: — a project that will apply a user-defined command to each file in a project.
- *Combining*: — a project that can be used to apply a user-defined command when any files in a project have changed.

Properties and configurations

Properties are attached to project nodes. They are usually used in the build process, for example, to define C preprocessor symbols. You can assign different values to the same property, based on a configuration: for example, you can assign one value to a C preprocessor symbol for release and a different value for a debug build.

Folders

Projects can contain *folders,* which are used to group related files. Automated grouping uses the files' extensions to, for example, put all .c files in one folder, etc. Grouping also can be done manually by explicitly creating a file within a folder. Note that these project folders do not map onto directories in the file system, they are used solely to structure the display of content shown in the **Project Explorer**.

Source files

Source files are all the files used to build a product. These include source code files and also section-placement files, memory-map files, and script files. All the source files you use for a particular product, or for a suite of related products, are managed in a CrossStudio project. A project can also contain files that are not directly used by CrossStudio to build a product but contain information you use during development, such as documentation. You edit source files during development using CrossStudio's built-in text editor, and you organize files into a target (described next) to define the build-system inputs for creating the product.

The source files of your project can be placed in folders or directly in the project. Ideally, the paths to files placed in a project should be relative to the project directory, but at times you might want to refer to a file in an absolute location and this is supported by the project system.

When you add a file to a project, the project system detects whether the file is in the project directory. If a file is not in the project directory, the project system tries to make a relative path from the file to the project directory. If the file isn't relative to the project directory, the project system detects whether the file is relative to the **$(StudioDir)** directory; if so, the filename is defined using **$(StudioDir)**. If a file is not relative to the project directory or to **$(StudioDir)**, the full, absolute pathname is used.

The project system will allow (with a warning) duplicate files to be put into a project.

The project system uses a file's extension to determine the appropriate build action to perform on the file:

- A file with the extension **.c** will be compiled by a C compiler.
- A file with the extension **.s** or **.asm** will be compiled by an assembler.
- A file with the extension **.cpp** or **.cxx** will be compiled by a C++ compiler.
- A file with the object-file extension **.o** or **.hzo** will be linked.
- A file with the library-file extension **.a** or **.hza** will be linked.

- A file with the extension **.xml** will be opened and its file type determined by the XML document type.
- Files with other file extensions will not be compiled or linked.

You can modify this behavior by setting a file's **File Type** property with the **Common** configuration selected in the **Properties** window, which enables files with non-standard extensions to be compiled by the project system.

Solution links

You can create links to existing project files from a solution, which enables you to create hierarchical builds. For example, you could have a solution that builds a library together with a stub test driver executable. You can link to that solution from your current solution by right-clicking the solution node of the **Project Explorer** and selecting **Add Existing Project**. Your current solution can then use the library built by the other project.

Session files

When you exit CrossWorks, details of your current session are stored in a *session file*. Session files are text files, with the file extension **hzs**, that contain details such as which files you have opened in the editor and what breakpoints you have set in the **Breakpoint** window.

Creating a project

You can create a new solution for each project or place multiple projects in an existing solution.

To create a new project in an existing solution:

1. Choose **Project** > **Add New Project**.
2. In the **New Project** wizard, select the type of project you wish to create and specify where it will be placed.
3. Ensure that **Add the project to current solution** is checked.
4. Click **OK** to go to next stage or **Cancel** to cancel the project's creation.

The project name must be unique to the solution and, ideally, the project directory should be relative to the solution directory. The project system will use the project directory as the *current directory* when it builds your project. Once complete, the **Project Explorer** displays the new solution, project, and files contained in the project. To add another project to the solution, repeat the above steps.

To create a new project in a new solution:

1. Choose **File** > **New Project** or press **Ctrl**+**Shift**+**N**.
2. Select the type of project you wish to create and where it will be placed.
3. Click **OK**.

Adding existing files to a project

You can add existing files to a project in a number of ways.

To add existing files to the active project:

- Choose **Project** > **Add Existing File**.

Using the **Open File** dialog, navigate to the directory containing the files and select the ones you wish to add to the project.

- Click **OK**.

The selected files are added to the folders whose filter matches the extension of each of the files. If no filter matches a file's extension, the file is placed underneath the project node.

To add existing files to a specific project:

1. In the **Project Explorer**, right-click the project to which you wish to add a new file.
2. Choose **Add Existing File**.

To add existing files to a specific folder:

1. In the **Project Explorer**, right-click the folder to which you wish to add a new file.
2. Choose **Add Existing File**.

The files are added to the specified folder without using filter matching.

Adding new files to a project

You can add new files to a project in a number of ways.

To add new files to the active project:

- Choose **Project** > **Add New File** or press **Ctrl**+**N**.

To add a new file to a project:

1. In the **Project Explorer**, right-click the project to which you wish to add a new file.
2. Choose **Add New File**.

When adding a new file, CrossStudio displays the **New File** dialog, from which you can choose the type of file to add, its filename, and where it will be stored. Once created, the new file is added to the folder whose filter matches the extension of the newly added file. If no filter matches the newly added file extension, the new file is placed underneath the project node.

To add new files to a folder:

1. In the **Project Explorer**, right-click the folder to which you wish to add a new file.
2. Choose **Add New File**.

The new file is added to the folder without using filter matching.

Removing a file, folder, project, or project link

You can remove whole projects, folders, or files from a project, or you can remove a project from a solution, using the **Remove** button on the **Project Explorer** toolbar. Note that removing a source file from a project does not remove it from disk.

To remove an item from the solution:

1. In the **Project Explorer**, select the item to remove.
2. Choose **Edit** > **Delete** or press **Del**.

—or—

1. In the **Project Explorer**, right-click the item to remove.
2. Choose **Remove**.

Project macros

You can use macros to modify the way the project system refers to files.

Macros are divided into four classes:

- *System macros* defined by CrossStudio relay information about the environment, such as paths to common directories.
- *Global macros* are saved in the environment and are shared across all solutions and projects. Typically, you would set up paths to libraries and any external items here.
- *Project macros* are saved as project properties in the project file and can define values specific to the solution or project in which they are defined.
- *Build macros* are generated by the project system when you build your project.

System macros

System macros are defined by CrossStudio itself and as such are read-only. System macros can be used in project properties, environment settings and to refer to files. See **System macros list** for the list of System macros.

Global macros

To define a global macro:

1. Choose **Project** > **Macros**.
2. Select the **Global** tab.
3. Set the macro using the syntax *name* = *replacement text*.

Project macros

To define a project macro:

1. Choose **Project** > **Macros**.
2. Select the **Project** tab.
3. Select the solution or project to which the macro should apply.
4. Set the macro using the syntax *name* = *replacement text*.

Alternatively, you can set the project macros from the **Properties** window:

1. Select the appropriate solution/project in the **Project Explorer**.
2. In the **Properties** window's **General Options** group, select the **Macros** property.
3. Click the ellipsis button on the right.
4. Set the macro using the syntax *name* = *replacement text*.

Build macros

Build macros are defined by the project system for a build of a given project node. See **Build macros list** for the list of build macros.

Using macros

You can use a macro for a project property or environment setting by using the $(macro) syntax. For example, the **Object File Name** property has a default value of `$(IntDir)/$(InputName)$(OBJ)`.

Building your application

CrossStudio builds your application using the resources and build rules it finds in your solution.

When CrossStudio builds your application, it tries to avoid building files that have not changed since they were last built. It does this by comparing the modification dates of the generated files with the modification dates of the dependent files together with the modification dates of the properties that pertain to the build. But if you are copying files, sometimes the modification dates may not be updated when the file is copied—in this instance, it is wise to use the **Rebuild** command rather than the **Build** command.

You can see the build rationale CrossStudio currently is using by setting the **Environment Properties** > **Build Settings** > **Show Build Information** property. To see the build commands themselves, set the **Environment Properties** > **Build Settings Echo Build Command** property.

You may have a solution that contains several interdependent projects. Typically, you might have several executable projects and some library projects. The **Project Dependencies** dialog specifies the dependencies between projects and to see the effect of those dependencies on the solution build order. Note that dependencies can be set on a per-configuration basis, but the default is for dependencies to be defined in the **Common** configuration.

You will also notice that a new folder titled **Dependencies** has appeared in the **Project Explorer**. This folder contains the list of newly generated files and the files from which they were generated. To see if one of files can be decoded and displayed in the editor, right-click the file to see if the **View** command is available on the shortcut menu.

If you have the **Symbols** window open, it will be updated with the symbol and section information of all executable files built in the solution.

When CrossStudio builds projects, it uses the values set in the **Properties** window. To generalize your builds, you can define macro values that are substituted when the project properties are used. These macro values can be defined globally at the solution and project level, and can be defined on a per-configuration basis. You can view and update the macro values using **Project** > **Macros**.

The combination of configurations, properties with inheritance, dependencies, and macros provides a very powerful build-management system. However, such systems can become complicated. To understand the implications of changing build settings, right-click a node in the **Project Explorer** and select **Properties** to view a dialog that shows which macros and build steps apply to that project node.

To build all projects in the solution:

1. Choose **Build** > **Build Solution** or press **Shift+F7**.

—or—

1. Right-click the solution in the **Project Explorer** window.
2. Choose **Build** from the shortcut menu.

To build a single project:

1. Select the required project in the **Project Explorer**.
2. Choose **Build** > **Build** or press **F7**.

—or—

1. Right-click the project in the **Project Explorer**.
2. Choose **Build**.

To compile a single file:

1. In the **Project Explorer**, click to select the source file to compile.
2. Choose **Build** > **Compile** or press **Ctrl+F7**.

—or—

1. In the **Project Explorer**, right-click the source file to compile.
2. Choose **Compile** from the shortcut menu.

Correcting errors after building

The results of a build are recorded in a **Build Log** that is displayed in the **Output** window. Errors are highlighted in red, warnings are highlighted in yellow. Double-clicking an error, warning, or note will move the insertion point to the line of source code that triggered that log entry.

You can move forward and backward through errors using **Search** > **Next Location** and **Search** > **Next Location**.

When you build a single project in a single configuration, the **Transcript** will display the memory used by the application and a summary for each memory area.

Creating variants using configurations

CrossStudio provides a facility to build projects in various configurations. Project configurations are used to create different software builds for your projects.

A configuration defines a set of project property values. For example, the output of a compilation can be put into different directories, dependent upon the configuration. When you create a solution, some default project configurations are created.

Build configurations and their uses

Configurations are typically used to differentiate debug builds from release builds. For example, the compiler options for debug builds will differ from those of a release build: a debug build will set options so the project can be debugged easily, whereas a release build will enable optimization to reduce program size or to increase its speed. Configurations have other uses; for example, you can use configurations to produce variants of software, such as custom libraries for several different hardware variants.

Configurations inherit properties from other configurations. This provides a single point of change for definitions common to several configurations. A particular property can be overridden in a particular configuration to provide configuration-specific settings.

When a solution is created, two configurations are generated — **Debug** and **Release** — and you can create additional configurations by choosing **Build** > **Build Configurations**. Before you build, ensure that the appropriate configuration is set using **Build** > **Set Active Build Configuration** or, alternatively, the **Active Configuration** combo box in the **Project Explorer**. You should also ensure that the appropriate build properties are set in the **Properties** window.

Selecting a configuration

To set the configuration that affects your building and debugging, use the combo box in the **Project Explorer** or select **Build** > **Set Active Build Configuration**

Creating a configuration

To create your own configurations, select **Build** > **Build Configurations** to invoke the **Configurations** dialog. The **New** button will produce a dialog allowing you to name your configuration. You can now specify the existing configurations from which your new configuration will inherit values.

Deleting a configuration

You can delete a configuration by selecting it and clicking the **Remove** button. This deletion cannot be undone or canceled, so beware.

Private configurations

Some configurations are defined purely for inheriting and, as such, should not appear in the **Build** combo box. When you select a configuration in the **Configuration** dialog, you can choose to hide that configuration.

Project properties

For solutions, projects, folders, and files, properties can be defined that are used by the project system in the build process. These property values can be viewed and modified by using the **Properties** window in conjunction with the **Project Explorer**. As you select items in the **Project Explorer**, the **Properties** window will list the set of relevant properties.

Some properties are only applicable to a given item type. For example, linker properties are only applicable to a project that builds an executable file. However, other properties can be applied either at the file, project, or solution project node. For example, a compiler property can be applied to a solution, project, or individual file. By setting a property at the solution level, you enable all files of the solution to use that property's value.

Unique properties

A unique property has *one* value. When a build is done, the value of a unique property is the first one defined in the project hierarchy. For example, the **Treat Warnings As Errors** property could be set to **Yes** at the solution level, which would then be applicable to every file in the solution that is compiled, assembled, and linked. You can then selectively define property values for other project items. For example, a particular source file may have warnings you decide are allowable, so you set the **Treat Warnings As Errors** to **No** for that particular file.

Note that, when the **Properties** window displays a project property, it will be shown in bold if it has been defined for unique properties. The inherited or default value will be shown if it hasn't been defined.

solution — **Treat Warnings As Errors = Yes**
project1 — Treat Warnings As Errors = Yes
file1 — Treat Warnings As Errors = Yes
file2 — **Treat Warnings As Errors = No**
project2 — **Treat Warnings As Errors = No**
file1 — Treat Warnings As Errors = No
file2 — **Treat Warnings As Errors = Yes**

In the above example, the files will be compiled with these values for **Treat Warnings As Errors**:

project1/file1	Yes
project1/file2	No
project2/file1	No
project2/file2	Yes

Aggregate properties

An aggregating property collects all the values defined for it in the project hierarchy. For example, when a C file is compiled, the **Preprocessor Definitions** property will take all the values defined at the file, project, and solution levels. Note that the **Properties** window *will not* show the inherited values of an aggregating property.

solution — **Preprocessor Definitions = SolutionDef**
project1 — Preprocessor Definitions =
file1 — Preprocessor Definitions =
file2 — **Preprocessor Definitions = File1Def**
project2 — **Preprocessor Definitions = ProjectDef**
file1 — Preprocessor Definitions=
file2 — **Preprocessor Definitions = File2Def**

In the above example, the files will be compiled with these preprocessor definitions:

project1/file1	SolutionDef
project1/file2	SolutionDef, File1Def
project2/file1	SolutionDef, ProjectDef
project2/file2	SolutionDef, ProjectDef, File2Def

Configurations and property values

Property values are defined for a configuration so you can have different values for a property for different builds. A given configuration can inherit the property values of other configurations. When the project system requires a property value, it checks for the existence of the property value in current configuration and then in the set of inherited configurations. You can specify the set of inherited configurations using the **Configurations** dialog.

A special configuration named **Common** is always inherited by a configuration. The **Common** configuration allows you to set property values that will apply to all configurations you create. You can select the **Common** configuration using the **Configurations** combo box of the properties window. If you are modifying a property value of your project, you almost certainly want each configuration to inherit it, so ensure that the **Common** configuration is selected.

If the property is unique, the build system will use the one defined for the particular configuration. If the property isn't defined for this configuration, the build system uses an arbitrary one from the set of inherited configurations.

If the property is still undefined, the build system uses the value for the **Common** configuration. If it is still undefined, the build system tries to find the value in the next higher level of the project hierarchy.

solution [Common] — Preprocessor Definitions = CommonSolutionDef
solution [Debug] — Preprocessor Definitions = DebugSolutionDef
solution [Release] — Preprocessor Definitions = ReleaseSolutionDef
project1 - Preprocessor Definitions =
file1 - Preprocessor Definitions =
file2 [Common] — Preprocessor Definitions = CommonFile1Def
file2 [Debug] — Preprocessor Definitions = DebugFile1Def
project2 [Common] — Preprocessor Definitions = ProjectDef
file1 — Preprocessor Definitions =
file2 [Common] - Preprocessor Definitions = File2Def

In the above example, the files will be compiled with these preprocessor definitions when in **Debug** configuration…

File	Setting
project1/file1	CommonSolutionDef, DebugSolutionDef
project1/file2	CommonSolutionDef, DebugSolutionDef,CommonFile1Def, DebugFile1Def
project2/file1	CommonSolutionDef, DebugSolutionDef, ProjectDef
project2/file2	ComonSolutionDef, DebugSolutionDef, ProjectDef, File2Def

…and the files will be compiled with these **Preprocessor Definitions** when in **Release** configuration:

File	Setting
project1/file1	CommonSolutionDef, ReleaseSolutionDef
project1/file2	CommonSolutionDef, ReleaseSolutionDef, CommonFile1Def
project2/file1	CommonSolutionDef, ReleaseSolutionDef, ProjectDef
project2/file2	ComonSolutionDef, ReleaseSolutionDef, ProjectDef, File2Def

Dependencies and build order

You can set up dependency relationships between projects using the **Project Dependencies** dialog. Project dependencies make it possible to build solutions in the correct order and, where the target permits, to load and delete applications and libraries in the correct order. A typical usage of project dependencies is to make an executable project dependent upon a library executable. When you elect to build the executable, the build system will ensure that the library it depends upon is up to date. In the case of a dependent library, the output file of the library build is supplied as an input to the executable build, so you don't have to worry about it.

Project dependencies are stored as project properties and, as such, can be defined differently based upon the selected configuration. You almost always want project dependencies to be independent of the configuration, so the **Project Dependencies** dialog selects the **Common** configuration by default.

To make one project dependent upon another:

1. Choose **Project** > **Project Dependencies**.
2. From the **Project** dropdown, select the target project that depends upon other projects.
3. In the **Depends Upon** list box, select the projects the target project depends upon and deselect the projects it does not depend upon.

Some items in the **Depends Upon** list box may be dimmed, indicating that a circular dependency would result if any of those projects were selected. In this way, CrossStudio prevents you from constructing circular dependencies using the **Project Dependencies** dialog.

If your target supports loading multiple projects, the **Build Order** also reflects the order in which projects are loaded onto the target. Projects will load, in order, from top to bottom. Generally, libraries need to be loaded before the applications that use them, and you can ensure this happens by making the application dependent upon the library. With this dependency set, the library gets built and loaded before the application does.

Applications are deleted from a target in reverse of their build order; in this way, applications are removed before the libraries on which they depend.

Linking and section placement

Executable programs consist of a number of sections. Typically, there are program sections for code, initialized data, and zeroed data. There is often more than one code section and they must be placed at specific addresses in memory.

To describe how the program sections of your program are positioned in memory, the CrossWorks project system uses *memory-map* files and *section-placement* files. These XML-formatted files are described in **Memory Map file format** and **Section Placement file format**. They can be edited with the CrossWorks text editor. The memory-map file specifies the start address and size of target memory segments. The section-placement file specifies where to place program sections in the target's memory segments. Separating the memory map from the section-placement scheme enables a single hardware description to be shared across projects and also enables a project to be built for a variety of hardware descriptions.

For example, a memory-map file representing a device with two memory segments called **FLASH** and **SRAM** could look something like this in the memory-map editor.

```
<Root name="Device1" >
  <MemorySegment name="FLASH" start="0x10000000" size="0x10000" />
  <MemorySegment name="SRAM" start="0x20000000" size="0x1000" />
```

A corresponding section-placement file will refer to the memory segments of the memory-map file and will list the sections to be placed in those segments. This is done by using a memory-segment name in the section-placement file that matches the corresponding memory-segment name in the memory-map file.

For example, a section-placement file that places a section called **.stack** in the **SRAM** segment and the **.vectors** and **.text** sections in the **FLASH** segment would look like this:

```
<Root name="Flash Section Placement" >
  <MemorySegment name="FLASH" >
    <ProgramSection name=".vectors" load="Yes" />
    <ProgramSection name=".text" load="Yes" />
  </MemorySegment>
  <MemorySegment name="SRAM" >
   <ProgramSection name=".stack" load="No" />
  </MemorySegment>
</Root>
```

Note that the order of section placement within a segment is top down; in this example **.vectors** is placed at lower addresses than **.text**.

The memory-map file and section-placement file to use for linkage can be included as a part of the project or, alternatively, they can be specified in the project's **linker properties**.

You can create a new program section using either the assembler or the compiler. For the C/C++ compiler, this can be achieved using __**attribute**__ on declarations. For example:

```
void foobar(void) __attribute__ ((section(".foo")));
```

This will allocate **foobar** in the section called **.foo**. Alternatively, you can specify the names for the code, constant, data, and zeroed-data sections of an entire compilation unit by using the **Section Options** properties.

You can now place the section into the section placement file using the editor so that it will be located after the vectors sections as follows:

```
<Root name="Flash Section Placement" >
  <MemorySegment name="FLASH" >
    <ProgramSection name=".vectors" load="Yes" />
    <ProgramSection name=".foo" load="Yes" />
    <ProgramSection name=".text" load="Yes" />
  </MemorySegment>
  <MemorySegment name="SRAM" >
   <ProgramSection name=".stack" load="No" />
  </MemorySegment>
</Root>
```

If you are modifying a section-placement file that is supplied in the CrossWorks distribution, you will need to import it into your project using the **Project Explorer**.

Sections containing code and constant data should have their **load** property set to **Yes**. Some sections don't require any loading, such as stack sections and zeroed-data sections; such sections should have their **load** property set to **No**.

Some sections that are loaded then need to be copied to sections that aren't yet loaded. This is required for initialized data sections and to copy code from slow memory regions to faster ones. To do this, the **runin** attribute should contain the name of a section in the section-placement file to which the section will be copied.

For example, initialized data is loaded into the **.data_load** section and then is copied into the **.data_run** section using:

```
<Root name="Flash Section Placement" >
  <MemorySegment name="FLASH" >
    <ProgramSection name=".vectors" load="Yes" />
    <ProgramSection name=".text" load="Yes" />
    <ProgramSection name=".data_load" load="Yes" runin="data_run" />
  </MemorySegment>
  <MemorySegment name="SRAM" >
    <ProgramSection name=".data_run" load="No" />
    <ProgramSection name=".stack" load="No" />
  </MemorySegment>
</Root>
```

The startup code will need to copy the contents of the **.data_load** section to the **.data_run** section. To enable this, symbols are generated marking the start and end addresses of each section. For each section, a start symbol called __ *section-name* _**start**__ and an end symbol called __ *section-name* _**end**__ are generated. These symbols can be used to copy the sections from their load positions to their run positions.

For example, the **.data_load** section can be copied to the **data_run** section using the following call to `memcpy`.

```
/* Section image located in flash */
extern const unsigned char __data_load_start__[];
extern const unsigned char __data_load_end__[];
/* Where to locate the section image in RAM. */
extern unsigned char __data_run_start__[];
extern unsigned char __data_run_end__[];
/* Copy image from flash to RAM. */
memcpy(__data_run_start__,
       __data_load_start__,
       __data_load_end__ - __data_load_start__);
```

Using source control

Source control is an essential tool for individuals or development teams. CrossStudio integrates with several popular source-control systems to provide this feature for files in your CrossWorks projects.

Source-control capability is implemented by a number of third-party providers, but the set of functions provided by CrossWorks aims to be provider independent.

Source control capabilities

The source-control integration capability provides:

- Connecting to the source-control database (sometimes called a *repository*) and mapping files in the CrossWorks project to those in source control.
- Showing the source-control status of files in the project.
- Adding files in the project to source control. This operation is called *Add To Source Control*.
- Fetching files in the project from source control. This operation is called *Get Latest Version*.
- Locking and unlocking files in the project for editing. The lock operation is called *Check Out*. The unlock operation is called *Undo Check Out*. These are optional for some source-control providers.
- Comparing a file in the project with the latest version in source control. This operation is called *Show Differences*.
- Merging a file in the project with the latest version in source control with reference to the original version. This operation is called *Merge* and requires an external, three-way merge tool.
- Committing changes made to project files into source control. This operation is called *Check In*.

Choosing your source-control provider

The source-control system you are using must be enabled.

To enable a provider:

1. Choose **Tools** > **Options** or press **Alt+,**.
2. Select the **Source Control** category in the options dialog.
3. Set the **Source Control Provider** to the appropriate provider.
4. Set the provider-specific options.
5. Set **Enable Source Code Control Integration** to **Yes**.

Note: The source-control provider information is stored in the CrossWorks global environment, so you can only use one provider for all your CrossWorks projects.

Connecting to the source-control system

You must connect each CrossWorks project individually to the source-control system.

To connect to the source-control system:

- Choose **VCS** > **Connect**.

The login dialog that appears enables you to specify your user name and password, and to select the source-control database. These details will be saved in the session file (the password is encrypted) so you won't need to specify this information each time the project is loaded.

To map files in the project to those in the source-control system, specify a local root directory and the corresponding directory in source control (called the **remote root**). Once you have provided this information, the files in your project that are within the local root directory are considered to be in, or can be added to, source control.

After the login dialog, you will be presented with a dialog where you specify the local and remote roots. The local root can be selected using a directory browser and the remote root can be selected using the source-control explorer. With both browsers, you can create new directories—such as when starting a new project or if you don't have any projects in source control.

Opening a project from source control

To fetch a project in source control to a local directory:

1. Choose **Source Control** > **Open Solution**. This will show the login dialog and then the source-control explorer.
2. Select a CrossWorks project file (the extension is **.{hzp}**) using the file list of the source-control explorer.
3. Use the mappings dialog that appears to specify the local root directory, i.e., where you want the project files to go.
4. A dialog will list the files to get from source control and, after confirmation, those files are fetched and the project file is loaded into CrossWorks.

Files source-control status

Determining the source-control status of a file can be an expensive operation. CrossWorks will do this when:

- A file node is selected by the **Project Explorer**.
- The source-control status is displayed in the **Project Explorer** and the file node is visible there, too.
- Before a recursive source-control operation.
- After a source-control operation.

A file will be in one of the following states:

- *Controlled*: The file is in source control.
- *Not Controlled*: The file is not in source control.
- *Checked Out*: The file is checked out.
- *Old*: The file is older than the most-recent version in source control.
- *Checked Out and Old*: Both of the above.

If the file has been modified, its status is displayed in red in the **Project Explorer**. Note that, if a file is not under the local root, it will not have a source-control status.

You can reset any stored source-control file status by choosing **Project** > **Source Control** > **Refresh Status**.

Source-control operations

Source-control operations can be performed on single files or recursively on multiple files in the **Project Explorer** hierarchy. Single-file operations are available on the **Source Control** toolbar and on the text editor's shortcut menu. All operations are available using the **VCS** menu. The operations are described in terms of the **Project Explorer** shortcut menu.

Adding files to source control

To add a single file to the source-control system:

1. In the **Project Explorer**, right-click a file.
2. Choose **Add To Source Control**.
3. Optionally, add a comment to help distinguish this version of the file later.
4. Click **OK**.

To add multiple files to the source-control system:

1. In the **Project Explorer**, right-click a solution, project, or folder.
2. Choose **Add To Source Control (Recursive)**.
3. The dialog will list the files that can be added, i.e., those whose status is **Not Controlled**.
4. In that dialog, you can deselect any files you don't want to add to source control.
5. You may optionally add a comment.
6. Click **OK**.

Checking files out

To check out a single file in the project:

1. In the **Project Explorer**, right-click the file to check out.
2. Choose **Check Out**.
3. Add a comment.
4. Click **OK**.

To check out multiple files in the project:

1. In the **Project Explorer**, right-click the solution, project, or folder to check out.
2. Choose **Source Control** > **Check Out (Recursive)**.

The dialog box will list the files that can be checked out, i.e., ones whose status is **Controlled**.

1. In the dialog, deselect the files you do not wish want to check out and add a comment.
2. Click **OK**.

Checking files in

To check in a single file:

1. In the **Project Explorer**, right-click the file to check in.
2. Choose **Source Control** > **Check In**.
3. Enter an optional comment.
4. Click **OK**.

To check in multiple files:

1. In the **Project Explorer**, right-click a solution, project, or folder.
2. From the shortcut menu, choose **Source Control** > **Check In (Recursive)**.

The dialog box will list the files that can be checked in.

1. In the dialog, deselect the items you do not wish to check in and add a comment.
2. Click **OK**.

Undoing a check out

To undo the check out of a single file:

1. In the **Project Explorer**, right-click a file node.
2. Choose **Source Control** > **Undo Check Out**.

To undo check out of multiple files:

1. In the **Project Explorer**, right-click a solution, project, or folder.
2. Choose **Source Control** > **Undo Check Out (Recursive)**.

The dialog box will list the files that have a status of **Checked Out** and are therefore eligible for this operation.

1. In the dialog, deselect the files you do not wish to revert.
2. Click **OK**.

Getting the latest version

To get the latest version of a single file:

1. In the **Project Explorer**, right-click the file to get.
2. Choose **Source Control** > **Get Latest Version**.

To get the latest version of multiple files:

1. In the **Project Explorer**, right-click a solution, project, or folder.
2. Choose **Source Control** > **Get Latest Version (Recursive)**.

The dialog will list the files that have a status of **Controlled**, **Checked Out** or **Old**.

1. In the dialog, you can deselect the files you don't want to get.
2. Click **OK**.

Showing differences between files

To show the differences between the file in the project and the version checked into source control, do the following:

1. In the **Project Explorer**, right-click the file.
2. From the shortcut menu, choose **Source Control** > **Show Differences**.

You can use an external diff tool in preference to the built-in CrossWorks diff tool. To define the diff command line CrossWorks generates, choose **Tools** > **Options** > **Source Control** > **Diff Command Line**. The command line is defined as a list of strings to avoid problems with spaces in arguments. The diff command line can contain the following macros:

- **$(localfile)** The filename of the file in the project.
- **$(remotefile)** The filename of the latest version of the file in source control.
- **$(localname)** A display name for $(localfile).
- **$(remotename)** A display name for $(remotefile).

Merging files

To use merging, you must have a merge tool. To define the merge command line CrossWorks generates, choose **Tools** > **Options** > **Source Control** > **Merge Command Line**. The command line is defined as a list of strings to avoid problems with spaces in arguments. The merge command line can contain the following macros:

- **$(localfile)** The filename of the file in the project.
- **$(remotefile)** The filename of the latest version of the file in source control.
- **$(commonfile)** The filename of the version of the file you originally edited and the file which will be produced by the merge tool.
- **$(localname)** A display name for $(localfile).
- **$(remotename)** A display name for $(remotefile).
- **$(commonname)** A display name for $(commonfile).

To merge the file in the project and the version checked into source control:

1. In the **Project Explorer**, right-click the file node.
2. Choose **Source Control** > **Merge**.

When the external tool has finished, if $(commonfile) has been modified, you will be asked if you want to overwrite the file in the project with $(commonfile).

Source-control explorer

The **Source Control Explorer** dialog is used when selecting the remote root directory and when **Project** > **Source Control** > **Open Solution From** is selected.

To activate the Source Control Explorer:

- Choose **VCS** > **Explorer**.

The **Source Control Explorer** lists the directories and files in source control. You can use the directory side of the dialog to create new directories and to refresh the list, if that is required by the source-control provider.

Source-control properties

When a file in the project is in source control, the **Properties** window shows the following properties in the **Source Control Options** group:

Property	Description
Checked Out	If **Yes**, the file is checked out by you to the project location; if **No**, the file is not checked out.
Different	If **Yes**, the checked-out file differs from the version in the source control system; if **No**, they are identical.
File Path	The pathname of the file in the source-control system.
Local Revision	The revision number/name of the local file.
Old Version	If **Yes**, the file in the project location is an old version compared to the latest version in the source-control system.
Provider Status	The source-control provider status of the file.
Remote Revision	The revision number/name of the most-recent version in source control.
Status	The source-control status of the file.

Visual SourceSafe provider

The Visual SourceSafe source-control provider is implemented using the SourceSafe OLE Automation interface provided in ssapi.dll. The two versions of ssapi.dll are supported by CrossWorks, they can be identified (using the OLE/COM Object Viewer of Visual Studio) as:

- Visual SourceSafe 6.0 Type Library (Ver 5.1)
- Visual SourceSafe 8.0 Type Library (Ver 5.2)

Other versions have not been tested. If you get an error when connecting to the database, try registering the type library using the DOS command `regsvr32 ssapi.dll`.

Provider-specific options

The following environment options are supported:

Property	Description
Exclusive Checkout	If **Yes**, check outs will be exclusive.

Connecting to the source-control system

When connecting to source control, the working directory of the SourceSafe root **$** is used to initially set the file mappings. Subsequently, the provider will not change the working directory of SourceSafe or refer to it for operations.

Source-control operations

In general, all CrossWorks source-control operations map directly to SourceSafe operations with default flag behavior.

The **Add To Source Control** operation will create SourceSafe projects, if they don't exist.

Source-control operations are performed one file at a time, there are no recursive SourceSafe project operations.

SourceOffSite provider

The SourceGear SourceOffsite source-control provider has been tested with SourceOffSite Classic 3.5.3 server.

Provider-specific options

The following environment options are supported:

Property	Description
Home Directory	The directory to the `servers` subdirectory in your SOS client installation.
Port	The port number used by your SOS server.
Server	The name or IP address of the computer running the SOS server.

Connecting to the source-control system

When connecting to source control, the working directory of the SourceSafe root **$** is used initially to set the file mappings. Subsequently, the provider will not change the working directory of SourceSafe or refer to it for operations.

Source-control operations

For each server connection, SourceOffSite maintains a file that has control information about the local files that have been fetched. The SourceOffSite provider will keep this file updated as it performs single-file operations. However, to update the project and files list, you will need to use the directory list's shortcut menu to choose **VCS** > **Refresh Status**.

In general, all CrossWorks source-control operations map directly to SourceSafe operations with default flag behavior.

The **Add To Source Control** operation will create SourceSafe projects, if they don't exist.

The operations are performed one file at a time, i.e., there are no recursive SourceSafe project operations.

Subversion provider

The Subversion source-control provider has been tested with SVN 1.4.3.

Provider-specific options

The following environment options are supported:

Property	Description
Executable	The path to the `svn` executable.
Lock Supported	If **Yes**, check out and undo check out operations are supported. Check out will issue the `svn lock` command; check in and undo check out will issue the `svn unlock` command.
Repository URL	The Subversion URL to the repository.

Connecting to the source-control system

When connecting to source control, the provider checks if the local root is in SVN control. If this is the case, the local and remote root will be set accordingly. If the local root is not in SVN control after you have set the remote root, a `svn checkout -N` command will be issued to make the local root SVN controlled. This command will also copy any files in the remote root to the local root.

The user name and password you enter will be supplied with each svn command the provider issues.

Source control operations

The CrossWorks source-control operations have been implemented using SVN commands. There are no multiple file operations, each operation is done on a single file and is committed as part of the operation.

Operation	Command
Get Status	`svn status --show-updates` for local directories that are in SVN control. `svn info` for directories in the repository.
Add To Source Control	`svn add` and `svn commit` for each directory not in SVN control. `svn add` and `svn commit` for the file.
Get Latest	`svn update -N` for each directory not in SVN control. Optional removal of the local file for overwrite case and `svn update`.

Check Out	Optional removal of the local file for overwrite case and `svn update`. `svn lock` to lock the file.
Undo Check Out	`svn unlock` to unlock the file. Optional `svn update` to get the latest version.
Check In	`svn commit` for the file.
Source Control Explorer	`svn list` with a remote root. `svn mkdir` to create directories in the repository.

CVS provider

The CVS source-control provider has been tested with CVSNT 2.5.03. The CVS source-control provider uses the CVS `rls` command to browse the repository—this command is implemented in CVS 1.12 but usage of '.' as the root of the module name is not supported.

Provider-specific options

The following environment options are supported:

Property	Description
CVSROOT	The CVSROOT value to access the repository.
Edit/Unedit Supported	If **Yes**, **Check Out** and **Undo Check Out** commands are supported. Any check-out operation will issue the `cvs edit` command; any check-in or undo-check-out operation will issue the `cvs unedit` command; the status operation will issue the `cvs ss` command.
Executable	The path to the `cvs` executable.
Login/Logout Required	If **Yes**, **Connect** will issue the `cvs login` command.

Connecting to the source-control system

When connecting to source control, the provider checks if the local root is in CVS control. If this is the case, the local and remote root will be set accordingly. If the local root is not in CVS control after you have set the remote root, a `cvs checkout -l -d` command will be issued to make the local root CVS controlled. This command will also copy any files in the remote root to the local root.

Source-control operations

The CrossWorks source-control operations have been implemented using CVS commands. There are no multiple-file operations, each operation is done on a single file and committed as part of the operation.

Operation	Command
Get Status	`cvs status` and optional `cvs editors` for local directories in CVS control. `cvs rls -e` for directories in the repository.
Add To Source Control	`cvs add` for each directory not in CVS control. `cvs add` for the file. `cvs commit` for the file and directories.

Get Latest	`cvs update -l -d` for each directory not in CVS control. `cvs update` to merge the local file. `cvs update -C` to overwrite the local file.
Check Out	Optional `cvs update -C` to get the latest version. `cvs edit` to lock the file.
Undo Check Out	`cvs unedit` to unlock the file. Optional `cvs update` to get the latest version.
Check In	`cvs commit` for the file.
Source Control Explorer	`cvs rls -e` with a remote root starting with '.'. `cvs import` to create directories in the repository.

Package management

Additional target-support functions can be added to, and removed from, CrossWorks with *packages*.

A CrossWorks package is an archive file containing a collection of target-support files. Installing a package involves copying the files it contains to an appropriate destination directory and registering the package with CrossWorks's package system. Keeping target-support files separate from the main CrossWorks installation allows us to support new hardware and issue bug fixes for existing hardware-support files between CrossWorks releases, and it allows third parties to develop their own support packages.

Installing packages

Use the **Package Manager** to automate the download, installation, upgrade and removal of packages.

To activate the Package Manager:

- Choose **Tools** > **Manage Packages**.

In some situations, such as using CrossWorks on a computer without Internet access or when you want to install packages that are not on the Rowley Associates website, you cannot use the **Package Manager** to install packages and it will be necessary to manually install them.

To manually install a package:

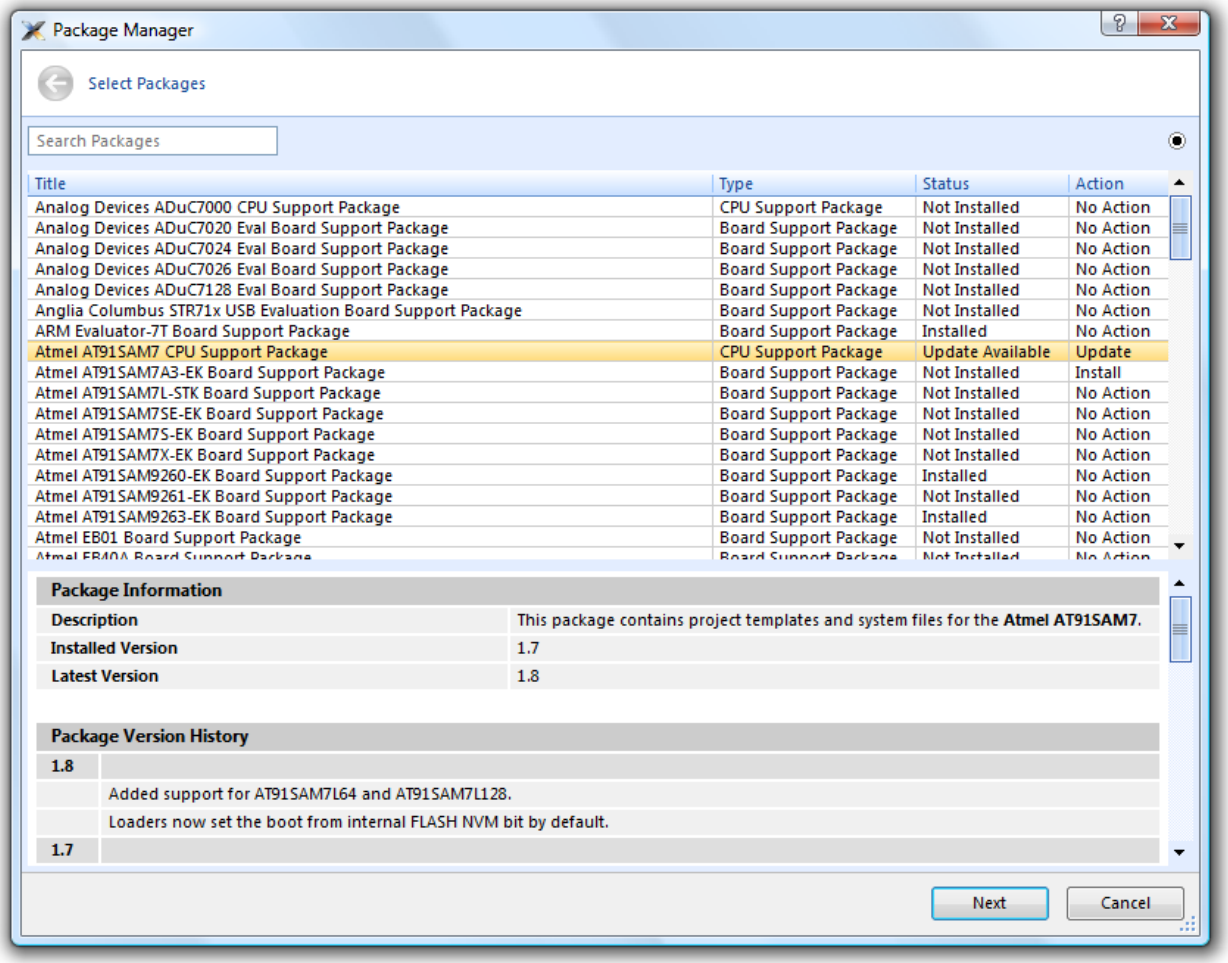
1. Choose **Tools** > **Packages** > **Manually Install Packages**.
2. Select the package file(s) you want to install.
3. Click **Open** to install the packages.

Choose **Tools** > **Show Installed Packages** to see more information on the installed packages.

The **Package Manager** window will remove manually installed packages.

The package manager

The **Package Manager** manages the support packages installed on your system. It lists the available packages, shows the installed packages, and allows you to install, update, reinstall, and remove them.



To activate the Package Manager:

- Choose **Tools** > **Manage Packages**.

Filtering the package list

By default, the **Package Manager** lists all available and installed packages. You can filter the displayed packages in a number of ways.

To filter by package status:

- Click on the disclosure icon near the top-right corner of the dialog.
- Use the pop-up menu to choose how to filter the list of packages.

The list-filter choices are:

- **Display All** — Show all packages irrespective of their status.
- **Display Not Installed** — Show packages that are available but are not currently installed.

- **Display Installed** — Only show packages that are installed.
- **Display Updates** — Only show packages that are installed but are not up-to-date because a newer version is available.

You can also filter the list of packages by the text in the package's title and documentation.

To filter packages by keyword:

- Type the keyword into the **Search Packages** box at the top-left corner of the dialog.

Installing a package

The package-installation operation downloads a package to **$(PackagesDir)/downloads**, if it has not been downloaded already, and unpacks the files contained within the package to their destination directory.

To install a package:

1. Choose **Tools** > **Packages** > **Install Package** (this is equivalent to choosing **Tools** > **Manage Packages** and setting the status filter to **Display Not Installed**).
2. Select the package or packages you wish to install.
3. Right-click the selected packages and choose **Install Selected Packages** from the shortcut menu.
4. Click **Next**; you will be see the actions the **Package Manager** is about to carry out.
5. Click **Next** and the **Package Manager** will install the selected packages.
6. When installation is complete, click **Finish** to close the **Package Manager**.

Updating a package

The package-update operation first removes existing package files, then it downloads the updated package to **$(PackagesDir)/downloads** and unpacks the files contained within the package to their destination directory.

To update a package:

1. Choose **Tools** > **Packages** > **Update Packages** (this is equivalent to clicking **Tools** > **Package Manager** and setting the status filter to **Display Updates**).
2. Select the package or packages you wish to update.
3. Right-click the selected packages and choose **Update Selected Packages** from the shortcut menu.
4. Click **Next**; you will see the actions the **Package Manager** is about to carry out.
5. Click **Next** and the **Package Manager** will update the package(s).
6. When the update is complete, click **Finish** to close the **Package Manager**.

Removing a package

The package-remove operation removes all the files that were extracted when the package was installed.

To remove a package:

1. Choose **Tools** > **Packages** > **Remove Packages** (this is equivalent to choosing **Tools** > **Package Manager** and setting the status filter to **Display Installed**).
2. Select the package or packages you wish to remove.
3. Right-click the selected packages and choose **Remove Selected Packages** from the shortcut menu.
4. Click **Next**; you will see the actions the **Package Manager** is about to carry out.
5. Click **Next** and the **Package Manager** will remove the package(s).
6. When the operation is complete, click **Finish** to close the **Package Manager**.

Reinstalling a package

The package-reinstall operation carries out a package-remove operation followed by a package-install operation.

To reinstall a package:

1. Choose **Tools** > **Packages** > **Reinstall Packages** (this is equivalent to choosing **Tools** > **Package Manager** and setting the status filter to **Display Installed**).
2. Select the package or packages you wish to reinstall.
3. Right-click the packages to reinstall and choose **Reinstall Selected Packages** from the shortcut menu.
4. Click **Next**; you will see the actions the **Package Manager** is about to carry out.
5. Click **Next** and the **Package Manager** will reinstall the packages.
6. When the operation is complete, click **Finish** to close the **Package Manager**.

Exploring your application

In this section, we discuss the CrossStudio tools that help you examine how your application is built.

Project explorer

The **Project Explorer** is the user interface of the CrossWorks project system. It organizes your projects and files and provides access to the commands that operate on them. A toolbar at the top of the window offers quick access to commonly used commands for the selected project node or the active project. Right-click to reveal a shortcut menu with a larger set of commands that will work on the selected project node, ignoring the active project.

The selected project node determines what operations you can perform. For example, the **Compile** operation will compile a single file if a file project node is selected; if a folder project node is selected, each of the files in the folder are compiled.

You can select project nodes by clicking them in the **Project Explorer**. Additionally, as you switch between files in the editor, the selection in the **Project Explorer** changes to highlight the file you're editing.

To activate the Project Explorer:

- Choose **View** > **Project Explorer** or press **Ctrl**+**Alt**+**P**.

Left-click operations

The following operations are available in the **Project Explorer** with a left-click of the mouse:

Action	Description
Single click	Select the node. If the node is already selected and is a solution, project, or folder node, a rename editor appears.
Double click	Double-clicking a solution node or folder node will open/close the node. Double-clicking a project node sets it as the active project. Double-clicking a file opens the file with the default editor for that file's type.

Toolbar commands

The following buttons are on the toolbar:

Button	Description
	Adds a new file to the active project using the **New File** dialog.
	Adds existing files to the active project.

	Removes files, folders, projects, and links from the project.
	Creates a new folder in the active project.
	Menu of build operations.
	Disassembles the active project.
	Menu of **Project Explorer** options.
	Displays the properties dialog for the selected item.

Shortcut menu commands

The shortcut menu, displayed by right-clicking, contains the commands listed below.

For solutions:

Item	Description
Build and Batch Build	Build all projects under the solution in the current or batch build configuration.
Rebuild and Batch Rebuild	Rebuild all projects under the solution in the current or batch build configuration.
Clean and Batch Clean	Remove all output and intermediate build files for the projects under the solution in the current or batch build configuration.
Export Build and Batch Export Build	Create an editor with the build commands for the projects under the solution in the current or batch build configuration.
Add New Project	Add a new project to the solution.
Add Existing Project	Create a link from an existing solution to this solution.
Paste	Paste a copied project into the solution.
Remove	Remove the link to another solution from the solution.
Rename	Rename the solution node.
Source Control Operations	Source-control operations on the project file and recursive operations on all files in the solution.

Edit Solution As Text	Create an editor containing the project file.
Save Solution As	Change the filename of the project file—note that the saved project file is not reloaded.
Properties	Show the **Properties** dialog with the solution node selected.

For projects:

Item	Description
Build and Batch Build	Build the project in the current or batch build configuration.
Rebuild and Batch Rebuild	Reuild the project in the current or batch build configuration.
Clean and Batch Clean	Remove all output and intermediate build files for the project in the current or batch build configuration.
Export Build and Batch Export Build	Create an editor with the build commands for the project in the current or batch build configuration.
Link	Perform the project node build operation: link for an Executable project type, archive for a Library project type, and the combine command for a Combining project type.
Set As Active Project	Set the project to be the active project.
Debugging Commands	For Executable and Externally Built Executable project types, the following debugging operations are available on the project node: **Start Debugging**, **Step Into Debugging**, **Reset And Debug**, **Start Without Debugging**, **Attach Debugger**, and **Verify**.
Memory-Map Commands	For Executable project types that don't have memory-map files in the project and have the memory-map file project property set, there are commands to view the memory-map file and to import it into the project.
Section-Placement Commands	For Executable project types that don't have section-placement files in the project but have the section-placement file project property set, there are commands to view the section-placement file and to import it into the project.
Target Processor	For Executable and Externally Built Executable project types that have a Target Processor property group, the selected target can be changed.

Add New File	Add a new file to the project.
Add Existing File	Add an existing file to the project.
New Folder	Create a new folder in the project.
Cut	Cut the project from the solution.
Copy	Copy the project from the solution.
Paste	Paste a copied folder or file into the project.
Remove	Remove the project from the solution.
Rename	Rename the project.
Source Control Operations	Source-control, recursive operations on all files in the project.
Find in Project Files	Run **Find in Files** in the project directory.
Properties	Show the **Project Manager** dialog and select the project node.

For folders:

Item	Description
Add New File	Add a new file to the folder.
Add Existing File	Add an existing file to the folder.
New Folder	Create a new folder in the folder.
Cut	Cut the folder from the project or folder.
Copy	Copy the folder from the project or folder.
Paste	Paste a copied folder or file into the folder.
Remove	Remove the folder from the project or folder.
Rename	Rename the folder.
Source Control Operations	Source-control recursive operations on all files in the folder.
Compile	Compile each file in the folder.
Properties	Show the properties dialog with the folder node selected.

For files:

Item	Description
Open	Edit the file with the default editor for the file's type.
Open With	Edit the file with a selected editor. You can choose from the **Binary Editor**, **Text Editor**, and **Web Browser**.
Select in File Explorer	Create a operating system file system window with the file selected.
Compile	Compile the file.
Export Build	Create an editor window containing the commands to compile the file in the active build configuration.
Exclude From Build	Set the **Exclude From Build** property to **Yes** for this project node in the active build configuration.
Disassemble	Disassemble the output file of the compile into an editor window.
Preprocess	Run the C preprocessor on the file and show the output in an editor window.
Cut	Cut the file from the project or folder.
Copy	Copy the file from the project or folder.
Remove	Remove the file from the project or folder.
Import	Import the file into the project.
Source Control Operations Source-control operations on the file.	
Properties	Show the properties dialog with the file node selected.

Source navigator window

One of the best ways to find your way around your source code is using the **Source Navigator**. It parses the active project's source code and organizes classes, functions, and variables in various ways.

The main part of the **Source Navigator** window provides an overview of your application's functions, classes, and variables.

CrossStudio displays these icons to the left of each object:

Icon	Description
	A C or C++ structure or a C++ namespace.
	A C++ class.
	A C++ member function declared `private` or a function declared with `static` linkage.
	A C++ member function declared `protected`.
	A C++ member function declared `public` or a function declared with `extern` linkage.
	A C++ member variable declared `private` or a variable declared with `static` linkage.
	A C++ member variable declared `protected`.
	A C++ member variable declared `public` or a variable declared with `extern` linkage.

To activate the Source Navigator:

- Choose **Tools** > **Source Navigator** or press **Ctrl+Alt+N**.

Re-parsing after editing

The **Source Navigator** does not update automatically, only when you ask it to.

To parse source files manually, click the **Refresh** button on the **Source Navigator** toolbar.

CrossStudio re-parses any changed files and updates the **Source Navigator** with the changes. Progress information and any errors are sent to the Source Navigator Log in the **Output** window when parsing.

Sorting and grouping

You can group objects by their type; that is, whether they are classes, functions, namespaces, structures, or variables. Each object is placed into a folder according to its type.

To group objects by type:

1. On the **Source Navigator** toolbar, click the arrow to the right of the **Cycle Grouping** button.
2. Choose **Group By Type**

Symbol browser

The **Symbol Browser** shows useful information about your linked application and complements the information displayed in the **Project Explorer** window. You can select different ways to filter and group the information in the **Symbol Browser** to provide an at-a-glance overview of your application. You can use the **Symbol Browser** to *drill down* to see the size and location of each part of your program. The way symbols are sorted and grouped is saved between runs; so, when you rebuild an application, CrossStudio automatically updates the **Symbol Browser** so you can see the effect of your changes on the memory layout of your program.

User interface

Button	Description
	Groups symbols by source filename.
	Groups symbols by symbol type (equates, functions, labels, sections, and variables).
	Groups symbols by the section where they are defined.
	Moves the cursor to the statement that defined the symbol.
	Determines what columns to display.

The main part of the **Symbol Browser** displays each symbol (both external and static) that is linked into an application. CrossStudio displays the following icons to the left of each symbol:

Icon	Description
	**Private Equate** A private symbol not defined relative to a section.
	**Public Equate** A public symbol that is not defined relative to a section.
	**Private Function** A private function symbol.
	**Public Function** A public function symbol.
	**Private Label** A private data symbol, defined relative to a section.
	**Public Label** A public data symbol, defined relative to a section.

	**Section** A program section.

Choosing what to show

To activate the Symbol Browser window:

- Choose **Tools** > **Symbol Browser** or press **Ctrl**+**Alt**+**Y**.

You can choose to display the following fields for each symbol:

- *Value*: The value of the symbol. For labels, code, and data symbols, this will be the address of the symbol. For absolute or symbolic equates, this will be the value of the symbol.
- *Range*: The range of addresses the code or data item covers. For code symbols that correspond to high-level functions, the range is the range of addresses used for that function's code. For data addresses that correspond to high-level **static** or **extern** variables, the range is the range of addresses used to store that data item. These ranges are only available if the corresponding source file was compiled with debugging information turned on: if no debugging information is available, the range will simply be the first address of the function or data item.
- *Size*: The size, in bytes, of the code or data item. The **Size** column is derived from the **Range** of the symbol: if the symbol corresponds to a high-level code or data item and has a range, **Size** is calculated as the difference between the start and end addresses of the range. If a symbol has no range, the size column is blank.
- *Section*: The section in which the symbol is defined. If the symbol is not defined within a section, the **Section** column is blank.
- *Type*: The high-level type for the data or code item. If the source file that defines the symbol is compiled with debugging information turned off, type information is not available and the **Type** column is blank.

Initially the **Range** and **Size** columns are shown in the **Symbol Browser**. To select which columns to display, use the **Field Chooser** button on the **Symbol Browser** toolbar.

To select the fields to display:

1. Click the **Field Chooser** button on the **Symbol Browser** toolbar.
2. Select the fields you wish to display and deselect the fields you wish to hide.

Organizing and sorting symbols

When you group symbols by section, each symbol is grouped underneath the section in which it is defined. Symbols that are absolute or are not defined within a section are grouped beneath '(No Section)'.

To group symbols by section:

1. On the **Symbol Browser** toolbar, click the arrow next to the **Cycle Grouping** button.

2. From the pop-up menu, choose **Group By Section**.

The **Cycle Grouping** icon will change to indicate that the **Symbol Browser** is grouping symbols by section.

When you group symbols by type, each symbol is classified as one of the following:

- An *Equate* has an absolute value and is not defined as relative to, or inside, a section.
- A *Function* is defined by a high-level code sequence.
- A *Variable* is defined by a high-level data declaration.
- A *Label* is defined by an assembly language module. *Label* is also used when high-level modules are compiled with debugging information turned off.

When you group symbols by source file, each symbol is grouped underneath the source file in which it is defined. Symbols that are absolute, are not defined within a source file, or are compiled without debugging information, are grouped beneath '(Unknown)'.

To group symbols by type:

1. On the **Symbol Browser** toolbar, click the arrow next to the **Cycle Grouping** button.
2. Choose **Group By Type** from the pop-up menu.

The **Cycle Grouping** icon will change to indicate that the **Symbol Browser** is grouping symbols by type.

To group symbols by source file:

1. On the **Symbol Browser** toolbar, click the arrow next to the **Cycle Grouping** button.
2. Choose **Group By Source File**.

The **Cycle Grouping** icon will change to indicate that the **Symbol Browser** is grouping symbols by source file.

When you sort symbols alphabetically, all symbols are displayed in a single list in alphabetical order.

To list symbols alphabetically:

1. On the **Symbol Browser** toolbar, click the arrow next to the **Cycle Grouping** button.
2. Choose **Sort Alphabetically**.

The **Cycle Grouping** icon will change to indicate that the **Symbol Browser** is grouping symbols alphabetically.

Filtering and finding symbols

When you're dealing with big projects with hundreds, or even thousands, of symbols, a way to filter those symbols in order to isolate just the ones you need is very useful. The **Symbol Browser**'s toolbar provides an editable *combo*box} you can use to specify the symbols you'd like displayed. You can type '*' to match a sequence of zero or more characters and '?' to match exactly one character.

The symbols are filtered and redisplayed as you type into the combo box. Typing the first few characters of a symbol name is usually enough to narrow the display to the symbol you need. *Note:* the C compiler prefixes all high-level language symbols with an underscore character, so the variable `extern int u` or the function `void fn(void)` have low-level symbol names _u and _fn. The **Symbol Browser** uses the low-level symbol name when displaying and filtering, so you must type the leading underscore to match high-level symbols.

To display symbols that start with a common prefix:

- Type the desired prefix text into the combo box, optionally followed by a "*".

For instance, to display all symbols that start with "**i2c**_", type "**i2c**_" and all matching symbols are displayed—you don't need to add a trailing "*" in this case, because it is implied.

To display symbols that end with a common suffix:

- Type '*' into the combo box, followed by the required suffix.

For instance, to display all symbols that end in '_data', type '*_data' and all matching symbols are displayed—in this case, the leading '*' is required.

When you have found the symbol you're interested in and your source files have been compiled with debugging information turned on, you can jump to a symbol's definition using the **Go To Definition** button.

To jump to the definition of a symbol:

1. Select the symbol from the list of symbols.
2. On the **Symbol Browser** toolbar, click **Go To Definition**.

—or—

1. Right-click the symbol in the list of symbols.
2. Choose **Go To Definition** from the shortcut menu.

Watching symbols

If a symbol's range and type is known, you can add it to the most recently opened **Watch** window or **Memory** window.

To add a symbol to the Watch window:

1. In the **Symbol Browser**, right-click the symbol you wish to add to the **Watch** window.
2. On the shortcut menu, choose **Add To Watch**.

To add a symbol to the Memory window:

1. In the **Symbol Browser**, right-click the symbol you wish to add to the **Memory** window.

2. Choose **Locate Memory** from the shortcut menu.

Using size information

Here are a few common ways to use the **Symbol Browser**:

What function uses the most code space? What requires the most data space?

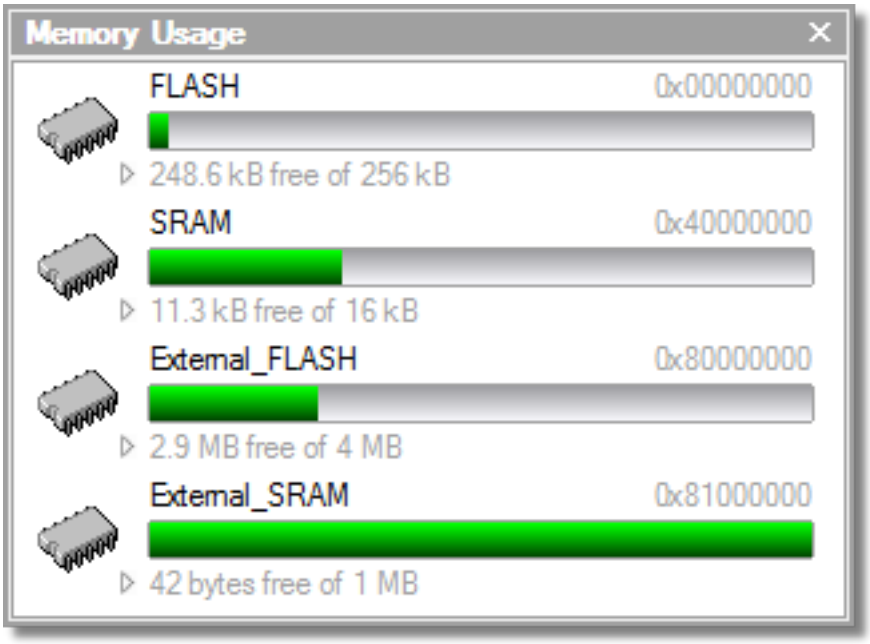
1. Choose **Tools** > **Symbol Browser** or press **Ctrl+Alt+Y**.
2. In the **Grouping** button menu on the **Symbol Browser** toolbar, select **Group By Type**.
3. Ensure the **Size** field is checked in the **Field Chooser** button's menu.
4. Ensure that the filter on the **Symbol Browser** toolbar is empty.
5. Click on the **Size** field in the header to sort by data size.
6. The sizes of variables and of functions are shown in separate lists.

What's the overall size of my application?

1. Choose **Tools** > **Symbol Browser** or press **Ctrl+Alt+Y**.
2. In the **Grouping** button menu on the **Symbol Browser** toolbar, select **Group By Section**.
3. Ensure the **Range** and **Size** fields are checked in the **Field Chooser** button's menu.
4. Read the section sizes and ranges of each section in the application.

Memory usage window

The **Memory Usage** window displays a graphical summary of how memory has been used in each memory segment of a linked application.



Each bar represents an entire memory segment. Green represents the area of the segment that contains code or data.

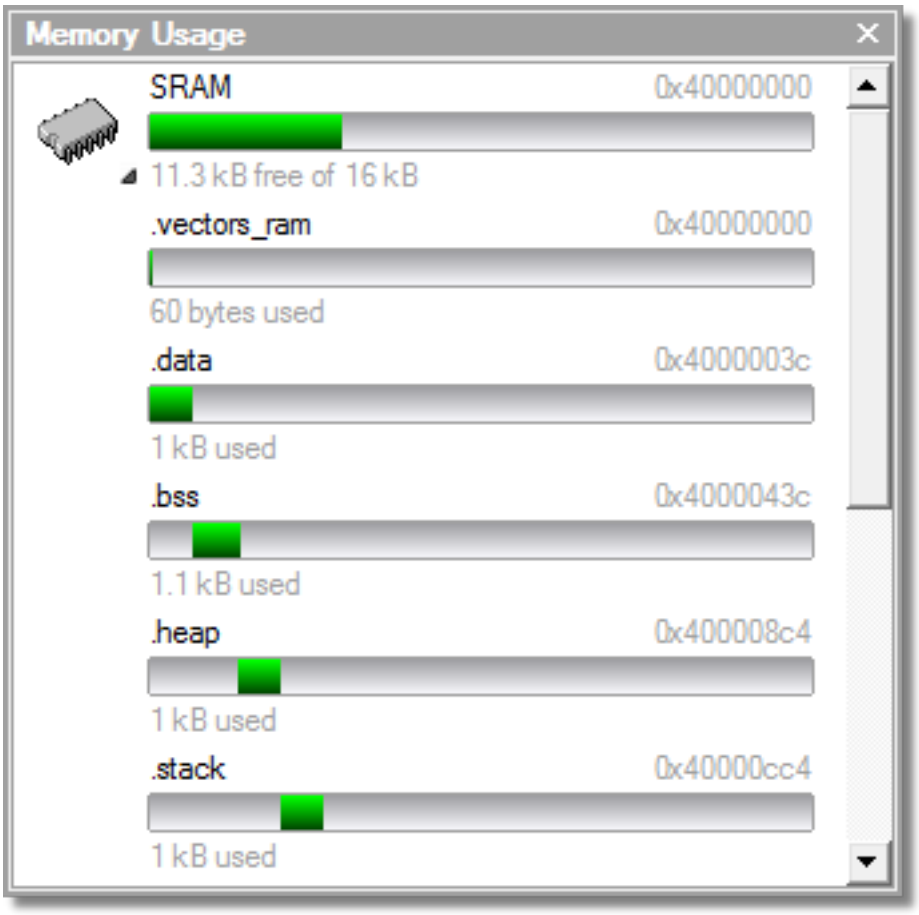
To activate the Memory Usage window:

- Choose **View** > **Memory Usage** or press **Ctrl**+**Alt**+**Z**.

The memory-usage graph will only be visible if your active project's target is an executable file and the file exists. If the executable file has not been linked by CrossStudio, memory-usage information may not be available.

Displaying section information

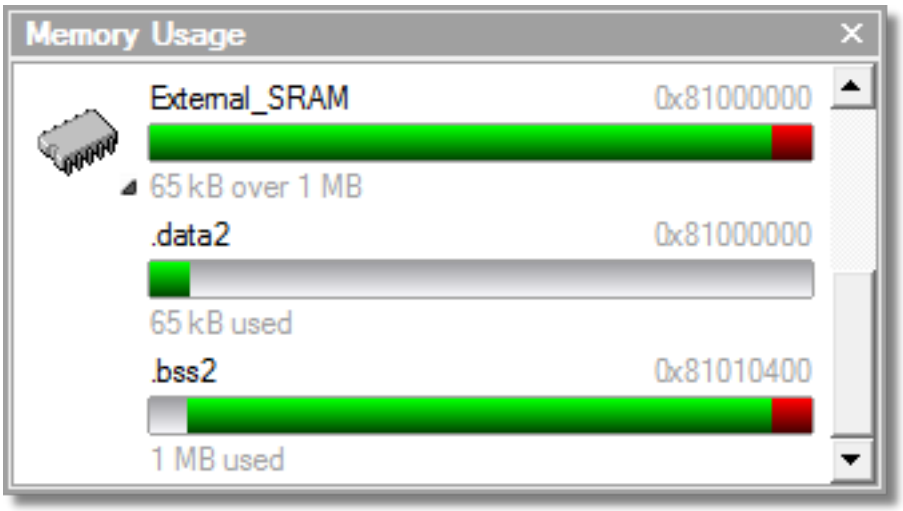
The **Memory Usage** window can also be used to visualize how program sections have been placed in memory. To display the program sections, simply click the memory segment to expand it; or, alternatively, right-click and choose **Show Memory Sections** from the shortcut menu.



Each bar represents an entire memory segment. Green represents the area of the segment that contains the program section.

Displaying segment overflow

The **Memory Usage** window also displays segment overflows when the total size of the program sections placed in a segment is larger than the segment size. When this happens, the segment and section bars represents the total memory used, green areas represent the code or data within the segment, and red areas represent code or data placed outside the segment.



Getting more-detailed information

If you require more-detailed information than that provided by the **Memory Usage** window, such as the location of specific objects within memory, use the **Symbol browser**.

Editing your code

CrossStudio has a built-in editor that allows you to edit text, but some features make it particularly well suited to editing code.

You can open multiple code editors to view or edit project source code, and you can copy and paste among them. The **Windows** menu contains a list of all open code editors.

The code editor supports the language of the source file it is editing, showing code with syntax highlighting and offering smart indenting.

You can open a code editor in several ways, some of which are:

- By double-clicking a file in the **Project Explorer** or by right-clicking a file and selecting **Open** from the shortcut menu.
- Using the **File** > **New File** or **File** > **Open** commands.

Elements of the code editor

The code editor is composed of several elements, which are described here.

- *Code pane*: The area where you edit code. You can set options that affect the code pane's text indents, tabs, drag-and-drop behavior, and so forth.
- *Margin gutter*: A gray area on the left side of the code editor where margin indicators such as breakpoints, bookmarks, and shortcuts are displayed. Clicking this area sets a breakpoint on the corresponding line of code.
- *Horizontal and vertical scroll bars*: You can scroll the code pane horizontally and vertically to view code that extends beyond the edges of the pane.

Basic editing

This section is a whirlwind tour of the basic editing features CrossStudio's code editor provides.

Whether you are editing code, HTML, or plain text, the code editor is just like many other text editors or word processors. For code that is part of a project, the project's programming language support provides syntax highlighting (colorization), indentation, and so on.

This section *is not* a reference for everything the code editor provides; for that, look in the following sections.

Moving the insertion point

The most common way to navigate through text is to use use the keyboard's cursor keys, the scroll bars, or the mouse's scroll wheel.

The keystrokes most commonly used to navigate through a document are:

Keystroke	Description
Up	Moves the insertion point up one line
Down	Moves the insertion point down one line
Left	Moves the insertion point left one character
Right	Moves the insertion point right one character
Home	Moves the insertion point to the first character on the line; pressing **Home** a second time moves the insertion point to the leftmost column
End	Moves the insertion point to the end of the line
PageUp	Moves the insertion point up one page
PageDown	Moves the insertion point down one page
Ctrl+Left	Moves the insertion point left one word
Ctrl+Right	Moves the insertion point right one word
Ctrl+Home	Moves the insertion point to the start of the document
Ctrl+End	Moves the insertion point to the end of the document
Alt+Up	Moves the insertion point up five lines
Alt+Down	Moves the insertion point down five lines
Ctrl+Up	Scrolls the document up one line in the window without moving the insertion point
Ctrl+Down	Scrolls the document down one line in the window without moving the insertion point

Selecting text

You can select text by using the keyboard or the mouse.

To select text with the keyboard:

- Hold down the **Shift** key while using the cursor keys.

To select text with the mouse:

1. Click the start of the selection.
2. Drag the mouse to mark the selection.
3. Release the left mouse button to end selection.

Adding text

The editor has two text-input modes:

- *Insertion mode*: As you type on the keyboard, text is entered at the insertion point and any text to the right of the cursor is shifted along. A visual indication of insertion mode is that the cursor is a flashing line.
- *Overstrike mode*: As you type on the keyboard, text at the insertion point is replaced with your typing. A visual indication of insertion mode is that the cursor is a flashing block.

Insert and overstrike modes are common to *all* editors: if one editor is in insert mode, *all* editors are in insert mode. To configure the cursor appearance, choose **Tools** > **Options**.

To toggle between insertion and overstrike mode:

- Click **Insert**.

If overstrike mode is enabled, the **OVR** status indicator will be enabled and the cursor will change to the overstrike cursor.

To add or insert text:

1. Move the insertion point to the place text is to be inserted.
2. Enter the text using the keyboard.

To overwrite characters in an existing line, press the **Insert** key to put the editor into overstrike mode.

Deleting text

The text editor supports the following common editing keystrokes:

Keystroke	Description
Backspace	Deletes the character before the insertion point
Delete	Deletes the character after the insertion point
Ctrl+Backspace	Deletes one word before the insertion point
Ctrl+Delete	Deletes one word after the insertion point

To delete characters or words:

1. Place the insertion point before the word or letter you want to delete.
2. Press **Delete** as many times as needed.

—or—

1. Place your cursor at the end of the letter or word you want to delete.
2. Press **Backspace** as many times as needed.

To delete text that spans more than a few characters:

1. Select the text you want to delete.
2. Press **Delete** or **Backspace** to delete it.

Using the clipboard

To copy selected text to the clipboard:

- Choose **Edit** > **Copy** or press **Ctrl**+**C**.

The Windows standard key sequence **Ctrl**+**Ins** also copies text to the clipboard.

To cut selected text to the clipboard:

- Choose **Edit** > **Cut** or press **Ctrl**+**X**.

The Windows standard key sequence **Shift**+**Del** also cuts text to the clipboard.

To insert the clipboard content at the insertion point:

- Choose **Edit** > **Paste** or press **Ctrl**+**V**.

The Windows standard key sequence **Shift**+**Ins** also inserts the clipboard content at the insertion point.

Undo and redo

The editor has an **Undo** facility to undo previous editing actions. The **Redo** feature can be used to re-apply previously undone actions.

To undo one editing action:

- Choose **Edit** > **Undo** or press **Ctrl**+**Z**.

The Windows standard key sequence **Alt**+**Backspace** also undoes an edit.

To undo multiple editing actions:

1. On the **Standard** toolbar, click the arrow next to the **Undo** button.
2. Select the editing operations to undo.

To undo all edits:

- Choose **Edit** > **Advanced** > **Undo All** or press **Ctrl**+**K, Ctrl**+**Z**.

To redo one editing action:

- Choose **Edit** > **Redo** or press **Ctrl**+**Y**.

The Windows standard key sequence **Alt**+**Shift**+**Backspace** also redoes an edit.

To redo multiple editing actions:

1. On the **Standard** toolbar, click the arrow next to the **Redo** tool button.
2. From the pop-up menu, select the editing operations to redo.

To redo all edits:

- Choose **Edit** > **Advanced** > **Redo All** or press **Ctrl**+**K, Ctrl**+**Y**.

Drag and drop

You can select text, then drag it to another location. You can drop the text at a different location in the same window or in another one.

To drag and drop text:

1. Select the text you want to move.
2. Press and hold the mouse button to drag the selected text to where you want to place it.
3. Release the mouse button to drop the text.

Dragging text *moves* it to the new location. To *copy* it to a new location, hold down the **Ctrl** key while dragging the text: the cursor changes to indicate a copy operation. Press the **Esc** key while dragging text to cancel the drag-and-drop edit.

By default, drag-and drop-editing is *disabled* and you must enable it if you want to use it.

To enable or disable drag-and-drop editing:

1. Choose **Tools** > **Options** or press **Alt+,**.
2. Click **Text Editor**.
3. Set **Allow Drag and Drop Editing** to **Yes** to enable or to **No** to disable drag-and-drop editing.

Searching

To find text in the current file:

1. Press **Ctrl+F**.
2. Enter the string to search for.

As you type, the editor searches the file for a match. The pop-up shows how many matches are in the current file. To move through the matches while the **Find** box is still active, press **Tab** or **F3** to move to the next match and **Shift+Tab** or **Shift+F3** to move to the previous match.

Advanced editing

You can do anything using its basic code-editing features, but the CrossStudio text editor has a host of labor-saving features that make editing programs a snap.

This section describes the code-editor features intended to make editing source code easier.

Indenting source code

The editor can increase or decrease the indentation level of a selection.

To increase indentation of selected text:

- Choose **Selection** > **Increase Line Indent** or press **Tab**.

To decrease indentation of selected text:

- Choose **Selection** > **Decrease Line Indent** or press **Shift+Tab**.

Commenting out sections of code

To comment selected text:

- Choose **Selection** > **Comment** or press **Ctrl**+/.

To uncomment selected text:

- Choose **Selection** > **Uncomment** or press **Ctrl**+**Shift**+/.

You can also toggle the commenting of a selection by typing /. This has no menu equivalent.

Changing letter case

The editor can change the case of the current word or the selection. The editor will change the case of the selection, if there is a selection, otherwise it will change the case of word at the insertion point.

To change text to uppercase:

- Choose **Selection** > **Make Uppercase** or press **Ctrl**+**Shift**+**U**.

This changes, for instance, 'Hello' to 'HELLO'.

To change text to lowercase:

- Choose **Selection** > **Make Lowercase** or press **Ctrl**+**U**.

This changes, for instance, 'Hello' to 'hello.'

To switch between uppercase and lowercase:

- Choose **Selection** > **Switch Case**.

This changes, for instance, 'Hello' to 'hELLO.'

With large software teams or imported source code, sometimes identifiers don't conform to your local coding style. To assist in conversion between two common coding styles for identifiers, CrossStudio's editor offers the following two shortcuts:

To change from split case to camel case:

- Choose **Selection** > **Camel Case** or press **Ctrl**+**K, Ctrl**+**Shift**+**U**.

This changes, for instance 'this_is_wrong' to 'thisIsWrong.'

To change from camel case to split case:

- Choose **Selection** > **Split Case** or press **Ctrl**+**K, Ctrl**+**U**.

This changes, for instance 'thisIsWrong' to 'this_is_wrong.'

Indenting

The editor uses the **Tab** key to increase or decrease the indentation level. The indentation size can be changed in the **Language Properties** pane of the editor's **Properties** window, as can all the indent-related features listed below.

To change the indentation size:

1. Set the **Indent Size** property for the required language.

The editor can optionally use tab characters to fill whitespace when indenting. The use of tabs for filling whitespace can be selected in the editor's **Language** settings.

To select tab or space fill when indenting:

1. Set the **Use Tabs** property for the required language. *Note:* changing this setting does not add or remove existing tabs from files, the change will only affect new indents.

The editor can assist with source code indentation while inserting text. There are three levels of indentation assistance:

- *None*: The indentation of the source code is left to the user.
- *Indent*: This is the default. The editor maintains the current indentation level. When you press **Return** or **Enter**, the editor moves the insertion point down one line and indented to the same level as the now-previous line.
- *Smart*: The editor analyzes the source code to compute the appropriate indentation level for each line. You can change how many lines before the cursor position will be analyzed for context. The smart-indent mode can be configured to indent either open and closing braces or the lines following the braces.

Changing indentation options:

To change the indentation mode:

1. Set the **Indent Mode** property for the required language.

To change whether opening braces are indented in smart-indent mode:

1. Set the **Indent Opening Brace** property for the required language.

To change whether closing braces are indented in smart-indent mode:

1. Set the **Indent Closing Brace** property for the required language.

To change the number of previous lines used for context in smart-indent mode:

1. Set the **Indent Context Lines** property for the required language.

Bookmarks

To edit a document elsewhere and then return to your current location, add a bookmark.

Dropping bookmarks

To place a bookmark:

1. Move the insertion point to the line you wish to bookmark.
2. Choose **Edit** > **Bookmarks** > **Toggle Bookmark** or press **Ctrl+F2**.

A temporary bookmark symbol appears next to the line in the indicator margin to show the bookmark is set.

You will also find the **Toggle Bookmark** button in the **Text Edit** toolbar.

Moving through bookmarks you've set

You can move through the bookmarks you've set in the current document or in all documents:

Keystroke	Description
F2	Moves to the next bookmark in the current document
BookmarkOrevInDocument	Moves to the previous bookmark in the current document
Ctrl+Q, F2	Moves to the first bookmark in the current document
Ctrl+Q, Shift+F2	Moves to the last bookmark in the current document

To navigate forward through bookmarks:

1. Choose **Edit** > **Bookmarks** > **Next Bookmark In Document** or press **F2**.
2. The editor moves the insertion point to the next bookmark in the document.

If there is no following bookmark, the insertion point is moved to the first bookmark in the document.

To navigate backward through bookmarks:

1. Choose **Edit** > **Bookmarks** > **Previous Bookmark In Document** or press **Shift+F2**.
2. The editor moves the insertion point to the previous bookmark in the document.

If there is no previous bookmark, the insertion point is moved to the last bookmark in the document.

Removing bookmarks

To remove a bookmark:

1. Move the insertion point to the line containing the bookmark.
2. Choose **Edit** > **Bookmarks** > **Toggle Bookmark** or press **Ctrl+F2**.

The bookmark symbol disappears, indicating the bookmark has been removed.

To remove all bookmarks in a document:

- Choose **Edit** > **Bookmarks** > **Clear Bookmarks In Document** or press **Ctrl+K, F2**.

Find and Replace window

The **Find and Replace** window allows you to search for and replace text in the current document or in a range of specified files.

To activate the find-and-replace window:

- Choose **Search** > **Find And Replace** or press **Ctrl+Alt+F**.

To find text in a single file:

- Select **Current Document** in the context combo box.
- Enter the string to be found in the text edit input.
- If the search will be case sensitive, set the **Match case** option.
- If the search will be for a whole word—i.e., there will be whitespace, such as spaces or the beginning or end of the line, on both sides of the string being searched for—set the **Whole word** option.
- If the search string is a regular expression, set the **Use regexp** option.
- Click the **Find** button to find all occurrences of the string in the current document.

To find and replace text in a single file:

- Click the **Replace** button on the toolbar.
- Enter the string to search for into the **Find what** input.
- Enter the replacement string into the **Replace with** input. If the search string is a regular expression, the *n* back-reference can be used in the replacement string to reference captured text.
- If the search will be case sensitive, set the **Match case** option.
- If the search will be for a whole word—i.e., there will be whitespace, such as spaces or the beginning or end of the line, on both sides of the string being searched for—set the **Match whole word** option.
- If the search string is a regular expression, set the **Use regular expression** option.
- Click the **Find Next** button to find next occurrence of the string, then click the **Replace** button to replace the found string with the replacement string; or click **Replace All** to replace all occurrences of the search string without prompting.

To find text in multiple files:

- Click the **Find In Files** button on the toolbar.
- Enter the string to search for into the **Find what** input.
- Select the appropriate option in the **Look in** input to select whether to carry out the search in all open documents, all documents in the current project, all documents in the current solution, or all files in a specified folder.
- If you have specified that you want to search in a folder, select the folder you want to search by entering its path in the **Folder** input and use the **Look in files matching** input to specify the type of files you want to search.

- If the search will be case sensitive, set the **Match case** option.
- If the search will be for a whole word—i.e., there will be whitespace, such as spaces or the beginning or end of the line, on both sides of the string being searched for—set the **Match whole word** option.
- If the search string is a regular expression, set the **Use regular expression** option.
- Click the **Find All** button to find all occurrences of the string in the specified files, or click the **Bookmark All** button to bookmark all the occurrences of the string in the specified files.

To replace text in multiple files:

- Click the **Replace In Files** button on the toolbar.
- Enter the string to search for into the **Find what** input.
- Enter the replacement string into the **Replace with** input. If the search string is a regular expression, the *n* back-reference can be used in the replacement string to reference captured text.
- Select the appropriate option in the **Look in** input to select whether you want to carry out the search and replace in all open documents, all documents contained in the current project, all documents in the current solution, or all files in a specified folder.
- If you have specified that you want to search in a folder, select the folder you want to search by entering its path in the **Folder** input and use the **Look in files matching** input to specify the type of files you want to search.
- If the search will be case sensitive, set the **Match case** option.
- If the search will be for a whole word—i.e., there will be whitespace, such as spaces or the beginning or end of the line, on both sides of the string being searched for—set the **Match whole word** option.
- If the search string is a regular expression, set the **Use regular expression** option.
- Click the **Replace All** button to replace all occurrences of the string in the specified files.

Clipboard-ring window

The code editor captures all cut and copy operations, and stores the cut or copied item on the *clipboard ring*. The clipboard ring stores the last 20 cut or copied text items, but you can configure the maximum number by using the environment options dialog. The clipboard ring is an excellent place to store scraps of text when you're working with many documents and need to cut and paste between them.

To activate the clipboard ring:

- Choose **Edit** > **Clipboard Ring** > **Clipboard Ring** or press **Ctrl**+**Alt**+**C**.

—or—

- Right-click the toolbar area to display the **View** menu.
- Choose **Clipboard Ring** from the shortcut menu.

To paste from the clipboard ring:

1. Cut or copy some text from your code. The last item you cut or copy into the clipboard ring is the current item for pasting.
2. Press **Ctrl**+**Shift**+**V** to paste the clipboard ring's current item into the current document.
3. Repeatedly press **Ctrl**+**Shift**+**V** to cycle through the entries in the clipboard ring until you get to the one you want to permanently paste into the document. Each time you press **Ctrl**+**Shift**+**V**, the editor replaces the last entry you pasted from the clipboard ring, so you end up with just the last one you selected. The item you stop on then becomes the current item.
4. Move to another location or cancel the selection. You can use **Ctrl**+**Shift**+**V** to paste the current item again or to cycle the clipboard ring to a new item.

Clicking an item in the clipboard ring makes it the current item.

To paste an item from the clipboard ring into the current document:

1. Move the cursor to where you want to paste the item into the document.
2. Display the pop-up menu of the item to paste by clicking the arrow on the right of the item.
3. Choose *Paste*.

—or—

1. Make the item you want to paste the current item by clicking it.
2. Move the cursor to where you want to paste the item into the document.
3. Press **Ctrl**+**Shift**+**V**.

To paste all items into a document:

To paste all items on the clipboard ring into the current document, move the cursor to where you want to paste the items and do one of the following:

- Choose **Edit** > **Clipboard Ring** > **Paste All**.

—or—

- On the **Clipboard Ring** toolbar, click the **Paste All** button.

To remove an item from the clipboard ring:

1. Display the pop-up menu of the item to delete by clicking the arrow at the right of the item.
2. Choose **Delete**.

To remove all items from the clipboard ring:

- Choose **Edit** > **Clipboard** > **Clear Clipboard**.

—or—

- On the **Clipboard Ring** toolbar, click the **Clear Clipboard Ring** button.

To configure the clipboard ring:

1. Choose **Tools** > **Options** or press **Alt+,**.
2. Click the **Windows** button to show the **Clipboard Ring Options** group.
3. Select **Preserve Contents Between Runs** to save the content of the clipboard ring between runs, or deselect it to start with an empty clipboard ring.
4. Change **Maximum Items Held In Ring** to configure the maximum number of items stored on the clipboard ring.

Regular expressions

The editor can search and replace text using *regular expressions*. A regular expression is a string that uses special characters to describe and reference patterns of text. The regular expression system used by the editor is modeled on Perl's regexp language. For more information on regular expressions, see *Mastering Regular Expressions*, Jeffrey E F Freidl, ISBN 0596002890.

Summary of special characters

The following table summarizes the special characters the CrossStudio editor supports

Pattern	Description
\d	Match a numeric character.
\D	Match a non-numeric character.
\s	Match a whitespace character.
\S	Match a non-whitespace character.
\w	Match a word character.
\W	Match a non-word character.
[*c*]	Match set of characters; e.g., [ch] matches characters c or h. A range can be specified using the '-' character; e.g., '[0-27-9]' matches if the character is 0, 1, 2, 7 8, or 9. A range can be negated using the '^' character; e.g., '[^a-z]' matches if the character is anything other than a lowercase alphabetic character.
c	Match the literal character *c*. For example, you would use '*' to match the character '*'.
\a	Match ASCII bell character (ASCII code 7).
\f	Match ASCII form feed character (ASCII code 12).
\n	Match ASCII line feed character (ASCII code 10).
\r	Match ASCII carriage return character (ASCII code 13).
\t	Match ASCII horizontal tab character (ASCII code 9).
\v	Match ASCII vertical tab character.
\x*hhhh*	Match Unicode character specified by hexadecimal number *hhhh*.
.	Match any character.

*	Match zero or more occurrences of the preceding expression.
+	Match one or more occurrences of the preceding expression.
?	Match zero or one occurrences of the preceding expression.
{*n*}	Match *n* occurrences of the preceding expression.
{*n*,}	Match at least *n* occurrences of the preceding expression.
{,*m*}	Match at most *m* occurrences of the preceding expression.
{*n*,*m*}	Match at least *n* and at most *m* occurrences of the preceding expression.
^	Beginning of line.
$	End of line.
\b	Word boundary.
\B	Non-word boundary.
(*e*)	Capture expression *e*.
n	Back-reference to *n*th captured text.

Examples

The following regular expressions can be used with the editor's search-and-replace operations. To use the regular expression mode, the **Use regular expression** checkbox must be set in the search-and-replace dialog. Once enabled, regular expressions can be used in the **Find what** search string. The **Replace With** strings can use the "*n*" back-reference string to reference any captured strings.

"Find what"	"Replace With"	Description
u\w.d		Search for any-length string containing one or more word characters beginning with the character 'u' and ending in the character 'd'.
^.*;$		Search for any lines ending in a semicolon.

(typedef.+\s+)(\S+);	\1TEST_\2;	Find C type definition and insert the string 'TEST' onto the beginning of the type name.

Debugging windows

This section describes the windows you can use to debug your application.

Locals window

The **Locals** window displays a list of all variables that are in scope of the selected stack frame in the **Call Stack**.

The **Locals** window has a toolbar and a main data display.

Button	Description
x_2	Displays the selected item in binary.
x_8	Displays the selected item in octal.
x_{10}	Displays the selected item in decimal.
x_{16}	Displays the selected item in hexadecimal.
±x	Displays the selected item as a signed decimal.
'x'	Displays the selected item as a character or Unicode character.
	Sets the range displayed in the active **Memory** window to where the selected item is stored.
	Sorts the variables alphabetically by name.
	Sorts the variables numerically by address or register number (default).

Using the Locals window

The **Locals** window shows the local variables of the active function when the debugger is stopped. The contents of the **Locals** window changes when you use the **Debug Location** toolbar items or select a new frame in the **Call Stack** window. When the program stops at a breakpoint, or is stepped, the **Locals** window updates to show the active stack frame. Items that have changed since they were previously displayed are highlighted in red.

To activate the Locals window:

- Choose **Debug** > **Locals** or press **Ctrl+Alt+L**.

When you select a variable in the main part of the display, the display-format button highlighted on the **Locals** window toolbar changes to show the selected item's display format.

To change the display format of a local variable:

- Right-click the item to change.

- From the shortcut menu, choose the desired display format.

—or—

- Click the item to change.
- On the **Locals** window toolbar, select the desired display format.

To modify the value of a local variable:

- Click the value of the local variable to modify.
- Enter the new value for the local variable. Prefix hexadecimal numbers with **0x**, binary numbers with **0b**, and octal numbers with **0**.

—or—

- Right-click the value of the local variable to modify.
- From the shortcut menu, select one of the commands to modify the local variable's value.

Globals window

The **Globals** window displays a list of all variables that are global to the program. The operations available on the entries in this window are the same as the **Watch** window, except you cannot add or delete variables from the **Globals** window.

Globals window user interface

The **Globals** window consists of a toolbar and main data display.

Globals toolbar

Button	Description
	Displays the selected item in binary.
	Displays the selected item in octal.
	Displays the selected item in decimal.
	Displays the selected item in hexadecimal.
	Displays the selected item as a signed decimal.
	Displays the selected item as a character or Unicode character.
	Sets the range displayed in the active memory window to where the selected item is stored.
	Sorts the variables alphabetically by name.
	Sorts the variables numerically by address or register number (default).

Using the Globals window

The **Globals** window shows the global variables of the application when the debugger is stopped. When the program stops at a breakpoint, or is stepped, the **Globals** window updates to show the active stack frame and new variable values. Items that have changed since they were previously displayed are highlighted in red.

To activate the Globals window:

- Choose **Debug** > **Other Windows** > **Globals** or press **Ctrl+Alt+G**.

Changing the display format

When you select a variable in the main part of the display, the display-format button highlighted on the **Globals** window toolbar changes to show the item's display format.

To change the display format of a global variable:

- Right-click the item to change.
- From the shortcut menu, choose the desired display format.

—or—

- Click the item to change.
- On the **Globals** window toolbar, select the desired display format.

To modify the value of a global variable:

- Click the value of the global variable to modify.
- Enter the new value for the global variable. Prefix hexadecimal numbers with **0x**, binary numbers with **0b**, and octal numbers with **0**.

Watch window

The **Watch** window provides a means to evaluate expressions and to display the results of those expressions. Typically, expressions are just the name of a variable to be displayed, but they can be considerably more complex; see **Debug expressions**. *Note:* expressions are always evaluated when your program stops, so the expression you are watching is the one that is in scope of the stopped program position.

The **Watch** window is divided into a toolbar and the main data display.

Button	Description
	Displays the selected item in binary.
	Displays the selected item in octal.
	Displays the selected item in decimal.
	Displays the selected item in hexadecimal.
	Displays the selected item as a signed decimal.
	Displays the selected item as a character or Unicode character.
	Sets the range displayed in the active memory window to the address where the selected item is stored.
	Sorts the watch items alphabetically by name.
	Sorts the watch items numerically by address or register number (default).
	Remove the selected watch item.
	Remove all the watches.

Right-clicking a watch item shows a shortcut menu with commands that are not available from the toolbar.

Button	Description
	View pointer or array as a null-terminated string.
	View pointer or array as an array.

x	View pointer value.
=0	Set watch value to zero.
=1	Set watch value to one.
x↑	Increment watched variable by one.
x↓	Decrement watched variable by one.
-x	Negated watched variable.
x̄	Invert watched variable.
	View the properties of the watch value.

You can view details of the watch item using the **Properties** window.

Filename

The filename context of the watch item.

Line number

The line number context of the watch item.

(Name)

The name of the watch item.

Address

The address or register of the watch item.

Expression

The debug expression of the watch item.

Previous Value

The previous watch value.

Size In Bytes

The size of the watch item in bytes.

Type

The type of the watch item.

Value

The value of the watch item.

Using the Watch window

Each expression appears as a row in the display. Each row contains the expression and its value. If the value of an expression is structured (for example, an array), you can open the structure to see its contents.

The display updates each time the debugger locates to source code. So it will update each time your program stops on a breakpoint, or single steps, and whenever you traverse the call stack. Items that have changed since they were previously displayed are highlighted in red.

To activate the Watch window:

- Choose **Debug** > **Other Windows** > **Watch** > **Watch 1** or press **Ctrl+T, W, 1**.

You can show other **Watch** windows similarly.

You can add a new expression to be watched by clicking and typing into the last entry in the **Watch** window. You can change an expression by clicking its entry and editing its contents.

When you select a variable in the main part of the display, the display format button highlighted on the **Watch** window toolbar changes to show the item's display format.

To change the display format of an expression:

- Right-click the item to change.
- From the shortcut menu, choose the desired display format.

—or—

- Click the item to change.
- On the **Watch** window toolbar, select the desired display format.

The selected display format will then be used for all subsequent displays and will be preserved after the debug session stops.

For C programs, the interpretation of pointer types can be changed by right-clicking and selecting from the shortcut menu. A pointer can be interpreted as:

- a null-terminated ASCII string
- an array
- an integer
- dereferenced

To modify the value of an expression:

- Click the value of the local variable to modify.
- Enter the new value of the local variable. Prefix hexadecimal numbers with **0x**, binary numbers with **0b**, and octal numbers with **0**.

—or—

- Right-click the value of the local variable to modify.
- From the shortcut menu, choose one of the commands to modify the variable's value.

Register window

The **Register** windows show the values of both CPU registers and the processor's special function or peripheral registers. Because microcontrollers are becoming very highly integrated, it's not unusual for them to have hundreds of special function registers or peripheral registers, so CrossStudio provides four register windows. You can configure each register window to display one or more register groups for the processor being debugged.

A **Register** window has a toolbar and a main data display.

Button	Description
	Displays the CPU, special function register, and peripheral register groups.
	Displays the selected item in binary.
	Displays the selected item in octal.
	Displays the selected item in decimal.
	Displays the selected item in hexadecimal.
	Displays the selected item as a signed decimal.
	Displays the selected item as a character or Unicode character.
	Force reads a register, ignoring the access property of the register.
	Updates the selected register group.
	Sets the active memory window to the address and size of the selected register group.

Using the registers window

Both CPU registers and special function registers are shown in the main part of the **Registers** window. When the program stops at a breakpoint, or is stepped, the **Registers** windows update to show the current values of the registers. Items that have changed since they were previously displayed are highlighted in red.

To activate the first register window:

- Choose **Debug** > **Other Windows** > **Registers** > **Registers 1** or press **Ctrl+T, R, 1**.

Other register windows can be similarly activated.

Displaying CPU registers

The values of the CPU registers displayed in the **Registers** window depend up upon the selected context. The selected context can be:

- The register state the CPU stopped in.
- The register state when a function call occurred using the Call Stack window.
- The register state of the currently selected thread using the the **Threads** window.
- The register state you supplied with the **Debug** > **Locate** operation.

To display a group of CPU registers:

- On the **Registers** window toolbar, click the **Groups** button.
- From the pop-up menu, select the register groups to display and deselect the ones to hide.

You can deselect all CPU register groups to allow more space in the display for special function registers or peripheral registers. So, for instance, you can have one register window showing the CPU registers and other register windows showing different peripheral registers.

Displaying special function or peripheral registers

The **Registers** window shows the set of register groups defined in the memory-map file the application was built with. If there is no memory-map file associated with a project, the **Registers** window will show only the CPU registers.

To display a special function or peripheral register:

- On the **Registers** toolbar, click the **Groups** button.
- From the pop-up menu, select the register groups to display and deselect the ones to hide.

Changing display format

When you select a register in the main part of the display, the display-format button highlighted on the **Registers** window toolbar changes to show the item's display format.

To change the display format of a register:

- Right-click the item to change.
- From the shortcut menu, choose the desired display format.

—or—

- Click the item to change.
- On the **Registers** window toolbar, select the desired display format.

Modifying register values

To modify the value of a register:

- Click the value of the register to modify.
- Enter the new value for the register. Prefix hexadecimal numbers with **0x**, binary numbers with **0b**, and octal numbers with **0**.

—or—

- Right-click the value of the register to modify.
- From the shortcut menu, choose one of the commands to modify the register value.

Modifying the saved register value of a function or thread may not be supported.

Memory window

The **Memory** windows show the contents of the connected target's memory areas.

To activate the first Memory window:

- Choose **Debug** > **Other Windows** > **Memory** > **Memory 1** or press **Ctrl+T, M, 1**.

There are four memory window in total and you can display other memory windows similarly.

The memory window does not show the complete address space of the target; instead you must enter both the start address and the number of bytes to display. You can specify the start address and size using *debugger expressions*, which enables you to position the memory display at the start address of a variable or to use a value in a register. You can also specify whether you want the expressions to be evaluated each time the **Memory** window is updated, or you can re-evaluate them yourself with the press of a button. Memory windows update each time your program stops on a breakpoint after a or single step, and whenever you traverse the call stack. If any values that were previously displayed have changed, they are highlighted in red.

Memory window user interface

The **Memory** window has a toolbar and a main data display.

Button	Description
Address	Start address to display (a debugger expression).
Size	Number of bytes to display (a debugger expression).
	Select binary display.
	Select octal display.
	Select unsigned decimal display.
	Select signed decimal display.
	Select hexadecimal display.
	Select byte display, which includes an ASCII display.
	Select 2-byte display.
	Select 4-byte display.

	Evaluate the address and size expressions, and update the **Memory** window.
	Move the data display up one line.
	Move the data display down one line.
	Move the data display up by **Size** bytes.
	Move the data display down by **Size** bytes.

Left-click commands

The following operations are available by left-clicking the mouse:

Action	Description
Single Click	First click selects the line, second click selects the displayed memory value. Once the memory value is selected, it can be modified by entering a new value. Note that the input radix is the same as the display radix; i.e., **0x** is not required to specify a hex number.

Shortcut menu commands

The shortcut menu contains the following commands:

Action	Description
Auto Evaluate	Re-evaluate **Address** and **Size** each time the **Memory** window is updated.
Set Number of Columns	Set the number of columns to display, the default being 8.
Access Memory By Display Width	Access memory in terms of the display width.
Export To Binary Editor	Create a binary editor with the current **Memory** window contents.
Save As	Save the current **Memory** window contents to a file. Supported file formats are **Binary File**, **Motorola S-Record File**, **Intel Hex File**, **TI Hex File**, and **Hex File**.
Load From	Load the current **Memory** window from a file. Supported file formats are **Binary File**, **Motorola S-Record File**, **Intel Hex File**, **TI Hex File**, and **Hex File**.

Using the memory window

Display formats

You can set the **Memory** window to display 8-bit, 16-bit, and 32-bit values that are formatted as hexadecimal, decimal, unsigned decimal, octal, or binary. You can also specify how many columns to display.

You can change a value in the **Memory** window by clicking the value to change and editing it as a text field. Note that, when you modify memory values, you need to prefix hexadecimal numbers with **0x**, binary numbers with **0b**, and octal numbers with **0**.

Saving memory contents

You can save the displayed contents of the **Memory** window to a file in various formats. Alternatively, you can export the contents to a binary editor to work on them.

You can save the displayed memory values as a binary file, Motorola S-record file, Intel hex file, or a Texas Instruments TXT file.

To save the current state of memory to a file:

- Select the start address and number of bytes to save by editing the **Start Address** and **Size** fields in the **Memory** window toolbar.
- Right-click the main memory display.
- From the shortcut menu, select **Save As**, then choose the format from the submenu.

To export the current state of memory to a binary editor:

- Select the start address and number of bytes to save by editing the **Start Address** and **Size** fields in the **Memory** window toolbar.
- Right-click the main memory display.
- Choose **Export to Binary Editor** from the shortcut menu.

Note that subsequent modifications in the binary editor will not modify memory in the target.

Breakpoints window

The **Breakpoints** window manages the list of currently set breakpoints on the solution. Using the **Breakpoints** window, you can:

- Enable, disable, and delete existing breakpoints.
- Add new breakpoints.
- Show the status of existing breakpoints.

Breakpoints are stored in the session file, so they will be remembered each time you work on a particular project. When running in the debugger, you can set breakpoints on assembly code addresses. These low-level breakpoints appear in the **Breakpoints** window for the duration of the debug run but are not saved when you stop debugging.

When a breakpoint is reached, the matching breakpoint is highlighted in the **Breakpoints** window.

Breakpoints window layout

The **Breakpoints** window has a toolbar and a main breakpoint display.

Button	Description
	Creates a new breakpoint using the **New Breakpoint** dialog.
	Toggles the selected breakpoint between enabled and disabled states.
	Removes the selected breakpoint.
	Moves the cursor to the statement at which the selected breakpoint is set.
	Deletes all breakpoints.
	Disables all breakpoints.
	Enables all breakpoints.
	Creates a new breakpoint group and makes it active.

The main part of the **Breakpoints** window shows what breakpoints have been set and the state they are in. You can organize breakpoints into folders, called *breakpoint groups*.

CrossStudio displays these icons to the left of each breakpoint:

Icon	Description
	**Enabled breakpoint** An enabled breakpoint will stop your program running when the breakpoint condition is met.
	**Disabled breakpoint** A disabled breakpoint will not stop the program when execution passes through it.
	**Invalid breakpoint** An invalid breakpoint is one where the breakpoint cannot be set; for example, no executable code is associated with the source code line where the breakpoint is set or the processor does not have enough hardware breakpoints.

Showing the Breakpoints window

To activate the Breakpoints window:

- Choose **Breakpoints** > **Breakpoints** or press **Ctrl+Alt+B**.

Managing single breakpoints

You can manage breakpoints in the **Breakpoint** window.

To delete a breakpoint:

- In the **Breakpoints** window, click the breakpoint to delete.
- From the **Breakpoints** window toolbar, click the **Delete** Breakpoint} button.

To edit the properties of a breakpoint:

- In the **Breakpoints** window, right-click the breakpoint to edit.
- Choose **Edit Breakpoint** from the shortcut menu.
- Edit the breakpoint in the **New Breakpoint** dialog.

- To toggle the enabled state of a breakpoint:

- In the **Breakpoints** window, right-click the breakpoint to enable or disable.
- Choose **Enable/Disable Breakpoint** from the shortcut menu.

—or—

- In the **Breakpoints** window, click the breakpoint to enable or disable.

- Press **Ctrl+F9**.

Breakpoint groups

Breakpoints are divided into *breakpoint groups*. You can use breakpoint groups to specify sets of breakpoints that are applicable to a particular project in the solution or for a particular debug scenario. Initially, there is a single breakpoint group, named *Default*, to which all new breakpoints are added.

To create a new breakpoint group:

- From the **Breakpoints** window toolbar, click the **New Breakpoint Group** button.

—or—

- From the **Debug** menu, choose **Breakpoints** then **New Breakpoint Group**.

—or—

- Right-click anywhere in the **Breakpoints** window.
- Choose **New Breakpoint Group** from the shortcut menu.

In the **New Breakpoint Group** dialog, enter the name of the breakpoint group.

When you create a breakpoint, it is added to the active breakpoint group.

To make a group the active group:

- In the **Breakpoints** window, right-click the breakpoint group to make active.
- Choose **Set as Active Group** from the shortcut menu.

To delete a breakpoint group:

- In the **Breakpoints** window, right-click the breakpoint group to delete.
- Choose **Delete Breakpoint Group** from the shortcut menu.

You can enable all breakpoints within a group at once.

To enable all breakpoints in a group:

- In the **Breakpoints** window, right-click the breakpoint group to enable.
- Choose **Enable Breakpoint Group** from the shortcut menu.

You can disable all breakpoints within a group at once.

To disable all breakpoints in a group:

- In the **Breakpoints** window, right-click the breakpoint group to disable.

- Choose **Disable Breakpoint Group** from the shortcut menu.

Managing all breakpoints

You can delete, enable, or disable all breakpoints at once.

To delete all breakpoints:

- Choose **Breakpoints** > **Clear All Breakpoints** or press **Ctrl+Shift+F9**.

—or—

- On the **Breakpoints** window toolbar, click the **Delete All Breakpoints** button.

To enable all breakpoints:

- Choose **Breakpoints** > **Enable All Breakpoints**.

—or—

- On the **Breakpoints** window toolbar, click the **Enable All Breakpoints** button.

To disable all breakpoints:

- Choose **Breakpoints** > **Disable All Breakpoints**.

—or—

- On the **Breakpoints** window toolbar, click the **Disable All Breakpoints** button.

Call Stack window

The **Call Stack** window displays the list of function calls (stack frames) that were active when program execution halted. When execution halts, CrossStudio populates the call-stack window from the active (currently executing) task. For simple, single-threaded applications not using the CrossWorks tasking library, there is only a single task; but for multi-tasking programs that use the CrossWorks Tasking Library, there may be any number of tasks. CrossStudio updates the **Call Stack** window when you change the active task in the **Threads** window.

The **Call Stack** window has a toolbar and a main call-stack display.

Button	Description
	Moves the cursor to where the call was made to the selected frame.
	Sets the debugger context to the selected stack frame.
	Moves the debugger context down one stack to the called function.
	Moves the debugger context up one stack to the calling function.
	Selects the fields to display for each entry in the call stack.
	Sets the debugger context to the most recent stack frame and moves the cursor to the currently executing statement.

The main part of the **Call Stack** window displays each unfinished function call (active stack frame) at the point when program execution halted. The most recent stack frame is displayed at the bottom of the list and the oldest is displayed at the top of the list.

CrossStudio displays these icons to the left of each function name:

Icon	Description
	Indicates the stack frame of the current task.
	Indicates the stack frame selected for the debugger context.
	Indicates that a breakpoint is active and when the function returns to its caller.

These icons can be overlaid to show, for instance, the debugger context and a breakpoint on the same stack frame.

Showing the call-stack window

To activate the Call Stack window:

- Choose **Debug** > **Call Stack** or press **Ctrl+Alt+S**.

Configuring the call-stack window

Each entry in the **Call Stack** window displays the function name and, additionally, parameter names, types, and values. You can configure the **Call Stack** window to show varying amounts of information for each stack frame. By default, CrossStudio displays all information.

To show or hide a field:

1. On the **Call Stack** toolbar, click the **Options** button on the far right.
2. Select the fields to show, and deselect the ones that should be hidden.

Changing the debugger context

You can select the stack frame for the debugger context from the **Call Stack** window.

To move the debugger context to a specific stack frame:

- In the **Call Stack** window, double-click the stack frame to move to.

—or—

- In the **Call Stack** window, select the stack frame to move to.
- On the **Call Stack** window's toolbar, click the **Switch To Frame** button.

—or—

- In the **Call Stack** window, right-click the stack frame to move to.
- Choose **Switch To Frame** from the shortcut menu.

The debugger moves the cursor to the statement where the call was made. If there is no debug information for the statement at the call location, CrossStudio opens a disassembly window at the instruction.

To move the debugger context up one stack frame:

- On the **Call Stack** window's toolbar, click the **Up One Stack Frame** button.

—or—

- On the **Debug Location** toolbar, click the **Up One Stack Frame** button.

—or—

- Press **Alt+-**.

The debugger moves the cursor to the statement where the call was made. If there is no debug information for the statement at the call location, CrossStudio opens a disassembly window at the instruction.

To move the debugger context down one stack frame:

- On the **Call Stack** window's toolbar, click the **Down One Stack Frame** button.

—or—

- On the **Debug Location** toolbar, click the **Down One Stack Frame** button.

—or—

- Press **Alt++**.

The debugger moves the cursor to the statement where the call was made. If there is no debug information for the statement at the call location, CrossStudio opens a disassembly window at the instruction.

Setting a breakpoint on a return to a function

To set a breakpoint on return to a function:

- In the **Call Stack** window, click the stack frame on the function to stop at on return.
- On the **Build** toolbar, click the **Toggle Breakpoint** button.

—or—

- In the **Call Stack** window, click the stack frame on the function to stop at on return.
- Press **F9**.

—or—

- In the **Call Stack** window, right-click the function to stop at on return.
- Choose **Toggle Breakpoint** from the shortcut menu.

Threads window

The **Threads** window displays the set of executing contexts on the target processor structured as a set of queues.

To activate the Threads window:

- Choose **Debug** > **Threads** or press **Ctrl**+**Alt**+**H**.

The window is populated using the threads script, which is a JavaScript program store in a file whose file-type property is "Threads Script" (or is called `threads.js`) and is in the project that is being debugged.

When debugging starts, the threads script is loaded and the {function init()} is called to determine which columns are displayed in the **Threads** window.

When the application stops on a breakpoint, the {function update()} is called to create entries in the **Threads** window corresponding to the columns that have been created together with the saved execution context (register state) of the thread. By double-clicking one of the entries, the debugger displays its saved execution context—to put the debugger back into the default execution context, use **Show Next Statement**.

Writing the threads script

The threads script controls the **Threads** window with the **Threads** object.

The methods {Threads.setColumns} and {Threads.setSortByNumber} can be called from the {function init()}.

```
function init()
{
  Threads.setColumns("Name", "Priority", "State", "Time");
  Threads.setSortByNumber("Time");
}
```

The above example creates the named columns **Name**>, **Priority**, **State**, and **Time** in the **Threads** window, with the **Time** column sorted numerically rather than alphabetically.

If you don't supply the {function init()} in the threads script, the **Threads** window will create the default columns **Name**, **Priority**, and **State**.

The methods {Threads.clear()}, {Threads.newqueue()}, and {Threads.add()} can be called from the {function update()}.

The **Threads.clear()** method clears the **Threads** window.

The {Threads.newqueue()} function takes a string argument and creates a new, top-level entry in the **Threads** window. Subsequent entries added to this window will go under this entry. If you don't call this, new entries will all be at the top level of the **Threads** window.

The {Threads.add()} function takes a variable number of string arguments, which should correspond to the number of columns displayed by the **Threads** window. The last argument to the {Threads.add()} function should be an array (possibly empty) containing the registers of the thread or, alternatively, a handle that can be supplied a call to the threads script {function getregs(handle)}, which will return an array when the thread is selected in the **Threads** window. The array containing the registers should have elements in the same order in which they are displayed in the CPU **Registers** display—typically this will be in register-number order, e.g., **r0**, **r1**, and so on.

```
function update()
{
  Threads.clear();
  Threads.newqueue("My Tasks");
  Threads.add("Task1", "0", "Executing", "1000", [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
  Threads.add("Task2", "1", "Waiting", "2000", [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16])
}
```

The above example will create a fixed output on the **Threads** window and is here to demonstrate how to call the methods.

To get real thread state, you need to access the debugger from the threads script. To do this, you can use the JavaScript method {Debug.evaluate("expression")}, which will evaluate the string argument as a debug expression and return the result. The returned result will be an object if you evaluate an expression that denotes a structure or an array. If the expression denotes a structure, each field can be accessed by using its field name.

So, if you have structs in the application as follows…

```
struct task {
  char *name;
  unsigned char priority;
  char *state;
  unsigned time;
  struct task *next;
  unsigned registers[17];
  unsigned thread_local_storage[4];
};
struct task task2 = { "Task2", 1, "Waiting", 2000, 0, { 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14
struct task task1 = { "Task1", 0, "Executing", 1000, &task2, { 0,1,2,3,4,5,6,7,8,9,10,11,1
```

…you can {update()} the **Threads** window using the following:

```
task1 = Debug.evaluate("task1");
Threads.add(task1.name, task1.priority, task1.state, task1.time, task1.registers);
```

You can use pointers and C-style cast to enable linked-list traversal.

```
var next = Debug.evaluate("&task1");
```

```
while (next)
  {
    var xt = Debug.evaluate("*(struct task*)"+next);
    Threads.add(xt.name, xt.priority, xt.state, xt.time, xt.registers);
    next=xt.next;
  }
```

Note that, if the threads script goes into an endless loop, the debugger—and consequently CrossStudio—will become unresponsive and you will need to kill CrossStudio using a task manager. Therefore, the above loop is better coded as follows:

```
var next = Debug.evaluate("&;task1");
var count=0;
while (next && count > 10)
  {
    var xt = Debug.evaluate("*(struct task*)"+next);
    Threads.add(xt.name, xt.priority, xt.state, xt.time, xt.registers);
    next=xt.next;
    count++;
  }
```

You can speed up the **Threads** window update by not supplying the registers of the thread to the {Threads.add()} function. To do this, you should supply a handle/pointer to the thread as the last argument to the {Threads.add()} function. For example:

```
var next = Debug.evaluate("&task1");
var count=0;
while (next && count > 10)
  {
    var xt = Debug.evaluate("*(struct task*)"+next);
    Threads.add(xt.name, xt.priority, xt.state, xt.time, next);
    next=xt.next;
    count++;
  }
```

When the thread is selected, the **Threads** window will call {getregs(x)} in the threads script. That function should return the array of registers, for example:

```
function getregs(x)
{
  return Debug.evaluate("((struct task*)"+x+")->registers");
}
```

If you use thread local storage, implementing the {gettls(x)} function enables you to return an expression for the debugger to evaluate when the base address of the thread local storage is accessed, for example:

```
function gettls(x)
{
  return "((struct task*)"+x+")->thread_local_storage";
}
```

Execution Counts window

The **Execution Counts** window shows a list of source locations and the number of times those source locations have been executed. This window is only available for targets that support the collection of jump trace information.

To activate the Execution Counts window:

- Choose **Debug** > **Other Windows** > **Execution Counts**.

The count value displayed is the number of times the first instruction of the source code location has been executed. The source locations displayed are target dependent: they could represent each statement of the program or each jump target of the program. If however the debugger is in intermixed or disassembly mode then the count values will be displayed on a per instruction basis.

The execution counts window is updated each time your program stops and the window is visible so if you have this window displayed then single stepping may be slower than usual.

Trace window

The trace window displays historical information on the instructions executed by the target.

To activate the Trace window:

- Choose **Debug** > **Other Windows** > **Execution Trace**.

The type and number of the trace entries depends upon the target that is connected when gathering trace information. Some targets may trace all instructions, others may trace jump instructions, and some may trace modifications to variables. You'll find the trace capabilities of your target on the shortcut menu.

Each entry in the trace window has a unique number, and the lower the number the earlier the trace. You can click on the header to show earliest to latest or the latest to earliest trace entries. If a trace entry can have source code located to it then double-clicking the trace entry will show the appropriate source display.

Some targets may provide timing information which will be displayed in the ticks column.

The trace window is updated each time the debugger stops when it is visible so single stepping is likely to be slower if you have this window displayed.

Debug file search editor

When a program is built with debugging enabled, the debugging information contains the paths and filenames of all the source files for the program in order to allow the debugger to find them. If a program or library linked into the program is on a different machine than the one on which it was compiled, or if the source files were moved after the program was compiled, the debugger will not be able to find the source files.

In this situation, the simplest way to help CrossStudio find the source files is to add the directory containing the source files to one of its source-file search paths. Alternatively, if CrossStudio cannot find a source file, it will prompt you for its location and will record its new location in the source-file map.

Debug source-file search paths

Debug's source-file search paths can be used to help the debugger locate source files that are no longer located where they were at compile time. When a source file cannot be found, the search-path directories will be checked, in turn, to see if they contain the source file. CrossStudio maintains two debug source-file search paths:

- *Project-session search path*: This path is for the current project session and does not apply to all projects.
- *The global search path*: This system-wide path applies to all projects.

The project-session search path is checked before the global search path.

To edit the debug search paths:

- Choose **Debug** > **Options** > **Search Paths**.

Debug source file map

If a source file cannot be found while debugging and the debugger has to prompt the user for its location, the results are stored in the debug source file map. The debug source file map simply correlates, or *maps*, the original pathnames to the new locations. When a file cannot be found at its original location or in the debug search paths, the debug source file map is checked to see if a new location has been recorded for the file or if the user has specified that the file does not exist. Each project session maintains its own source file map, the map is not shared by all projects.

To view the debug source file map:

- Choose **Debug** > **Options** > **Search Paths**.

To remove individual entries from the debug source file map:

- Choose **Debug** > **Options** > **Search Paths**.

- Right-click the mapping to delete.
- Choose **Delete Mapping** from the shortcut menu.

To remove all entries from the debug source file map:

- Choose **Debug** > **Options** > **Search Paths**.
- Right-click any mapping.
- Choose **Delete All Mappings** from the shortcut menu.

Breakpoint expressions

The debugger can set breakpoints by evaluating simple C-like expressions. Note that the exact capabilities offered by the hardware to assist in data breakpointing will vary from target to target; please refer to the particular target interface you are using and the capabilities of your target silicon for exact details. The simplest expression supported is a symbol name. If the symbol name is a function, a breakpoint occurs when the first instruction of the symbol is about to be executed. If the symbol name is a variable, a breakpoint occurs when the symbol has been accessed; this is termed a *data breakpoint*. For example, the expression `x` will breakpoint when *x* is accessed. You can use a debug expression (see **Debug expressions**) as a breakpoint expression. For example, `x[4]` will breakpoint when element 4 of array *x* is accessed, and `@sp` will breakpoint when the *sp* register is accessed.

Data breakpoints can be specified, using the == operator, to occur when a symbol is accessed with a specific value. The expression `x == 4` will breakpoint when *x* is accessed and its value is 4. The operators <, >=, >;, >=, ==, and != can be used similarly. For example, `@sp <= 0x1000` will breakpoint when register **sp** is accessed and its value is less than or equal to 0x1000.

You can use the operator '&' to mask the value you wish to breakpoint on. For example, `(x & 1) == 1` will breakpoint when *x* is accessed and has an odd value.

You can use the operator '&&' to combine comparisons. For example…

```
(x >= 2) && (x <= 14)
```

…will breakpoint when *x* is accessed and its value is between 2 and 14.

You can specify an arbitrary memory range using an array cast expression. For example, `(char[256])(0x1000)` will breakpoint when the memory region 0x1000–0x10FF is accessed.

You can specify an inverse memory range using the **!** operator. For example `!(char[256])(0x1000)` will breakpoint when memory outside the range 0x1000–0x10FF is accessed.

Debug expressions

The debugger can evaluate simple expressions that can be displayed in the **Watch** window or as a tool-tip in the code editor.

The simplest expression is an identifier the debugger tries to interpret in the following order:

- an identifier that exists in the scope of the current context.
- the name of a global identifier in the program of the current context.

Numbers can be used in expressions. Hexadecimal numbers must be prefixed with `0x`.

Registers can be referenced by prefixing the register name with `@`.

The standard C and C++ operators `!`, ~, `*`, `/`, %, +, `-`, >>, <<, <, <=, >, >=, ==, `|`, &, ^, &&, and `||` are supported on numeric types.

The standard assignment operators =, +=, `-=`, `*=`, `/=`, %=, >>, >>=, <<=, &=, `|=`, ^= are supported on numeric types.

The array subscript operator '[]' is supported on array and pointer types.

The structure access operator '.' is supported on structured types (this also works on pointers to structures), and -> works similarly.

The dereference operator (prefix '*') is supported on pointers, the address-of (prefix '&') and **sizeof** operators are supported.

The `addressof(` *filename* `,` *linenumber* `)` operator will return the address of the specified source code line number.

Function calling with parameters and return results.

Casting to basic pointer types is supported. For example, (unsigned char *)0x300 can be used to display the memory at a given location.

Casting to basic array types is supported. For example, (unsigned char[256])0x100 can be used to reference a memory region.

Operators have the precedence and associativity one expects of a C-like programming language.

Utility windows

This section describes the utility windows.

Output window

The **Output** window contains logs and transcripts from various systems within CrossStudio. Most notably, it contains the *Transcript* and *Source Navigator Log*.

Transcript

The Transcript contains the results of the last build or target operation. It is cleared on each build. Errors detected by CrossStudio are shown in red and warnings are shown in yellow. Double-clicking an error or warning in the build log will open the offending file at the error position. The commands used for the build can be echoed to the build log by setting the **Echo Build Command Lines** environment option. The transcript also shows a trace of the high-level loading and debug operations carried out on the target. For downloading, uploading, and verification operations, it displays the time it took to carry out each operation. The log is cleared for each new download or debug session.

Navigator Log

The Source Navigator Log displays a list of files the Source Navigator has parsed and the time it took to parse each file.

To activate the Output window:

- Choose **View** > **Output** or press **Ctrl**+**Alt**+**O**.

To show a specific log:

- On the **Output** window toolbar, click the log combo box.
- From the list, click the log to display.

—or—

- Choose **View** > **Logs** and select the log to display.

Properties window

The **Properties** window displays properties of the current CrossStudio object. Using the **Properties** window, you can set the build properties of your project, modify the editor defaults, and change target settings.

To activate the Properties window:

- Choose **View** > **Properties Window** or press **Ctrl**+**Alt**+**Enter**.

The **Properties** window is organized as a set of key–value pairs. As you select one of the keys, help text explains the purpose of the property. Because properties are numerous and can be specific to a particular product build, consider this help to be the definitive help on the property.

You can divide the properties display into categories or, alternatively, display it as a flat list that is sorted alphabetically.

A combo-box enables you to change the properties and explains which properties you are looking at.

Some properties have actions associated with them—you can find these by right-clicking the property key. Most properties that represent filenames can be opened this way.

When the **Properties** window is displaying project properties, you'll find some properties displayed in bold. This means the property value hasn't been inherited. If you wish to inherit rather than define such a property, right-click the property and select **Inherit** from the shortcut menu.

Targets window

The **Targets** window (and its associated menu) displays the set of target interfaces you can connect to in order to download and debug your programs. Using the **Targets** window in conjunction with the **Properties** window enables you to define new targets based on the specific target types supported by the particular CrossStudio release.

To activate the Targets window:

- Choose **View** > **Targets** or press **Ctrl**+**Alt**+**T**.

You can connect, disconnect, and reconnect to a target system. You can also use the **Targets** window to reset and load programs.

Targets window layout

Button	Description
	Connects the target interface selected in the **Targets** window.
	Disconnects the connected target interface.
	Reconnects the connected target interface.
	Resets the connected target interface.
	Displays the properties of the selected target interface.

Managing connections to target devices

To connect a target:

- In the **Targets** window, double-click the target to connect.

—or—

- Choose **Target** > **Connect** and click the target to connect.

—or—

1. In the **Targets** window, click the target to connect.

2. On the **Targets** window toolbar, click the **Connect** button

—or—

1. In the **Targets** window, right-click the target to connect.
2. Choose **Connect**.

To disconnect a target:

- Choose **Target** > **Disconnect** or press **Ctrl+T, D**.

—or—

- On the **Targets** window toolbar, click the **Disconnect** button.

—or—

1. Right-click the connected target in the **Targets** window.
2. Choose **Disconnect** from the shortcut menu.

Alternatively, connecting a different target will disconnect the current target connection.

You can disconnect and reconnect a target in a single operation using the reconnect feature. This may be useful if the target board has been power cycled, or reset manually, because it forces CrossStudio to resynchronize with the target.

To reconnect a target:

- Choose **Target** > **Reconnect** or press **Ctrl+T, E**.

—or—

- On the **Targets** window toolbar, click the **Reconnect** button.

—or—

1. In the **Targets** window, right-click the target to reconnect.
2. Choose **Reconnect** from the shortcut menu.

Automatic target connection

You can configure CrossStudio to automatically connect to the last-used target interface when loading a solution.

To enable or disable automatic target connection:

1. Choose **View** > **Targets** or press **Ctrl+Alt+T**.

2. Click the disclosure arrow on the **Targets** window toolbar.
3. Select or deselect **Automatically Connect When Starting Debug**.

Resetting the target

Reset of the target is typically handled by the system when you start debugging. However, you can manually reset the target from the **Targets** window.

To reset the connected target:

- Choose **Project** > **Reset And Debug** or press **Ctrl**+**Alt**+**F5**.

—or—

- On the **Targets** window toolbar, click the **Reset** button.

Creating a new target interface

To create a new target interface:

1. From the **Targets** window shortcut menu, click **New Target Interface**. A menu will display the types of target interface that can be created.
2. Select the type of target interface to create.

Setting target interface properties

All target interfaces have a set of properties. Some properties are read-only and provide information about the target, but others are modifiable and allow the target interface to be configured. Target interface properties can be viewed and edited using CrossStudio's property system.

To view or edit target properties:

- Select a target.
- Select the **Properties** option from the target's shortcut menu.

The **Targets** window provides the facility to restore the target definitions to the default set. Restoring the default target definitions will undo any of the changes you have made to the targets and their properties, therefore it should be used with care.

To restore the default target definitions:

1. Select **Restore Default Targets** from the **Targets** window shortcut menu.

2. Click **Yes** when the systems asks whether you want to restore the default targets.

Importing and exporting target definitions

You can import and export your target-interface definitions. This may be useful if you make a change to the default set of target definitions and want to share it with another user or use it on another machine.

To export the current set of target-interface definitions:

- Choose **Export Target Definitions To XML** from the **Targets** window shortcut menu.
- Specify the location and name of the file to which you want to save the target definitions and click **Save**.

To import an existing set of target-interface definitions:

- Select **Import Target Definitions From XML** from the **Targets** window shortcut menu.
- Select the file from which you want to load the target definitions and click **Open**.

Downloading programs

Program download is handled automatically by CrossStudio when you start debugging. However, you can download arbitrary programs to a target using the **Targets** window.

To download a program to the currently selected target:

- In the **Targets** window, right-click the selected target.
- Choose **Download File**.
- From the **Download File** menu, select the type of file to download.
- In the **Open File** dialog, select the executable file to download and click **Open** to download the file.

CrossStudio supports the following file formats when downloading a program:

- Binary
- Intel Hex
- Motorola S-record
- CrossWorks native object file (AVR, MSP430, and MAXQ products)
- Texas Instruments text file

Verifying downloaded programs

You can verify a target's contents against arbitrary programs on disk using the **Targets** window.

To verify a target's contents against a program:

1. In the **Targets** window, right-click the selected target.

2. Choose **Verify File**.
3. From the **Verify File** menu, select the type of file to verify.
4. In the **Open File** dialog, select the executable file to verify and click **Open** to verify the file.

CrossStudio supports the same file types for verification as for downloading.

Erasing target memory

Usually, erasing target memory is done when CrossStudio downloads a program, but you can erase a target's memory manually.

To erase all target memory:

1. In the **Targets** window, right-click the target to erase.
2. Choose **Erase All** from the shortcut menu.

To erase part of target memory:

1. In the **Targets** window, right-click the target to erase.
2. Choose **Erase Range** from the shortcut menu.

Terminal emulator window

The **Terminal Emulator** window contains a basic serial-terminal emulator that allows you to receive and transmit data over a serial interface.

To activate the Terminal Emulator window:

- Choose **Tools** > **Terminal Emulator** > **Terminal Emulator** or press **Ctrl**+**Alt**+**M**.

To use the terminal emulator:

1. Set the required terminal emulator properties.
2. Connect the terminal emulator to the communications port by clicking the button on the toolbar or by selecting **Connect** from the shortcut menu.

Once connected, any input in the **Terminal Emulator** window will be transmitted to the communications port and any data received from the communications port will be displayed on the terminal.

Connection may be refused if the communication port is in use by another application or if the port doesn't exist.

To disconnect the terminal emulator:

1. Disconnect the communications port by clicking the **Disconnect** icon on the toolbar or by right-clicking to select **Disconnect** from the shortcut menu.

This will release the communications port so it can be used by other applications.

Supported control codes

The terminal supports a limited set of control codes:

Control code	Description
<BS>	Backspace
<LF>	Linefeed
<LF>	Linefeed {mo Why duplicated?}
<ESC>[{attr1};...;{attrn}m	Set display attributes. The attributes 2-Dim, 5-Blink, 7-Reverse, and 8-Hidden are not supported.

Script Console window

The **Script Console** window provides interactive access to the JavaScript interpreter and JavaScript classes that are built into CrossStudio. The interpreter is an implementation of the 3rd edition of the ECMAScript standard. The interpreter has an additional function property of the global object that enable files to be loaded into the interpreter.

The JavaScript method **load(** *filepath* **)** loads and executes the JavaScript contained in *filepath* returns a Boolean indicating success.

To activate the Script Console window:

- Choose **View** > **Script Console** or press **Ctrl+Alt+J**.

Debug Immediate window

The **Debug Immediate** window allows you to type in debug expressions and display the results. All results are displayed in the format specified by the **Default Display Mode** property found in the **Debugging** group in the **Environment Options** dialog.

To activate the Envronment Options dialog:

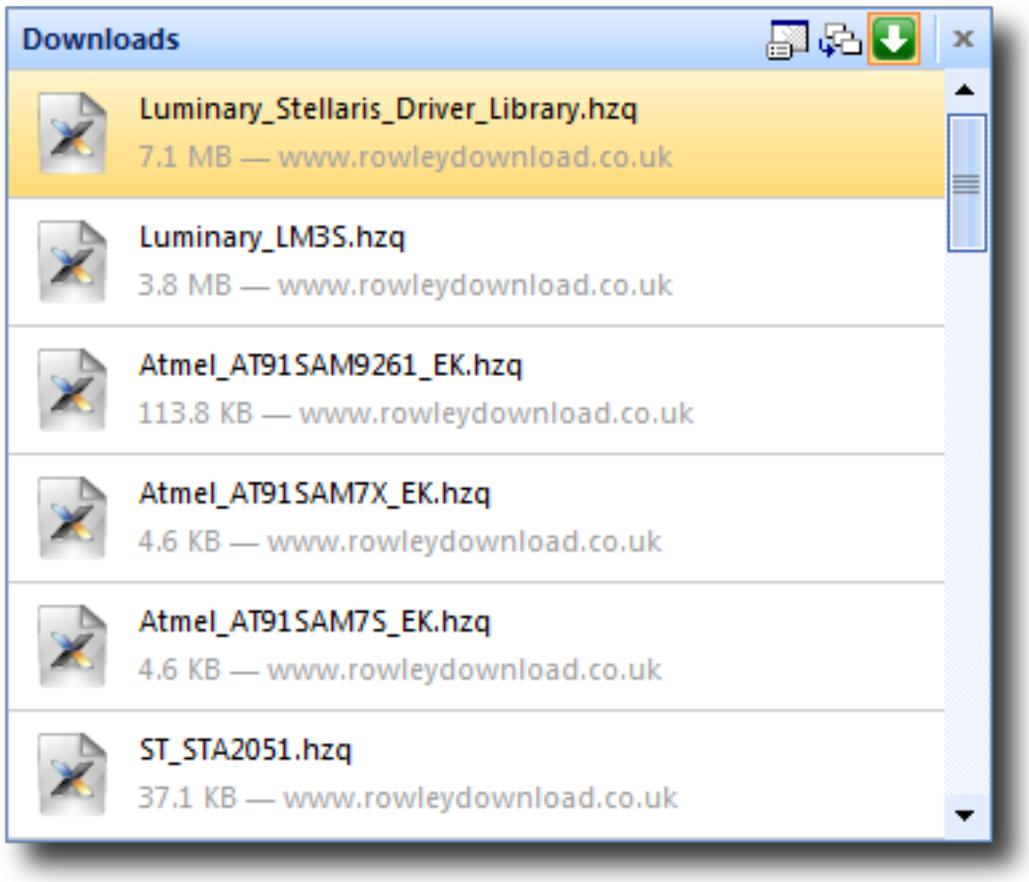
- Choose **Tools** > **Options** or press **Alt+,**.

To activate the Debug Immediate window:

- Choose **Debug** > **Other Windows** > **Debug Immediate**.

Downloads window

The **Downloads Window** displays a historical list of files downloaded over the Internet by CrossStudio.

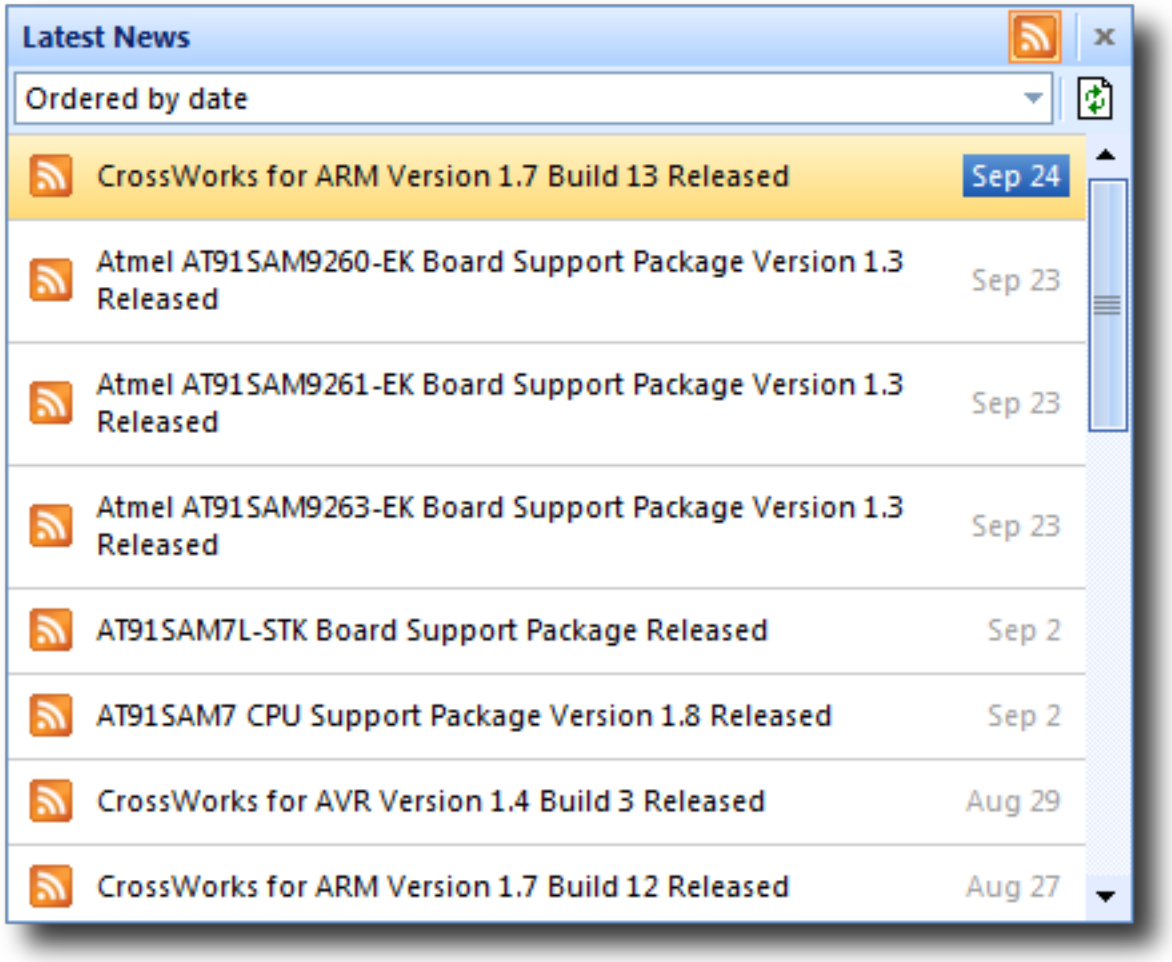


To activate the Downloads window:

- Choose **Tools** > **Downloads Window**.

Latest News window

The **Latest News** window displays a historical list of news articles from the Rowley Associates website.



To activate the Latest News window:

- Choose **Help** > **Latest News**.

Memory-map editor

memory-map files are tree-structured descriptions of the target memory map. They are used by the compiler to ensure correct placement of program sections, and by the debugger so it knows which addresses are valid on the target and which program sections to load. You can also use them files to direct the debugger to display memory-mapped peripherals. Usually, you don't need to modify memory-map files—they will be set up for the particular targets CrossStudio supports—but it is useful to view them with the memory-map editor.

To open memory-map files, choose **File** > **Open** and select the XML file that contains the memory map or, alternatively, use the **View Memory Map** option on the shortcut menu of the **Project Explorer**.

The memory-map editor provides a tree-structured view of the memory space of a target. The memory map consists of a set of different node types arranged in a hierarchy. These nodes have properties that can be modified using the **Properties** window when the node is selected. These properties and the placement of nodes within the memory map are used as input to the program-building process so the linker knows where sections should be placed. Additionally, the debugger uses the information in memory-map files to enable register and memory displays.

The memory-map editor supports the following node types:

Root

The top node of the memory map.

Memory Segment

A range of addresses that represents a region in target memory.

Program Section

Represents a program section of your application.

Register Group

Represents an area in memory that contains a group of related registers.

Register

Represents a memory-mapped register.

Bit Field

Part of a memory-mapped register.

The following statements hold regarding the creation and movement of nodes within a memory:

- Memory segments can be within the **Root** segment.
- Program sections must be within a memory segment.
- Register groups can be within the Root or within a memory segment.
- Registers can be within memory segments or register groups.
- Bitfields can be within registers.

All nodes have both mandatory and optional properties. The value of the mandatory Name property should be unique within the memory map.

Memory-segment and register-group properties

Start Address

A hexadecimal number stating where memory begins (lowest address).

Start Address Symbol

The name of a linker symbol to generate with the value of the Start Address.

Size

A hexadecimal number specifying the size in bytes of the memory segment.

Size Symbol

The name of a linker symbol to generate with the value of the Size.

Access Type

Specifies whether the memory segment is read only or read/write.

Program section properties

Start Address

An optional hexadecimal value representing the absolute load position of the section. If this isn't set, the relative placement of the program section within the memory segment will determine the load position of the section.

Size

An optional decimal value specifying the size in bytes of the program section.

Load

Specifies whether the section should be loaded by the debugger.

Alignment

An optional decimal value specifying the alignment requirements of the section.

Section To Run In

An optional name of another program section to which this program section will be copied.

Input Section Names

The optional names of the files that will be placed in this section.

Register properties

Start Address

A hexadecimal value specifying where the register is placed.

Start Address Symbol

The name of a linker symbol to generate with the value of the Start Address.

Register Type

Optional, a C type specifying how you want to display the register. This defaults to the word length of the target processor.

Endian

Optional, specifies the byte order of a multibyte register. This defaults to the byte order of the target processor.

Bitfield properties

Bit Offset

A decimal value that sets the starting bit position of the bit field. Bit 0 is the first bit position.

Bit Length

A decimal value that defines the number of bits in the field.

The memory-map editor shares many of the attributes of the text editor, and the same key-bindings—for example, cut, copy and paste—are accessible from the **Edit** menu. But in addition to the standard editor capabilities, the memory-map editor supports moving nodes up and down within a hierarchy. This enables adjusting the sequence of program sections.

Environment options dialog

The **Environment Options** dialog enables you to modify settings that apply to all uses of a CrossWorks installation.

Building Environment Options

Build Options

Property	Description
Automatically Build Before Debug `Environment/Build/Build Before Debug` – Boolean	Enables auto-building of a project before downloading if it is out of date.
Build Macros `Environment/Macros/Global Macros` – StringList	Build macros that are shared across all solutions and projects e.g. paths to library files.
Confirm Debugger Stop `Environment/Build/Confirm Debugger Stop` – Boolean	Present a warning when you start to build that requires the debugger to stop.
Echo Build Command Lines `Environment/Build/Show Command Lines` – Boolean	Selects whether build command lines are written to the build log.
Echo Raw Error/Warning Output `Environment/Build/Show Unparsed Error Output` – Boolean	Selects whether the unprocessed error and warning output from tools is displayed in the build log.
Find Error After Building `Environment/Build/Find Error After Build` – Boolean	Moves the cursor to the first diagnostic after a build completes with errors.
Keep Going On Error `Environment/Build/Keep Going On Error` – Boolean	Build doesn't stop on error.
Save Project File Before Building `Environment/Build/Save Project File On Build` – Boolean	Selects whether to save the project file prior to build.
Show Build Information `Environment/Build/Show Build Information` – Boolean	Show build information.
Show Error Window on Build Error `Environment/Build/Show Error Window on Build Error` – Boolean	Shows the Errors window if there is a build error.
Toolchain Root Directory `Environment/Build/Tool Chain Root Directory` – String	Specifies where to find the toolchain (compilers etc).

Building Options

Property	Description

Link In debugIO Default `Build/Link DebugIO Default` – Boolean	Specifies the default for debugio to be linked into an executable.
Link In printf Default `Build/Link Printf Default` – Boolean	Specifies the default for printf to be linked into an executable.
Link In scanf Default `Build/Link Scanf Default` – Boolean	Specifies the default for scanf to be linked into an executable.

Compatibility Options

Property	Description
Build command line generation `ARM/Build/Command Line Generation Version` – Enumeration	Generate build command lines for the specified CrossWorks for ARM version.
CrossWorks for ARM Installation Directory `ARM/Build/StudioDir Directory` – DirPath	The CrossWorks for ARM installation directory to be used for building - the value $(StudioDir) is set to.
Default GCC Target `ARM/Build/GCC Target Default` – Enumeration	Specifies the default GCC compiler target to use.
GCC supports section renaming `ARM/Build/GCC Can Rename Sections` – Boolean	GCC supports the -mtext=t, -mdata=d, -mbss=b, -mrodata=r section renaming options.

Window Options

Property	Description
Show Build Log On Build `Environment/Show Transcript On Build` – Boolean	Show the build log when a build starts.

Debugging Environment Options

Breakpoint Options

Property	Description
Clear Disassembly Breakpoints On Debug Stop `Environment/Debugger/Clear Disassembly Breakpoint` – Boolean	Clear Disassembly Breakpoints On Debug Stop
Initial Breakpoint Is Set `Environment/Debugger/Set Initial Breakpoint` – Enumeration	Specify when the initial breakpoint should be set
Set Initial Breakpoint At `Environment/Debugger/Initial Breakpoint` – String	An initial breakpoint to set if no other breakpoints exist

Debugging Options

Property	Description
Default debugIO implementation `ARM/Debug/Default debugIO Implementation` – Enumeration	The default debugIO implemenation.
TLS Expression `Environment/Debugger/TLS Expression` – String	Default expression the debugger evaluates to get the base of Thread Local Storage
UAL Disassemble `ARM/Debug/UAL Disassemble` – Boolean	Generate UAL disassembly mnemonics

Display Options

Property	Description
Close Disassembly On Mode Switch `Environment/Debugger/Close Disassembly On Mode Switch` – Boolean	Close Disassembly On Mode Switch
Data Tips Display a Maximum Of `Environment/Debugger/Maximum Array Elements Displayed` – IntegerRange	Selects the maximum number of array elements displayed in a datatip.
Default Display Mode `Environment/Debugger/Default Variable Display Mode` – Enumeration	Selects the format that data values are shown in.

Display Floating Point Number In `Environment/Debugger/Floating Point Format Display` – Custom	The printf format directive used to display floating point numbers.
Maximum Backtrace Calls `Environment/Debugger/Maximum Backtrace Calls` – IntegerRange	Selects the maximum number of calls when backtracing.
Prompt To Display If More Than `Environment/Debugger/Array Elements Prompt Size` – IntegerRange	The array size to display with prompt.
Show CPU Registers In Locals Window `Environment/Debugger/Locals Display Registers` – Boolean	Specify whether the locals window should display CPU registers
Show Labels In Disassembly `Environment/Debugger/Disassembly Show Labels` – Boolean	Show Labels In Disassembly
Show Source In Disassembly `Environment/Debugger/Disassembly Show Source` – Boolean	Show Source In Disassembly
Show char * as null terminated string `Environment/Debugger/Display Char Ptr As String` – Boolean	Show char * as null terminated string
Source Path `Environment/Debugger/Source Path` – StringList	Global search path to find source files.

Extended Data Tips Options

Property	**Description**
ASCII `Environment/Debugger/Extended Tooltip Display Mode/ASCII` – Boolean	Selects ASCII extended datatips.
Binary `Environment/Debugger/Extended Tooltip Display Mode/Binary` – Boolean	Selects Binary extended datatips.
Decimal `Environment/Debugger/Extended Tooltip Display Mode/Decimal` – Boolean	Selects Decimal extended datatips.
Hexadecimal `Environment/Debugger/Extended Tooltip Display Mode/Hexadecimal` – Boolean	Selects Hexadecimal extended datatips.
Octal `Environment/Debugger/Extended Tooltip Display Mode/Octal` – Boolean	Selects Octal extended datatips.

Unsigned Decimal `Environment/Debugger/Extended Tooltip Display Mode/Unsigned Decimal` – Boolean	Selects Unsigned Decimal extended datatips.

Target Options

Property	Description
Automatically Connect When Starting Debug `Target/Auto Connect` – Boolean	Enable automatic connection to last connected target when debug start pressed.
Automatically Disconnect When Stopping Debug `Target/Auto Disconnect` – Boolean	Enable automatic disconnection on debug stop.
Check Project And Target Processor Compatibility `Target/Enable Processor Check` – Boolean	Verify that the project-defined processor is compatible with the connected target processor.
Enable Differential Download `Target/Enable Differential Download` – Boolean	Verify the contents of memory prior to download and only download the code and data that is different.
Identify Target On Connect `Target/Identify` – Boolean	Note that turning this off may make a malfunctioning target connection appear as if it is working.
Reset Target When Restarting `Environment/Debugger/Restart Resets Target` – Boolean	Specify whether the target should be reset when restarting the debug session
Step Using Hardware Step `Environment/Debugger/Step Using Hardware Step` – Boolean	Step using hardware single stepping rather than setting breakpoints
Verify Program After Download `Target/Enable Load Verification` – Boolean	Verify that a program has been successfully downloaded after download.

Window Options

Property	Description
Clear Debug Terminal On Run `Environment/Clear Debug Terminal On Run` – Boolean	Clear the debug terminal automatically when a program is run.
Hide Output Window On Successful Load `Debugging/Hide Transcript On Successful Load` – Boolean	Hide the Output window when a load completes without error.
Show Target Log On Load `Debugging/Show Transcript On Load` – Boolean	Show the target log when a load starts.

IDE Environment Options

Browser Options

Property	Description
Text Size `Environment/Browser/Text Size` – Enumeration	Sets the text size of the integrated HTML and help browser.
Underline Hyperlinks In Browser `Environment/Browser/Underline Web Links` – Boolean	Enables underlining of hypertext links in the integrated HTML and help browser.

Directory Options

Property	Description
Package Directory `Environment/Package/Destination Directory` – String	Specifies the directory packages are installed to.

File Extension Options

Property	Description
ELF Executable File Extensions `ElfDwarf/Environment/Executable File Extensions` – StringList	The file extensions used for ELF executable files.
ELF Object File Extensions `ElfDwarf/Environment/Object File Extensions` – StringList	The file extensions used for ELF object files.

File Search Options

Property	Description
Files To Search `Find In Files/File Type` – StringList	The wildcard used to match files in Find In Files searches.
Find History `Find In Files/Find History` – StringList	The list of strings recently used in searches.
Folder History `Find In Files/Folder History` – StringList	The set of folders recently used in file searches.
Match Case `Find In Files/Match Case` – Boolean	Whether the case of letters must match exactly when searching.

Match Whole Word `Find In Files/Match Whole Word` – Boolean	Whether the whole word must match when searching.
Replace History `Find In Files/Replace History` – StringList	The list of strings recently used in searches.
Search Dependencies `Find In Files/Search Dependencies` – Boolean	Controls searching of dependent files.
Search In `Find In Files/Context` – Enumeration	Where to look to find files.
Use Regular Expressions `Find In Files/Use RegExp` – Boolean	Whether to use a regular expression or plain text search.

Internet Options

Property	Description
Check For Latest News `Environment/Internet/RSS Update` – Boolean	Specifies whether to enable downloading of the Latest News RSS feeds.
Check For Packages `Environment/Internet/Check Packages` – Boolean	Specifies whether to enable downloading of the list of available packages.
Check For Updates `Environment/Internet/Check Updates` – Boolean	Specifies whether to enable checking for software updates.
Enable Connection Debugging `Environment/Internet/Enable Debugging` – Boolean	Controls debugging traces of internet connections and downloads.
External Web Browser `Environment/External Web Browser` – FileName	The path to the external web browser to use when accessing non-local files.
HTTP Proxy Host `Environment/Internet/HTTP Proxy Server` – String	Specifies the IP address or hostname of the HTTP proxy server. If empty, no HTTP proxy server will be used.
HTTP Proxy Port `Environment/Internet/HTTP Proxy Port` – IntegerRange	Specifies the HTTP proxy server's port number.
Maximum Download History Items `Environment/Internet/Max Download History Items` – IntegerRange	The maximum amount of download history kept in the downloads window.

Launcher Options

Property	Description

Launch Latest Installations Only `Environment/Launcher Use Latest Installations Only` – Boolean	Specifies whether the CrossStudio launcher should only consider the latest installations of CrossWorks when deciding which one to use.
Launcher Enabled `Environment/Launcher Enabled` – Boolean	Specifies whether the CrossStudio launcher should be used when the operating system or an external application requests a file to be opened.

Print Options

Property	Description
Bottom Margin `Environment/Printing/Bottom Margin` – IntegerRange	The page's bottom margin in millimetres.
Left Margin `Environment/Printing/Left Margin` – IntegerRange	The page's left margin in millimetres.
Page Orientation `Environment/Printing/Orientation` – Enumeration	The page's orientation.
Page Size `Environment/Printing/Page Size` – Enumeration	The page's size.
Right Margin `Environment/Printing/Right Margin` – IntegerRange	The page's right margin in millimetres.
Top Margin `Environment/Printing/Top Margin` – IntegerRange	The page's top margin in millimetres.

Startup Options

Property	Description
Allow Multiple CrossStudios `Environment/Permit Multiple Studio Instances` – Boolean	Allow more than one CrossStudio to run at the same time.
Load Most-Recent Solution At Startup `Environment/Startup Action` – Boolean	Enables loading the previous session's solution on startup.
New Project Directory `Environment/General/Solution Directory` – String	The directory where projects are created.

Project Templates File `Environment/General/Project Templates` – String	The project templates file.
Splash Screen `Environment/Splash Screen` – Enumeration	How to display the splash screen on startup.

Status Bar Options

Property	Description
(Visible) `Environment/Status Bar` – Boolean	Show or hide the status bar.
Show Build Status Pane `Environment/General/Status Bar/Show Build Status` – Boolean	Show or hide the Build pane in the status bar.
Show Caret Position Pane `Environment/General/Status Bar/Show Caret Pos` – Boolean	Show or hide the Caret Position pane in the status bar.
Show Insert/Overwrite Status Pane `Environment/General/Status Bar/Show Insert Mode` – Boolean	Show or hide the Insert/Overwrite pane in the status bar.
Show Read-Only Status Pane `Environment/General/Status Bar/Show Read Only` – Boolean	Show or hide the Read Only pane in the status bar.
Show Size Grip `Environment/General/Status Bar/Show Size Grip` – Boolean	Show or hide the status bar size grip.
Show Target Pane `Environment/General/Status Bar/Show Target` – Boolean	Show or hide the Target pane in the status bar.
Show Time Pane `Environment/General/Status Bar/Show Time` – Boolean	Show or hide the Time pane in the status bar.

User Interface Options

Property	Description
Application Main Font `Environment/Application Main Font` – Font	The font to use for the user interface as a whole.
Application Monospace Font `Environment/Application Monospace Font` – Font	The fixed-size font to use for the user interface as a whole.

Error Display Timeout `Environment/Error Display Timeout` – IntegerRange	The minimum time, in seconds, that errors are shown for in the status bar.
Errors Are Displayed `Environment/Error Display Mode` – Enumeration	How errors are reported in CrossStudio.
File Size Display Units `Environment/Size Display Unit` – Enumeration	How to display sizes of items in the user interface. SI defines 1kB=1000 bytes, IEC defines 1kiB=1024 bytes, Alternate SI defines 1kB=1024 bytes.
Number File Names in Menus `Environment/Number Menus` – Boolean	Number the first nine file names in menus for quick keyboard access.
Show Large Icons In Toolbars `Environment/General/Large Icons` – Boolean	Show large or small icons on toolbars.
Show Ribbon `Environment/General/Ribbon/Show` – Boolean	Show or hide the ribbon.
Show Window Selector On Ctrl+Tab `Environment/Show Selector` – Boolean	Present the Window Selector on Next Window and Previous Window commands activated from the keyboard.
User Interface Theme `Environment/General/Skin` – Enumeration	The theme that CrossStudio uses.
Window Menu Contains At Most `Environment/Max Window Menu Items` – IntegerRange	The maximum number of windows appearing in the Windows menu.

Programming Language Environment Options

Assembly Language Settings

Property	Description
Column Guide Columns `Text Editor/Indent/Assembly Language/Column Guides` – String	The columns that guides are drawn for.
Indent Closing Brace `Text Editor/Indent/Assembly Language/Close Brace` – Boolean	Indent the closing brace of compound statements.
Indent Context `Text Editor/Indent/Assembly Language/Context Lines` – IntegerRange	The number of lines to use for context when indenting.
Indent Mode `Text Editor/Indent/Assembly Language/Indent Mode` – Enumeration	How to indent when a new line is inserted.
Indent Opening Brace `Text Editor/Indent/Assembly Language/Open Brace` – Boolean	Indent the opening brace of compound statements.
Indent Size `Text Editor/Indent/Assembly Language/Size` – IntegerRange	The number of columns to indent a code block.
Tab Size `Text Editor/Indent/Assembly Language/Tab Size` – IntegerRange	The number of columns between tabstops.
Use Tabs `Text Editor/Indent/Assembly Language/Use Tabs` – Boolean	Insert tabs when indenting.
User-Defined Keywords `Text Editor/Indent/Assembly Language/Keywords` – StringList	Additional identifiers to highlight as keywords.

C and C++ Settings

Property	Description
Column Guide Columns `Text Editor/Indent/C and C++/Column Guides` – String	The columns that guides are drawn for.

Indent Closing Brace `Text Editor/Indent/C and C++/Close Brace` – Boolean	Indent the closing brace of compound statements.
Indent Context `Text Editor/Indent/C and C++/Context Lines` – IntegerRange	The number of lines to use for context when indenting.
Indent Mode `Text Editor/Indent/C and C++/Indent Mode` – Enumeration	How to indent when a new line is inserted.
Indent Opening Brace `Text Editor/Indent/C and C++/Open Brace` – Boolean	Indent the opening brace of compound statements.
Indent Size `Text Editor/Indent/C and C++/Size` – IntegerRange	The number of columns to indent a code block.
Tab Size `Text Editor/Indent/C and C++/Tab Size` – IntegerRange	The number of columns between tabstops.
Use Tabs `Text Editor/Indent/C and C++/Use Tabs` – Boolean	Insert tabs when indenting.
User-Defined Keywords `Text Editor/Indent/C and C++/Keywords` – StringList	Additional identifiers to highlight as keywords.

Default Settings

Property	Description
Column Guide Columns `Text Editor/Indent/Default/Column Guides` – String	The columns that guides are drawn for.
Indent Closing Brace `Text Editor/Indent/Default/Close Brace` – Boolean	Indent the closing brace of compound statements.
Indent Context `Text Editor/Indent/Default/Context Lines` – IntegerRange	The number of lines to use for context when indenting.
Indent Mode `Text Editor/Indent/Default/Indent Mode` – Enumeration	How to indent when a new line is inserted.

Indent Opening Brace `Text Editor/Indent/Default/Open Brace` – Boolean	Indent the opening brace of compound statements.
Indent Size `Text Editor/Indent/Default/Size` – IntegerRange	The number of columns to indent a code block.
Tab Size `Text Editor/Indent/Default/Tab Size` – IntegerRange	The number of columns between tabstops.
Use Tabs `Text Editor/Indent/Default/Use Tabs` – Boolean	Insert tabs when indenting.
User-Defined Keywords `Text Editor/Indent/Default/Keywords` – StringList	Additional identifiers to highlight as keywords.

Java Settings

Property	Description
Column Guide Columns `Text Editor/Indent/Java/Column Guides` – String	The columns that guides are drawn for.
Indent Closing Brace `Text Editor/Indent/Java/Close Brace` – Boolean	Indent the closing brace of compound statements.
Indent Context `Text Editor/Indent/Java/Context Lines` – IntegerRange	The number of lines to use for context when indenting.
Indent Mode `Text Editor/Indent/Java/Indent Mode` – Enumeration	How to indent when a new line is inserted.
Indent Opening Brace `Text Editor/Indent/Java/Open Brace` – Boolean	Indent the opening brace of compound statements.
Indent Size `Text Editor/Indent/Java/Size` – IntegerRange	The number of columns to indent a code block.
Tab Size `Text Editor/Indent/Java/Tab Size` – IntegerRange	The number of columns between tabstops.
Use Tabs `Text Editor/Indent/Java/Use Tabs` – Boolean	Insert tabs when indenting.
User-Defined Keywords `Text Editor/Indent/Java/Keywords` – StringList	Additional identifiers to highlight as keywords.

Source Control Environment Options

Configuration Options

Property	Description
Enable Source Control Integration `Environment/Source Code Control/Enabled` – Boolean	Source Control integration enabled.
Source Control Provider `Environment/Source Code Control/Provider` – Enumeration	The source control provider to use.

External Tools

Property	Description
Diff Command Line `Environment/Source Code Control/DiffCommand` – StringList	The diff command line
Merge Command Line `Environment/Source Code Control/MergeCommand` – StringList	The merge command line

Preference Options

Property	Description
Inhibit Add Dialog `Environment/Source Code Control/InhibitAddDialog` – Boolean	Inhibits the dialog when adding a file to source control.
Inhibit Check In Dialog `Environment/Source Code Control/InhibitCheckinDialog` – Boolean	Inhibits the dialog when checking in a file to source control.
Inhibit Check Out Dialog `Environment/Source Code Control/InhibitCheckoutDialog` – Boolean	Inhibits the dialog when checking out a file from source control.
Inhibit Check Out On Edit Dialog `Environment/Source Code Control/InhibitCheckoutOnEditDialog` – Boolean	Inhibits the check out on edit dialog.
Inhibit Get Latest Dialog `Environment/Source Code Control/InhibitGetLatestDialog` – Boolean	Inhibits the dialog when updating the local version from the source control version.

Inhibit Undo Check Out Dialog `Environment/Source Code Control/InhibitUndoCheckoutDialog` – Boolean	Inhibits the dialog when undoing a checkout to source control.

Text Editor Environment Options

Cursor Fence Options

Property	Description
Bottom Margin `Text Editor/Margins/Bottom` – IntegerRange	The number of lines in the bottom margin.
Keep Cursor Within Fence `Text Editor/Margins/Enabled` – Boolean	Enable margins to fence and scroll around the cursor.
Left Margin `Text Editor/Margins/Left` – IntegerRange	The number of characters in the left margin.
Right Margin `Text Editor/Margins/Right` – IntegerRange	The number of characters in the right margin.
Top Margin `Text Editor/Margins/Top` – IntegerRange	The number of lines in the right margin.

Editing Options

Property	Description
Allow Drag and Drop Editing `Text Editor/Drag Drop Editing` – Boolean	Enables dragging and dropping of selections in the text editor.
Bold Popup Diagnostic Messages `Text Editor/Bold Popup Diagnostics` – Boolean	Displays popup diagnostic messages in bold for easier reading.
Check Spelling `Text Editor/Spell Checking` – Boolean	Enable spell checking in comments.
Code Completion `Text Editor/Code Completion` – Boolean	Enable or disable code completion assistance.
Column-mode Tab `Text Editor/Column Mode Tab` – Boolean	Tab key moves to the next textual column using the line above.
Confirm Modified File Reload `Text Editor/Confirm Modified File Reload` – Boolean	Display a confirmation prompt before reloading a file that has been modified on disk.
Copy Action When Nothing Selected `Text Editor/Copy Action` – Enumeration	What Copy copies when nothing is selected.
Copy On Mouse Select `Text Editor/Copy On Mouse Select` – Boolean	Automatically copy text to clipboard when marking a selection with the mouse.
Cut Action When Nothing Selected `Text Editor/Cut Action` – Enumeration	What Cut cuts when nothing is selected.

Cut Single Blank Line `Text Editor/Cut Blank Lines` – Boolean	Selects whether to place text on the clipboard when a single blank line is cut. When set to
Diagnostic Cycle Mode `Text Editor/Diagnostic Cycle Mode` – Enumeration	Iterates through diagnostics either from most severe to least severe or in reported order.
Edit Read-Only Files `Text Editor/Edit Read Only` – Boolean	Allow editing of read-only files.
Enable Popup Diagnostics `Text Editor/Enable Popup Diagnostics` – Boolean	Enables on-screen diagnostics in the text editor.
Enable Virtual Space `Text Editor/Enable Virtual Space` – Boolean	Permit the cursor to move into locations that do not currently contain text.
Expand Templates On Space `Text Editor/Auto Expand Templates` – Boolean	Enables automatic expansion of templates when the space key is pressed.
FIXME Tag List `Text Editor/FIXME Tags` – StringList	Set the tags to display as FIXMEs.
Numeric Keypad Editing `Text Editor/Numeric Keypad Enabled` – Boolean	Selects whether the numeric keypad plus and minus buttons copy and cut text.
Paste On Mouse Middle Button `Text Editor/Paste On Mouse Middle Button` – Boolean	Paste text from clipboard when mouse middle button is pressed.
Undo And Redo Behavior `Text Editor/Undo Mode` – Enumeration	How Undo and Redo group your typing when it is undone and redone.

Find And Replace Options

Property	**Description**
Case Sensitive Matching `Text Editor/Find/Match Case` – Boolean	Enables or disables the case sensitivity of letters when searching.
Find History `Text Editor/Find/History` – StringList	The list of strings recently used in searches.
Regular Expression Matching `Text Editor/Find/Use RegExp` – Boolean	Enables regular expression matching rather than plain text matching.
Replace History `Text Editor/Replace/History` – StringList	The list of strings recently used in replaces.
Whole Word Matching `Text Editor/Find/Match Whole Word` – Boolean	Enables or disables whole word matching when searching.

International

Property	Description
Default Text File Encoding `Text Editor/Default Codec` – Enumeration	The encoding to use if not overridden by a project property or file is not in a known format.

Save Options

Property	Description
Backup File History Depth `Text Editor/Backup File Depth` – IntegerRange	The number of backup files to keep when saving an existing file.
Delete Trailing Space On Save `Text Editor/Delete Trailing Space On Save` – Boolean	Deletes trailing whitespace from each line when a file is saved.
Tab Cleanup On Save `Text Editor/Cleanup Tabs On Save` – Enumeration	Cleans up tabs when a file is saved.

Visual Appearance

Property	Description
Font `Text Editor/Font` – Font	The font to use for text editors.
Hide Cursor When Typing `Text Editor/Hide Cursor When Typing` – Boolean	Hide or show the I-beam cursor when you start to type.
Highlight Cursor Line `Text Editor/Highlight Cursor Line` – Boolean	Enable or disable visually highlighting the cursor line.
Horizontal Scroll Bar `Text Editor/HScroll Bar` – Enumeration	Show or hide the horizontal scroll bar.
Insert Caret Style `Text Editor/Insert Caret Style` – Enumeration	How the caret is displayed with the editor in insert mode.
Line Numbers `Text Editor/Line Number Mode` – Enumeration	How often line numbers are displayed in the margin.
Mate Matching Mode `Text Editor/Mate Matching Mode` – Enumeration	Controls when braces, brackets, and parentheses are matched.
Overwrite Caret Style `Text Editor/Overwrite Caret Style` – Enumeration	How the caret is displayed with the editor in overwrite mode.

Show Diagnostic Icons In Gutter `Text Editor/Diagnostic Icons` – Boolean	Enables display of diagnostic icons in the icon gutter.
Show Icon Gutter `Text Editor/Icon Gutter` – Boolean	Show or hide the left-hand gutter containing breakpoint, bookmark, and optional diagnostic icons.
Show Mini Toolbar `Text Editor/Mini Toolbar` – Boolean	Show the mini toolbar when selecting text with the mouse.
Use I-beam Cursor `Text Editor/Ibeam cursor` – Boolean	Show an I-beam or arrow cursor in the text editor.
Vertical Scroll Bar `Text Editor/VScroll Bar` – Enumeration	Show or hide the vertical scroll bar.

Windows Environment Options

Call Stack Options

Property	Description
Show Call Address `Environment/Call Stack/Show Call Address` – Boolean	Enables the display of the call address in the call stack.
Show Call Source Location `Environment/Call Stack/Show Call Location` – Boolean	Enables the display of the call source location in the call stack.
Show Parameter Names `Environment/Call Stack/Show Parameter Names` – Boolean	Enables the display of parameter names in the call stack.
Show Parameter Types `Environment/Call Stack/Show Parameter Types` – Boolean	Enables the display of parameter types in the call stack.
Show Parameter Values `Environment/Call Stack/Show Parameter Values` – Boolean	Enables the display of parameter values in the call stack.

Clipboard Ring Options

Property	Description
Maximum Items Held In Ring `Environment/Clipboard Ring/Max Entries` – IntegerRange	The maximum number of items held on the clipboard ring before they are recycled.
Preserve Contents Between Runs `Environment/Clipboard Ring/Save` – Boolean	Save the clipboard ring across CrossStudio runs.

Outline Window Options

Property	Description
Group #define Directives `Windows/Outline/Group Defines` – Boolean	Group consecutive #define and #undef preprocessor directives.
Group #if Directives `Windows/Outline/Group Ifs` – Boolean	Group lines contained betwen #if, #else, and #endif preprocessor directives.
Group #include Directives `Windows/Outline/Group Includes` – Boolean	Group consecutive #include preprocessor directives.

Group Top-Level Declarations `Windows/Outline/Group Top Level Items` – Boolean	Group consecutive top-level variable and type declarations.
Group Visibility `Windows/Outline/Group Visibility` – Boolean	Group class members by public, protected, and private visibility.
Hide #region Prefix `Windows/Outline/Hide Region Prefix` – Boolean	Hides the '#region' prefix from groups and shows only the group name.
Refresh Outline and Preview `Windows/Outline/Preview Refresh Mode` – Enumeration	How the Preview pane refreshes its contects.

Project Explorer Options

Property	Description
Add Filename Replace Macros `Environment/Project Explorer/Filename Replace Macros` – StringList	Macros (system and global) used to replace the start of a filename on project file addition.
Color Project Nodes `Environment/Project Explorer/Color Nodes` – Boolean	Show the project nodes colored for identification in the Project Explorer.
Output Files Folder `Environment/Project Explorer/Show Output Files` – Boolean	Show the build output files in an Output Files folder in the project explorer.
Read-Only Data In Code `Environment/Project Explorer/Statistics Read-Only Data Handling` – Boolean	Configures whether read-only data contributes to the Code or Data statistic.
Show Dependencies `Environment/Project Explorer/Dependencies Display` – Enumeration	Controls how the dependencies are displayed.
Show File Count on Folder `Environment/Project Explorer/Count Files` – Boolean	Show the number of files contained in a folder as a badge in the Project Explorer.
Show Properties `Environment/Project Explorer/Properties Display` – Enumeration	Controls how the properties are displayed.
Show Statistics Rounded `Environment/Project Explorer/Statistics Format` – Boolean	Show exact or rounded sizes in the project explorer.
Source Control Status Column `Environment/Project Explorer/Show Source Code Control Status` – Boolean	Show the source control status column in the project explorer.

Statistics Column `Environment/Project Explorer/Statistics Display` – Boolean	Show the code and data size columns in the Project Explorer.
Synchronize Explorer With Editor `Environment/Project Explorer/Sync Editor` – Boolean	Synchronizes the Project Explorer with the document being edited.
Use Common Properties Folder `Environment/Project Explorer/Common Properties Display` – Boolean	Controls how common properties are displayed.

Properties Window Options

Property	Description
Properties Displayed `Environment/General/Properties Displayed` – Enumeration	Set how the properties are displayed.
Show Property Details `Environment/General/Property View Details` – Boolean	Show or hide the property description.

Windows Window Options

Property	Description
Buffer Grouping `Environment/Windows/Grouping` – Enumeration	How the files are grouped or listed in the Windows window.
Show File Path as Tooltip `Environment/Windows/Show Filename Tooltips` – Boolean	Show the full file name as a tooltip when hovering over files in the Windows window.
Show Line Count and File Size `Environment/Windows/Show Sizes` – Boolean	Show the number of lines and size of each file in the windows list.

ARM target support

When a target-specific executable project is created using the **New Project Wizard**, the following default files are added to the project:

- *Target* `_Startup.s` — The target-specific startup code. See **Target startup code**.
- `crt0.s` — The CrossWorks standard C runtime. See **Startup code**.
- *Target*_MemoryMap.xml — The target-specific memory map file for the board. See **memory-map files**. Note that, for some targets, a general linker placement file may not be suitable. In these cases, there will be two memory-map files: one for a flash build and one for a RAM build.
- `flash_placement.xml` — The linker placement file for a flash build.
- `sram_placement.xml` — The linker placement file for a RAM build.
- *Target* `_Target.js` — The target script file. See **Target script file**.

Initially, shared versions of these files are added to the project. If you want to modify any these shared files, select the file in the **Project**Explorer} and then click the **Import** option from the shortcut menu. This will copy a writable version of the file into your project directory and change the path in the **Project Explorer** to that of the local version. You can then make changes to the local file without affecting the shared copy of it.

The following list describes the typical flow of a C program created with CrossStudio's project templates:

- The processor starts executing at address 0x0000000, which is the reset exception vector. The exception-vector table can be found in the target-specific startup code (see **Target startup code**), and is put into the program section **.vectors**, which is positioned at address 0x00000000 by the target-specific memory-map file.
- The processor jumps to the **reset_handler** label in the target-specific startup code, which configures the target (see **Target startup code**).
- When the target is configured, the target-specific startup code jumps to the **_start** entry point in the C runtime code, which sets up the C runtime environment (see **Startup code**).
- When the C runtime environment has been set up, the C runtime code jumps to the C entry-point function, **main**.
- When the program returns from main, it re-enters the C runtime code, executes the destructors and enters an endless loop.

Target startup code

The following section describes the role of the target-specific startup code.

When you create a new project to produce an executable file using a target-specific project template, a file containing the default startup code for the target will be added to the project. Initially, a shared version of this file will be added to the project; if you want to modify this file, select the file in the **Project Explorer** and select **Import** to copy the file to your project directory.

ARM Target startup code

The target startup file typically consists of the exception vector table and the default set of exception handlers.

- **_vectors** — This is the exception vector table. It is put into its own **.vectors** section in order to ensure that it is always placed at address 0x00000000. The vector table contains jump instructions to the particular exception handlers. It is recommended that absolute jump instructions are used `ldr pc, [pc, #handler_address - . - 8]` rather than relative branch instructions `b handler_address` since many devices shadow the memory at address zero to start execution but the program will be linked to run at a different address.
- **reset_handler** — This is the main reset handler function and typically is the main entry point of an executable. The reset handler will usually carry out any target-specific initialization and then will jump to the **_start** entry point. In a C system, the **_start** entry point is in the **crt0.s** file. During development it is usual to replace this jump with an endless loop which will stop the device running potentially dangerous in-development code directly out of reset.
- **undef_handler** — This is the default, undefined-instruction exception handler.*
- **swi_handler** — This is the default, software-interrupt exception handler.*
- **pabort_handler** — This is the default, prefetch-abort exception handler.*
- **dabort_handler** — This is the default, data-abort exception handler.*
- **irq_handler** — This is the default, IRQ-exception handler.*
- **fiq_handler** — This is the default, FIQ-exception handler.*

* Declared as a weak symbol to allow the user to override the implementation.

Note that ARM exception handlers must be written in ARM assembly code. The CPU or board support package of the project you have created will typically supply an ARM assembly-coded **irq_handler** implementation that will enable you to write interrupt service routines as C functions.

Cortex-M Target startup code

The target startup file typically consists of the exception vector table and the default set of exception handlers.

- **_vectors** — This is the exception vector table. It is put into its own **.vectors** section in order to ensure that it is always placed at address 0x00000000.

The vector table is structured as follows:

- The first entry is the initial value of the stack pointer.
- The second entry is the address of the reset handler function. The reset handler will usually carry out any target-specific initialization and then jump to the **_start** entry point. In a C system, the **_start** entry point is in the `crt0.s` file. During development it is usual to replace this jump with an endless loop which will stop the device running potentially dangerous in-development code directly out of reset.
- The following 15 entries are the addresses of the standard Cortex-M exception handlers ending with the **SysTick_ISR** entry.
- Subsequent entries are addresses of device-specific interrupt sources and their associated handlers.

For each exception handler, a weak symbol is declared that will implement an endless loop. You can implement your own exception handler as a regular C function. Note that the name of the C function must match the name in the startup code e.g. **void SysTick_ISR(void)**. You can use the C preprocessor to rename the symbol in the startup code if you have existing code with different exception handler names e.g. **SysTick_ISR=SysTick_Handler**.

Startup code

The following section describes the role of the C runtime-startup code, **crt0.s** (and the Cortex-M3/Thumb-2 equivalent **thumb_crt0.s**).

When you create a new project to produce an executable file using a target-specific project template, the **crt0.s** file is added to the project. Initially, a shared version of this file is added to the project. If you want to modify this file, right-click it in the **Project Explorer** and then select **Import** from the shortcut menu to copy the file to your project directory.

The entry point of the C runtime-startup code is **_start**. In a typical system, this will be called by the target-specific startup code after it has initialized the target.

The C runtime carries out the following actions:

- Initialize the stacks.
- If required, copy the contents of the **.data** (initialized data) section from non-volatile memory.
- If required, copy the contents of the **.fast** section from non-volatile memory to SRAM.
- Initialize the **.bss** section to zero.
- Initialize the heap.
- Call constructors.
- If compiled with **FULL_LIBRARY**, get the command line from the host using **debug_getargs** and set registers to supply **argc** and **argv** to **main**.
- Call the **main** entry point.

On return from **main** or when **exit** is called…

- If compiled with **FULL_LIBRARY**, call destructors.
- If compiled with **FULL_LIBRARY**, call **atexit** functions.
- If compiled with **FULL_LIBRARY**, call **debug_exit** while supplying the return result from **main**.
- Wait in exit loop.

Program sections

The following program sections are used for the C runtime in section-placement files:

Section name	Description
.vectors	The exception vector table.
.init	Startup code that runs before the call to the application's **main** function.
.ctors	Static constructor function table.
.dtors	Static destructor function table.

.text	The program code.
.fast	Code to copy from flash to RAM for fast execution.
.data	The initialized static data.
.bss	The zeroed static data.
.rodata	The read-only constants and literals of the program.
.ARM.exidx	The C++ exception table.

Stacks

The ARM maintains six separate stacks. The position and size of these stacks are specified in the project's section-placement or memory-map file by the following program sections:

Section Name	Linker Size Symbol	Description
`.stack`	`__STACKSIZE__`	System and User mode stack.
`.stack_svc`	`__STACKSIZE_SVC__`	Supervisor mode stack
`.stack_irq`	`__STACKSIZE_IRQ__`	IRQ mode stack
`.stack_fiq`	`__STACKSIZE_FIQ__`	FIQ mode stack
`.stack_abt`	`__STACKSIZE_ABT__`	Abort mode stack
`.stack_und`	`__STACKSIZE_UND__`	Undefined mode stack

For Cortex-M devices the following stacks and linker symbol stack sizes are defined:

Section Name	Linker Size Symbol	Description
`.stack`	`__STACKSIZE__`	Main stack.
`.stack_process`	`__STACKSIZE_PROCESS__`	Process stack.

The **crt0.s** startup code references these sections and initializes each of the stack-pointer registers to point to the appropriate location. To change the location in memory of a particular stack, the section should be moved to the required position in the section-placement or memory-map file.

Should your application not require one or more of these stacks, you can remove those sections from the memory-map file or set the size to 0 and remove the initialization code from the `crt0.s` file.

The .data section

The **.data** section contains the initialized data. If the run address is different from the load address, as it would be in a flash-based application in order to allow the program to run from reset, the `crt0.s` startup code will copy the **.data** section from the load address to the run address before calling the **main** entry point.

The .fast section

For performance reasons, it is a common requirement for embedded systems to run critical code from fast memory; the **.fast** section can be used to simplify this. If the **.fast** section's run address is different from the load address, the `crt0.s` startup code will copy the **.fast** section from the load address to the run address before calling the **main** entry point.

The .bss Section

The **.bss** section contains the zero-initialized data. The startup code in `crt0.s` references the **.bss** section and sets its contents to zero.

The heap

The position and size of the heap is specified in the project's section-placement or memory-map file by the **.heap** program section.

The startup code in `crt0.s` references this section and initializes the heap. To change the position of the heap, the section should be moved to the required position in the section-placement or memory-map file.

There is a **Heap Size** linker project property you can modify in order to alter the heap size. For compatibility with earlier versions of CrossStudio, you can also specify the heap size using the heap section's **Size** property in the section-placement or memory-map file.

Should your application not require the heap functions, you can remove the heap section from the memory-map file or set the size to zero and remove the heap-initialization code from the `crt0.s` file.

memory-map files

CrossStudio's memory-map files are XML files and are used…

- *Linking*: …by the linker, to describe how to lay out a program in memory.
- *Loading*: …by the loader, to check whether a program being downloaded will actually fit into the target's memory.
- *Debugging*: …by the debugger, to describe the location and types of memory a target has. This information is used to decide how to debug the program—for example, whether to set hardware or software breakpoints on particular memory location.

Section placement files map program sections used in your program into the memory spaces defined in the memory map. For instance, it's common for code and read-only data to be programmed into non-volatile flash memory, whereas read-write data needs to be mapped onto either internal or external RAM.

Memory map files are provided in the CPU support package you are using and are referenced in executable projects by the **Memory Map File** project property. Section-placement files are provided in the base CrossWorks distribution.

ARM section placement

The following placement files are supplied for ARM targets:

File	Description
`flash_placement.xml`	Single FLASH segment with internal RAM segment and optional external RAM segment.
`flash_run_text_from_ram_placement.xml`	Single FLASH segment with internal RAM segment and optional external RAM segments. Text section is copied from FLASH to RAM.
`internal_sram_placement.xml`	Single internal RAM segment.
`flash_placement.xml`	Two FLASH segments with internal RAM segment and optional external RAM segment.
`internal_sram_placement.xml`	Internal RAM segment and optional external RAM segment.

Cortex-M section placement

The following placement files are supplied for Cortex-M targets:

File	Description

`flash_placement.xml`	Two FLASH segments and two RAM segments.
`flash_placement2.xml`	One FLASH segment and two RAM segments.
`flash_to_ram_placement.xml`	One FLASH segment and one RAM segment. Text section is copied from FLASH to RAM.
`ram_placement.xml`	Two RAM segments.

The memory segments defined in the section placement files have macro-expandable names which can be defined using the **Section Placement Macros** project property.

Some of the section placement files have a macro-expandable start attribute in the first program section. You can use this to reserve space at the beginning of the memory segment.

Project configurations

When you create a new project a default set of build configurations are created. These configurations vary depending on the CPU support package you are using and the type of project you create.

Executable project types

For **Executable** projects, some CPU support packages include the memory configuration in the build configuration. The following describes the default set of project configurations for this type of project:

Private configurations

Configuration Name	Description	
ARM	Compile and assemble for ARM instruction set. Link ARM version of libraries.	
THUMB	Compile and assemble for Thumb instruction set. Link Thumb version of libraries.	
Flash	Load into, and run from, flash memory.	
RAM	Load into, and run from, RAM.	
Debug	Compile and assemble with debug information and with optimization disabled.	
Release	Compile and assemble without debug information and with optimization enabled at level 1.	

Public configurations

Configuration Name	Inherited configurations	
ARM Flash Debug	ARM, Flash, Debug	
ARM Flash Release	ARM, Flash, Release	
ARM RAM Debug	ARM, RAM, Debug	
ARM RAM Release	ARM, RAM, Release	
THUMB Flash Debug	THUMB, Flash, Debug	

THUMB Flash Release	THUMB, Flash, Release
THUMB RAM Debug	THUMB, RAM, Debug
THUMB RAM Release	THUMB, RAM, Release

For **Executable** project types with CPU support packages that do not specify the memory configuration in the build configuration, you will get the following configurations:

Configuration Name	Description
ARM Debug	Compile/assemble for ARM instruction set. Link ARM version of libraries. Compile/assemble with debug information and with optimization disabled.
ARM Release	Compile/assemble for ARM instruction set. Link ARM version of libraries. Compile/assemble without debug information and with optimization enabled.
Thumb Debug	Compile/assemble for Thumb instruction set. Link Thumb version of libraries. Compile/assemble with debug information and with optimization disabled.
Thumb Release	Compile/assemble for Thumb instruction set. Link Thumb version of libraries. Compile/assemble without debug information and with optimization enabled.

The CPU support packages that create configurations which have no memory configuration will provide a project **Placement** property that enables the memory configuration to be selected.

Note: *Cortex-M CPU support packages will not create any ARM configurations.*

Library project types

Crossworks for ARM provides two library project types with associated build configurations. The **Static Library** project will create configurations based on combinations of ARM/THUMB and Debug/Release. When you have created a library project of this form, you will need to set the required ARM architecture, byte order (endian) and floating-point ABI project properties. The **Static Library with Configurations** project will create configurations based on combinations of:

- ARM architecture.
- ARM vs THUMB.
- Byte order (endianness).
- Floating-point ABI.
- ABI type.
- Double as float.
- Optimization for speed vs size. Debug vs Release.

For example, **V5TE VFP ARM LE SoftFP EABI Fast Debug** is a configuration for a V5TE architecture device with a VFP, ARM instruction set, little-endian byte order, soft floating point, EABI procedure calling, double is supported, do speed optimization rather than size optimization, and include debug information.

The CPU support package you are using may support a library project type—in this case the project configurations created will be based on combinations of ARM/THUMB and Debug/Release.

Externally Built Executable project types

The set of build configurations created with **Externally Built Executable** project types will either match those created for an **Executable** project types, or will have no build configurations created. The memory configuration selected for debug will be specified by the build configuration, or if no build configurations are available, by the value of the **Placement** project property.

Target script file

The target-interface system uses CrossStudio's JavaScript (ECMAScript) interpreter to support board-specific and target-specific behavior.

The main use for this is to support non-standard target and board reset schemes and to configure the target after reset using the **Reset Script** and **Loader Reset Script** facilities, described later.

The target script system can also be used to carry out target-specific operations when the target interface connects or disconnects, or when the debugger uses the **Connect**, **Disconnect**, **Stop**, and **Run** scripts, described later.

In order to reduce script duplication, when the target interface runs a reset, attach, run, or stop script, it first looks in the current active project for a file whose project property **File Type** is set to **Reset Script**. If a file of this type is found, it will be loaded prior to executing the scripts; each of the scripts can then call functions defined in this script file.

Attach script

The **Attach Script** property in the **Target** project-property group specifies the script to be executed when the debugger first attaches to an application. This can be after a download or reset before the program is run, or after an attach to a running application. The aim of the attach script is to carry out any target-specific configuration before the debugger first attaches to the application being debugged.

See arm_target_script_TargetInterface for a description of the **TargetInterface** object the attach script uses to access the target hardware.

Connect script

The **Connect Script** property in the **Target** project-property group specifies the script to be executed when the user connects to the target interface.

See arm_target_script_TargetInterface for a description of the **TargetInterface** object the connect script uses to access the target hardware.

Disconnect script

The **Disconnect Script** property of the **Target** project-property group specifies the script to be executed when the user disconnects from the target interface.

See arm_target_script_TargetInterface for a description of the **TargetInterface** object the disconnect script uses to access the target hardware.

Loader reset script

The **Loader Reset Script** property in the **Target** project-property group specifies the script to be executed in order to reset and configure the target prior to downloading a loader application. It does essentially the same job as the **Reset Script** property, but it will be used only prior to downloading a loader application, thereby allowing a loader to have a different reset script than the application. If this property is not defined, the script defined by the **Reset Script** property will be used.

See **arm_target_script_TargetInterface** for a description of the **TargetInterface** object the loader reset script uses to access the target hardware.

Reset script

The **Reset Script** property in the **Target** project-property group defines a script to execute in order to reset and configure the target.

The aim of the reset script is to get the processor into a known state. When the script has executed, the processor should be reset, stopped on the first instruction and configured appropriately.

As an example, the following script demonstrates the reset script for an Evaluator 7T target board with a memory configuration that re-maps SRAM to start from 0x00000000. The {Evaluator7T_Reset} function carries out the standard ARM reset and stops the processor prior to executing the first instruction. The {Evaluator7T_ResetWithRamAtZero} function calls this reset function and then configures target memory by accessing the configuration registers directly. See **arm_target_script_TargetInterface** for a description of the **TargetInterface** object the reset script uses to access the target hardware.

```
function Evaluator7T_Reset()
{
  TargetInterface.setNSRST(0);
  TargetInterface.setICEBreakerBreakpoint(0, 0x00000000, 0xFFFFFFFF,
                                          0x00000000, 0xFFFFFFFF, 0x100, 0xF7);
  TargetInterface.setNSRST(1);
  TargetInterface.waitForDebugState(1000);
  TargetInterface.trst();
}
function Evaluator7T_ResetWithRamAtZero()
{
  Evaluator7T_Reset();
  /*************************************************************************
  Register settings for the following memory configuration:
      +----------------------+
      | ROMCON0 - 512K FLASH | 0x01800000 - 0x0187FFFF
      +----------------------+
```

```
      | ROMCON2 - 256K SRAM  | 0x00040000 - 0x0007FFFF
      +----------------------+
      | ROMCON1 - 256K SRAM  | 0x00000000 - 0x0003FFFF
      +----------------------+
*********************************************************************/
  TargetInterface.pokeWord(0x03FF0000, 0x07FFFFA0); // SYSCFG
  TargetInterface.pokeWord(0x03FF3000, 0x00000000); // CLKCON
  TargetInterface.pokeWord(0x03FF3008, 0x00000000); // EXTACON0
  TargetInterface.pokeWord(0x03FF300C, 0x00000000); // EXTACON1
  TargetInterface.pokeWord(0x03FF3010, 0x0000003E); // EXTDBWIDTH
  TargetInterface.pokeWord(0x03FF3014, 0x18860030); // ROMCON0
  TargetInterface.pokeWord(0x03FF3018, 0x00400010); // ROMCON1
  TargetInterface.pokeWord(0x03FF301C, 0x00801010); // ROMCON2
  TargetInterface.pokeWord(0x03FF3020, 0x08018020); // ROMCON3
  TargetInterface.pokeWord(0x03FF3024, 0x0A020040); // ROMCON4
  TargetInterface.pokeWord(0x03FF3028, 0x0C028040); // ROMCON5
  TargetInterface.pokeWord(0x03FF302C, 0x00000000); // DRAMCON0
  TargetInterface.pokeWord(0x03FF3030, 0x00000000); // DRAMCON1
  TargetInterface.pokeWord(0x03FF3034, 0x00000000); // DRAMCON2
  TargetInterface.pokeWord(0x03FF3038, 0x00000000); // DRAMCON3
  TargetInterface.pokeWord(0x03FF303C, 0x9C218360); // REFEXTCON
```

Run script

The **Run Script** property in the **Target Script Options** project-property group is used to define a script to be executed when the target enters run state. This can be when the application is run for the first time or when the **Debug** > **Go** operation is carried out after the application has hit a breakpoint or was stopped using the **Debug** > **Break** operation. The aim of the run script is to carry out any target-specific operations after the debugger has finished accessing target memory. This can be useful, for example, to re-enable caches previously disabled by the stop script.

See arm_target_script_TargetInterface for a description of the **TargetInterface** object the run script uses to access the target hardware.

Stop script

The **Stop Script** property in the **Target Script Options** project-property groups is used to define a script that is executed when the target enters debug/stopped state. This can be after the application hits a breakpoint or when the **Debug** > **Break** operation is carried out. The aim of the stop script is to carry out any target-specific operations before the debugger starts accessing target memory. This is particularly useful when debugging

applications that have caches enabled, because the script can disable and flush the caches, giving the debugger access to the current memory state.

See arm_target_script_TargetInterface for a description of the **TargetInterface** object the stop script uses to access the target hardware.

Debug Interface Reset Script

The **Debug Interface Reset Script** property held in the **Target Script Options** project property groups is used to define a script that is executed when CrossWorks resets the debug interface. This should not affect the target processor and will be executed for example when the debugger attaches to a running target. Use this script if you don't want Crossworks to execute a TRST to reset the JTAG TAP, for example if the device has a JTAG router.

See arm_target_script_TargetInterface for a description of the **TargetInterface** object which is used by the debug interface reset script to access the target hardware.

TAP Reset Script

The **TAP Reset Script** property held in the **Target Script Options** project-property groups is used to define a script that is executed when CrossWorks resets the JTAG connection when exploring the JTAG chain. This script can be used to configure a JTAG router that would be reset when the standard TRST sequence is applied.

See arm_target_script_TargetInterface for a description of the **TargetInterface** object the TAP Reset Script uses to access the target hardware.

Program loading

CrossStudio for ARM supports flash programming (and subsequent debugging) by loading a program—the *loader executable*, or *loader* —into the target's RAM and transmitting to it the data to be programmed.

The **Loader File Path** project property is part of a project's configuration. It specifies the location of the loader executable to be used; if this property is defined, the loader executable will be downloaded and run on the target prior to downloading the main application.

In addition, the **Loader File Type** project property must be specified. This tells CrossStudio how to communicate with the loader. The various communication mechanisms available are explained in more detail later. The **Load File Type** property may be set to one of the following:

- *LIBMEM RPC Loader*: Calls to LIBMEM library functions will be made directly, using a remote prodedure call mechanism.
- *Comms Channel Loader*: The ARM debug comms channel is used to communicate with the loader.
- *Fast Comms Channel Loader*: The ARM debug comms channel is used to communicate with the loader. This scheme is significantly faster for downloading than **Comms Channel Loader** because it makes the assumption that the loader is always ready to read data and therefore does not check the ARM comms-channel status before sending data to it. This may not be suitable for all targets or loaders. If you experience reliability problems when downloading and verifying programs while using this setting, try reverting to the **Comms Channel Loader**.
- *RAM Loader*: The target's RAM is used to communicate with the loader.

To write your own loader programs, see **LIBMEM loader library**.

Debug Capabilities

The particular debugging capabilities provided in CrossWorks for ARM depends upon the particular ARM device being used. The following table summarizes the CrossStudio debug facilities available for each ARM device type:

ARM Debug Architecture	Software Breakpoints	Hardware Breakpoints	Break on Exception	Monitor Mode	Memory Access	Debug I/O
ARM7	Unlimited (1 hardware breakpoint used)	2	No	Yes	Stop CPU or Monitor Mode	Stop CPU or DCC
ARM9	Unlimited (1 hardware breakpoint used on ARM920T/ ARM922T)	2	Yes	Yes	Stop CPU or Monitor Mode	Stop CPU or DCC
ARM11	Unlimited	8 (6 instruction and 2 data)	Yes	No	Stop CPU	Stop CPU or DCC
Cortex-M3	Unlimited	Max. 12 (8 instruction, 4 data)	Yes	No	Real Time	Stop CPU or Real Time
Cortex-M1/ M0	Unlimited	Max. 6 (4 instruction, 2 data)	Yes	No	Real Time	Stop CPU or Real Time
Cortex-A/R	Unlimited	8 (6 instruction and 2 data)	Yes	No	Stop CPU	Stop CPU or DCC
XScale	Unlimited	4 (2 instruction, 2 data)	Yes	No	Stop CPU	Stop CPU

Common debug features

Single stepping is implemented by setting a hardware breakpoint on the next instruction that will execute in the current execution thread. Therefore, you will not single step into a different thread of execution, unless code is shared; and, if you have used all the hardware breakpoints, you won't be able to single step.

Software breakpoints are implemented by overwriting the instruction at the desired breakpoint address with a breakpoint instruction. Restarting from a software breakpoint uses the built-in ARM simulator, unless the instruction cannot be simulated, in which case the instruction is written back to memory and single stepped.

The project properties **Read-only Software Breakpoints** and **Read-write Software Breakpoints** control how software breakpoints are used in memory areas marked ReadOnly and ReadWrite in the current project's memory-map file.

The project property **Startup Completion Point** is used to specify the address of a symbol that has a breakpoint on it. When the startup completion point is hit, software breakpoints will be used and debugIO will be enabled. This enables debugging of applications that copy code into RAM on startup.

ARM7 and ARM9

These ARM devices provide two hardware-breakpoint units that can be configured as program or data breakpoints.

There is no software-breakpoint instruction on ARM7TDMI, ARM720T, and ARM920T devices. To implement software breakpoints, one of the hardware-breakpoint units is programmed to break on the execution of the ARM opcode 0xdfffdfff or 0xdffedffe and, consequently, the Thumb opcode 0xdfff and 0xdffe.

Data breakpoints can only be set on ranges of aligned powers of 2. So *char*, *short*, and *int/long* variables can have breakpoints set on them, but larger variables are unlikely to meet the requirement for aligned powers of 2. Data-valued breakpoints such as **count==3** are supported, as are masked data-valued breakpoints such as **(x & 1)==1**.

The hardware breakpoints can be chained together to allow breakpoint sequencing. When you are connected to the target, use the breakpoint-edit dialog or the breakpoint properties to change the **Action** to **Set Chain** on the first breakpoint, and change the **Action** of the second breakpoint to **Stop (When Chain Set)**.

ARM9 devices have a vector-catch capability that can be set in the exceptions group of the **Breakpoints** window to enable a breakpoint when an exception occurs.

The debug communication channel (DCC) can be used to implement debug I/O, which depends on the setting of the **DebugIO Implementation** project property. Using the DCC to implement debug I/O enables interrupts to be serviced during debug I/O.

The DCC is also used to implement communications with the debug handler, if the project property **Use Debug Handler** is set. You can build the debug handler into your application by adding the file `$(StudioDir)/source/ARMDIDebugHandler.s` to your project. When you have the debug handler in your project, you can enable the project property **Monitor Mode Debug** to allow interrupts to be serviced when a breakpoint is hit. To do this, you must set the prefetch and data-abort exception vectors to jump to the symbols **dbg_pabort_handler** and **dbg_dabort_handler**, respectively. You can also enable the project property **Monitor Mode Memory**, in which case CrossWorks will access memory using the debug handler when the application is running. You must arrange for your application to call the function **dbg_poll** at regular intervals, which will enable interrupts to be serviced while the debugger is accessing memory.

ARM11

These devices provide 6 hardware instruction breakpoints and 2 hardware data breakpoints. Data-valued breakpoints are not supported.

- Vector catching is supported
- Debug I/O is supported by stopping the CPU or the DCC.
- Memory access is supported by stopping the CPU.
- Monitor mode is not supported.

Cortex-M

Cortex-M devices have a variable number of instruction breakpoints and data breakpoints. Typically, Cortex-M3 parts have six instruction breakpoints and four data breakpoints, Cortex-M1/M0 parts have four instruction and two data breakpoints. Note that the instruction breakpoints work only on the internal code memory of the Cortex-M devices. If you have external flash on your Cortex-M device and software breakpoints in flash aren't supported, a data breakpoint is used, which will stop the processor after the instruction has executed.

Data breakpoints can only be set on ranges of aligned powers of 2. So *char*, *short*, and *int/long* variables can have breakpoints set on them, but larger variables are unlikely to meet the requirement for aligned powers of 2. One data-valued breakpoint, such as **count==3**, is optionally supported on some Cortex-M3 devices.

- Vector catching is supported.
- Debug I/O is supported by stopping the CPU or polling memory.
- The internal data and system memories and the external memories of Cortex-M devices can be accessed without stopping the CPU. When accessing the internal code memory of Cortex-M devices, the CPU is stopped.
- Monitor mode is not supported.

Cortex-A and Cortex-R

Cortex-A and Cortex-R devices provide six hardware instruction breakpoints and two hardware data breakpoints. Data-valued breakpoints are not supported.

- Vector catching is supported.
- Debug I/O is supported by stopping the CPU or the DCC.
- Memory access is supported by stopping the CPU.
- Monitor mode is not supported.

XScale

XScale devices have two instruction breakpoints and two data breakpoints. The data breakpoints are supported on *int/long* variables only.

- Vector catching is supported.
- Debug I/O is supported by stopping the CPU.
- Memory access is supported by stopping the CPU.
- Monitor mode is not supported.

Target interfaces

A target interface is a mechanism for communicating with, and controlling, a target. A target can be either a physical hardware device or a software simulation of a device. CrossStudio has a **Targets** window for viewing and manipulating target interfaces. For more information, see **Targets window**.

Before you can use a target interface, you must *connect* to it. You can only connect to one target interface at a time. For more information, see **Connecting to a target**.

All target interfaces have a set of properties. The properties provide information on the connected target and allow the target interface to be configured. For more information, see **Viewing and editing target properties**.

Target Interface	ARM7	ARM9	ARM11	XScale	Cortex-M (JTAG)	Cortex-M (SWD)	Cortex-A/R
CrossConnect for ARM	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Generic ARM Debug Interface	Yes	Yes	No	No	Yes	Yes	No
Generic FT2232 Device	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Macraigor Systems's Wiggler for ARM	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Segger J-Link	Yes	Yes	No	No	Yes	Yes	No
CrossStudio ARM Simulator	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ST-Link	No	No	No	No	Yes	Yes	No
ST-Link/V2	No	No	No	No	Yes	Yes	No
KPandE UNIT Interface DLL	No	No	No	No	Yes	No	No
Kinetis OSJTAG	No	No	No	No	Yes	No	No

Stellaris ICDI	No	No	No	No	Yes	No	No

Note that the Amontec JTAGkey and Olimex ARM-USB-OCD are FT2232-based devices.

See **Debug Capabilities** for details about the debug support Crossworks provides for the various devices.

Note that the Segger J-Link, ST-Link, and PandE UNIT Interface DLL target interfaces require other files that are supplied by the vendor of the target interface.

The Segger J-Link target interface's **J-Link DLL File** property should point at the file `JLinkARM.dll` on Windows and to `JLinkARM.so` on Linux. Go to **http://www.segger.com/cms/jlink-software.html** for the latest downloads.

The ST-Link's **ST-LINK DLL File** property should point at the file `STLinkUSBDriver.dll` that is supplied in the *ST-Link Utility*, found here:

http://www.st.com/internet/com/SOFTWARE_RESOURCES/TOOL/DEVICE_PROGRAMMER/um0892.zip

The PandE UNIT Interface DLL's **File Path** property should point at the file `unit_ngs_arm.dll`. Contact Rowley Associates for the latest information on where to find this.

Do not copy the above files into the CrossWorks distribution—just reference the files where they have been installed.

ARM Simulator target interface

The ARM Simulator target interface provides access to CrossStudio's ARM instruction set simulator (ISS). The ISS simulates the ARM V4T, ARM V5TE, ARM V6-M, and ARM V7-M instruction sets, as defined in appropriate ARM Architecture Reference Manuals. The ARM architecture, core type, and memory byte order to be simulated are specified by the project's code-generation properties.

The instruction set simulator (ISS) supports MCR and MRC access to the 16 primary registers of the System Control coprocessor (CP15), as defined in the ARM Architecture Reference Manual. The MMU is simulated, but the cache is not. The ISS supports MCR and MRC access to the Debug Communication Channel (CP14), as defined in the ARM7TDMI Technical Reference Manual. The ISS supports a limited subset of VFP instructions (CP10 and CP11) that enables C programs that use the VFP to execute.

The ISS implements a three-word, instruction-prefetch buffer.

The memory system simulated by the ISS is implemented by the dynamic link library and associated parameter defined in the project's simulator properties.

The ISS supports program loading and debugging with an unlimited number of breakpoints. The ISS supports instruction tracing, execution counts, exception-vector trapping, and exception-vector triggering.

CrossConnect Target Interface

Interface

Property	Description
Information `interfaceInformation` – String	Interface connection information.
Model `modelInformation` – String	CrossConnect Model.
Serial Number `connectedSerialNumber` – String	The serial number of the currently connected CrossConnect.
Target Voltage `target_voltage` – String	The target's JTAG referece voltage.
Version `interfaceVersion` – String	The target interface version number.

JTAG

Property	Description
Adaptive Clocking `adaptiveClocking` – Enumeration	Specifies whether JTAG adaptive clocking should be used.
JTAG Clock Divider `jtagDivider` – IntegerRange	The amount to divide the JTAG clock frequency.
Target Interface Type `tif` – Enumeration	The target interface type.

Loader

Property	Description
Erase All `loaderEraseAll` – Boolean	If set to **Yes**, all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
Erase All Timeout `loaderEraseAllTimeout` – IntegerRange	The timeout period for an erase all operation in milliseconds.

Loader Timeout `loaderTimeout` – IntegerRange	The timeout period for loader operations in milliseconds.
No Load Sections `noLoadSections` – StringList	Names of (loadable) sections not to load.

Target

Property	Description
Connection `Connection` – String	The USB serial number of the CrossConnect to use.
Device Type `device_id` – String	The detected type of the currently connected target device.
Fast Memory Accesses `fastMemoryAccessesEnabled` – Boolean	Specifies whether fast memory accesses should be used for ARM7, ARM9 and Cortex-M3 targets. With this option set to **Yes** the target interface will not wait for a memory access to complete before moving onto the next - this means it relies on the JTAG interface being slower than the memory interface. If your target is running slowly, or has slow memory you may experience problems reading from or writing to memory with this option enabled in which case you should set this option to **No**. The default setting of this property on this target interface is **Yes**, this is because the implementation of slow memory accesses is considerably slower than fast accesses on this target interface - if you experience problems reading from or writing to memory you may find you achieve better performance by reducing the JTAG clock frequency using the **JTAG Clock Divider** property rather than disabling this option.
Memory Access Timeout `memoryAccessTimeout` – IntegerRange	The timeout period for memory accesses in milliseconds.
Processor Stop Timeout `stopTimeout` – IntegerRange	The timeout period for stopping the processor in milliseconds.
Restrict Memory Accesses `restrictMemoryAccesses` – Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.

Segger J-Link Target Interface

J-Link

Property	Description
DLL Version – String	The J-Link DLL version.
Firmware Version – String	The J-Link firmware version.
Hardware Version – String	The J-Link hardware version.
J-Link DLL File `JLinkARMDLLFileName` – FileName	The file path of the JLinkARM.dll to use.
J-Link Script File `JLinkScriptFileName` – FileName	The file path of the .JLinkScript to use.
Reset Type `resetType` – IntegerRange	The reset strategy to use.
Speed `speed` – IntegerRange	The required JTAG/SWD clock frequency in kHz (0 to auto-detect best possible).
Supply Power `supplyPower` – Enumeration	The J-Link supplies power to the target.
Verify Read Operations `checkModeAfterRead` – Enumeration	The CPU mode is checked after each read operation.

JTAG

Property	Description
Target Interface Type `tif` – Enumeration	The target interface type.

Loader

Property	Description

Erase All `loaderEraseAll` – Boolean	If set to **Yes**, all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
Erase All Timeout `loaderEraseAllTimeout` – IntegerRange	The timeout period for an erase all operation in milliseconds.
Loader Timeout `loaderTimeout` – IntegerRange	The timeout period for loader operations in milliseconds.
No Load Sections `noLoadSections` – StringList	Names of (loadable) sections not to load.

Target

Property	Description
Current Speed – IntegerRange	The JTAG/SWD clock frequency the J-Link is currently using.
Device Type `device_id` – String	The detected type of the currently connected target device.
Processor Stop Timeout `stopTimeout` – IntegerRange	The timeout period for stopping the processor in milliseconds.
Restrict Memory Accesses `restrictMemoryAccesses` – Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.
Target Voltage – String	The target reference voltage.

Olimex ARM-USB-OCD Target Interface

Interface

Property	Description
Device Driver `deviceDriver` – Enumeration	Specifies which device driver to use.
Serial Number `connectedSerialNumber` – String	The serial number of the currently connected FT2232.
Use Serial Number `connectToSerialNumber` – String	The serial number of the FT2232 device you want to connect to. If multiple FT2232 devices are connected to your system, this property allows you to specify which one to use. If no serial number is specified, the first available FT2232 device will be used.
Version `interfaceVersion` – String	The target interface version number.

JTAG

Property	Description
Adaptive Clocking `adaptiveClocking` – Enumeration	Specifies whether JTAG adaptive clocking should be used.
JTAG Clock Divider `jtagDivider` – IntegerRange	The amount to divide the JTAG clock frequency.
Target Interface Type `tif` – Enumeration	The target interface type.
nTRST Open Drain `trstOpenDrain` – Boolean	Specifies whether the nTRST signal is open-drain or push-pull.

Loader

Property	Description
Erase All `loaderEraseAll` – Boolean	If set to **Yes**, all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.

Erase All Timeout `loaderEraseAllTimeout` – IntegerRange	The timeout period for an erase all operation in milliseconds.
Loader Timeout `loaderTimeout` – IntegerRange	The timeout period for loader operations in milliseconds.
No Load Sections `noLoadSections` – StringList	Names of (loadable) sections not to load.

Target

Property	Description
Device Type `device_id` – String	The detected type of the currently connected target device.
Fast Memory Accesses `fastMemoryAccessesEnabled` – Boolean	Specifies whether fast memory accesses should be used for ARM7, ARM9 and Cortex-M3 targets. With this option set to **Yes** the target interface will not wait for a memory access to complete before moving onto the next - this means it relies on the JTAG interface being slower than the memory interface. If your target is running slowly, or has slow memory you may experience problems reading from or writing to memory with this option enabled in which case you should set this option to **No**. The default setting of this property on this target interface is **Yes**, this is because the implementation of slow memory accesses is considerably slower than fast accesses on this target interface - if you experience problems reading from or writing to memory you may find you achieve better performance by reducing the JTAG clock frequency using the **JTAG Clock Divider** property rather than disabling this option.
Memory Access Timeout `memoryAccessTimeout` – IntegerRange	The timeout period for memory accesses in milliseconds.
Processor Stop Timeout `stopTimeout` – IntegerRange	The timeout period for stopping the processor in milliseconds.
Restrict Memory Accesses `restrictMemoryAccesses` – Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.

Amontec JTAGkey Target Interface

Interface

Property	Description
Device Driver `deviceDriver` – Enumeration	Specifies which device driver to use.
Serial Number `connectedSerialNumber` – String	The serial number of the currently connected FT2232.
Use Serial Number `connectToSerialNumber` – String	The serial number of the FT2232 device you want to connect to. If multiple FT2232 devices are connected to your system, this property allows you to specify which one to use. If no serial number is specified, the first available FT2232 device will be used.
Version `interfaceVersion` – String	The target interface version number.

JTAG

Property	Description
Adaptive Clocking `adaptiveClocking` – Enumeration	Specifies whether JTAG adaptive clocking should be used.
JTAG Clock Divider `jtagDivider` – IntegerRange	The amount to divide the JTAG clock frequency.
Target Interface Type `tif` – Enumeration	The target interface type.
nSRST Open Drain `srstOpenDrain` – Boolean	Specifies whether the nSRST signal is open-drain or push-pull.
nTRST Open Drain `trstOpenDrain` – Boolean	Specifies whether the nTRST signal is open-drain or push-pull.

Loader

Property	Description

Erase All `loaderEraseAll` – Boolean	If set to **Yes**, all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
Erase All Timeout `loaderEraseAllTimeout` – IntegerRange	The timeout period for an erase all operation in milliseconds.
Loader Timeout `loaderTimeout` – IntegerRange	The timeout period for loader operations in milliseconds.
No Load Sections `noLoadSections` – StringList	Names of (loadable) sections not to load.

Target

Property	Description
Device Type `device_id` – String	The detected type of the currently connected target device.
Fast Memory Accesses `fastMemoryAccessesEnabled` – Boolean	Specifies whether fast memory accesses should be used for ARM7, ARM9 and Cortex-M3 targets. With this option set to **Yes** the target interface will not wait for a memory access to complete before moving onto the next - this means it relies on the JTAG interface being slower than the memory interface. If your target is running slowly, or has slow memory you may experience problems reading from or writing to memory with this option enabled in which case you should set this option to **No**. The default setting of this property on this target interface is **Yes**, this is because the implementation of slow memory accesses is considerably slower than fast accesses on this target interface - if you experience problems reading from or writing to memory you may find you achieve better performance by reducing the JTAG clock frequency using the **JTAG Clock Divider** property rather than disabling this option.
Memory Access Timeout `memoryAccessTimeout` – IntegerRange	The timeout period for memory accesses in milliseconds.
Processor Stop Timeout `stopTimeout` – IntegerRange	The timeout period for stopping the processor in milliseconds.
Restrict Memory Accesses `restrictMemoryAccesses` – Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.

PandE UNIT Interface DLL Target Interface

Loader

Property	Description
Erase All `loaderEraseAll` – Boolean	If set to **Yes**, all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
Erase All Timeout `loaderEraseAllTimeout` – IntegerRange	The timeout period for an erase all operation in milliseconds.
Loader Timeout `loaderTimeout` – IntegerRange	The timeout period for loader operations in milliseconds.
No Load Sections `noLoadSections` – StringList	Names of (loadable) sections not to load.

P&E UNIT Interface DLL

Property	Description
API Version – String	The API version of the P&E UNIT Interface DLL.
File Path `PandEMicroDriver` – FileName	The file path of the unit_ngs_arm.dll to use.
Port `port` – IntegerRange	The port number to connect to.

Target

Property	Description
Device Type `device_id` – String	The detected type of the currently connected target device.
Processor Stop Timeout `stopTimeout` – IntegerRange	The timeout period for stopping the processor in milliseconds.
Restrict Memory Accesses `restrictMemoryAccesses` – Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.

ST-LINK Target Interface

JTAG

Property	Description
Target Interface Type `tif` – Enumeration	The target interface type.

Loader

Property	Description
Erase All `loaderEraseAll` – Boolean	If set to **Yes**, all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
Erase All Timeout `loaderEraseAllTimeout` – IntegerRange	The timeout period for an erase all operation in milliseconds.
Loader Timeout `loaderTimeout` – IntegerRange	The timeout period for loader operations in milliseconds.
No Load Sections `noLoadSections` – StringList	Names of (loadable) sections not to load.

ST-LINK

Property	Description
Firmware Version – String	The Main, JTAG and SWIM firmware versions.
ST-LINK DLL File `STLinkUSBDriver` – FileName	The file path of the STLinkUSBDriver.dll to use.

Target

Property	Description
Device Type `device_id` – String	The detected type of the currently connected target device.

Processor Stop Timeout `stopTimeout` – IntegerRange	The timeout period for stopping the processor in milliseconds.
Restrict Memory Accesses `restrictMemoryAccesses` – Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.
Voltage – String	The target reference voltage.

ST-LINK/V2 Target Interface

JTAG

Property	Description
Target Interface Type `tif` – Enumeration	The target interface type.

Loader

Property	Description
Erase All `loaderEraseAll` – Boolean	If set to **Yes**, all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
Erase All Timeout `loaderEraseAllTimeout` – IntegerRange	The timeout period for an erase all operation in milliseconds.
Loader Timeout `loaderTimeout` – IntegerRange	The timeout period for loader operations in milliseconds.
No Load Sections `noLoadSections` – StringList	Names of (loadable) sections not to load.

ST-LINK

Property	Description
Firmware Version – String	The Main, JTAG and SWIM firmware versions.

Target

Property	Description
Device Type `device_id` – String	The detected type of the currently connected target device.
Processor Stop Timeout `stopTimeout` – IntegerRange	The timeout period for stopping the processor in milliseconds.

Restrict Memory Accesses `restrictMemoryAccesses` – Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.
Voltage – String	The target reference voltage.

Kinetis OSJTAG Target Interface

Kinetis OSJTAG

Property	Description
Firmware Version – String	The Firmware version of the Kinetis OSJTAG.

Loader

Property	Description
Erase All `loaderEraseAll` – Boolean	If set to **Yes**, all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
Erase All Timeout `loaderEraseAllTimeout` – IntegerRange	The timeout period for an erase all operation in milliseconds.
Loader Timeout `loaderTimeout` – IntegerRange	The timeout period for loader operations in milliseconds.
No Load Sections `noLoadSections` – StringList	Names of (loadable) sections not to load.

Target

Property	Description
Device Type `device_id` – String	The detected type of the currently connected target device.
Processor Stop Timeout `stopTimeout` – IntegerRange	The timeout period for stopping the processor in milliseconds.
Restrict Memory Accesses `restrictMemoryAccesses` – Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.

Stellaris ICDI Target Interface

Loader

Property	Description
Erase All `loaderEraseAll` – Boolean	If set to **Yes**, all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
Erase All Timeout `loaderEraseAllTimeout` – IntegerRange	The timeout period for an erase all operation in milliseconds.
Loader Timeout `loaderTimeout` – IntegerRange	The timeout period for loader operations in milliseconds.
No Load Sections `noLoadSections` – StringList	Names of (loadable) sections not to load.

Target

Property	Description
Device Type `device_id` – String	The detected type of the currently connected target device.
Processor Stop Timeout `stopTimeout` – IntegerRange	The timeout period for stopping the processor in milliseconds.
Restrict Memory Accesses `restrictMemoryAccesses` – Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.

Macraigor Wiggler (20 and 14 pin) Target Interface

Connection

Property	Description
Parallel Port `portName` – String	The parallel port connection to use to connect to target.
Parallel Port Address `portAddress` – String	The base address of the currently connected parallel port.
Parallel Port Sharing `portSharing` – Boolean	Specifies whether sharing of the parallel port with other device drivers or programs is permitted.

Interface

Property	Description
Version `interfaceVersion` – String	The target interface version number.

JTAG

Property	Description
Invert nSRST `invertNSRST` – Boolean	Specify whether the nSRST signal should be inverted.
JTAG Clock Divider `jtagDivider` – IntegerRange	The amount to divide the JTAG clock frequency.
Target Interface Type `tif` – Enumeration	The target interface type.

Loader

Property	Description
Erase All `loaderEraseAll` – Boolean	If set to **Yes**, all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.

Erase All Timeout `loaderEraseAllTimeout` – IntegerRange	The timeout period for an erase all operation in milliseconds.
Loader Timeout `loaderTimeout` – IntegerRange	The timeout period for loader operations in milliseconds.
No Load Sections `noLoadSections` – StringList	Names of (loadable) sections not to load.

Target

Property	Description
Device Type `device_id` – String	The detected type of the currently connected target device.
Fast Memory Accesses `fastMemoryAccessesEnabled` – Boolean	Specifies whether fast memory accesses should be used for ARM7, ARM9 and Cortex-M3 targets. With this option set to **Yes** the target interface will not wait for a memory access to complete before moving onto the next - this means it relies on the JTAG interface being slower than the memory interface. If your target is running slowly, or has slow memory you may experience problems reading from or writing to memory with this option enabled in which case you should set this option to **No**. The default setting of this property on this target interface is **Yes**, this is because the implementation of slow memory accesses is considerably slower than fast accesses on this target interface - if you experience problems reading from or writing to memory you may find you achieve better performance by reducing the JTAG clock frequency using the **JTAG Clock Divider** property rather than disabling this option.
Memory Access Timeout `memoryAccessTimeout` – IntegerRange	The timeout period for memory accesses in milliseconds.
Processor Stop Timeout `stopTimeout` – IntegerRange	The timeout period for stopping the processor in milliseconds.
Restrict Memory Accesses `restrictMemoryAccesses` – Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.

Generic FT2232 Target Interface

FT2232 Pin Configuration

Property	Description
Connected LED Inversion Mask `connectedLedXORMask` – IntegerHex	Specifies the FT2232 output pin(s) to invert when setting 'connected' LED.
Connected LED Mask `connectedLedMask` – IntegerHex	Specifies the FT2232 output pin(s) to use for the 'connected' LED.
Output Pins `outputDirection` – IntegerHex	Specifies the FT2232 pins that are to be configured for output.
Output Value `outputValue` – IntegerHex	Specifies the initial value of the FT2232 output pins on connection.
Output Value 2 `outputValue2` – IntegerHex	If non-zero the 2nd initial value of the FT2232 output pins on connection.
Running LED Inversion Mask `runningLedXORMask` – IntegerHex	Specifies the FT2232 output pin(s) to invert when setting the 'running' LED.
Running LED Mask `runningLedMask` – IntegerHex	Specifies the FT2232 output pin(s) to use for the 'running' LED
SWD Direction Inversion Mask `swdDirectionXORMask` – IntegerHex	Specifies the FT2232 output pin(s) to invert to set serial wire debug to output.
SWD Direction Mask `swdDirectionMask` – IntegerHex	Specifies the FT2232 output pin(s) to use to set serial wire debug to output.
SWD Enable Inversion Mask `swdEnableXORMask` – IntegerHex	Specifies the FT2232 output pin(s) to invert when enabling serial wire .
SWD Enable Mask `swdEnableMask` – IntegerHex	Specifies the FT2232 output pin(s) to use when enabling serial wire debug.
nSRST Inversion Mask `srstXORMask` – IntegerHex	Specifies the FT2232 output pin(s) to invert when setting the nSRST signal.
nSRST Mask `srstMask` – IntegerHex	Specifies the FT2232 output pin(s) to use for the nSRST signal.
nTRST Inversion Mask `trstXORMask` – IntegerHex	Specifies the FT2232 output pin(s) to invert when setting the nTRST signal.
nTRST Mask `trstMask` – IntegerHex	Specifies the FT2232 output pin(s) to use for the nTRST signal.

FT2232 USB

Property	Description

Channel `channel` – Enumeration	Specifies the FT2232 channel to use
PID `usbPid` – StringList	Specifies the USB product ID of the FT2232 device.
VID `usbVid` – String	Specifies the USB vendor ID of the FT2232 device.

Interface

Property	Description
Device Driver `deviceDriver` – Enumeration	Specifies which device driver to use.
Serial Number `connectedSerialNumber` – String	The serial number of the currently connected FT2232.
Use Serial Number `connectToSerialNumber` – String	The serial number of the FT2232 device you want to connect to. If multiple FT2232 devices are connected to your system, this property allows you to specify which one to use. If no serial number is specified, the first available FT2232 device will be used.
Version `interfaceVersion` – String	The target interface version number.

JTAG

Property	Description
Adaptive Clocking `adaptiveClocking` – Enumeration	Specifies whether JTAG adaptive clocking should be used.
JTAG Clock Divider `jtagDivider` – IntegerRange	The amount to divide the JTAG clock frequency.
Target Interface Type `tif` – Enumeration	The target interface type.

Loader

Property	Description

Erase All `loaderEraseAll` – Boolean	If set to **Yes**, all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
Erase All Timeout `loaderEraseAllTimeout` – IntegerRange	The timeout period for an erase all operation in milliseconds.
Loader Timeout `loaderTimeout` – IntegerRange	The timeout period for loader operations in milliseconds.
No Load Sections `noLoadSections` – StringList	Names of (loadable) sections not to load.

Target

Property	Description
Device Type `device_id` – String	The detected type of the currently connected target device.
Fast Memory Accesses `fastMemoryAccessesEnabled` – Boolean	Specifies whether fast memory accesses should be used for ARM7, ARM9 and Cortex-M3 targets. With this option set to **Yes** the target interface will not wait for a memory access to complete before moving onto the next - this means it relies on the JTAG interface being slower than the memory interface. If your target is running slowly, or has slow memory you may experience problems reading from or writing to memory with this option enabled in which case you should set this option to **No**. The default setting of this property on this target interface is **Yes**, this is because the implementation of slow memory accesses is considerably slower than fast accesses on this target interface - if you experience problems reading from or writing to memory you may find you achieve better performance by reducing the JTAG clock frequency using the **JTAG Clock Divider** property rather than disabling this option.
Memory Access Timeout `memoryAccessTimeout` – IntegerRange	The timeout period for memory accesses in milliseconds.
Processor Stop Timeout `stopTimeout` – IntegerRange	The timeout period for stopping the processor in milliseconds.
Restrict Memory Accesses `restrictMemoryAccesses` – Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.

Generic Target Interface

Generic

Property	Description
Applicable Host OS `host` – StringList	The names of host OS that are supported.
Generic DLL File `DLLFileName` – FileName	The file path of the .dll to use.

C Library User Guide

This section describes the library and how to use and customize it.

The libraries supplied with CrossWorks have all the support necessary for input and output using the standard C functions **printf** and **scanf**, support for the **assert** function, both 32-bit and 64-bit floating point, and are capable of being used in a multi-threaded environment. However, to use these facilities effectively you will need to customize the low-level details of *how* to input and output characters, what to do when an assertion fails, how to provide protection in a multithreaded environment, and how to use the available hardware to the best of its ability.

Floating point

The CrossWorks C library uses IEEE floating point format as specified by the ISO 60559 standard with restrictions.

This library favors code size and execution speed above absolute precision. It is suitable for applications that need to run quickly and not consume precious resources in limited environments. The library does not implement features rarely used by simple applications: floating point exceptions, rounding modes, and subnormals.

NaNs and infinities are supported and correctly generated. The only rounding mode supported is round-to-nearest. Subnormals are always flushed to a correctly-signed zero. The mathematical functions use stable approximations and do their best to cater ill-conditioned inputs.

Single and double precision

CrossWorks C allows you to choose whether the **double** data type uses the IEC 60559 32-bit or 64-bit format. The following sections describe the details of why you would want to choose a 32-bit **double** rather than a 64-bit **double** in many circumstances.

Why choose 32-bit doubles?

Many users are surprised when using **float** variables exclusively that sometimes their calculations are compiled into code that calls for **double** arithmetic. They point out that the C standard allows **float** arithmetic to be carried out only using **float** operations and not to automatically promote to the **double** data type of classic K&R C.

This is valid point. However, upon examination, even the simplest calculations can lead to **double** arithmetic. Consider:

```
// Compute sin(2x)
float sin_two_x(float x)
{
  return sinf(2.0 * x);
}
```

This looks simple enough. We're using the **sinf** function which computes the sine of a **float** and returns a **float** result. There appears to be no mention of a **double** anywhere, yet the compiler generates code that calls **double** support routines—but why?

The answer is that the constant **2.0** is a **double** constant, not a **float** constant. That is enough to force the compiler to convert both operands of the multiplication to **double** format, perform the multiplication in **double** precision, and then convert the result back to **float** precision. To avoid this surprise, the code should have been written:

```
// Compute sin(2x)
float sin_two_x(float x)
{
  return sinf(2.0F * x);
}
```

This uses a single precision floating-point constant **2.0F**. It's all too easy to forget to correctly type your floating-point constants, so if you compile your program with **double** meaning the same as **float**, you can forget all about adding the '**F**' suffix to your floating point constants.

As an aside, the C99 standard is very strict about the way that floating-point is implemented and the latitude the compiler has to rearrange and manipulate expressions that have floating-point operands. The compiler cannot second-guess user intention and use a number of useful mathematical identities and algebraic simplifications

because in the world of IEC 60559 arithmetic many algebraic identities, such as **x * 1 = x**, do not hold when **x** takes one of the special values NaN, infinity, or negative zero.

More reasons to choose 32-bit doubles

Floating-point constants are not the only silent way that **double** creeps into your program. Consider this:

```
void write_results(float x)
{
  printf("After all that x=%f\n", x);
}
```

Again, no mention of a **double** anywhere, but **double** support routines are now required. The reason is that ISO C requires that **float** arguments are promoted to **double** when they are passed to the non-fixed part of variadic functions such as **printf**. So, even though your application may never mention **double**, **double** arithmetic may be required simply because you use **printf** or one of its near relatives.

If, however, you compile your code with 32-bit doubles, then there is no requirement to promote a **float** to a **double** as they share the same internal format.

Why choose 64-bit doubles?

If your application requires very accurate floating-point, more precise than the seven decimal digits supported by the **float** format, then you have little option but to use **double** arithmetic as there is no simple way to increase the precision of the **float** format. The **double** format delivers approximately 15 decimal digits of precision.

Multithreading

The CrossWorks libraries support multithreading, for example, where you are using CTL or a third-party real-time operating system (RTOS).

Where you have single-threaded processes, there is a single flow of control. However, in multithreaded applications there may be several flows of control which access the same functions, or the same resources, concurrently. To protect the integrity of resources, any code you write for multithreaded applications must be *reentrant* and *thread-safe*.

Reentrancy and thread safety are both related to the way functions in a multithreaded application handle resources.

Reentrant functions

A reentrant function does not hold static data over successive calls and does not return a pointer to static data. For this type of function, the caller provides all the data that the function requires, such as pointers to any workspace. This means that multiple concurrent calls to the function do not interfere with each other, that the function can be called in mainline code, and that the function can be called from an interrupt service routine.

Thread-safe functions

A thread-safe function protects shared resources from concurrent access using locks. In C, local variables are held in processor registers or are on the stack. Any function that does not use static data, or other shared resources, is thread-safe. In general, thread-safe functions are safe to call from any thread but cannot be called directly, or indirectly, from an interrupt service routine.

Thread safety in the CrossWorks library

In the CrossWorks C library:

- some functions are inherently thread-safe, for example **strcmp**.
- some functions, such as **malloc**, are not thread-safe by default but can be made thread-safe by implementing appropriate lock functions.
- other functions are only thread-safe if passed appropriate arguments, for example **tmpnam**.
- some functions are never thread-safe, for example **setlocale**.

We define how the functions in the C library can be made thread-safe if needed. If you use a third-party library in a multi-threaded system and combine it with the CrossWorks C library, you will need to ensure that the third-party library can be made thread-safe in just the same way that the CrossWorks C library can be made thread-safe.

Implementing mutual exclusion in the C library

The CrossWorks C library ships as standard with callouts to functions that provide thread-safety in a multithreaded application. If your application has a single thread of execution, the default implementation of these functions does nothing and your application will run without modification.

If your application is intended for a multithreaded environment and you wish to use the CrossWorks C library, you must implement the following locking functions:

- **__heap_lock** and **__heap_unlock** to provide thread-safety for all heap operations such as **malloc**, **free**, and **realloc**.
- **__printf_lock** and **__printf_unlock** to provide thread-safety for **printf** and relatives.
- **__scanf_lock** and **__scanf_unlock** to provide thread-safety for **scanf** and relatives.
- **__debug_io_lock** and **__debug_io_unlock** to provide thread-safety for semi-hosting support in the CrossStudio I/O function.

If you create a CTL project using the **New Project** wizard, CrossWorks provides implementations of these using CTL event sets. You're free to reimplement them as you see fit.

If you use a third-party RTOS with the CrossWorks C library, you will need to use whatever your RTOS provides for mutual exclusion, typically a semaphore, a mutex, or an event set.

Input and output

The C library provides all the standard C functions for input and output except for the essential items of where to output characters printed to **stdout** and where to read characters from **stdin**.

If you want to output to a UART, to an LCD, or input from a keyboard using the standard library print and scan functions, you need to customize the low-level input and output functions.

Customizing putchar

To use the standard output functions **putchar**, **puts**, and **printf**, you need to customize the way that characters are written to the standard output device. These output functions rely on a function **__putchar** that outputs a character and returns an indication of whether it was successfully written.

The prototype for **__putchar** is

```
int __putchar(int ch);
```

Sending all output to the CrossStudio virtual terminal

You can send all output to the CrossStudio virtual terminal by supplying the following implementation of the**__putchar** function in your code:

```
#include <__cross_studio_io.h>
int __putchar(int ch)
{
  return debug_putchar(ch);
}
```

This hands off output of the character **ch** to the low-level debug output routine, **debug_putchar**.

Whilst this is an adequate implementation of **__putchar**, it does consume stack space for an unnecessary nested call and associated register saving. A better way of achieving the same result is to define the low-level symbol for **__putchar** to be equivalent to the low-level symbol for **debug_putchar**. To do this, we need to instruct the linker to make the symbols equivalent.

To do this using the HCC environment:

- Select the project node in the **Project Explorer**.
- Display the **Properties Window**.
- Enter the text `-D___putchar=_debug_putchar` into the **Additional Options** property of the **Linker Options** group.

Note that there are three leading underscores in **___putchar** and a single leading underscore in **_debug_putchar** because the C compiler automatically prepends an underscore to all global symbols.

To do this using the GCC environment:

- Select the project node in the **Project Explorer**.
- Display the **Properties Window**.
- Enter the text `__putchar=debug_putchar` into the **Linker** > **Linker Symbol Definitions** property of the **Linker Options** group.

Sending all output to another device

If you need to output to a physical device, such as a UART, the following notes will help you:

- If the character cannot be written for any reason, **putchar** *must* return **EOF**. Just because a character can't be written immediately is not a reason to return **EOF**: you can busy-wait or tasking (if applicable) to wait until the character is ready to be written.
- The higher layers of the library do not translate C's end of line character '**n**' before passing it to **putchar**. If you are directing output to a serial line connected to a terminal, for instance, you will most likely need to output a carriage return and line feed when given the character '**n**' (ASCII code 10).

The standard functions that perform input and output are the **printf** and **scanf** functions.These functions convert between internal binary and external printable data. In some cases, though, you need to read and write formatted data on other channels, such as other RS232 ports. This section shows how you can extend the I/O library to best implement these function.

Classic custom printf-style output

Assume that we need to output formatted data to two UARTs, numbered 0 and 1, and we have a functions **uart0_putc** and **uart1_putc** that do just that and whose prototypes are:

```
int uart0_putc(int ch);
int uart1_putc(int ch);
```

These functions return a positive value if there is no error outputting the character and EOF if there was an error.

Using a classic implementation, you would use **sprintf** to format the string for output and then output it:

```
void uart0_printf(const char *fmt, ...)
{
  char buf[80], *p;
  va_list ap;
  va_start(ap, fmt);
  vsnprintf(buf, sizeof(buf), fmt, ap);
  for (p = buf; *p; ++p)
    uart0_putc(*p);
  va_end(ap);
}
```

We would, of course, need an identical routine for outputting to the other UART. This code is portable, but it requires an intermediate buffer of 80 characters. On small systems, this is quite an overhead, so we could reduce the buffer size to compensate. Of course, the trouble with that means that the maximum number of characters that can be output by a single call to **uart0_printf** is also reduced. What would be good is a way to output characters to one of the UARTs without requiring an intermediate buffer.

CrossWorks printf-style output

CrossWorks provides a solution for just this case by using some internal functions and data types in the CrossWorks library. These functions and types are define in the header file <__vfprintf.h>.

The first thing to introduce is the **__printf_t** type:

```
typedef struct
{
  int is_string;
  size_t charcount;
  size_t maxchars;
  union
  {
    char *string;
    int (*output_fn)(int);
   } u;
} __printf_t;
```

This type is used by the library functions to direct what the formatting routines do with each character they need to output. The **is_string** member discriminates the union **u** and directs whether the character should be appended to the string pointed to by **string** or output using **output_fn**. The member **charcount** counts the number of characters currently output, and **maxchars** defines the maximum number of characters output by the formatting routine **__vfprintf**.

We can use this type and function to rewrite **uart0_printf**:

```
int uart0_printf(const char *fmt, ...)
{
  int n;
  va_list ap;
  __printf_t iod;
  va_start(ap, fmt);
  iod.is_string = 0;
  iod.maxchars = INT_MAX;
  iod.u.output_fn = uart0_putc;
  n = __vfprintf(&iod, fmt, ap);
  va_end(ap);
  return n;
}
```

This function has no intermediate buffer: when a character is ready to be output by the formatting routine, it calls the **output_fn** function in the descriptor **iod** to output it immediately. The maximum number of characters isn't limited as the **maxchars** member is set to **INT_MAX**. if you wanted to limit the number of characters output you can simply set the **maxchars** member to the appropriate value before calling **__vfprintf**.

We can adapt this function to take a UART number as a parameter:

```
int uart_printf(int uart, const char *fmt, ...)
{
  int n;
  va_list ap;
  __printf_t iod;
  va_start(ap, fmt);
  iod.is_string = 0;
  iod.maxchars = INT_MAX;
  iod.u.output_fn = uart ? uart1_putc : uart0_putc;
  n = __vfprintf(&iod, fmt, ap);
  va_end(ap);
  return n;
}
```

Now we can use:

```
uart_printf(0, "This is uart %d
...", 0);
uart_printf(1, "..and this is uart %d
", 1);
```

__vfprintf returns the actual number of characters printed, which you may wish to dispense with and make the **uart_printf** routine return **void**.

Extending input functions

The formatted input functions would be implemented in the same manner as the output functions: read a string into an intermediate buffer and parse using **sscanf**. However, we can use the low-level routines in the CrossWorks library for formatted input without requiring the intermediate buffer.

The type **__stream_scanf_t** is:

```
typedef struct
{
  char is_string;
  int (*getc_fn)(void);
  int (*ungetc_fn)(int);
} __stream_scanf_t;
```

The function **getc_fn** reads a single character from the UART, and **ungetc_fn** pushes back a character to the UART. You can push at most one character back onto the stream.

Here's an implementation of functions to read and write from a single UART:

```
static int uart0_ungot = EOF;
int uart0_getc(void)
{
  if (uart0_ungot)
    {
      int c = uart0_ungot;
      uart0_ungot = EOF;
      return c;
    }
  else
    return read_char_from_uart(0);
}

int uart0_ungetc{int c)
{
  uart0_ungot = c;
}
```

You can use these two functions to perform formatted input using the UART:

```
int uart0_scanf(const char *fmt, ...)
{
  __stream_scanf_t iod;
  va_list a;
  int n;
  va_start(a, fmt);
  iod.is_string = 0;
  iod.getc_fn = uart0_getc;
  iod.ungetc_fn = uart0_ungetc;
  n = __vfscanf((__scanf_t *)&iod, (const unsigned char *)fmt, a);
  va_end(a);
  return n;
}
```

Using this template, we can add functions to do additional formatted input from other UARTs or devices, just as we did for formatted output.

Locales

The CrossWorks C library supports wide characters, multi-byte characters and locales. However, as not all programs require full localization, you can tailor the exact support provided by the CrossWorks C library to suit your application. These sections describe how to add new locales to your application and customize the runtime footprint of the C library.

Unicode, ISO 10646, and wide characters

The ISO standard 10646 is identical to the published Unicode standard and the CrossWorks C library uses the Unicode 6.0 definition as a base. Hence, whenever you see the term 'Unicode' in this document, it is equivalent to Unicode 6.0 and ISO/IEC 10646:2011.

The CrossWorks C library is compiled using a 32-bit wide character. The types **wchar_t** and **wint_t** are 32-bit integers that can hold any Unicode code point. Unicode only defines the code points 0 through 0x10FFFF with all others being undefined at this time.

The wide character type will hold Unicode code points in a locale that is defined to use Unicode and character type functions such as **wisalpha** will work correctly on all Unicode code points.

Multi-byte characters

CrossWorks supports multi-byte encoding and decoding of characters. Most new software on the desktop uses Unicode internally and UTF-8 as the external, on-disk encoding for files and for transport over 8-bit mediums such as network connections.

However, in embedded software there is still a case to use code pages, such as ISO-Latin1, to reduce the footprint of an application whilst also providing extra characters that do not form part of the ASCII character set.

The CrossWorks C library can support both models and you can choose a combination of models, dependent upon locale, or construct a custom locale.

The standard C and POSIX locales

The standard C locale is called simply ‘C’. In order to provide POSIX compatibility, the name ‘POSIX’ is a synonym for ‘C’.

The C locale is fixed and supports only the ASCII character set with character codes 0 through 127. There is no multi-byte character support, so the character encoding between wide and narrow characters is simply one-to-one: a narrow character is converted to a wide character by zero extension. Thus, ASCII encoding of narrow characters is compatible with the ISO 10646 (Unicode) encoding of wide characters in this locale.

Additional locales in source form

The CrossWorks C library provides only the 'C' locale; if you need other locales, you must provide those by linking them into your application. We have constructed a number of locales from the Unicode Common Locale Data Repository (CLDR) and provided them in source form for you to include in your application.

A C library locale is divided into two parts:

- the locale's date, time, numeric, and monetary formatting information
- how to convert between multi-byte characters and wide characters by the functions in the C library.

The first, the locale data, is independent of how characters are represented. The second, the code set in use, defines how to map between narrow, multi-byte, and wide characters.

The locales we provide define the Hungarian locale as 'HU_hu'. However, you may want to use either a UTF-8 encoding or ISO-Latin2 encoding for multi-byte characters, and these encodings are known by the names 'utf8' and 'iso-8859-2'.

You combine the two encodings using a period, so the C locale 'HU_hu.utf8' denotes a Hungarian locale that encodes multi-byte characters as UTF-8 and 'HU_hu.iso-8859-2' denotes a Hungarian locale that encodes multi-byte characters using the ISO 8859-2 code page.

Installing a locale

If the locale you request using **setlocale** is neither 'C' nor 'POSIX', the C library calls the function **__user_find_locale** to find a user-supplied locale. The standard implementation of this function is to return a null pointer which indicates that no additional locales are installed and, hence, no locale matches the request.

The prototype for **__user_find_locale** is:

```
const __RAL_locale_t *__user_find_locale(const char *locale);
```

The parameter **locale** is the locale to find and includes the code set. The locale name is identical to the name passed to **setlocale** when you select a locale.

Now let's install the Hungarian locale using both UTF-8 and ISO 8859-2 encodings. The UTF-8 codecs are included in the CrossWorks C library, but the Hungarian locale and the ISO 8859-2 codec are not.

You will find the file **locale_hu_HU.c** in the CrossWorks **src** directory along with the file **codeset_iso_8859_2.c**. Add these two files to your project.

Although this adds the data needed for the locales, it does not make these available for the C library: we need to write some code for **__user_find_locale** to return the appropriate locales.

To create the locales, we need to add the following code and data to tie everything together:

```
#include <__crossworks.h>
static const __RAL_locale_t hu_HU_utf8 = {
  &locale_hu_HU,
  &codeset_utf8
};
static const __RAL_locale_t hu_HU_iso_8859_2 = {
  &locale_hu_HU,
  &codeset_iso_8859_2
};
const __RAL_locale_t *
__user_find_locale(const char *locale)
{
  if (strncmp(locale, "hu_HU.utf8") == 0)
    return &hu_HU_utf8;
  else if (strncmp(locale, "hu_HU.iso_8859_2") == 0)
    return &hu_HU_iso_8859_2;
  else
    return 0;
}
```

Complete API reference

This section contains a complete reference to the CrossWorks C library API.

<assert.h>

Describes the diagnostic facilities which you can build into your application.

<cross_studio_io.h>

Describes the virtual console services and semi-hosting support that CrossStudio provides to help you when developing your applications.

<ctype.h>

Describes the character classification and manipulation functions.

<errno.h>

Describes the macros and error values returned by the C library.

<float.h>

Defines macros that expand to various limits and parameters of the standard floating point types.

<intrinsics.h>

Describes ARM-specific intrinsic functions.

<libarm.h>

Describes ARM-specific library functions.

<limits.h>

Describes the macros that define the extreme values of underlying C types.

<locale.h>

Describes support for localization specific settings.

<math.h>

Describes the mathematical functions provided by the C library.

<setjmp.h>

Describes the non-local goto capabilities of the C library.

<stdarg.h>

Describes the way in which variable parameter lists are accessed.

<stddef.h>

Describes standard type definitions.

<stdio.h>

Describes the formatted input and output functions.

<stdlib.h>

Describes the general utility functions provided by the C library.

<string.h>

Describes the string handling functions provided by the C library.

<time.h>

Describes the functions to get and manipulate date and time information provided by the C library.

<wchar.h>

Describes the facilities you can use to manipulate wide characters.

<assert.h>

API Summary

Macros	
assert	Allows you to place assertions and diagnostic tests into programs
Functions	
__assert	User defined behaviour for the assert macro

__assert

Synopsis

```
void __assert(const char *expression,
              const char *filename,
              int line);
```

Description

There is no default implementation of **__assert**. Keeping **__assert** out of the library means that you can can customize its behaviour without rebuilding the library. You must implement this function where **expression** is the stringized expression, **filename** is the filename of the source file and **line** is the linenumber of the failed assertion.

assert

Synopsis

```
#define assert(e) ..
```

Description

If **NDEBUG** is defined as a macro name at the point in the source file where <**assert.h**> is included, the **assert** macro is defined as:

```
#define assert(ignore) ((void)0)
```

If **NDEBUG** is not defined as a macro name at the point in the source file where <**assert.h**> is included, the **assert** macro expands to a **void** expression that calls **__assert**.

```
#define assert(e) ((e) ? (void)0 : __assert(#e, __FILE__, __LINE__))
```

When such an **assert** is executed and **e** is false, **assert** calls the **__assert** function with information about the particular call that failed: the text of the argument, the name of the source file, and the source line number. These are the stringized expression and the values of the preprocessing macros **__FILE__** and **__LINE__**.

Note

The **assert** macro is redefined according to the current state of **NDEBUG** each time that <**assert.h**> is included.

<cross_studio_io.h>

API Summary

File Functions	
debug_clearerr	Clear error indicator
debug_fclose	Closes an open stream
debug_feof	Check end of file condition
debug_ferror	Check error indicator
debug_fflush	Flushes buffered output
debug_fgetc	Read a character from a stream
debug_fgetpos	Return file position
debug_fgets	Read a string
debug_filesize	Return the size of a file
debug_fopen	Opens a file on the host PC
debug_fprintf	Formatted write
debug_fprintf_c	Formatted write
debug_fputc	Write a character
debug_fputs	Write a string
debug_fread	Read data
debug_freopen	Reopens a file on the host PC
debug_fscanf	Formatted read
debug_fscanf_c	Formatted read
debug_fseek	Set file position
debug_fsetpos	Teturn file position
debug_ftell	Return file position
debug_fwrite	Write data
debug_remove	Deletes a file on the host PC
debug_rename	Renames a file on the host PC
debug_rewind	Set file position to the beginning
debug_tmpfile	Open a temporary file
debug_tmpnam	Generate temporary filename
debug_ungetc	Push a character
debug_vfprintf	Formatted write
debug_vfscanf	Formatted read

Debug Terminal Output Functions	
debug_printf	Formatted write
debug_printf_c	Formatted write
debug_putchar	Write a character
debug_puts	Write a string
debug_vprintf	Formatted write
Debug Terminal Input Functions	
debug_getch	Blocking character read
debug_getchar	Line-buffered character read
debug_getd	Line-buffered double read
debug_getf	Line-buffered float read
debug_geti	Line-buffered integer read
debug_getl	Line-buffered long read
debug_getll	Line-buffered long long read
debug_gets	String read
debug_getu	Line-buffered unsigned integer
debug_getul	Line-buffered unsigned long read
debug_getull	Line-buffered unsigned long long read
debug_kbhit	Polled character read
debug_scanf	Formatted read
debug_scanf_c	Formatted read
debug_vscanf	Formatted read
Debugger Functions	
debug_abort	Stop debugging
debug_break	Stop target
debug_enabled	Test if debug input/output is enabled
debug_exit	Stop debugging
debug_getargs	Get arguments
debug_loadsymbols	Load debugging symbols
debug_runtime_error	Stop and report error
debug_unloadsymbols	Unload debugging symbols
Misc Functions	
debug_getenv	Get environment variable value
debug_perror	Display error
debug_system	Execute command

debug_time	get time

debug_abort

Synopsis

```
void debug_abort(void);
```

Description

debug_abort causes the debugger to exit and a failure result is returned to the user.

debug_break

Synopsis

```
void debug_break(void);
```

Description

debug_break causes the debugger to stop the target and position the cursor at the line that called debug_break.

debug_clearerr

Synopsis

```
void debug_clearerr(DEBUG_FILE *stream);
```

Description

debug_clearerr clears any error indicator or end of file condition for the **stream**.

debug_enabled

Synopsis

```
int debug_enabled(void);
```

Description

debug_enabled returns non-zero if the debugger is connected - you can use this to test if a debug input/output functions will work.

debug_exit

Synopsis

```
void debug_exit(int result);
```

Description

debug_exit causes the debugger to exit and **result** is returned to the user.

debug_fclose

Synopsis

```
int debug_fclose(DEBUG_FILE *stream);
```

Description

debug_fclose flushes any buffered output of the **stream** and then closes the stream.

debug_fclose returns 0 on success or -1 if there was an error.

debug_feof

Synopsis

```
int debug_feof(DEBUG_FILE *stream);
```

Description

debug_feof returns non-zero if the end of file condition is set for the **stream**.

debug_ferror

Synopsis

```
int debug_ferror(DEBUG_FILE *stream);
```

Description

debug_ferror returns non-zero if the error indicator is set for the **stream**.

debug_fflush

Synopsis

```
int debug_fflush(DEBUG_FILE *stream);
```

Description

debug_fflush flushes any buffered output of the **stream**.

debug_fflush returns 0 on success or -1 if there was an error.

debug_fgetc

Synopsis

```
int debug_fgetc(DEBUG_FILE *stream);
```

Description

debug_fgetc reads and returns the next character on **stream** or -1 if no character is available.

debug_fgetpos

Synopsis

```
int debug_fgetpos(DEBUG_FILE *stream,
                  long *pos);
```

Description

debug_fgetpos is equivalent to **debug_fseek** .

debug_fgets

Synopsis

```
char *debug_fgets(char *s,
                  int n,
                  DEBUG_FILE *stream);
```

Description

debug_fgets reads at most **n**-1 characters or the characters up to (and including) a newline from the input **stream** into the array pointed to by **s**. A null character is written to the array after the input characters.

debug_fgets returns **s** on success, or 0 on error or end of file.

debug_filesize

Synopsis

```
int debug_filesize(DEBUG_FILE *stream);
```

Description

debug_filesize returns the size of the file associated with the **stream** in bytes.

debug_filesize returns -1 on error.

debug_fopen

Synopsis

```
DEBUG_FILE *debug_fopen(const char *filename,
                        const char *mode);
```

Description

debug_fopen opens the **filename** on the host PC and returns a stream or **0** if the open fails. The **filename** is a host PC filename which is opened relative to the debugger working directory. The **mode** is a string containing one of:

- **r** open file for reading.
- **w** create file for writing.
- **a** open or create file for writing and position at the end of the file.
- **r**+ open file for reading and writing.
- **w**+ create file for reading and writing.
- **a**+ open or create text file for reading and writing and position at the end of the file.

followed by one of:

- **t** for a text file.
- **b** for a binary file.

debug_fopen returns a stream that can be used to access the file or **0** if the open fails.

debug_fprintf

Synopsis

```
int debug_fprintf(DEBUG_FILE *stream,
                  const char *format,
                  ...);
```

Description

debug_fprintf writes to **stream**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. The **format** string is a standard C printf format string. The actual formatting is performed on the host by the debugger and therefore **debug_fprintf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_fprintf returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

debug_fprintf_c

Synopsis

```
int debug_fprintf_c(DEBUG_FILE *stream,
                    __code const char *format,
                    ...);
```

Description

debug_fprintf_c is equivalent to **debug_fprintf** with the format string in code memory.

debug_fputc

Synopsis

```
int debug_fputc(int c,
                DEBUG_FILE *stream);
```

Description

debug_fputc writes the character **c** to the output **stream**.

debug_fputc returns the character written or -1 if an error occurred.

debug_fputs

Synopsis

```
int debug_fputs(const char *s,
                DEBUG_FILE *stream);
```

Description

debug_fputs writes the string pointed to by **s** to the output **stream** and appends a new-line character. The terminating null character is not written.

debug_fputs returns -1 if a write error occurs; otherwise it returns a nonnegative value.

debug_fread

Synopsis

```
int debug_fread(void *ptr,
                int size,
                int nobj,
                DEBUG_FILE *stream);
```

Description

debug_fread reads from the input **stream** into the array **ptr** at most **nobj** objects of size **size**.

debug_fread returns the number of objects read. If this number is different from **nobj** then **debug_feof** and **debug_ferror** can be used to determine status.

debug_freopen

Synopsis

```
DEBUG_FILE *debug_freopen(const char *filename,
                          const char *mode,
                          DEBUG_FILE *stream);
```

Description

debug_freopen is the same as **debug_open** except the file associated with the **stream** is closed and the opened file is then associated with the **stream**.

debug_fscanf

Synopsis

```
int debug_fscanf(DEBUG_FILE *stream,
                 const char *format,
                 ...);
```

Description

debug_fscanf reads from the input **stream**, under control of the string pointed to by **format**, that specifies how subsequent arguments are converted for input. The **format** string is a standard C scanf format string. The actual formatting is performed on the host by the debugger and therefore **debug_fscanf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_fscanf returns number of characters read, or a negative value if an output or encoding error occurred.

debug_fscanf_c

Synopsis

```
int debug_fscanf_c(DEBUG_FILE *stream,
                   __code const char *format,
                   ...);
```

Description

debug_fscanf_c is equivalent to **debug_fscanf** with the format string in code memory.

debug_fseek

Synopsis

```
int debug_fseek(DEBUG_FILE *stream,
                long offset,
                int origin);
```

Description

debug_fseek sets the file position for the **stream**. A subsequent read or write will access data at that position. The **origin** can be one of:

- **0** sets the position to **offset** bytes from the beginning of the file.
- **1** sets the position to **offset** bytes relative to the current position.
- **2** sets the position to **offset** bytes from the end of the file.

Note that for text files **offset** must be zero. **debug_fseek** returns zero on success, non-zero on error.

debug_fsetpos

Synopsis

```
int debug_fsetpos(DEBUG_FILE *stream,
                  const long *pos);
```

Description

debug_fsetpos is equivalent to **debug_fseek** with 0 as the **origin**.

debug_ftell

Synopsis

```
long debug_ftell(DEBUG_FILE *stream);
```

Description

debug_ftell returns the current file position of the **stream**.

debug_ftell returns -1 on error.

debug_fwrite

Synopsis

```
int debug_fwrite(void *ptr,
                 int size,
                 int nobj,
                 DEBUG_FILE *stream);
```

Description

debug_fwrite write to the output **stream** from the array **ptr** at most **nobj** objects of size **size**.

debug_fwrite returns the number of objects written. If this number is different from **nobj** then **debug_feof** and **debug_ferror** can be used to determine status.

debug_getargs

Synopsis

```
int debug_getargs(unsigned bufsize,
                  unsigned char *buf);
```

Description

debug_getargs stores the debugger command line arguments into the memory pointed at by **buf** up to a maximum of **bufsize** bytes. The command line is stored as a C **argc** array of null terminated string and the number of entries is returned as the result.

debug_getch

Synopsis

```
int debug_getch(void);
```

Description

debug_getch reads one character from the Debug Terminal. This function will block until a character is available.

debug_getchar

Synopsis

```
int debug_getchar(void);
```

Description

debug_getchar reads one character from the **Debug Terminal**. This function uses line input and will therefore block until characters are available and ENTER has been pressed.

debug_getchar returns the character that has been read.

debug_getd

Synopsis

```
int debug_getd(double *);
```

Description

debug_getd reads a double from the **Debug Terminal**. The number is written to the double object pointed to by **d**.

debug_getd returns zero on success or -1 on error.

debug_getenv

Synopsis

```
char *debug_getenv(char *name);
```

Description

debug_getenv returns the value of the environment variable **name** or 0 if the environment variable cannot be found.

debug_getf

Synopsis

```
int debug_getf(float *f);
```

Description

debug_getf reads an float from the **Debug Terminal**. The number is written to the float object pointed to by **f**.

debug_getf returns zero on success or -1 on error.

debug_geti

Synopsis

```
int debug_geti(int *i);
```

Description

debug_geti reads an integer from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with **0b** it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the integer object pointed to by **i**.

debug_geti returns zero on success or -1 on error.

debug_getl

Synopsis

```
int debug_getl(long *l);
```

Description

debug_getl reads a long from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the long object pointed to by **l**.

debug_getl returns zero on success or -1 on error.

debug_getll

Synopsis

```
int debug_getll(long long *ll);
```

Description

debug_getll reads a long long from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with **0b** it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the long long object pointed to by **ll**.

debug_getll returns zero on success or -1 on error.

debug_gets

Synopsis

```
char *debug_gets(char *s);
```

Description

debug_gets reads a string from the Debug Terminal in memory pointed at by **s**. This function will block until ENTER has been pressed.

debug_gets returns the value of **s**.

debug_getu

Synopsis

```
int debug_getu(unsigned *u);
```

Description

debug_getu reads an unsigned integer from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with **0b** it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the unsigned integer object pointed to by **u**.

debug_getu returns zero on success or -1 on error.

debug_getul

Synopsis

```
int debug_getul(unsigned long *ul);
```

Description

debug_getul reads an unsigned long from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with **0b** it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the long object pointed to by **ul**.

debug_getul returns zero on success or -1 on error.

debug_getull

Synopsis

```
int debug_getull(unsigned long long *ull);
```

Description

debug_getull reads an unsigned long long from the **Debug Terminal**. If the number starts with **0x** it is interpreted as a hexadecimal number, if it starts with **0** it is interpreted as an octal number, if it starts with **0b** it is interpreted as a binary number, otherwise it is interpreted as a decimal number. The number is written to the long long object pointed to by **ull**.

debug_getull returns zero on success or -1 on error.

debug_kbhit

Synopsis

```
int debug_kbhit(void);
```

Description

debug_kbhit polls the Debug Terminal for a character and returns a non-zero value if a character is available or 0 if not.

debug_loadsymbols

Synopsis

```
void debug_loadsymbols(const char *filename,
                       const void *address,
                       const char *breaksymbol);
```

Description

debug_loadsymbols instructs the debugger to load the debugging symbols in the file denoted by **filename**. The **filename** is a (macro expanded) host PC filename which is relative to the debugger working directory. The **address** is the load address which is required for debugging position independent executables, supply **NULL** for regular executables. The **breaksymbol** is the name of a symbol in the filename to set a temporary breakpoint on or **NULL**.

debug_perror

Synopsis

```
void debug_perror(const char *s);
```

Description

debug_perror displays the optional string **s** on the **Debug Terminal** together with a string corresponding to the errno value of the last Debug IO operation.

debug_printf

Synopsis

```
int debug_printf(const char *format,
                 ...);
```

Description

debug_printf writes to the **Debug Terminal**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. The **format** string is a standard C printf format string. The actual formatting is performed on the host by the debugger and therefore **debug_printf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_printf returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

debug_printf_c

Synopsis

```
int debug_printf_c(__code const char *format,
                   ...);
```

Description

debug_printf_c is equivalent to **debug_printf** with the format string in code memory.

debug_putchar

Synopsis

```
int debug_putchar(int c);
```

Description

debug_putchar write the character **c** to the Debug Terminal.

debug_putchar returns the character written or -1 if a write error occurs.

debug_puts

Synopsis

```
int debug_puts(const char *);
```

Description

debug_puts writes the string **s** to the Debug Terminal followed by a new-line character.

debug_puts returns -1 if a write error occurs, otherwise it returns a nonnegative value.

debug_remove

Synopsis

```
int debug_remove(const char *filename);
```

Description

debug_remove removes the filename denoted by **filename** and returns **0** on success or **-1** on error. The **filename** is a host PC filename which is relative to the debugger working directory.

debug_rename

Synopsis

```
int debug_rename(const char *oldfilename,
                 const char *newfilename);
```

Description

debug_rename renames the file denoted by **oldpath** to **newpath** and returns zero on success or non-zero on error. The **oldpath** and **newpath** are host PC filenames which are relative to the debugger working directory.

debug_rewind

Synopsis

```
void debug_rewind(DEBUG_FILE *stream);
```

Description

debug_rewind sets the current file position of the **stream** to the beginning of the file and clears any error and end of file conditions.

debug_runtime_error

Synopsis

```
void debug_runtime_error(const char *error);
```

Description

debug_runtime_error causes the debugger to stop the target, position the cursor at the line that called debug_runtime_error, and display the null-terminated string pointed to by **error**.

debug_scanf

Synopsis

```
int debug_scanf(const char *format,
                ...);
```

Description

debug_scanf reads from the **Debug Terminal**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for input. The **format** string is a standard C scanf format string. The actual formatting is performed on the host by the debugger and therefore **debug_scanf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_scanf returns number of characters read, or a negative value if an output or encoding error occurred.

debug_scanf_c

Synopsis

```
int debug_scanf_c(__code const char *format,
                  ...);
```

Description

debug_scanf_c is equivalent to **debug_scanf** with the format string in code memory.

debug_system

Synopsis

```
int debug_system(char *command);
```

Description

debug_system executes the **command** with the host command line interpreter and returns the commands exit status.

debug_time

Synopsis

```
unsigned long debug_time(unsigned long *ptr);
```

Description

debug_time returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC), according to the system clock of the host computer. The return value is stored in ***ptr** if **ptr** is not NULL.

debug_tmpfile

Synopsis

```
DEBUG_FILE *debug_tmpfile(void);
```

Description

debug_tmpfile creates a temporary file on the host PC which is deleted when the stream is closed.

debug_tmpnam

Synopsis

```
char *debug_tmpnam(char *str);
```

Description

debug_tmpnam returns a unique temporary filename. If **str** is **NULL** then a static buffer is used to store the filename, otherwise the filename is stored in **str**. On success a pointer to the string is returned, on failure **0** is returned.

debug_ungetc

Synopsis

```
int debug_ungetc(int c,
                 DEBUG_FILE *stream);
```

Description

debug_ungetc pushes the character **c** onto the input **stream**. If successful **c** is returned, otherwise -1 is returned.

debug_unloadsymbols

Synopsis

```
void debug_unloadsymbols(const char *filename);
```

Description

debug_unloadsymbols instructs the debugger to unload the debugging symbols (previously loaded by a call to **debug_loadsymbols**) in the file denoted by **filename**. The **filename** is a host PC filename which is relative to the debugger working directory.

debug_vfprintf

Synopsis

```
int debug_vfprintf(DEBUG_FILE *stream,
                   const char *format,
                   __va_list);
```

Description

debug_vfprintf is equivalent to **debug_fprintf** with arguments passed using **stdarg**.h rather than a variable number of arguments.

debug_vfscanf

Synopsis

```
int debug_vfscanf(DEBUG_FILE *stream,
                  const char *format,
                  __va_list);
```

Description

debug_vfscanf is equivalent to **debug_fscanf** with arguments passed using **stdarg**.h rather than a variable number of arguments.

debug_vprintf

Synopsis

```
int debug_vprintf(const char *format,
                  __va_list);
```

Description

debug_vprintf is equivalent to **debug_printf** with arguments passed using **stdarg**.h rather than a variable number of arguments.

debug_vscanf

Synopsis

```
int debug_vscanf(const char *format,
                 __va_list);
```

Description

debug_vscanf is equivalent to **debug_scanf** with arguments passed using **stdarg**.h rather than a variable number of arguments.

<ctype.h>

API Summary

Classification functions	
isalnum	Is character alphanumeric?
isalpha	Is character alphabetic?
isblank	Is character a space or horizontal tab?
iscntrl	Is character a control character?
isdigit	Is character a decimal digit?
isgraph	Is character any printing character except space?
islower	Is character a lowercase letter?
isprint	Is character printable?
ispunct	Is character a punctuation mark?
isspace	Is character a whitespace character?
isupper	Is character an uppercase letter?
isxdigit	Is character a hexadecimal digit?
Conversion functions	
tolower	Convert uppercase character to lowercase
toupper	Convert lowercase character to uppercase

isalnum

Synopsis

```
int isalnum(int c);
```

Description

isalnum returns nonzero (true) if and only if the value of the argument **c** is an alphabetic or numeric character.

isalpha

Synopsis

```
int isalpha(int c);
```

Description

isalpha returns nonzero (true) if and only if **isupper** or **islower** return true for value of the argument **c**.

isblank

Synopsis

```
int isblank(int c);
```

Description

isblank returns nonzero (true) if and only if the value of the argument **c** is either a space character (`' '`) or the horizontal tab character (`'\\t'`).

iscntrl

Synopsis

```
int iscntrl(int c);
```

Description

iscntrl returns nonzero (true) if and only if the value of the argument **c** is a control character. Control characters have values 0 through 31 and the single value 127.

isdigit

Synopsis

```
int isdigit(int c);
```

Description

isdigit returns nonzero (true) if and only if the value of the argument **c** is a digit.

isgraph

Synopsis

```
int isgraph(int c);
```

Description

isgraph returns nonzero (true) if and only if the value of the argument **c** is any printing character except space (' ').

islower

Synopsis

```
int islower(int c);
```

Description

islower returns nonzero (true) if and only if the value of the argument **c** is an lowercase letter.

isprint

Synopsis

```
int isprint(int c);
```

Description

isprint returns nonzero (true) if and only if the value of the argument **c** is any printing character including space (' ').

ispunct

Synopsis

```
int ispunct(int c);
```

Description

ispunct returns nonzero (true) for every printing character for which neither **isspace** nor **isalnum** is true.

isspace

Synopsis

```
int isspace(int c);
```

Description

isspace returns nonzero (true) if and only if the value of the argument **c** is a standard white-space character. The standard white-space characters are space (' '), form feed ('\\f'), new-line ('\\n'), carriage return ('\\r'), horizontal tab ('\\t'), and vertical tab ('\v').

isupper

Synopsis

```
int isupper(int c);
```

Description

isupper returns nonzero (true) if and only if the value of the argument **c** is an uppercase letter.

isxdigit

Synopsis

```
int isxdigit(int c);
```

Description

isxdigit returns nonzero (true) if and only if the value of the argument **c** is a hexadecimal digit.

tolower

Synopsis

```
int tolower(int c);
```

Description

tolower converts an uppercase letter to a corresponding lowercase letter. If the argument **c** is a character for which **isupper** is true, **tolower** returns the corresponding lowercase letter; otherwise, the argument is returned unchanged.

toupper

Synopsis

```
int toupper(int c);
```

Description

toupper converts a lowercase letter to a corresponding uppercase letter. If the argument **c** is a character for which **islower** is true, **toupper** returns the corresponding uppercase letter; otherwise, the argument is returned unchanged.

<errno.h>

API Summary

Macros	
errno	Allows you to access the errno implementation
Error numbers	
EDOM	Domain error
EILSEQ	Illegal byte sequence
ERANGE	Result too large or too small
Functions	
__errno	User-defined behavior for the errno macro

EDOM

Synopsis

```
#define EDOM     0x01
```

Description

EDOM - an input argument is outside the defined domain of the mathematical function.

EILSEQ

Synopsis

```
#define EILSEQ   0x02
```

Description

EILSEQ - A wide-character code has been detected that does not correspond to a valid character, or a byte sequence does not form a valid wide-character code.

ERANGE

Synopsis

```
#define ERANGE   0x03
```

Description

ERANGE - the result of the function is too large (overflow) or too small (underflow) to be represented in the available space.

__errno

Synopsis

```
int *__errno(void);
```

Description

There is no default implementation of **__errno**. Keeping **__errno** out of the library means that you can can customize its behavior without rebuilding the library. A default implementation could be

```
static int errno;

int *_errno(void) { return &errno; }
```

errno

Synopsis

```
#define errno (*__errno())
```

Description

errno macro expands to a function call to **__errno** that returns a pointer to an **int**. This function can be implemented by the application to provide a thread specific errno.

The value of **errno** is zero at program startup, but is never set to zero by any library function. The value of **errno** may be set to a nonzero value by a library function, and this effect is documented in each function that does so.

Note

The ISO standard does not specify whether **errno** is a macro or an identifier declared with external linkage. Portable programs must not make assumptions about the implementation of **errno**.

<float.h>

API Summary

Double exponent minimum and maximum values	
DBL_MAX_10_EXP	The maximum exponent value in base 10 of a double
DBL_MAX_EXP	The maximum exponent value of a double
DBL_MIN_10_EXP	The minimal exponent value in base 10 of a double
DBL_MIN_EXP	The minimal exponent value of a double
Implementation	
DBL_DIG	The number of digits of precision of a double
DBL_MANT_DIG	The number of digits in a double
DECIMAL_DIG	The number of decimal digits that can be rounded without change
FLT_DIG	The number of digits of precision of a float
FLT_EVAL_METHOD	The evaluation format
FLT_MANT_DIG	The number of digits in a float
FLT_RADIX	The radix of the exponent representation
FLT_ROUNDS	The rounding mode
Float exponent minimum and maximum values	
FLT_MAX_10_EXP	The maximum exponent value in base 10 of a float
FLT_MAX_EXP	The maximum exponent value of a float
FLT_MIN_10_EXP	The minimal exponent value in base 10 of a float
FLT_MIN_EXP	The minimal exponent value of a float
Double minimum and maximum values	
DBL_EPSILON	The difference between 1 and the least value greater than 1 of a double
DBL_MAX	The maximum value of a double
DBL_MIN	The minimal value of a double
Float minimum and maximum values	
FLT_EPSILON	The difference between 1 and the least value greater than 1 of a float
FLT_MAX	The maximum value of a float
FLT_MIN	The minimal value of a float

DBL_DIG

Synopsis

```
#define DBL_DIG                         15
```

Description

DBL_DIG specifies The number of digits of precision of a **double**.

DBL_EPSILON

Synopsis

```
#define DBL_EPSILON 2.2204460492503131E-16
```

Description

DBL_EPSILON the minimum positive number such that 1.0 + DBL_EPSILON != 1.0.

DBL_MANT_DIG

Synopsis

```
#define DBL_MANT_DIG                    53
```

Description

DBL_MANT_DIG specifies the number of base FLT_RADIX digits in the mantissa part of a **double**.

DBL_MAX

Synopsis

```
#define DBL_MAX    1.7976931348623157E+308
```

Description

DBL_MAX is the maximum value of a **double**.

DBL_MAX_10_EXP

Synopsis

```
#define DBL_MAX_10_EXP                +308
```

Description

DBL_MAX_10_EXP is the maximum value in base 10 of the exponent part of a **double**.

DBL_MAX_EXP

Synopsis

```
#define DBL_MAX_EXP                 +1024
```

Description

DBL_MAX_EXP is the maximum value of base **FLT_RADIX** in the exponent part of a **double**.

DBL_MIN

Synopsis

```
#define DBL_MIN    2.2250738585072014E-308
```

Description

DBL_MIN is the minimum value of a **double**.

DBL_MIN_10_EXP

Synopsis

```
#define DBL_MIN_10_EXP               -307
```

Description

DBL_MIN_10_EXP is the minimum value in base 10 of the exponent part of a **double**.

DBL_MIN_EXP

Synopsis

```
#define DBL_MIN_EXP                 -1021
```

Description

DBL_MIN_EXP is the minimum value of base **FLT_RADIX** in the exponent part of a **double**.

DECIMAL_DIG

Synopsis

```
#define DECIMAL_DIG             17
```

Description

DECIMAL_DIG specifies the number of decimal digits that can be rounded to a floating-point number without change to the value.

FLT_DIG

Synopsis

```
#define FLT_DIG                  6
```

Description

FLT_DIG specifies The number of digits of precision of a **float**.

FLT_EPSILON

Synopsis

```
#define FLT_EPSILON 1.19209290E-07F // decimal constant
```

Description

FLT_EPSILON the minimum positive number such that 1.0 + FLT_EPSILON != 1.0.

FLT_EVAL_METHOD

Synopsis

```
#define FLT_EVAL_METHOD 0
```

Description

FLT_EVAL_METHOD specifies that all operations and constants are evaluated to the range and precision of the type.

FLT_MANT_DIG

Synopsis

```
#define FLT_MANT_DIG            24
```

Description

FLT_MANT_DIG specifies the number of base FLT_RADIX digits in the mantissa part of a **float**.

FLT_MAX

Synopsis

```
#define FLT_MAX     3.40282347E+38F
```

Description

FLT_MAX is the maximum value of a **float**.

FLT_MAX_10_EXP

Synopsis

```
#define FLT_MAX_10_EXP          +38
```

Description

FLT_MAX_10_EXP is the maximum value in base 10 of the exponent part of a **float**.

FLT_MAX_EXP

Synopsis

```
#define FLT_MAX_EXP             +128
```

Description

FLT_MAX_EXP is the maximum value of base **FLT_RADIX** in the exponent part of a **float**.

FLT_MIN

Synopsis

```
#define FLT_MIN     1.17549435E-38F
```

Description

FLT_MIN is the minimum value of a **float**.

FLT_MIN_10_EXP

Synopsis

```
#define FLT_MIN_10_EXP          -37
```

Description

FLT_MIN_10_EXP is the minimum value in base 10 of the exponent part of a **float**.

FLT_MIN_EXP

Synopsis

```
#define FLT_MIN_EXP           -125
```

Description

FLT_MIN_EXP is the minimum value of base **FLT_RADIX** in the exponent part of a **float**.

FLT_RADIX

Synopsis

```
#define FLT_RADIX                2
```

Description

FLT_RADIX specifies the radix of the exponent representation.

FLT_ROUNDS

Synopsis

```
#define FLT_ROUNDS 1
```

Description

FLT_ROUNDS specifies the rounding mode of floating-point addition is round to nearest.

<intrinsics.h>

API Summary

Misc Intrinsics	
__breakpoint	BKPT instruction
__clrex	CLREX instruction
__clz	CLZ instruction
__dbg	DBG instruction
__dmb	DMB instruction
__dsb	DSB instruction
__isb	ISB instruction
__nop	NOP instruction
__pld	PLD instruction
__pli	PLI instruction
__sev	SEV instruction
__swp	SWP instruction
__swpb	SWPB instruction
__wfe	WFE instruction
__wfi	WFI instruction
__yield	YIELD instruction
Coprocessor Intrinsics	
__cdp	CDP instruction
__cdp2	CDP2 instruction
__ldc	LDC instruction
__ldc2	LDC2 instruction
__ldc2_noidx	LDC2 instruction
__ldc2l	LDC2L instruction
__ldc2l_noidx	LDC2L instruction
__ldc_noidx	LDC instruction
__ldcl	LDCL instruction
__ldcl_noidx	LDCL instruction
__mcr	MCR instruction
__mcr2	MCR2 instruction
__mcrr	MCRR instruction

__mcrr2	MCRR2 instruction
__mrc	MRC instruction
__mrc2	MRC2 instruction
__mrrc	MRRC instruction
__mrrc2	MRRC2 instruction
__stc	STC instruction
__stc2	STC2 instruction
__stc2l	STC2L instruction
__stc_noidx	STC2L instruction
__stcl	STCL instruction
Interrupt Intrinsics	
__disable_fiq	Disable FIQ interrupts
__disable_interrupt	Disable interrupt
__disable_irq	Disable IRQ interrupts
__enable_fiq	Enable FIQ interrupts
__enable_interrupt	Enable interrupt
__enable_irq	Enable IRQ interrupts
VFP Intrinsics	
__fabs	VABS.F64 instruction
__fabsf	VABS.F32 instruction
__sqrt	VSQRT.F64 instruction
__sqrtf	VQSRT.F32 instruction
Register Intrinsics	
__get_APSR	Get APSR value
__get_BASEPRI	Get BASEPRI register value
__get_CONTROL	Get CONTROL register value
__get_CPSR	Get CPSR value
__get_FAULTMASK	Get FAULTMASK register value
__get_PRIMASK	Get PRIMASK register value
__set_APSR	Set APSR value
__set_BASEPRI	Set BASEPRI register value
__set_CONTROL	Set CONTROL register value
__set_CPSR	Set CPSR value
__set_FAULTMASK	Set FAULTMASK register value
__set_PRIMASK	Set PRIMASK register value

Load/Store Intrinsics	
__ldrbt	LDRBT instruction
__ldrex	LDREX instruction
__ldrexb	LDREXB instruction
__ldrexd	LDREXD instruction
__ldrexh	LDREXH instruction
__ldrht	LDRHT instruction
__ldrsbt	LDRSBT instruction
__ldrsht	LDRSHT instruction
__ldrt	LDRT instruction
__strbt	STRBT instruction
__strex	STREX instruction
__strexb	STREXB instruction
__strexd	STREXD instruction
__strexh	STREXH instruction
__strht	STRHT instruction
__strt	STRT instruction
DSP & SIMD Intrinsics	
__qadd	QADD instruction
__qadd16	QADD16 instruction
__qadd8	QADD8 instruction
__qasx	QASX instruction
__qdadd	QDADD instruction
__qdbl	QDBL instruction
__qdsub	QDSUB instruction
__qflag	Get Q flag value
__qsax	QSAX instruction
__qsub	QSUB instruction
__qsub16	QSUB16 instruction
__qsub8	QSUB8 instruction
__sadd16	SADD16 instruction
__sadd8	SADD8 instruction
__sasx	SASX instruction
__sel	SEL instruction
__shadd16	SHADD16 instruction

__shadd8	SHADD8 instruction
__shasx	SHASX instruction
__shsax	SHSAX instruction
__shsub16	SHSUB16 instruction
__shsub8	SHSUB8 instruction
__smlabb	SMLABB instruction
__smlabt	SMLABT instruction
__smlad	SMLAD instruction
__smladx	SMLADX instruction
__smlalbb	SMLALBB instruction
__smlalbt	SMLALBT instruction
__smlald	SMLALD instruction
__smlaldx	SMLALDX instruction
__smlaltb	SMLALTB instruction
__smlaltt	SMLALTT instruction
__smlatb	SMLATB instruction
__smlatt	SMLATT instruction
__smlawb	SMLAWB instruction
__smlawt	SMLAWT instruction
__smlsd	SMLSD instruction
__smlsdx	SMLSDX instruction
__smlsld	SMLSLD instruction
__smlsldx	SMLSLDX instruction
__smuad	SMUAD instruction
__smuadx	SMUADX instruction
__smulbb	SMULBB instruction
__smulbt	SMULBT instruction
__smultb	SMULTB instruction
__smultt	SMULTT instruction
__smulwb	SMULWB instruction
__smulwt	SMULWT instruction
__smusd	SMUSD instruction
__smusdx	SMUSDX instruction
__ssat	SSAT instruction
__ssat16	SSAT16 instruction

__ssax	SSAX instruction
__ssub16	SSUB16 instruction
__ssub8	SSUB8 instruction
__sxtab16	SXTAB16 instruction
__sxtb16	SXTB16 instruction
__uadd16	UADD16 instruction
__uadd8	UADD8 instruction
__uasx	UASX instruction
__uhadd16	UHADD16 instruction
__uhadd8	UHADD8 instruction
__uhasx	UHASX instruction
__uhsax	UHSAX instruction
__uhsub16	UHSUB16 instruction
__uhsub8	UHSUB8 instruction
__uqadd16	UQADD16 instruction
__uqadd8	UQADD8 instruction
__uqasx	UQASX instruction
__uqsax	UQSAX instruction
__uqsub16	USUB16 instruction
__uqsub8	UQSUB8 instruction
__usad8	USAD8 instruction
__usad8a	USADA8 instruction
__usat	USAT instruction
__usat16	USAT16 instruction
__usax	USAX instruction
__usub8	USUB8 instruction
__uxtab16	UXTAB16 instruction
__uxtb16	UXTB16 instruction
Reversing Intrinsics	
__rbit	RBIT instruction
__rev	REV instruction
__rev16	REV16 instruction
__revsh	REVSH instruction

__breakpoint

Synopsis

```
void __breakpoint(unsigned val);
```

Description

__**breakpoint** inserts a BKPT instruction where **val** is a compile time constant.

__cdp

Synopsis

```
void __cdp(unsigned coproc,
           unsigned opc1,
           unsigned crd,
           unsigned crn,
           unsigned crm,
           unsigned opc2);
```

Description

__**cdp** inserts a CDP instruction. All arguments are compile time constants.

__cdp2

Synopsis

```
void __cdp2(unsigned coproc,
            unsigned opc1,
            unsigned crd,
            unsigned crn,
            unsigned crm,
            unsigned opc2);
```

Description

__cdp2 inserts a CDP2 instruction. All arguments are compile time constants.

__clrex

Synopsis

```
void __clrex(void);
```

Description

__**clrex** inserts a CLREX instruction.

__clz

Synopsis

```
unsigned char __clz(unsigned val);
```

Description

__clz returns the number of leading zeros in **val**.

__dbg

Synopsis

```
void __dbg(unsigned option);
```

Description

__dbg inserts a DBG instruction where **option** is a compile time constant.

__disable_fiq

Synopsis

```
int __disable_fiq(void);
```

Description

__disable_fiq sets the F bit in the CPSR and returns the previous F bit value.

__disable_interrupt

Synopsis

```
void __disable_interrupt(void);
```

Description

__disable_interrupt set the PRIMASK for Cortex-M parts and sets the I and F bit in the CPSR for ARM parts.

__disable_irq

Synopsis

```
int __disable_irq(void);
```

Description

__disable_irq sets the I bit in the CPSR and returns the previous I bit value.

__dmb

Synopsis

```
void __dmb(void);
```

Description

__dmb inserts a DMB instruction.

__dsb

Synopsis

```
void __dsb(void);
```

Description

__dsb inserts a DSB instruction.

__enable_fiq

Synopsis

```
void __enable_fiq(void);
```

Description

__enable_fiq clears the F bit in the CPSR.

__enable_interrupt

Synopsis

```
void __enable_interrupt(void);
```

Description

__enable_interrupt clears the PRIMASK for Cortex-M parts and clears the I and F bit in the CPSR for ARM parts.

__enable_irq

Synopsis

```
void __enable_irq(void);
```

Description

__enable_irq clears the I bit in the CPSR.

__fabs

Synopsis

```
double __fabs(double val);
```

Description

__**fabs** inserts a VABS.F64 instruction. Returns the absolute value of **val**.

__fabsf

Synopsis

```
float __fabsf(float val);
```

Description

__fabsf inserts a VABS.F32 instruction. Returns the absolute value of **val**.

__get_APSR

Synopsis

```
unsigned __get_APSR(void);
```

Description

__get_APSR returns the value of the APSR/CPSR for Cortex-M/ARM parts.

__get_BASEPRI

Synopsis

```
unsigned __get_BASEPRI(void);
```

Description

__get_BASEPRI returns the value of the Cortex-M3/M4 BASEPRI register.

__get_CONTROL

Synopsis

```
unsigned __get_CONTROL(void);
```

Description

__get_CONTROL returns the value of the Cortex-M CONTROL register.

__get_CPSR

Synopsis

```
unsigned __get_CPSR(void);
```

Description

__get_CPSR returns the value of the ARM CPSR register.

__get_FAULTMASK

Synopsis

```
unsigned __get_FAULTMASK(void);
```

Description

__get_FAULTMASK returns the value of the Cortex-M3/M4 FAULTMASK register.

__get_PRIMASK

Synopsis

```
unsigned __get_PRIMASK(void);
```

Description

__get_PRIMASK returns the value of the Cortex-M PRIMASK register.

__isb

Synopsis

```
void __isb(void);
```

Description

__**isb** inserts a ISB instruction.

__ldc

Synopsis

```
void __ldc(unsigned coproc,
           unsigned Crd,
           unsigned *ptr);
```

Description

__**ldc** inserts a LDC instruction where **coproc** and **Crd** are compile time constants and **ptr** points to the word of data to load.

__ldc2

Synopsis

```
void __ldc2(unsigned coproc,
            unsigned Crd,
            unsigned *ptr);
```

Description

__**ldc2** inserts a LDC2 instruction where **coproc** and **Crd** are compile time constants and **ptr** points to the word of data to load.

__ldc2_noidx

Synopsis

```
void __ldc2_noidx(unsigned coproc,
                  unsigned Crd,
                  unsigned *ptr,
                  unsigned option);
```

Description

__ldc2_noidx inserts a LDC2 instruction where **coproc**, **Crd** and **option** are compile time constants and **ptr** points to the word of data to load.

__ldc2l

Synopsis

```
void __ldc2l(unsigned coproc,
             unsigned Crd,
             unsigned *ptr);
```

Description

__ldc2l inserts a LDC2L instruction where **coproc** and **Crd** are compile time constants and **ptr** points to the word of data to load.

__ldc2l_noidx

Synopsis

```
void __ldc2l_noidx(unsigned coproc,
                   unsigned Crd,
                   unsigned *ptr,
                   unsigned option);
```

Description

__ldc2l_noidx inserts a LDC2L instruction where **coproc**, **Crd** and **option** are compile time constants and **ptr** points to the word of data to load.

__ldc_noidx

Synopsis

```
void __ldc_noidx(unsigned coproc,
                 unsigned Crd,
                 unsigned *ptr,
                 unsigned option);
```

Description

__ldc_noidx inserts a LDC instruction where **coproc**, **Crd** and **option** are compile time constants and **ptr** points to the word of data to load.

__ldcl

Synopsis

```
void __ldcl(unsigned coproc,
            unsigned Crd,
            unsigned *ptr);
```

Description

__**ldcl** inserts a LDCL instruction where **coproc** and **Crd** are compile time constants and **ptr** points to the word of data to load.

__ldcl_noidx

Synopsis

```
void __ldcl_noidx(unsigned coproc,
                  unsigned Crd,
                  unsigned *ptr,
                  unsigned option);
```

Description

__ldcl_noidx inserts a LDCL instruction where **coproc**, **Crd** and **option** are compile time constants and **ptr** points to the word of data to load.

__ldrbt

Synopsis

```
unsigned __ldrbt(unsigned char *ptr);
```

Description

__**ldrbt** inserts a LDRBT instruction. Returns the byte of data at memory address **ptr**.

__ldrex

Synopsis

```
unsigned __ldrex(unsigned *ptr);
```

Description

__**ldrex** inserts a LDREX instruction. Returns the word of data at memory address **ptr**.

__ldrexb

Synopsis

```
unsigned __ldrexb(unsigned char *ptr);
```

Description

__**ldrexb** inserts a LDREXB instruction. Returns the byte of data at memory address **ptr**.

__ldrexd

Synopsis

```
unsigned long long __ldrexd(unsigned long long *ptr);
```

Description

__**ldrexd** inserts a LDREXD instruction. Returns the double word of data at memory address **ptr**.

__ldrexh

Synopsis

```
unsigned __ldrexh(unsigned short *ptr);
```

Description

__**ldrexh** inserts a LDREXH instruction. Returns the half word of data at memory address **ptr**.

__ldrht

Synopsis

```
unsigned __ldrht(unsigned short *ptr);
```

Description

__**ldrht** inserts a LDRHT instruction. Returns the half word of data at memory address **ptr**.

__ldrsbt

Synopsis

```
unsigned __ldrsbt(signed char *ptr);
```

Description

__**ldrsbt** inserts a LDRSBT instruction. Returns the sign extended byte of data at memory address **ptr**.

__ldrsht

Synopsis

```
unsigned __ldrsht(short *ptr);
```

Description

__ldrsht inserts a LDRSHT instruction. Returns the sign extended half word of data at memory address **ptr**.

__ldrt

Synopsis

```
unsigned __ldrt(unsigned *ptr);
```

Description

__ldrt inserts a LDRT instruction. Returns the word of data at memory address **ptr**.

__mcr

Synopsis

```
void __mcr(unsigned coproc,
           unsigned opc1,
           unsigned src,
           unsigned CRn,
           unsigned CRm,
           unsigned opc2);
```

Description

__mcr inserts a MCR instruction. Where **coproc**, **opc1**, **Crn**, **Crm** and **opc2** are compile time constants and **src** is the value to write.

__mcr2

Synopsis

```
void __mcr2(unsigned coproc,
            unsigned opc1,
            unsigned src,
            unsigned CRn,
            unsigned CRm,
            unsigned opc2);
```

Description

__**mcr2** inserts a MCR2 instruction. Where **coproc**, **opc1**, **Crn**, **Crm** and **opc2** are compile time constants and **src** is the value to write.

__mcrr

Synopsis

```
void __mcrr(unsigned coproc,
            unsigned opc1,
            unsigned src1,
            unsigned src2,
            unsigned CRn);
```

Description

__**mcrr** inserts a MCRR instruction. Where **coproc**, **opc1** and **Crn** are compile time constants and **src1**, **src2** are the values to write.

__mcrr2

Synopsis

```
void __mcrr2(unsigned coproc,
             unsigned opc1,
             unsigned src1,
             unsigned src2,
             unsigned CRn);
```

Description

__mcrr2 inserts a MCRR2 instruction. Where **coproc**, **opc1** and **Crn** are compile time constants and **src1**, **src2** are the values to write.

__mrc

Synopsis

```
unsigned __mrc(unsigned coproc,
               unsigned opc1,
               unsigned CRn,
               unsigned CRm,
               unsigned opc2);
```

Description

__**mrc** inserts a MRC instruction. Where **coproc**, **opc1**, **Crn**, **Crm** and **opc2** are compile time constants. __mrc returns the value read.

__mrc2

Synopsis

```
unsigned __mrc2(unsigned coproc,
                unsigned opc1,
                unsigned CRn,
                unsigned CRm,
                unsigned opc2);
```

Description

__**mrc2** inserts a MRC2 instruction. Where **coproc**, **opc1**, **Crn**, **Crm** and **opc2** are compile time constants. __mrc2 returns the value read.

__mrrc

Synopsis

```
void __mrrc(unsigned coproc,
            unsigned opc1,
            unsigned *dst1,
            unsigned *dst2,
            unsigned CRn);
```

Description

__**mrrc** inserts a MRRC instruction. Where **coproc**, **opc1** and **Crn** are compile time constants and **dst1**, **dst2** are the values read.

__mrrc2

Synopsis

```
void __mrrc2(unsigned coproc,
             unsigned opc1,
             unsigned *dst1,
             unsigned *dst2,
             unsigned CRn);
```

Description

__mrrc2 inserts a MRRC2 instruction. Where **coproc**, **opc1** and **Crn** are compile time constants and **dst1**, **dst2** are the values read.

__nop

Synopsis

```
void __nop(void);
```

Description

__**nop** inserts a NOP instruction.

__pld

Synopsis

```
void __pld(void *ptr);
```

Description

__pld inserts a PLD instruction. Where **ptr** specifies the memory address.

__pli

Synopsis

```
void __pli(void *ptr);
```

Description

__pli inserts a PLI instruction. Where **ptr** specifies the memory address.

__qadd

Synopsis

```
int __qadd(int val1,
           int val2);
```

Description

__qadd inserts a QADD instruction. Returns the 32-bit saturating signed equivalent of res = val1 + val2. This operation sets the Q flag if saturation occurs.

__qadd16

Synopsis

```
int16x2 __qadd16(int16x2 val1,
                 int16x2 val2);
```

Description

__**qadd16** inserts a QADD16 instruction. __**qadd16** returns the 16-bit signed saturated equivalent of

- res[0] = val1[0] + val2[0],
- res[1] = val1[1] + val2[1]

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__qadd8

Synopsis

```
int8x4 __qadd8(int8x4 val1,
               int8x4 val2);
```

Description

__qadd8 inserts a QADD8 instruction. **__qadd8** returns the 8-bit signed saturated equivalent of

- res[0] = val1[0] + val2[0]
- res[1] = val1[1] + val2[1]
- res[2] = val1[2] + val2[2]
- res[3] = val1[3] + val2[3]

  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__qasx

Synopsis

```
int16x2 __qasx(int16x2 val1,
               int16x2 val2);
```

Description

__**qasx** inserts a QASX instruction. __**qasx** returns the 16-bit signed saturated equivalent of

- res[0] = val1[1] - val2[1]
- res[1] = val1[1] + val2[0]

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__qdadd

Synopsis

```
int __qdadd(int val1,
            int val2);
```

Description

__qdadd inserts a QDADD instruction. **__qdadd** returns the 32-bit signed saturated equivalent of res = val1 + (2*val2). This operation sets the Q flag if saturation occurs.

__qdbl

Synopsis

```
int __qdbl(int val);
```

Description

__qdbl inserts a QADD instruction. **__qdbl** returns the 32-bit signed saturated equivalent of res = val + val. This operation sets the Q flag if saturation occurs.

__qdsub

Synopsis

```
int __qdsub(int val1,
            int val2);
```

Description

__qdsub inserts a QDSUB instruction. **__qdsub** returns the 32-bit signed saturated equivalent of val1 - (2*val2). This operation sets the Q flag if saturation occurs.

__qflag

Synopsis

```
int __qflag(void);
```

Description

__qflag returns the value of the Q flag.

__qsax

Synopsis

```
int16x2 __qsax(int16x2 val1,
               int16x2 val2);
```

Description

__**qsax** inserts a QSAX instruction. __**qsax** returns the 16-bit signed saturated equivalent of

- res[0] = val1[0] + val2[1]
- res[1] = val1[1] - val2[0]

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__qsub

Synopsis

```
int __qsub(int val1,
           int val2);
```

Description

__qsub inserts a QSUB instruction. **__qsub** returns the 32-bit signed saturated equivalent of res=val1-val2. This operation sets the Q flag if saturation occurs.

__qsub16

Synopsis

```
int16x2 __qsub16(int16x2 val1,
                 int16x2 val2);
```

Description

__qsub16 inserts a QSUB16 instruction. **__qsub16** returns the 16-bit signed saturated equivalent of

- res[0] = val1[0] - val2[0]
- res[1] = val1[1] - val2[1]

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__qsub8

Synopsis

```
int8x4 __qsub8(int8x4 val1,
               int8x4 val2);
```

Description

__qsub8 inserts a QSUB8 instruction. **__qsub8** returns the 8-bit signed saturated equivalent of

- res[0] = val1[0] - val2[0]
- res[1] = val1[1] - val2[1]
- res[2] = val1[2] - val2[2]
- res[3] = val1[3] - val2[3]

  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__rbit

Synopsis

```
unsigned __rbit(unsigned val);
```

Description

__rbit inserts a RBIT instruction. **__rbit** returns the bit reversed equivalent of **val**.

__rev

Synopsis

```
unsigned __rev(unsigned val);
```

Description

__**rev** inserts a REV instruction. __**rev** returns the equivalent of

- res[0] = val[3]
- res[1] = val[2]
- res[2] = val[1]
- res[3] = val[0]
  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__rev16

Synopsis

```
unsigned __rev16(unsigned val);
```

Description

__**rev16** inserts a REV16 instruction. __**rev16** returns the equivalent of

- res[0] = val[1]
- res[1] = val[0]
- res[2] = val[3]
- res[3] = val[2]
  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__revsh

Synopsis

```
unsigned __revsh(unsigned val);
```

Description

__**revsh** inserts a REVSH instruction. __**revsh** returns the 16-bit sign extended equivalent of

- res[0] = val[1]
- res[1] = val[0]
  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__sadd16

Synopsis

```
int16x2 __sadd16(int16x2 val1,
                 int16x2 val2);
```

Description

__**sadd16** inserts a SADD16 instruction. __**sadd16** returns the 16-bit signed equivalent of

- res[0] = val1[0] + val2[0]
- res[1] = val1[1] + val2[1]

  where [0] is the lower 16 bits and [1] is the upper 16 bits. The GE bits of the APSR are set.

__sadd8

Synopsis

```
int8x4 __sadd8(int8x4 val1,
               int8x4 val2);
```

Description

__**sadd8** inserts a SADD8 instruction. __**sadd8** returns the 8-bit signed equivalent of

- res[0] = val1[0] + val2[0]
- res[1] = val1[1] + val2[1]
- res[2] = val1[2] + val2[2]
- res[3] = val1[3] + val2[3]

  where [0] is the lower 8 bits and [3] is the upper 8 bits. The GE bits of the APSR are set.

__sasx

Synopsis

```
int16x2 __sasx(int16x2 val1,
               int16x2 val2);
```

Description

__**sasx** inserts a SASX instruction. __**sasx** returns the 16-bit signed equivalent of

- res[0] = val1[0] - val2[1]
- res[1] = val1[1] + val2[0]

  where [0] is the lower 16 bits and [1] is the upper 16 bits. The GE bits of the APSR are set.

__sel

Synopsis

```
uint8x4 __sel(uint8x4 val1,
              uint8x4 val2);
```

Description

__**sel** inserts a SEL instruction. __**sel** returns the equivalent of

- res[0] = GE[0] ? val1[0] : val2[0]
- res[1] = GE[1] ? val1[1] : val2[1]
- res[2] = GE[2] ? val1[2] : val2[2]
- res[3] = GE[3] ? val1[3] : val2[3]

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__set_APSR

Synopsis

```
void __set_APSR(unsigned val);
```

Description

__set_APSR sets the value of the APSR i.e. the condition bits and the GE bits.

__set_BASEPRI

Synopsis

```
void __set_BASEPRI(unsigned val);
```

Description

__set_BASEPRI sets the value of the Cortex-M3/M4 BASEPRI register.

__set_CONTROL

Synopsis

```
void __set_CONTROL(unsigned val);
```

Description

__set_CONTROL set the value of the Cortex-M CONTROL register.

__set_CPSR

Synopsis

```
void __set_CPSR(unsigned val);
```

Description

__set_CPSR sets the value of the ARM CPSR.

__set_FAULTMASK

Synopsis

```
void __set_FAULTMASK(unsigned val);
```

Description

__set_FAULTMASK sets the value of the Cortex-M3/M4 FAULTMASK register.

__set_PRIMASK

Synopsis

```
void __set_PRIMASK(unsigned val);
```

Description

__set_PRIMASK sets the value of the Cortex-M3/M4 PRIMASK register.

__sev

Synopsis

```
void __sev(void);
```

Description

__**sev** inserts a SEV instruction.

__shadd16

Synopsis

```
int16x2 __shadd16(int16x2 val1,
                  int16x2 val2);
```

Description

__**shadd16** inserts a SHADD16 instruction. __**shadd16** returns the 16-bit signed equivalent of

- res[0] = (val1[0] + val2[0])/2
- res[1] = (val1[1] + val2[1])/2

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__shadd8

Synopsis

```
int8x4 __shadd8(int8x4 val1,
                int8x4 val2);
```

Description

__**shadd8** inserts a SHADD8 instruction. __**shadd8** returns the 8-bit signed equivalent of

- res[0] = (val1[0] + val2[0])/2
- res[1] = (val1[1] + val2[1])/2
- res[2] = (val1[2] + val2[2])/2
- res[3] = (val1[3] + val2[3])/2

  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__shasx

Synopsis

```
int16x2 __shasx(int16x2 val1,
                int16x2 val2);
```

Description

__**shasx** inserts a SHASX instruction. __**shasx** returns the 16-bit signed equivalent of

- res[0] = (val1[0] - val2[1])/2
- res[1] = (val1[1] + val2[0])/2

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__shsax

Synopsis

```
int16x2 __shsax(int16x2 val1,
                int16x2 val2);
```

Description

__**shsax** inserts a SHSAX instruction. __**shsax** returns the 16-bit signed equivalent of

- res[0] = (val1[0] + val2[1])/2
- res[1] = (val1[1] - val2[0])/2

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__shsub16

Synopsis

```
int16x2 __shsub16(int16x2 val1,
                  int16x2 val2);
```

Description

__**shsub16** inserts a SHSUB16 instruction. __**shsub16** returns the 16-bit signed equivalent of

- res[0] = (val1[0] - val2[0])/2
- res[1] = (val1[1] - val2[1])/2

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__shsub8

Synopsis

```
int8x4 __shsub8(int8x4 val1,
                int8x4 val2);
```

Description

__**shsub8** inserts a SHSUB8 instruction. __**shsub8** returns the 8-bit signed equivalent of

- res[0] = (val1[0] - val2[0])/2
- res[1] = (val1[1] - val2[1])/2
- res[2] = (val1[2] - val2[2])/2
- res[3] = (val1[3] - val2[3])/2

  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__smlabb

Synopsis

```
int __smlabb(int16x2 val1,
             int16x2 val2,
             int val3);
```

Description

__**smlabb** inserts a SMLABB instruction. __**smlabb** returns the equivalent of

- res = val1[0] * val2[0] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smlabt

Synopsis

```
int __smlabt(int16x2 val1,
             int16x2 val2,
             int val3);
```

Description

__**smlabt** inserts a SMLABT instruction. __**smlabt** returns the equivalent of

- res = val1[0] * val2[1] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smlad

Synopsis

```
int __smlad(int16x2 val1,
            int16x2 val2,
            int val3);
```

Description

__**smlad** inserts a SMLAD instruction. __**smlad** returns the 16-bit signed equivalent of

- res = val1[0] * val2[0] + val1[1] * val2[1] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smladx

Synopsis

```
int __smladx(int16x2 val1,
             int16x2 val2,
             int val3);
```

Description

__**smladx** inserts a SMLADX instruction. __**smladx** returns the 16-bit signed equivalent of

- res = val1[0] * val2[1] + val1[1] * val2[0] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smlalbb

Synopsis

```
long long __smlalbb(int16x2 val1,
                    int16x2 val2,
                    long long val3);
```

Description

__**smlalbb** inserts a SMLALBB instruction. __**smlalbb** returns the equivalent of

- res = val1[0] * val2[0] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smlalbt

Synopsis

```
long long __smlalbt(int16x2 val1,
                    int16x2 val2,
                    long long val3);
```

Description

__**smlalbt** inserts a SMLALBT instruction. __**smlalbt** returns the equivalent of

- res = val1[0] * val2[1] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smlald

Synopsis

```
long long __smlald(int16x2 val1,
                   int16x2 val2,
                   long long val3);
```

Description

__**smlald** inserts a SMLALD instruction. __**smlald** returns the equivalent of

- res = val1[0] * val2[0] + val1[1] * val2[1] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smlaldx

Synopsis

```
long long __smlaldx(int16x2 val1,
                    int16x2 val2,
                    long long val3);
```

Description

__**smlaldx** inserts a SMLALDX instruction. __**smlaldx** returns the equivalent of

- res = val1[0] * val2[1] + val1[1] * val2[0] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smlaltb

Synopsis

```
long long __smlaltb(int16x2 val1,
                    int16x2 val2,
                    long long val3);
```

Description

__**smlaltb** inserts a SMLALTB instruction. __**smlaltb** returns the equivalent of

- res = val1[1] * val2[0] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smlaltt

Synopsis

```
long long __smlaltt(int16x2 val1,
                    int16x2 val2,
                    long long val3);
```

Description

__**smlaltt** inserts a SMLALTT instruction. __**smlaltt** returns the equivalent of

- res = val1[1] * val2[1] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smlatb

Synopsis

```
int __smlatb(int16x2 val1,
             int16x2 val2,
             int val3);
```

Description

__**smlatb** inserts a SMLATB instruction. __**smlatb** returns the equivalent of

- res = val1[1] * val2[0] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smlatt

Synopsis

```
int __smlatt(int16x2 val1,
             int16x2 val2,
             int val3);
```

Description

__**smlatt** inserts a SMLATT instruction. __**smlatt** returns the equivalent of

- res = val1[1] * val2[1] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smlawb

Synopsis

```
int __smlawb(int val1,
             int16x2 val2,
             int val3);
```

Description

__**smlawb** inserts a SMLAWB instruction. __**smlawb** returns the equivalent of

- res = (val1 * val2[0] + (val3 << 16)) >> 16
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smlawt

Synopsis

```
int __smlawt(int val1,
             int16x2 val2,
             int val3);
```

Description

__**smlawt** inserts a SMLAWT instruction. __**smlawt** returns the equivalent of

- res = (val1 * val2[1] + (val3 << 16)) >> 16
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smlsd

Synopsis

```
int __smlsd(int16x2 val1,
            int16x2 val2,
            int val3);
```

Description

__**smlsd** inserts a SMLSD instruction. __**smlsd** returns the equivalent of

- res = val1[0] * val2[0] - val1[1] * val2[1] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smlsdx

Synopsis

```
int __smlsdx(int16x2 val1,
             int16x2 val2,
             int val3);
```

Description

__**smlsdx** inserts a SMLSDX instruction. __**smlsdx** returns the equivalent of

- res = val1[0] * val2[1] - val1[1] * val2[0] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smlsld

Synopsis

```
long long __smlsld(int16x2 val1,
                   int16x2 val2,
                   long long val3);
```

Description

__**smlsld** inserts a SMLSLD instruction. __**smlsld** returns the equivalent of

- res = val1[0] * val2[0] - val1[1] * val2[1] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smlsldx

Synopsis

```
long long __smlsldx(int16x2 val1,
                    int16x2 val2,
                    long long val3);
```

Description

__**smlsldx** inserts a SMLSLDX instruction. __**smlsldx** returns the equivalent of

- res = val1[0] * val2[1] - val1[1] * val2[0] + val3
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smuad

Synopsis

```
int __smuad(int16x2 val1,
            int16x2 val2);
```

Description

__**smuad** inserts a SMUAD instruction. __**smuad** returns the equivalent of

- res = val1[0] * val2[0] + val1[1] * val2[1]
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smuadx

Synopsis

```
int __smuadx(int16x2 val1,
             int16x2 val2);
```

Description

__**smuadx** inserts a SMUADX instruction. __**smuadx** returns the equivalent of

- res = val1[0] * val2[1] + val1[1] * val2[0]
  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the Q flag if overflow occurs on the addition.

__smulbb

Synopsis

```
int __smulbb(int16x2 val1,
             int16x2 val2);
```

Description

__**smulbb** inserts a SMULBB instruction. __**smulbb** returns the equivalent of

- res = val1[0] * val2[0]
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smulbt

Synopsis

```
int __smulbt(int16x2 val1,
             int16x2 val2);
```

Description

__**smulbt** inserts a SMULBT instruction. __**smulbt** returns the equivalent of

- res = val1[0] * val2[1]
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smultb

Synopsis

```
int __smultb(int16x2 val1,
             int16x2 val2);
```

Description

__**smultb** inserts a SMULTB instruction. __**smultb** returns the equivalent of

- res = val1[1] * val2[0]
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smultt

Synopsis

```
int __smultt(int16x2 val1,
             int16x2 val2);
```

Description

__smultt inserts a SMULTT instruction. **__smultt** returns the equivalent of

- res = val1[1] * val2[1]
  where [1] is the lower 16 bits and [1] is the upper 16 bits.

__smulwb

Synopsis

```
int __smulwb(int16x2 val1,
             int val2);
```

Description

__**smulwb** inserts a SMULWB instruction. __**smulwb** returns the equivalent of

- res = (val1[0] * val2) >> 16
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smulwt

Synopsis

```
int __smulwt(int16x2 val1,
             int val2);
```

Description

__**smulwt** inserts a SMULWT instruction. __**smulwt** returns the equivalent of

- res = (val1[1] * val2) >> 16
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smusd

Synopsis

```
int __smusd(int16x2 val1,
            int16x2 val2);
```

Description

__**smusd** inserts a SMUSD instruction. __**smusd** returns the equivalent of

- res = val1[0] * val2[0] - val1[1] * val2[1]
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smusdx

Synopsis

```
int __smusdx(int16x2 val1,
             int16x2 val2);
```

Description

__**smusdx** inserts a SMUSDX instruction. __**smusdx** returns the equivalent of

- res = val1[0] * val2[1] - val1[1] * val2[0]
  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__sqrt

Synopsis

```
double __sqrt(double val);
```

Description

__**sqrt** inserts a VSQRT.F64 instruction.

__sqrtf

Synopsis

```
float __sqrtf(float val);
```

Description

__**sqrtf** inserts a VSQRT.F32 instruction.

__ssat

Synopsis

```
int __ssat(int val,
           unsigned sat);
```

Description

__**ssat** inserts a SSAT instruction. __**ssat** returns **val** saturated to the signed range of **sat** where **sat** is a compile time constant.

__ssat16

Synopsis

```
int16x2 __ssat16(int16x2 val,
                 unsigned sat);
```

Description

__**ssat16** inserts a SSAT16 instruction. __**ssat16** returns the equivalent of

- res[0] = val[0] saturated to the signed range of sat
- res[1] = val[1] saturated to the signed range of sat

  where [0] is the lower 16 bits and [1] is the upper 16 bits and **sat** is a compile time constant.

__ssax

Synopsis

```
int16x2 __ssax(int16x2 val1,
               int16x2 val2);
```

Description

__**ssax** inserts a SSAX instruction. __**ssax** returns the equivalent of

- res[0] = val1[0] + val2[1]
- res[1] = val1[1] - val2[0]

  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the GE bits.

__ssub16

Synopsis

```
int16x2 __ssub16(int16x2 val1,
                 int16x2 val2);
```

Description

__**ssub16** inserts a SSUB16 instruction. __**ssub16** returns the 16-bit signed equivalent of

- res[0] = val1[0] - val2[0]
- res[1] = val1[1] - val2[1]

  where [0] is the lower 16 bits and [1] is the upper 16 bits. The GE bits of the APSR are set.

__ssub8

Synopsis

```
int8x4 __ssub8(int8x4 val1,
               int8x4 val2);
```

Description

__ssub8 inserts a SSUB8 instruction. **__ssub8** returns the 8-bit signed equivalent of

- res[0] = val1[0] - val2[0]
- res[1] = val1[1] - val2[1]
- res[2] = val1[2] - val2[2]
- res[3] = val1[3] - val2[3]

  where [0] is the lower 8 bits and [3] is the upper 8 bits. The GE bits of the APSR are set.

__stc

Synopsis

```
void __stc(unsigned coproc,
           unsigned Crd,
           unsigned *ptr);
```

Description

__**stc** inserts a STC instruction where **coproc** and **Crd** are compile time constants and **ptr** points to the word of data to store.

__stc2

Synopsis

```
void __stc2(unsigned coproc,
            unsigned Crd,
            unsigned *ptr);
```

Description

__**stc2** inserts a STC2 instruction where **coproc** and **Crd** are compile time constants and **ptr** points to the word of data to store.

__stc2l

Synopsis

```
void __stc2l(unsigned coproc,
             unsigned Crd,
             unsigned *ptr);
```

Description

__**stc2l** inserts a STC2L instruction where **coproc** and **Crd** are compile time constants and **ptr** points to the word of data to store.

__stc_noidx

Synopsis

```
void __stc_noidx(unsigned coproc,
                 unsigned Crd,
                 unsigned *ptr,
                 unsigned option);
```

Description

__**stc_noidx** inserts a STC2L instruction where **coproc**, **Crd** and **option** are compile time constants and **ptr** points to the word of data to store.

__stcl

Synopsis

```
void __stcl(unsigned coproc,
            unsigned Crd,
            unsigned *ptr);
```

Description

__**stcl** inserts a STCL instruction where **coproc** and **Crd** are compile time constants and **ptr** points to the word of data to store.

__strbt

Synopsis

```
void __strbt(unsigned char val,
             unsigned char *ptr);
```

Description

__strbt inserts a STRBT instruction.

__strex

Synopsis

```
int __strex(unsigned val,
            unsigned *ptr);
```

Description

__**strex** inserts a STREX instruction.

__strexb

Synopsis

```
int __strexb(unsigned char val,
             unsigned *char ptr);
```

Description

__strexb inserts a STREXB instruction.

__strexd

Synopsis

```
int __strexd(unsigned long long val,
             unsigned *long long ptr);
```

Description

__**strexd** inserts a STREXD instruction.

__strexh

Synopsis

```
int __strexh(unsigned short val,
             unsigned *short ptr);
```

Description

__strexh inserts a STREXH instruction.

__strht

Synopsis

```
void __strht(unsigned short val,
             unsigned short *ptr);
```

Description

__**strht** inserts a STRHT instruction.

__strt

Synopsis

```
void __strt(unsigned val,
            unsigned *ptr);
```

Description

__**strt** inserts a STRT instruction.

__swp

Synopsis

```
unsigned __swp(unsigned val,
               unsigned *ptr);
```

Description

__**swp** inserts a SWP instruction.

__swpb

Synopsis

```
unsigned __swpb(unsigned char val,
                unsigned char *ptr);
```

Description

__**swpb** inserts a SWPB instruction.

__sxtab16

Synopsis

```
int16x2 __sxtab16(int16x2 val1,
                  uint8x4 val2);
```

Description

__**sxtab16** inserts a SXTAB16 instruction. __**sxtab16** returns the 16-bit signed equivalent of

- res[0] = val1[0] + (short)val2[0]
- res[1] = val1[1] + (short)val2[2]
  where res[0] & val1[0] are the lower 16 bits, res[1] & val1[1] are the upper 16 bits, val2[0] is the lower 8 bits and val2[2] is the 8 bits starting at bit position 16.

__sxtb16

Synopsis

```
int16x2 __sxtb16(int8x4 val);
```

Description

__sxtb16 inserts a SXTB16 instruction. **__sxtb16** returns the 16-bit signed equivalent of

- res[0] = (short)val[0]
- res[1] = (short)val[2]
  where res[0] is the lower 16 bits, res[1] is the upper 16 bits, val[0] is the lower 8 bits and val[2] is the 8 bits starting at bit position 16.

__uadd16

Synopsis

```
uint16x2 __uadd16(uint16x2 val1,
                  uint16x2 val2);
```

Description

__**uadd16** inserts a UADD16 instruction. __**uadd16** returns the 16-bit unsigned equivalent of

- res[0] = val1[0] + val2[0]
- res[1] = val1[1] + val2[1]

  where [0] is the lower 16 bits and [1] is the upper 16 bits. The GE bits of the APSR are set.

__uadd8

Synopsis

```
uint8x4 __uadd8(uint8x4 val1,
                uint8x4 val2);
```

Description

__**uadd8** inserts a UADD8 instruction. __**uadd8** returns the 8-bit unsigned equivalent of

- res[0] = val1[0] + val2[0]
- res[1] = val1[1] + val2[1]
- res[2] = val1[2] + val2[2]
- res[3] = val1[3] + val2[3]

  where [0] is the lower 8 bits and [3] is the upper 8 bits. The GE bits of the APSR are set.

__uasx

Synopsis

```
uint16x2 __uasx(uint16x2 val1,
                uint16x2 val2);
```

Description

__**uasx** inserts a UASX instruction. __**uasx** returns the 16-bit unsigned equivalent of

- res[0] = val1[0] - val2[1]
- res[1] = val1[1] + val2[0]

  where [0] is the lower 16 bits and [1] is the upper 16 bits. The GE bits of the APSR are set.

__uhadd16

Synopsis

```
uint16x2 __uhadd16(uint16x2 val1,
                   uint16x2 val2);
```

Description

__**uhadd16** inserts a UHADD16 instruction. __**uhadd16** returns the 16-bit unsigned equivalent of

- res[0] = (val1[0] + val2[0])/2
- res[1] = (val1[1] + val2[1])/2

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__uhadd8

Synopsis

```
uint8x4 __uhadd8(uint8x4 val1,
                 uint8x4 val2);
```

Description

__**uhadd8** inserts a UHADD8 instruction. __**uhadd8** returns the 8-bit unsigned equivalent of

- res[0] = (val1[0] + val2[0])/2
- res[1] = (val1[1] + val2[1])/2
- res[2] = (val1[2] + val2[2])/2
- res[3] = (val1[3] + val2[3])/2

  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__uhasx

Synopsis

```
uint16x2 __uhasx(uint16x2 val1,
                 uint16x2 val2);
```

Description

__**uhasx** inserts a UHASX instruction. __**uhasx** returns the 16-bit unsigned equivalent of

- res[0] = (val1[0] - val2[1])/2
- res[1] = (val1[1] + val2[0])/2

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__uhsax

Synopsis

```
uint16x2 __uhsax(uint16x2 val1,
                 uint16x2 val2);
```

Description

__**uhsax** inserts a UHSAX instruction. __**uhsax** returns the 16-bit unsigned equivalent of

- res[0] = (val1[0] + val2[1])/2
- res[1] = (val1[1] - val2[0])/2

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__uhsub16

Synopsis

```
uint16x2 __uhsub16(uint16x2 val1,
                   uint16x2 val2);
```

Description

__**uhsub16** inserts a UHSUB16 instruction. __**uhsub16** returns the 16-bit unsigned equivalent of

- res[0] = (val1[0] - val2[0])/2
- res[1] = (val1[1] - val2[1])/2

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__uhsub8

Synopsis

```
uint8x4 __uhsub8(uint8x4 val1,
                 uint8x4 val2);
```

Description

__**uhsub8** inserts a UHSUB8 instruction. __**uhsub8** returns the 8-bit unsigned equivalent of

- res[0] = (val1[0] - val2[0])/2
- res[1] = (val1[1] - val2[1])/2
- res[2] = (val1[2] - val2[2])/2
- res[3] = (val1[3] - val2[3])/2

  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__uqadd16

Synopsis

```
uint16x2 __uqadd16(uint16x2 val1,
                   uint16x2 val2);
```

Description

__**uqadd16** inserts a UQADD16 instruction. __**uqadd16** returns the 16-bit unsigned saturated equivalent of

- res[0] = val1[0] + val2[0]
- res[1] = val1[1] + val2[1]

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__uqadd8

Synopsis

```
uint8x4 __uqadd8(uint8x4 val1,
                 uint8x4 val2);
```

Description

__**uqadd8** inserts a UQADD8 instruction. __**uqadd8** returns the 8-bit unsigned saturated equivalent of

- res[0] = val1[0] + val2[0]
- res[1] = val1[1] + val2[1]
- res[2] = val1[2] + val2[2]
- res[3] = val1[3] + val2[3]

  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__uqasx

Synopsis

```
uint16x2 __uqasx(uint16x2 val1,
                 uint16x2 val2);
```

Description

__**uqasx** inserts a UQASX instruction. __**uqasx** returns the 16-bit signed saturated equivalent of

- res[0] = val1[0] - val2[1]
- res[1] = val1[1] + val2[0]

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__uqsax

Synopsis

```
uint16x2 __uqsax(uint16x2 val1,
                 uint16x2 val2);
```

Description

__**uqsax** inserts a UQSAX instruction. __**uqsax** returns the 16-bit signed saturated equivalent of

- res[0] = val1[0] + val2[1]
- res[1] = val1[1] - val2[0]

  where [0] is the lower 16 bits and [1] is the upper 16 bits.

__uqsub16

Synopsis

```
uint16x2 __uqsub16(uint16x2 val1,
                   uint16x2 val2);
```

Description

__**uqsub16** inserts a USUB16 instruction. __**uqsub16** returns the 16-bit unsigned equivalent of

- res[0] = val1[0] - val2[0]
- res[1] = val1[1] - val2[1]

  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__uqsub8

Synopsis

```
uint8x4 __uqsub8(uint8x4 val1,
                 uint8x4 val2);
```

Description

__**uqsub8** inserts a UQSUB8 instruction. __**uqsub8** returns the 8-bit unsigned saturated equivalent of

- res[0] = val1[0] - val2[0]
- res[1] = val1[1] - val2[1]
- res[2] = val1[2] - val2[2]
- res[3] = val1[3] - val2[3]

  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__usad8

Synopsis

```
unsigned __usad8(uint8x4 val1,
                 uint8x4 val2);
```

Description

__**usad8** inserts a USAD8 instruction. __**usad8** returns the 8-bit unsigned equivalent of

- res = abs(val1[0] - val2[0]) + abs(val1[1] - val2[1]) + (val1[2] - val2[2]) + (val1[3] - val2[3])
  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__usad8a

Synopsis

```
unsigned __usad8a(uint8x4 val1,
                  uint8x4 val2,
                  unsigned val3);
```

Description

__**usad8a** inserts a USADA8 instruction. __**usad8a** returns the 8-bit unsigned equivalent of

- res = abs(val1[0] - val2[0]) + abs(val1[1] - val2[1]) + (val1[2] - val2[2]) + (val1[3] - val2[3]) + val3
  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__usat

Synopsis

```
int __usat(int val,
           unsigned sat);
```

Description

__**usat** inserts a USAT instruction. __**usat** returns **val** saturated to the unsigned range of **sat** where **sat** is a compile time constant.

__usat16

Synopsis

```
int16x2 __usat16(int16x2 val,
                 const unsigned sat);
```

Description

__usat16 inserts a USAT16 instruction. **__usat16** returns the equivalent of

- res[0] = val[0] saturated to the unsigned range of sat
- res[1] = val[1] saturated to the unsigned range of sat

  where [0] is the lower 16 bits and [1] is the upper 16 bits and **sat** is a compile time constant.

__usax

Synopsis

```
int16x2 __usax(int16x2 val1,
               int16x2 val2);
```

Description

__**usax** inserts a USAX instruction. __**usax** returns the equivalent of

- res[0] = val1[0] + val2[1]
- res[1] = val1[1] - val2[0]

  where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the GE bits.

__usub8

Synopsis

```
uint8x4 __usub8(uint8x4 val1,
                uint8x4 val2);
```

Description

__**usub8** inserts a USUB8 instruction. __**usub8** returns the 8-bit unsigned equivalent of

- res[0] = val1[0] - val2[0]
- res[1] = val1[1] - val2[1]
- res[2] = val1[2] - val2[2]
- res[3] = val1[3] - val2[3]

  where [0] is the lower 8 bits and [3] is the upper 8 bits.

__uxtab16

Synopsis

```
int16x2 __uxtab16(int16x2 val1,
                  uint8x4 val2);
```

Description

__**uxtab16** inserts a UXTAB16 instruction. __**uxtab16** returns the 16-bit unsigned equivalent of

- res[0] = val1[0] + (unsigned short)val2[0]
- res[1] = val1[1] + (unsigned short)val2[2]
  where res[0] & val1[0] are the lower 16 bits, res[1] & val1[1] are the upper 16 bits, val2[0] is the lower 8 bits and val2[2] is the 8 bits starting at bit position 16.

__uxtb16

Synopsis

```
int16x2 __uxtb16(int8x4 val);
```

Description

__**uxtb16** inserts a UXTB16 instruction. __**uxtb16** returns the 16-bit unsigned equivalent of

- res[0] = (unsigned short)val[0]
- res[1] = (unsigned short)val[2]
  where res[0] is the lower 16 bits, res[1] is the upper 16 bits, val[0] is the lower 8 bits and val[2] is the 8 bits starting at bit position 16.

__wfe

Synopsis

```
void __wfe(void);
```

Description

__**wfe** inserts a WFE instruction.

__wfi

Synopsis

```
void __wfi(void);
```

Description

__**wfi** inserts a WFI instruction.

__yield

Synopsis

```
void __yield(void);
```

Description

__yield inserts a YIELD instruction.

<iso646.h>

Overview

The header <iso646.h> defines macros that expand to the corresponding tokens to ease writing C programs with keyboards that do not have keys for frequently-used operators.

API Summary

Macros	
and	Alternative spelling for logical and operator
and_eq	Alternative spelling for logical and-equals operator
bitand	Alternative spelling for bitwise and operator
bitor	Alternative spelling for bitwise or operator
compl	Alternative spelling for bitwise complement operator
not	Alternative spelling for logical not operator
not_eq	Alternative spelling for not-equal operator
or	Alternative spelling for logical or operator
or_eq	Alternative spelling for bitwise or-equals operator
xor	Alternative spelling for bitwise exclusive or operator
xor_eq	Alternative spelling for bitwise exclusive-or-equals operator

and

Synopsis

```
#define and     &&
```

Description

and defines the alternative spelling for &&.

and_eq

Synopsis

```
#define and_eq  &=
```

Description

and_eq defines the alternative spelling for &=.

bitand

Synopsis

```
#define bitand  &
```

Description

bitand defines the alternative spelling for &.

bitor

Synopsis

```
#define bitor   |
```

Description

bitor defines the alternative spelling for |.

compl

Synopsis

```
#define compl   ~
```

Description

compl defines the alternative spelling for ~.

not

Synopsis

```
#define not     !
```

Description

not defines the alternative spelling for `!`.

not_eq

Synopsis

```
#define not_eq  !=
```

Description

not_eq defines the alternative spelling for !=.

or

Synopsis

```
#define or      ||
```

Description

or defines the alternative spelling for ||.

or_eq

Synopsis

```
#define or_eq   |=
```

Description

or_eq defines the alternative spelling for |=.

xor

Synopsis

```
#define xor     ^
```

Description

xor defines the alternative spelling for ^.

xor_eq

Synopsis

```
#define xor_eq  ^=
```

Description

xor_eq defines the alternative spelling for ^=.

<libarm.h>

API Summary

Functions	
libarm_dcc_read	Read a word of data from the host over JTAG using the ARM's debug comms channel.
libarm_dcc_write	Write a word of data to the host over JTAG using the ARM debug comms channel.
libarm_disable_fiq	Disable FIQ interrupts.
libarm_disable_irq	Disable IRQ interrupts.
libarm_disable_irq_fiq	Disables IRQ and FIQ interrupts and return the previous enable state.
libarm_enable_fiq	Enable FIQ interrupts.
libarm_enable_irq	Enable IRQ interrupts.
libarm_enable_irq_fiq	Enable IRQ and FIQ interrupts.
libarm_get_cpsr	Get the value of the CPSR.
libarm_isr_disable_irq	Re-disable ARM's global interrupts from within an IRQ interrupt service routine.
libarm_isr_enable_irq	Re-enable ARM's global interrupts from within an IRQ interrupt service routine.
libarm_mmu_flat_initialise_level_1_table	Create a flat mapped level 1 translation table.
libarm_mmu_flat_initialise_level_2_small_page_table	Create a level 2 small page table for an address range.
libarm_mmu_flat_set_level_1_cacheable_region	Mark region of memory described by level 1 section descriptors as cacheable.
libarm_mmu_flat_set_level_2_small_page_cacheable_region	Mark region of memory described by level 2 small page table descriptors as cacheable.
libarm_restore_irq_fiq	Restores the IRQ and FIQ interrupt enable state.
libarm_run_dcc_port_server	Serve commands from the ARM's debug communication channel.
libarm_set_cpsr	Set the value of the CPSR.
libarm_set_fiq	Enables or disables FIQ interrupts.
libarm_set_irq	Enables or disables IRQ interrupts.

libarm_dcc_read

Synopsis

```
unsigned long libarm_dcc_read(void);
```

Description

libarm_dcc_read returns — The data read from the debug comms channel.

The ARM's debug comms channel is usually used by debuggers so reading from this port with a debugger attached can cause unpredictable results.

libarm_dcc_write

Synopsis

```
void libarm_dcc_write(unsigned long data);
```

Description

data — The data to write to the debug comms channel.

The ARM's debug comms channel is usually used by debuggers so writing to this port with a debugger attached can cause unpredictable results.

libarm_disable_fiq

Synopsis

```
void libarm_disable_fiq(void);
```

Description

This function disables FIQ interrupts by setting the F bit in the CPSR register.

Note that this function modifies the CPSR register's control field and therefore will only work when the CPU is executing in a privileged operating mode.

Example

```
// Disable FIQ interrupts
   libarm_disable_fiq();
```

libarm_disable_irq

Synopsis

```
void libarm_disable_irq(void);
```

Description

This function disables IRQ interrupts by setting the I bit in the CPSR register.

Note that this function modifies the CPSR register's control field and therefore will only work when the CPU is executing in a privileged operating mode.

Example

```
// Disable IRQ interrupts
   libarm_disable_irq();
```

libarm_disable_irq_fiq

Synopsis

```
int libarm_disable_irq_fiq(void);
```

Description

libarm_disable_irq_fiq returns — The IRQ and FIQ enable state prior to disabling the IRQ and FIQ interrupts.

This function disables both IRQ and FIQ interrupts, it also returns the previous IRQ and FIQ enable state so that it can be restored using **libarm_restore_irq_fiq**.

Note that this function modifies the CPSR register's control field and therefore will only work when the CPU is executing in a privileged operating mode.

Example

```
int s;
   // Disable IRQ and FIQ interrupts
   s = libarm_disable_irq_fiq();
   \hdots
   // Restore IRQ and FIQ interrupts
   libarm_restore_irq_fiq(s);
```

libarm_enable_fiq

Synopsis

```
void libarm_enable_fiq(void);
```

Description

This function enables FIQ interrupts by clearing the F bit in the CPSR register.

Note that this function modifies the CPSR register's control field and therefore will only work when the CPU is executing in a privileged operating mode.

Example

```
// Enable FIQ interrupts
   libarm_enable_fiq();
```

libarm_enable_irq

Synopsis

```
void libarm_enable_irq(void);
```

Description

This function enables IRQ interrupts by clearing the I bit in the CPSR register.

Note that this function modifies the CPSR register's control field and therefore will only work when the CPU is executing in a privileged operating mode.

Example:

```
// Enable IRQ interrupts
   libarm_enable_irq();
```

libarm_enable_irq_fiq

Synopsis

```
void libarm_enable_irq_fiq(void);
```

Description

libarm_enable_irq_fiq returns — The IRQ and FIQ enable state prior to enabling the IRQ and FIQ interrupts.

This function enables both IRQ and FIQ interrupts.

Note that this function modifies the CPSR register's control field and therefore will only work when the CPU is executing in a privileged operating mode.

Example

```
// Enable IRQ and FIQ interrupts
   libarm_enable_irq_fiq();
```

libarm_get_cpsr

Synopsis

```
unsigned long libarm_get_cpsr(void);
```

Description

libarm_get_cpsr returns — The value of the CPSR.

This function returns the value of the CPSR (Current Program Status Register).

libarm_isr_disable_irq

Synopsis

```
void libarm_isr_disable_irq(void);
```

Description

A call to **libarm_isr_enable_irq** must have been made prior to calling this function.

Note that this call should only be made from within an IRQ interrupt handler.

libarm_isr_enable_irq

Synopsis

```
void libarm_isr_enable_irq(void);
```

Description

ARM IRQ interrupts are automatically disabled on entry to an interrupt handler and subsequently re-enabled on exit. You can use **libarm_isr_enable_irq** to re-enable interrupts from within an interrupt handler so that higher-priority interrupts may interrupt the current interrupt handler.

This call must be accompanied with a call to **libarm_isr_disable_irq** prior to completion of the interrupt service routine.

Note that this function should only be called from within an IRQ interrupt handler and that calling this function changes the operating mode, and therefore the stack, so if it is being called from a C function you should not use any automatic variables within that function.

libarm_mmu_flat_initialise_level_1_table

Synopsis

```
void libarm_mmu_flat_initialise_level_1_table(void *translation_table);
```

Description

translation_table — A pointer to the start of the translation table.

This function creates a flat mapped (i.e. virtual addresses == physical addresses) level 1 MMU translation table at the location pointed to by **translation_table** (the translation table is 16BKytes in size).

Note that this function only initialises the translation table, it doesn't set the translation table base register.

libarm_mmu_flat_initialise_level_2_small_page_table

Synopsis

```
void libarm_mmu_flat_initialise_level_2_small_page_table(void *translation_table,
                                                         void *start,
                                                         size_t size,
                                                         void *coarse_page_tables);
```

Description

translation_table — A pointer to the start of the translation table.

start — A pointer to the start address of the address range.

size — The size of the address range in bytes.

coarse_page_tables — A pointer to the start address of the coarse page tables.

This function creates a level 2 small page table for the specified address range, it requires a level 1 translation table to be createdi using **libarm_mmu_flat_initialise_level_1_table** prior to calling.

libarm_mmu_flat_set_level_1_cacheable_region

Synopsis

```
void libarm_mmu_flat_set_level_1_cacheable_region(void *translation_table,
                                                  void *start,
                                                  size_t size);
```

Description

translation_table — A pointer to the start of the translation table.

start — A pointer to the start of the cacheable region.

size — The size of the cacheable region in bytes.

This function marks a region of memory described by level 1 section descriptors as cacheable, it requires a level 1 translation table to be created using **libarm_mmu_flat_initialise_level_1_table** prior to calling.

libarm_mmu_flat_set_level_2_small_page_cacheable_region

Synopsis

```
void libarm_mmu_flat_set_level_2_small_page_cacheable_region(void *translation_table,
                                                             void *start,
                                                             size_t size);
```

Description

translation_table — A pointer to the start of the translation table.

start — A pointer to the start address of the cacheable region.

size — The size of the cacheable region in bytes.

This function marks a region of memory described by level 2 small page table descriptors as cacheable, it requires a level 2 small page table table to be created using **libarm_mmu_flat_initialise_level_2_small_page_table** prior to calling.

libarm_restore_irq_fiq

Synopsis

```
void libarm_restore_irq_fiq(int disable_irq_fiq_return);
```

Description

disable_irq_fiq_return — The value returned from **libarm_disable_irq_fiq**.

This function restores the IRQ and FIQ enable state to the state it was in before a call to **libarm_disable_irq_fiq**.

Note that this function modifies the CPSR register's control field and therefore will only work when the CPU is executing in a privileged operating mode.

Example

```
int s;
   // Disable IRQ and FIQ interrupts
   s = libarm_disable_irq_fiq();
   \hdots
   // Restore IRQ and FIQ interrupts
   libarm_restore_irq_fiq(s);
```

libarm_run_dcc_port_server

Synopsis

```
void libarm_run_dcc_port_server(void);
```

Description

CrossWorks uses the ARM's debug communication channel to carry operations such as memory access, to do this a simple client server protocol is run over the channel. This function runs the debug communications channel server, it returns when the host terminates the server.

libarm_set_cpsr

Synopsis

```
void libarm_set_cpsr(unsigned long cpsr);
```

Description

cpsr — The value the CPSR should be set to.

This function sets the value of all fields of the CPSR (Current Program Status Register).

libarm_set_fiq

Synopsis

```
int libarm_set_fiq(int enable);
```

Description

enable — If non-zero FIQ interrupts will be enabled, otherwise they will be disabled.

libarm_set_fiq returns — The FIQ enable state prior to enabling the FIQ interrupt.

This function enables or disables FIQ interrupts. It modifies the CPSR register's control field and therefore will only work when the CPU is executing in a privileged operating mode.

Example

```
// Enable FIQ interrupts
   libarm_set_fiq(1);
   \hdots
   // Disable FIQ interrupts
   libarm_set_fiq(0);
```

libarm_set_irq

Synopsis

```
int libarm_set_irq(int enable);
```

Description

enable — If non-zero IRQ interrupts will be enabled, otherwise they will be disabled.

libarm_set_irq returns — The IRQ enable state prior to enabling the IRQ interrupt.

This function enables or disables IRQ interrupts. It modifies the CPSR register's control field and therefore will only work when the CPU is executing in a privileged operating mode.

Example

```
// Disable IRQ interrupts if enabled
   int en = libarm_set_irq(0);
   \hdots
   // Restore IRQ interrupts
   libarm_set_irq(en);
```

<limits.h>

API Summary

Long integer minimum and maximum values	
LONG_MAX	Maximum value of a long integer
LONG_MIN	Minimum value of a long integer
ULONG_MAX	Maximum value of an unsigned long integer
Character minimum and maximum values	
CHAR_MAX	Maximum value of a plain character
CHAR_MIN	Minimum value of a plain character
SCHAR_MAX	Maximum value of a signed character
SCHAR_MIN	Minimum value of a signed character
UCHAR_MAX	Maximum value of an unsigned char
Long long integer minimum and maximum values	
LLONG_MAX	Maximum value of a long long integer
LLONG_MIN	Minimum value of a long long integer
ULLONG_MAX	Maximum value of an unsigned long long integer
Short integer minimum and maximum values	
SHRT_MAX	Maximum value of a short integer
SHRT_MIN	Minimum value of a short integer
USHRT_MAX	Maximum value of an unsigned short integer
Integer minimum and maximum values	
INT_MAX	Maximum value of an integer
INT_MIN	Minimum value of an integer
UINT_MAX	Maximum value of an unsigned integer
Type sizes	
CHAR_BIT	Number of bits in a character

CHAR_BIT

Synopsis

```
#define CHAR_BIT   8
```

Description

CHAR_BIT is the number of bits for smallest object that is not a bit-field (byte).

CHAR_MAX

Synopsis

```
#define CHAR_MAX   255
```

Description

CHAR_MAX is the maximum value for an object of type **char**.

CHAR_MIN

Synopsis

```
#define CHAR_MIN   0
```

Description

CHAR_MIN is the minimum value for an object of type **char**.

INT_MAX

Synopsis

```
#define INT_MAX    2147483647
```

Description

INT_MAX is the maximum value for an object of type **int**.

INT_MIN

Synopsis

```
#define INT_MIN    (-2147483647 - 1)
```

Description

INT_MIN is the minimum value for an object of type **int**.

LLONG_MAX

Synopsis

```
#define LLONG_MAX  9223372036854775807LL
```

Description

LLONG_MAX is the maximum value for an object of type **long long int**.

LLONG_MIN

Synopsis

```
#define LLONG_MIN  (-9223372036854775807LL - 1)
```

Description

LLONG_MIN is the minimum value for an object of type **long long int**.

LONG_MAX

Synopsis

```
#define LONG_MAX   2147483647L
```

Description

LONG_MAX is the maximum value for an object of type **long int**.

LONG_MIN

Synopsis

```
#define LONG_MIN   (-2147483647L - 1)
```

Description

LONG_MIN is the minimum value for an object of type **long int**.

SCHAR_MAX

Synopsis

```
#define SCHAR_MAX  127
```

Description

SCHAR_MAX is the maximum value for an object of type **signed char**.

SCHAR_MIN

Synopsis

```
#define SCHAR_MIN  (-128)
```

Description

SCHAR_MIN is the minimum value for an object of type **signed char**.

SHRT_MAX

Synopsis

```
#define SHRT_MAX   32767
```

Description

SHRT_MAX is the minimum value for an object of type **short int**.

SHRT_MIN

Synopsis

```
#define SHRT_MIN   (-32767 - 1)
```

Description

SHRT_MIN is the minimum value for an object of type **short int**.

UCHAR_MAX

Synopsis

```
#define UCHAR_MAX  255
```

Description

UCHAR_MAX is the maximum value for an object of type **unsigned char**.

UINT_MAX

Synopsis

```
#define UINT_MAX   4294967295U
```

Description

UINT_MAX is the maximum value for an object of type **unsigned int**.

ULLONG_MAX

Synopsis

```
#define ULLONG_MAX 18446744073709551615ULL
```

Description

ULLONG_MAX is the maximum value for an object of type **unsigned long long int**.

ULONG_MAX

Synopsis

```
#define ULONG_MAX  4294967295UL
```

Description

ULONG_MAX is the maximum value for an object of type **unsigned long int**.

USHRT_MAX

Synopsis

```
#define USHRT_MAX  65535
```

Description

USHRT_MAX is the minimum value for an object of type **unsigned short int**.

<locale.h>

API Summary

Functions	
localeconv	Get current locale data
setlocale	Set Locale
Structures	
lconv	Formatting info for numeric values

lconv

Synopsis

```
typedef struct {
  char *currency_symbol;
  char *decimal_point;
  char frac_digits;
  char *grouping;
  char *int_curr_symbol;
  char int_frac_digits;
  char *mon_decimal_point;
  char *mon_grouping;
  char *mon_thousands_sep;
  char *negative_sign;
  char n_cs_precedes;
  char n_sep_by_space;
  char n_sign_posn;
  char *positive_sign;
  char p_cs_precedes;
  char p_sep_by_space;
  char p_sign_posn;
  char *thousands_sep;
} lconv;
```

Description

lconv structure holds formatting information on how numeric values are to be written.

Member	Description
current_symbol	Local currency symbol.
decimal_point	Decimal point separator.
frac_digits	Amount of fractional digits to the right of the decimal point for monetary quantities in the local format.
grouping	Specifies the amount of digits that form each of the groups to be separated by thousands_sep separator for non-monetary quantities.
int_curr_symbol	International currency symbol.
int_frac_digits	Amount of fractional digits to the right of the decimal point for monetary quantities in the international format.
mon_decimal_point	Decimal-point separator used for monetary quantities.
mon_grouping	Specifies the amount of digits that form each of the groups to be separated by mon_thousands_sep separator for monetary quantities.

mon_thousands_sep	Separators used to delimit groups of digits to the left of the decimal point for monetary quantities.
negative_sign	Sign to be used for negative monetary quantities.
n_cs_precedes	Whether the currency symbol should precede negative monetary quantities.
n_sep_by_space	Whether a space should appear between the currency symbol and negative monetary quantities.
n_sign_posn	Position of the sign for negative monetary quantities.
positive_sign	Sign to be used for nonnegative (positive or zero) monetary quantities.
p_cs_precedes	Whether the currency symbol should precede nonnegative (positive or zero) monetary quantities.
p_sep_by_space	Whether a space should appear between the currency symbol and nonnegative (positive or zero) monetary quantities.
p_sign_posn	Position of the sign for nonnegative (positive or zero) monetary quantities.
thousands_sep	Separators used to delimit groups of digits to the left of the decimal point for non-monetary quantities.

localeconv

Synopsis

```
localeconv(void);
```

Description

localeconv returns a pointer to a structure of type **lconv** with the corresponding values for the current locale filled in.

setlocale

Synopsis

```
char *setlocale(int category,
                const char *locale);
```

Description

setlocale sets the current locale. The **category** parameter can have the following values

Name	Locale affected
LC_ALL	Entire locale
LC_COLLATE	Affects **strcoll** and **strxfrm**
LC_CTYPE	Affects character handling
LC_MONETARY	Affects monetary formatting information
LC_NUMERIC	Affects decimal-point character in I/O and string formatting operations
LC_TIME	Affects **strftime**

The **locale** parameter contains the name of a C locale to set or if **NULL** is passed the current locale is not changed.

setlocale returns the name of the current locale.

Note

CrossWorks only supports the minimal "C" locale.

<math.h>

API Summary

Type Generic Macros	
fpclassify	Classify floating type
isfinite	Test for a finite value
isinf	Test for infinity
isnan	Test for NaN
isnormal	Test for a normal value
signbit	Test sign
Trigonometric functions	
cos	Compute cosine of a double
cosf	Compute cosine of a float
sin	Compute sine of a double
sinf	Compute sine of a float
tan	Compute tangent of a double
tanf	Compute tangent of a double
Inverse trigonometric functions	
acos	Compute inverse cosine of a double
acosf	Compute inverse cosine of a float
asin	Compute inverse sine of a double
asinf	Compute inverse sine of a float
atan	Compute inverse tangent of a double
atan2	Compute inverse tangent of a ratio of doubles
atan2f	Compute inverse tangent of a ratio of floats
atanf	Compute inverse tangent of a float
Exponential and logarithmic functions	
cbrt	Compute cube root of a double
cbrtf	Compute cube root of a float
exp	Compute exponential of a double
expf	Compute exponential of a float
frexp	Set exponent of a double
frexpf	Set exponent of a float
ldexp	Adjust exponent of a double

ldexpf	Adjust exponent of a float
log	Compute natural logarithm of a double
log10	Compute common logarithm of a double
log10f	Compute common logarithm of a float
logf	Compute natural logarithm of a float
pow	Raise a double to a power
powf	Raise a float to a power
scalbn	Scale a double
scalbnf	Scale a float
sqrt	Compute square root of a double
sqrtf	Compute square root of a float
Remainder functions	
fmod	Compute remainder after division of two doubles
fmodf	Compute remainder after division of two floats
modf	Break a double into integer and fractional parts
modff	Break a float into integer and fractional parts
Nearest integer functions	
ceil	Compute smallest integer not greater than a double
ceilf	Compute smallest integer not greater than a float
floor	Compute largest integer not greater than a float
floorf	Compute largest integer not greater than a float
Absolute value functions	
fabs	Compute absolute value of a double
fabsf	Compute absolute value of a float
hypot	Compute complex magnitude of two doubles
hypotf	Compute complex magnitude of two floats
Maximum, minimum, and positive difference functions	
fmax	Compute maximum of two doubles
fmaxf	Compute maximum of two floats
fmin	Compute minimum of two doubles
fminf	Compute minimum of two floats
Hyperbolic functions	
cosh	Compute hyperbolic cosine of a double
coshf	Compute hyperbolic cosine of a float
sinh	Compute hyperbolic sine of a double

sinhf	Compute hyperbolic sine of a float
tanh	Compute hyperbolic tangent of a double
tanhf	Compute hyperbolic tangent of a float
Inverse hyperbolic functions	
acosh	Compute inverse hyperbolic cosine of a double
acoshf	Compute inverse hyperbolic cosine of a float
asinh	Compute inverse hyperbolic sine of a double
asinhf	Compute inverse hyperbolic sine of a float
atanh	Compute inverse hyperbolic tangent of a double
atanhf	Compute inverse hyperbolic tangent of a float

acos

Synopsis

```
double acos(double x);
```

Description

acos returns the principal value, in radians, of the inverse circular cosine of **x**. The principal value lies in the interval [0, PI] radians.

If |**x**| > 1, **errno** is set to **EDOM** and **acos** returns **HUGE_VAL**.

If **x** is NaN, **acos** returns **x**. If |**x**| > 1, **acos** returns NaN.

acosf

Synopsis

```
float acosf(float x);
```

Description

acosf returns the principal value, in radians, of the inverse circular cosine of **x**. The principal value lies in the interval [0, PI] radians.

If |**a**| 1, **errno** is set to **EDOM** and **acosf** returns **HUGE_VAL**.

If **x** is NaN, **acosf** returns **x**. If |**x**| > 1, **acosf** returns NaN.

acosh

Synopsis

```
double acosh(double x);
```

Description

acosh returns the non-negative inverse hyperbolic cosine of **x**.

acosh(*x*) is defined as **log**(*x* + **sqrt**(*x*^2 − 1)), assuming completely accurate computation.

If **x** < 1, **errno** is set to **EDOM** and **acosh** returns **HUGE_VAL**.

If **x** < 1, **acosh** returns NaN.
If **x** is NaN, **acosh** returns NaN.

acoshf

Synopsis

```
float acoshf(float x);
```

Description

acoshf returns the non-negative inverse hyperbolic cosine of **x**.

acosh(*x*) is defined as **log**(*x* + **sqrt**(*x*^2 − 1)), assuming completely accurate computation.

If **x** < 1, **errno** is set to **EDOM** and **acoshf** returns **HUGE_VALF**.

If **x** < 1, **acoshf** returns NaN.
If **x** is NaN, **acoshf** returns that NaN.

asin

Synopsis

```
double asin(double x);
```

Description

asin returns the principal value, in radians, of the inverse circular sine of **x**. The principal value lies in the interval [−½π, +½π] radians.

If |**x**| > 1, **errno** is set to **EDOM** and **asin** returns **HUGE_VAL**.

If **x** is NaN, **asin** returns **x**. If |**x**| > 1, **asin** returns NaN.

asinf

Synopsis

```
float asinf(float x);
```

Description

asinf returns the principal value, in radians, of the inverse circular sine of **val**. The principal value lies in the interval [−½π, +½π] radians.

If |**x**| > 1, **errno** is set to **EDOM** and **asinf** returns **HUGE_VALF**.

If **x** is NaN, **asinf** returns **x**. If |**x**| > 1, **asinf** returns NaN.

asinh

Synopsis

```
double asinh(double x);
```

Description

asinh calculates the hyperbolic sine of **x**.

If |**x**| > ~709.782, **errno** is set to **EDOM** and **asinh** returns **HUGE_VAL**.

If **x** is +∞, −∞, or NaN, **asinh** returns |**x**|. If |**x**| > ~709.782, **asinh** returns +∞ or −∞ depending upon the sign of **x**.

asinhf

Synopsis

```
float asinhf(float x);
```

Description

asinhf calculates the hyperbolic sine of **x**.

If |**x**| > ~88.7228, **errnois** set to **EDOM** and **asinhf** returns **HUGE_VALF**.

If **x** is +∞, −∞, or NaN, **asinhf** returns |**x**|. If |**x**| > ~88.7228, **asinhf** returns +∞ or −∞ depending upon the sign of **x**.

atan

Synopsis

```
double atan(double x);
```

Description

atan returns the principal value, in radians, of the inverse circular tangent of **x**. The principal value lies in the interval [−½π, +½π] radians.

atan2

Synopsis

```
double atan2(double x,
             double y);
```

Description

atan2 returns the value, in radians, of the inverse circular tangent of **y** divided by **x** using the signs of **x** and **y** to compute the quadrant of the return value. The principal value lies in the interval [−½π/2, +½π] radians. If **x** = **y** = 0, **errno** is set to **EDOM** and **atan2** returns **HUGE_VAL**.

atan2(**x**, NaN) is NaN.
atan2(NaN, **x**) is NaN.
atan2(±0, +(anything but NaN)) is ±0.
atan2(±0, −(anything but NaN)) is ±π.
atan2(±(anything but 0 and NaN), 0) is ±½π.
atan2(±(anything but ∞ and NaN), +∞) is ±0.
atan2(±(anything but ∞ and NaN), −∞) is ±π.
atan2(±∞, +∞) is ±¼π.
atan2(±∞, −∞) is ±¾π.
atan2(±∞, (anything but 0, NaN, and ∞)) is ±½π.

atan2f

Synopsis

```
float atan2f(float x,
             float y);
```

Description

atan2f returns the value, in radians, of the inverse circular tangent of **y** divided by **x** using the signs of **x** and **y** to compute the quadrant of the return value. The principal value lies in the interval [−½π/2, +½π] radians.

If **x** = **y** = 0, **errno** is set to **EDOM** and **atan2f** returns **HUGE_VALF**.

atan2f(**x**, NaN) is NaN.
atan2f(NaN, **x**) is NaN.
atan2f(±0, +(anything but NaN)) is ±0.
atan2f(±0, −(anything but NaN)) is ±π.
atan2f(±(anything but 0 and NaN), 0) is ±½π.
atan2f(±(anything but ∞ and NaN), +∞) is ±0.
atan2f(±(anything but ∞ and NaN), −∞) is ±π.
atan2f(±∞, +∞) is ±¼π.
atan2f(±∞, −∞) is ±¾π.
atan2f(±∞, (anything but 0, NaN, and ∞)) is ±½π.

atanf

Synopsis

```
float atanf(float x);
```

Description

atanf returns the principal value, in radians, of the inverse circular tangent of **x**. The principal value lies in the interval [−½π, +½π] radians.

atanh

Synopsis

```
double atanh(double x);
```

Description

atanh returns the inverse hyperbolic tangent of **x**.

If |**x**| ≥ 1, **errno** is set to **EDOM** and **atanh** returns **HUGE_VAL**.

If |**x**| > 1 **atanh** returns NaN.
If **x** is NaN, **atanh** returns that NaN.
If **x** is 1, **atanh** returns ∞.
If **x** is −1, **atanh** returns −∞.

atanhf

Synopsis

```
float atanhf(float x);
```

Description

atanhf returns the inverse hyperbolic tangent of **val**.

If |**x**| ≥ 1, **errno** is set to **EDOM** and **atanhf** returns **HUGE_VALF**.

If |**val**| > 1 **atanhf** returns NaN. If **val** is NaN, **atanhf** returns that NaN. If **val** is 1, **atanhf** returns ∞. If **val** is −1, **atanhf** returns −∞.

cbrt

Synopsis

```
double cbrt(double x);
```

Description

cbrt computes the cube root of **x**.

cbrtf

Synopsis

```
float cbrtf(float x);
```

Description

cbrtf computes the cube root of **x**.

ceil

Synopsis

```
double ceil(double x);
```

Description

ceil computes the smallest integer value not less than **x**.

ceil (±0) is ±0. **ceil** (± ∞) is ± ∞.

ceilf

Synopsis

```
float ceilf(float x);
```

Description

ceilf computes the smallest integer value not less than **x**.

ceilf (±0) is ±0. **ceilf** (± ∞) is ± ∞.

COS

Synopsis

```
double cos(double x);
```

Description

cos returns the radian circular cosine of **x**.

If |**x**| > 10^9, **errno** is set to **EDOM** and **cos** returns **HUGE_VAL**.

If **x** is NaN, **cos** returns **x**. If |**x**| is ∞, **cos** returns NaN.

cosf

Synopsis

```
float cosf(float x);
```

Description

cosf returns the radian circular cosine of x.

If |**x**| > 10^9, **errno** is set to **EDOM** and **cosf** returns **HUGE_VALF**.

If **x** is NaN, **cosf** returns **x**. If |**x**| is ∞, **cosf** returns NaN.

cosh

Synopsis

```
double cosh(double x);
```

Description

cosh calculates the hyperbolic cosine of **x**.

If |**x**| > ~709.782, **errno** is set to **EDOM** and **cosh** returns **HUGE_VAL**.

If **x** is +∞, −∞, or NaN, **cosh** returns |**x**|.> If |**x**| > ~709.782, **cosh** returns +∞ or −∞ depending upon the sign of **x**.

coshf

Synopsis

```
float coshf(float x);
```

Description

coshf calculates the hyperbolic sine of **x**.

If |**x**| > ~88.7228, **errno** is set to **EDOM** and **coshf** returns **HUGE_VALF**.

If **x** is +∞, −∞, or NaN, **coshf** returns |**x**|.
If |**x**| > ~88.7228, **coshf** returns +∞ or −∞ depending upon the sign of **x**.

exp

Synopsis

```
double exp(double x);
```

Description

exp computes the base-*e* exponential of **x**.

If |**x**| > ~709.782, **errno** is set to **EDOM** and **exp** returns **HUGE_VAL**.

If **x** is NaN, **exp** returns NaN.
If **x** is ∞, **exp** returns ∞.
If **x** is −∞, **exp** returns 0.

expf

Synopsis

```
float expf(float x);
```

Description

expf computes the base-*e* exponential of **x**.

If |**x**| > ~88.722, **errno** is set to **EDOM** and **expf** returns **HUGE_VALF**. If **x** is NaN, **expf** returns NaN.
If **x** is ∞, **expf** returns ∞.
If **x** is −∞, **expf** returns 0.

fabs

Synopsis

```
double fabs(double x);
```

fabsf

Synopsis

```
float fabsf(float x);
```

Description

fabsf computes the absolute value of the floating-point number **x**.

floor

Synopsis

```
double floor(double);
```

floor computes the largest integer value not greater than **x**.

floor (±0) is ±0. **floor** (±∞) is ±∞.

floorf

Synopsis

```
float floorf(float);
```

floorf computes the largest integer value not greater than **x**.

floorf(±0) is ±0. **floorf**(±∞) is ±∞.

fmax

Synopsis

```
double fmax(double x,
            double y);
```

Description

fmax determines the maximum of **x** and **y**.

fmax (NaN, **y**) is **y**. **fmax** (**x**, NaN) is **x**.

fmaxf

Synopsis

```
float fmaxf(float x,
            float y);
```

Description

fmaxf determines the maximum of **x** and **y**.

fmaxf (NaN, **y**) is **y**. **fmaxf**(**x**, NaN) is **x**.

fmin

Synopsis

```
double fmin(double x,
            double y);
```

Description

fmin determines the minimum of **x** and **y**.

fmin (NaN, **y**) is **y**. **fmin** (x, NaN) is **x**.

fminf

Synopsis

```
float fminf(float x,
            float y);
```

Description

fminf determines the minimum of **x** and **y**.

fminf (NaN, **y**) is **y**. **fminf** (**x**, NaN) is **x**.

fmod

Synopsis

```
double fmod(double x,
            double y);
```

Description

fmod computes the floating-point remainder of **x** divided by **y**. #b #this returns the value **x** − *n* **y**, for some integer *n* such that, if **y** is nonzero, the result has the same sign as **x** and magnitude less than the magnitude of **y**.

If **y** = 0, **fmod** returns zero and **errno** is set to **EDOM**.

fmod (± 0, **y**) is ± 0 for **y** not zero.
fmod (∞, **y**) is NaN.
fmod (**x**, 0) is NaN.
fmod (x, ± ∞) is **x** for **x** not infinite.

fmodf

Synopsis

```
float fmodf(float x,
            float y);
```

Description

fmodf computes the floating-point remainder of **x** divided by **y**. **fmodf** returns the value **x** – *n* **y**, for some integer *n* such that, if **y** is nonzero, the result has the same sign as **x** and magnitude less than the magnitude of **y**.

If **y** = 0, **fmodf** returns zero and **errno** is set to **EDOM**.

fmodf (± 0, **y**) is ± 0 for **y** not zero.
fmodf (∞, **y**) is NaN.
fmodf (**x**, 0) is NaN.
fmodf (x, ± ∞) is **x** for **x** not infinite.

fpclassify

Synopsis

```
#define fpclassify(x) (sizeof(x) == sizeof(float) ? __float32_classify(x) :
 __float64_classify(x))
```

Description

fpclassify macro shall classify its argument value as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. The fpclassify macro returns the value of the number classification macro one of

- FP_ZERO
- FP_SUBNORMAL
- FP_NORMAL
- FP_INFINITE
- FP_NAN

frexp

Synopsis

```
double frexp(double x,
             int *exp);
```

Description

frexp breaks a floating-point number into a normalized fraction and an integral power of 2.

frexp stores power of two in the **int** object pointed to by **exp** and returns the value **x**, such that **x** has a magnitude in the interval [1/2, 1) or zero, and value equals **x** * 2^**exp**.

If **x** is zero, both parts of the result are zero.

If **x** is ∞ or NaN, **frexp** returns **x** and stores zero into the int object pointed to by **exp**.

frexpf

Synopsis

```
float frexpf(float x,
             int *exp);
```

Description

frexpf breaks a floating-point number into a normalized fraction and an integral power of 2.

frexpf stores power of two in the **int** object pointed to by **frexpf** and returns the value **x**, such that **x** has a magnitude in the interval [½, 1) or zero, and value equals **x** * 2^**exp**.

If **x** is zero, both parts of the result are zero.

If **x** is ∞ or NaN, **frexpf returns x** and stores zero into the int object pointed to by **exp**.

hypot

Synopsis

```
double hypot(double x,
             double y);
```

Description

hypot computes the square root of the sum of the squares of **x** and **y**, **sqrt**(**x*****x** + **y*****y**), without undue overflow or underflow. If **x** and **y** are the lengths of the sides of a right-angled triangle, then **hypot** computes the length of the hypotenuse.

If **x** or **y** is +∞ or −∞, **hypot** returns ∞.
If **x** or **y** is NaN, **hypot** returns NaN.

hypotf

Synopsis

```
float hypotf(float x,
             float y);
```

Description

hypotf computes the square root of the sum of the squares of **x** and **y**, **sqrtf**(**x*****x** + **y*****y**), without undue overflow or underflow. If **x** and **y** are the lengths of the sides of a right-angled triangle, then **hypotf** computes the length of the hypotenuse.

If **x** or **y** is +∞ or −∞, **hypotf** returns ∞. If **x** or **y** is NaN, **hypotf** returns NaN.

isfinite

Synopsis

```
#define isfinite(x) (sizeof(x) == sizeof(float) ? __float32_isfinite(x) :
 __float64_isfinite(x))
```

Description

isfinite determines whether **x** is a finite value (zero, subnormal, or normal, and not infinite or NaN). The isfinite macro returns a non-zero value if and only if its argument has a finite value.

isinf

Synopsis

```
#define isinf(x) (sizeof(x) == sizeof(float) ? __float32_isinf(x) : __float64_isinf(x))
```

Description

isinf determines whether **x** is an infinity (positive or negative). The determination is based on the type of the argument.

isnan

Synopsis

```
#define isnan(x) (sizeof(x) == sizeof(float) ? __float32_isnan(x) : __float64_isnan(x))
```

Description

isnan determines whether **x** is a NaN. The determination is based on the type of the argument.

isnormal

Synopsis

```
#define isnormal(x) (sizeof(x) == sizeof(float) ? __float32_isnormal(x) :
 __float64_isnormal(x))
```

Description

isnormal determines whether **x** is a normal value (zero, subnormal, or normal, and not infinite or NaN). The isnormal macro returns a non-zero value if and only if its argument has a normal value.

ldexp

Synopsis

```
double ldexp(double x,
             int exp);
```

Description

ldexp multiplies a floating-point number by an integral power of 2.

ldexp returns **x** * 2^**exp**.

If the result overflows, **errno** is set to **ERANGE** and **ldexp** returns **HUGE_VALF**.

If **x** is ∞ or NaN, **ldexp** returns **x**. If the result overflows, **ldexp** returns ∞.

ldexpf

Synopsis

```
float ldexpf(float x,
             int exp);
```

Description

ldexpf multiplies a floating-point number by an integral power of 2.

ldexpf returns **x** * 2^**exp**. If the result overflows, **errno** is set to **ERANGE** and **ldexpf** returns **HUGE_VALF**.

If **x** is ∞ or NaN, **ldexpf** returns **x**. If the result overflows, **ldexpf** returns ∞.

log

Synopsis

```
double log(double x);
```

Description

log computes the base-*e* logarithm of **x**.

If **x** = 0, **errno** is set to **ERANGE** and **log** returns −**HUGE_VAL**. If **x** < 0, **errno** is set to **EDOM** and **log** returns −**HUGE_VAL**.

If **x** < 0 or **x** = −∞, **log** returns NaN.
If **x** = 0, **log** returns −∞.
If **x** = ∞, **log** returns ∞.
If **x** = NaN, **log** returns **x**.

log10

Synopsis

```
double log10(double x);
```

Description

log10 computes the base-10 logarithm of **x**.

If **x** = 0, **errno** is set to **ERANGE** and **log10** returns −**HUGE_VAL**. If **x** < 0, **errno** is set to **EDOM** and **log10** returns −**HUGE_VAL**.

If **x** < 0 or **x** = −∞, **log10** returns NaN.
If **x** = 0, **log10** returns −∞.
If **x** = ∞, **log10** returns ∞.
If **x** = NaN, **log10** returns **x**.

log10f

Synopsis

```
float log10f(float x);
```

Description

log10f computes the base-10 logarithm of **x**.

If **x** = 0, **errno** is set to **ERANGE** and **log10f** returns −**HUGE_VALF**. If **x** < 0, **errno** is set to **EDOM** and **log10f** returns −**HUGE_VALF**.

If **x** < 0 or **x** = −∞, **log10f** returns NaN.
If **x** = 0, **log10f** returns −∞.
If **x** = ∞, **log10f** returns ∞.
If **x** = NaN, **log10f** returns **x**.

logf

Synopsis

```
float logf(float x);
```

Description

logf computes the base-*e* logarithm of **x**.

If **x** = 0, **errno** is set to **ERANGE** and **logf** returns −**HUGE_VALF**. If **x** < 0, **errno** is set to **EDOM** and **logf** returns −**HUGE_VALF**.

If **x** < 0 or **x** = −∞, **logf** returns NaN.
If **x** = 0, **logf** returns −∞.
If **x** = ∞, **logf** returns ∞.
If **x** = NaN, **logf** returns **x**.

modf

Synopsis

```
double modf(double x,
            double *iptr);
```

Description

modf breaks **x** into integral and fractional parts, each of which has the same type and sign as **x**.

The integral part (in floating-point format) is stored in the object pointed to by **iptr** and **modf** returns the signed fractional part of **x**.

modff

Synopsis

```
float modff(float x,
            float *iptr);
```

Description

modff breaks **x** into integral and fractional parts, each of which has the same type and sign as **x**.

The integral part (in floating-point format) is stored in the object pointed to by **iptr** and **modff** returns the signed fractional part of **x**.

pow

Synopsis

```
double pow(double x,
           double y);
```

Description

pow computes **x** raised to the power **y**.

If **x** < 0 and **y** ≤ 0, **errno** is set to **EDOM** and **pow** returns −**HUGE_VAL**. If **x** ≤ 0 and **y** is not an integer value, **errno** is set to **EDOM** and **pow** returns −**HUGE_VAL**.

If **y** = 0, **pow** returns 1.
If **y** = 1, **pow** returns **x**.
If **y** = NaN, **pow** returns NaN.
If **x** = NaN and **y** is anything other than 0, **pow** returns NaN.
If **x** < −1 or 1 < **x**, and **y** = +∞, **pow** returns +∞.
If **x** < −1 or 1 < **x**, and **y** = −∞, **pow** returns 0.
If −1 < **x** < 1 and **y** = +∞, **pow** returns +0.
If −1 < **x** < 1 and **y** = −∞, **pow** returns +∞.
If **x** = +1 or **x** = −1 and **y** = +∞ or **y** = −∞, **pow** returns NaN.
If **x** = +0 and **y** > 0 and **y** ≠ NaN, **pow** returns +0.
If **x** = −0 and **y** > 0 and **y** ≠ NaN or **y** not an odd integer, **pow** returns +0.
If **x** = +0 and **y** and **y** ≠ NaN, **pow** returns +∞.
If **x** = −0 and **y** > 0 and **y** ≠ NaN or **y** not an odd integer, **pow** returns +∞.
If **x** = −0 and **y** is an odd integer, **pow** returns −0.
If **x** = +∞ and **y** > 0 and **y** ≠ NaN, **pow** returns +∞.
If **x** = +∞ and **y** < 0 and **y** ≠ NaN, **pow** returns +0.
If **x** = −∞, **pow** returns **pow**(−0, **y**)
If **x** < 0 and **x** ≠ ∞ and **y** is a non-integer, **pow** returns NaN.

powf

Synopsis

```
float powf(float,
           float);
```

Description

powf computes **x** raised to the power **y**.

If **x** < 0 and **y** ≤ 0, **errno**. is set to **EDOM** and **powf** returns −**HUGE_VALF**. If **x** ≤ 0 and **y** is not an integer value, **errno** is set to **EDOM** and **pow** returns −**HUGE_VALF**.

If **y** = 0, **powf** returns 1.
If **y** = 1, **powf** returns **x**.
If **y** = NaN, **powf** returns NaN.
If **x** = NaN and **y** is anything other than 0, **powf** returns NaN.
If **x** < −1 or 1 < **x**, and **y** = +∞, **powf** returns +∞.
If **x** < −1 or 1 < **x**, and **y** = −∞, **powf** returns 0.
If −1 < **x** < 1 and **y** = +∞, **powf** returns +0.
If −1 < **x** < 1 and **y** = −∞, **powf** returns +∞.
If **x** = +1 or **x** = −1 and **y** = +∞ or **y** = −∞, **powf** returns NaN.
If **x** = +0 and **y** > 0 and **y** ≠ NaN, **powf** returns +0.
If **x** = −0 and **y** > 0 and **y** ≠ NaN or **y** not an odd integer, **powf** returns +0.
If **x** = +0 and **y** and **y** ≠ NaN, **powf** returns +∞.
If **x** = −0 and **y** > 0 and **y** ≠ NaN or **y** not an odd integer, **powf** returns +∞.
If **x** = −0 and **y** is an odd integer, **powf** returns −0.
If **x** = +∞ and **y** > 0 and **y** ≠ NaN, **powf** returns +∞.
If **x** = +∞ and **y** < 0 and **y** ≠ NaN, **powf** returns +0.
If **x** = −∞, **powf** returns **powf**(−0, **y**)
If **x** < 0 and **x** ≠ ∞ and **y** is a non-integer, **powf** returns NaN.

scalbn

Synopsis

```
double scalbn(double x,
              int exp);
```

Description

scalbn multiplies a floating-point number by an integral power of **DBL_RADIX**.

As floating-point arithmetic conforms to IEC 60559, **DBL_RADIX** is 2 and **scalbn** is (in this implementation) identical to **ldexp**.

scalbn returns **x** * **DBL_RADIX^exp**.

If the result overflows, **errno** is set to **ERANGE** and **scalbn** returns **HUGE_VAL**.

If **x** is ∞ or NaN, **scalbn** returns **x**.
If the result overflows, **scalbn** returns ∞.

See Also

ldexp

scalbnf

Synopsis

```
float scalbnf(float x,
              int exp);
```

Description

scalbnf multiplies a floating-point number by an integral power of **FLT_RADIX**.

As floating-point arithmetic conforms to IEC 60559, **FLT_RADIX** is 2 and **scalbnf** is (in this implementation) identical to **ldexpf**.

scalbnf returns **x** * **FLT_RADIX** ^**exp**.

If the result overflows, **errno** is set to **ERANGE** and **scalbnf** returns **HUGE_VALF**.

If **x** is ∞ or NaN, **scalbnf** returns **x**. If the result overflows, **scalbnf** returns ∞.

See Also

ldexpf

signbit

Synopsis

```
#define signbit(x) (sizeof(x) == sizeof(float) ? __float32_signbit(x) : __float64_signbit(x))
```

Description

signbit macro shall determine whether the sign of its argument value is negative. The signbit macro returns a non-zero value if and only if its argument value is negative.

sin

Synopsis

```
double sin(double x);
```

Description

sin returns the radian circular sine of **x**.

If |**x**| > 10^9, **errno** is set to **EDOM** and **sin** returns **HUGE_VAL**.

sin returns **x** if **x** is NaN. **sin** returns NaN if |**x**| is ∞.

sinf

Synopsis

```
float sinf(float x);
```

Description

sinf returns the radian circular sine of **x**.

If |**x**| > 10^9, **errno** is set to **EDOM** and **sinf** returns **HUGE_VALF**.

sinf returns **x** if **x** is NaN. **sinf** returns NaN if |**x**| is ∞.

sinh

Synopsis

```
double sinh(double x);
```

Description

sinh calculates the hyperbolic sine of **x**.

If |**x**| .782, **errno** is set to **EDOM** and **sinh** returns **HUGE_VAL**.

If **x** is +∞, −∞, or NaN, **sinh** returns |**x**|. If |**x**| > ~709.782, **sinh** returns +∞ or −∞ depending upon the sign of **x**.

sinhf

Synopsis

```
float sinhf(float x);
```

Description

sinhf calculates the hyperbolic sine of **x**.

If |**x**| > ~88.7228, **errno** is set to **EDOM** and **sinhf** returns **HUGE_VALF**.

If **x** is +∞, −∞, or NaN, **sinhf** returns |**x**|. If |**x**| > ~88.7228, **sinhf** returns +∞ or −∞ depending upon the sign of **x**.

sqrt

Synopsis

```
double sqrt(double x);
```

Description

sqrt computes the nonnegative square root of **x**. C90 and C99 require that a domain error occurs if the argument is less than zero. CrossWorks C deviates and always uses IEC 60559 semantics.

If **x** is +0, **sqrt** returns +0.
If **x** is −0, **sqrt** returns −0.
If **x** is ∞, **sqrt** returns ∞.
If **x** < 0, **sqrt** returns NaN.
If **x** is NaN, **sqrt** returns that NaN.

sqrtf

Synopsis

```
float sqrtf(float x);
```

Description

sqrtf computes the nonnegative square root of **x**. C90 and C99 require that a domain error occurs if the argument is less than zero. CrossWorks C deviates and always uses IEC 60559 semantics.

If **x** is +0, **sqrtf** returns +0.
If **x** is −0, **sqrtf** returns −0.
If **x** is ∞, **sqrtf** returns ∞.
If **x** < 0, **sqrtf** returns NaN.
If **x** is NaN, **sqrtf** returns that NaN.

tan

Synopsis

```
double tan(double x);
```

Description

tan returns the radian circular tangent of **x**.

If |**x**| > 10^9, **errno** is set to **EDOM** and **tan** returns **HUGE_VAL**.

If **x** is NaN, **tan** returns **x**. If |**x**| is ∞, **tan** returns NaN.

tanf

Synopsis

```
float tanf(float x);
```

Description

tanf returns the radian circular tangent of **x**.

If |**x**| > 10^9, **errno** is set to **EDOM** and **tanf** returns **HUGE_VALF**.

If **x** is NaN, tanf returns **x**. If |**x**| is ∞, tanf returns NaN.

tanh

Synopsis

```
double tanh(double x);
```

Description

tanh calculates the hyperbolic tangent of **x**.

If **x** is NaN, **tanh** returns NaN.

tanhf

Synopsis

```
float tanhf(float x);
```

Description

tanhf calculates the hyperbolic tangent of **x**.

If **x** is NaN, **tanhf** returns NaN.

<setjmp.h>

API Summary

Functions	
longjmp	Restores the saved environment
setjmp	Save calling environment for non-local jump

longjmp

Synopsis

```
void longjmp(jmp_buf env,
             int val);
```

Description

longjmp restores the environment saved by **setjmp** in the corresponding **env** argument. If there has been no such invocation, or if the function containing the invocation of **setjmp** has terminated execution in the interim, the behavior of **longjmp** is undefined.

After **longjmp** is completed, program execution continues as if the corresponding invocation of **setjmp** had just returned the value specified by **val**.

Note

longjmp cannot cause **setjmp** to return the value 0; if **val** is 0, **setjmp** returns the value 1.

Objects of automatic storage allocation that are local to the function containing the invocation of the corresponding **setjmp** that do not have **volatile** qualified type and have been changed between the **setjmp** invocation and **this** call are indeterminate.

setjmp

Synopsis

```
int setjmp(jmp_buf env);
```

Description

setjmp saves its calling environment in the **env** for later use by the **longjmp** function.

On return from a direct invocation **setjmp** returns the value zero. On return from a call to the **longjmp** function, the **setjmp** returns a nonzero value determined by the call to **longjmp**.

The environment saved by a call to **setjmp** consists of information sufficient for a call to the **longjmp** function to return execution to the correct block and invocation of that block, were it called recursively.

<stdarg.h>

API Summary

Macros	
va_arg	Get variable argument value
va_copy	Copy var args
va_end	Finish access to variable arguments
va_start	Start access to variable arguments

va_arg

Synopsis

```
type va_arg(va_list ap,
            type);
```

Description

va_arg expands to an expression that has the specified type and the value of the **type** argument. The **ap** parameter must have been initialized by **va_start** or **va_copy**, without an intervening invocation of **va_end**. You can create a pointer to a **va_list** and pass that pointer to another function, in which case the original function may make further use of the original list after the other function returns.

Each invocation of the **va_arg** macro modifies **ap** so that the values of successive arguments are returned in turn. The parameter type must be a type name such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to **type**.

If there is no actual next argument, or if type is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior of **va_arg** is undefined, except for the following cases:

- one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;
- one type is pointer to **void** and the other is a pointer to a character type.

The first invocation of the **va_arg** macro after that of the **va_start** macro returns the value of the argument after that specified by **parmN**. Successive invocations return the values of the remaining arguments in succession.

va_copy

Synopsis

```
void va_copy(va_list dest,
             val_list src);
```

Description

va_copy initializes **dest** as a copy of **src**, as if the **va_start** macro had been applied to **dest** followed by the same sequence of uses of the **va_arg** macro as had previously been used to reach the present state of **src**. Neither the **va_copy** nor **va_start** macro shall be invoked to reinitialize **dest** without an intervening invocation of the **va_end** macro for the same **dest**.

va_end

Synopsis

```
void va_end(va_list ap);
```

Description

va_end indicates a normal return from the function whose variable argument list **ap** was initialised by **va_start** or **va_copy**. The **va_end** macro may modify **ap** so that it is no longer usable without being reinitialized by **va_start** or **va_copy**. If there is no corresponding invocation of **va_start** or **va_copy**, or if **va_end** is not invoked before the return, the behavior is undefined.

va_start

Synopsis

```
void va_start(va_list ap,
              paramN);
```

Description

va_start initializes **ap** for subsequent use by the **va_arg** and **va_end** macros.

The parameter **parmN** is the identifier of the last fixed parameter in the variable parameter list in the function definition (the one just before the **', ...').**

The behaviour of **va_start** and **va_arg** is undefined if the parameter **parmN** is declared with the **register** storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions.

va_start must be invoked before any access to the unnamed arguments.

va_start and **va_copy** must not be invoked to reinitialize **ap** without an intervening invocation of the **va_end** macro for the same **ap**.

<stddef.h>

API Summary

Macros	
NULL	NULL pointer
offsetof	offsetof
Types	
ptrdiff_t	ptrdiff_t type
size_t	size_t type
wchar_t	wchar_t type

NULL

Synopsis

```
#define NULL 0
```

Description

NULL is the null pointer constant.

offsetof

Synopsis

```
#define offsetof(type, member)
```

Description

offsetof returns the offset in bytes to the structure **member**, from the beginning of its structure **type**.

ptrdiff_t

Synopsis

```
typedef __PTRDIFF_T ptrdiff_t;
```

Description

ptrdiff_t is the signed integral type of the result of subtracting two pointers.

size_t

Synopsis

```
typedef __SIZE_T size_t;
```

Description

size_t is the unsigned integral type returned by the sizeof operator.

wchar_t

Synopsis

```
typedef __WCHAR_T wchar_t;
```

Description

wchar_t is the wide character type.

<stdio.h>

API Summary

Character and string I/O functions	
getchar	Read a character from standard input
gets	Read a string from standard input
putchar	Write a character to standard output
puts	Write a string to standard output
Formatted output functions	
printf	Write formatted text to standard output
snprintf	Write formatted text to a string with truncation
sprintf	Write formatted text to a string
vprintf	Write formatted text to standard output using variable argument context
vsnprintf	Write formatted text to a string with truncation using variable argument context
vsprintf	Write formatted text to a string using variable argument context
Formatted input functions	
scanf	Read formatted text from standard input
sscanf	Read formatted text from string
vscanf	Read formatted text from standard using variable argument context
vsscanf	Read formatted text from a string using variable argument context

getchar

Synopsis

```
int getchar(void);
```

Description

getchar reads a single character from the standard input stream.

If the stream is at end-of-file or a read error occurs, **getchar** returns **EOF**.

gets

Synopsis

```
char *gets(char *s);
```

Description

gets reads characters from standard input into the array pointed to by **s** until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

gets returns **s** if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and **gets** returns a null pointer. If a read error occurs during the operation, the array contents are indeterminate and **gets** returns a null pointer.

printf

Synopsis

```
int printf(const char *format,
           ...);
```

Description

printf writes to the standard output stream using **putchar**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

printf returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Formatted output control strings

The format is composed of zero or more directives: ordinary characters (not '%', which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

Each conversion specification is introduced by the character '%'. After the '%' the following appear in sequence:

- Zero or more *flags* (in any order) that modify the meaning of the conversion specification.
- An optional *minimum field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag has been given) to the field width. The field width takes the form of an asterisk '*' or a decimal integer.
- An optional precision that gives the minimum number of digits to appear for the 'd', 'i', 'o', 'u', 'x', and 'X' conversions, the number of digits to appear after the decimal-point character for 'e', 'E', 'f', and 'F' conversions, the maximum number of significant digits for the 'g' and 'G' conversions, or the maximum number of bytes to be written for 's' conversions. The precision takes the form of a period '.' followed either by an asterisk '*' or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an int argument supplies the field width or precision. The arguments specifying field width, or precision, or both, must appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a '-' flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

Some CrossWorks library variants do not support width and precision specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Width/Precision Support** property of the project if you use these.

Flag characters

The flag characters and their meanings are:

'-'

The result of the conversion is left-justified within the field. The default, if this flag is not specified, is that the result of the conversion is left-justified within the field.

'+'

The result of a signed conversion *always* begins with a plus or minus sign. The default, if this flag is not specified, is that it begins with a sign only when a negative value is converted.

space

If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the space and '+' flags both appear, the space flag is ignored.

'#'

The result is converted to an *alternative form*. For 'o' conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both zero, a single '0' is printed). For 'x' or 'X' conversion, a nonzero result has '0x' or '0X' prefixed to it. For 'e', 'E', 'f', 'F', 'g', and 'G' conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For 'g' and 'F' conversions, trailing zeros are not removed from the result. As an extension, when used in 'p' conversion, the results has '#' prefixed to it. For other conversions, the behavior is undefined.

'0'

For 'd', 'i', 'o', 'u', 'x', 'X', 'e', 'E', 'f', 'F', 'g', and 'G' conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the '0' and '-' flags both appear, the '0' flag is ignored. For 'd', 'i', 'o', 'u', 'x', and 'X' conversions, if a precision is specified, the '0' flag is ignored. For other conversions, the behavior is undefined.

Length modifiers

The length modifiers and their meanings are:

'hh'

Specifies that a following 'd', 'i', 'o', 'u', 'x', or 'X' conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value will be converted to **signed char** or **unsigned char** before printing); or that a following 'n' conversion specifier applies to a pointer to a **signed char** argument.

'h'

Specifies that a following 'd', 'i', 'o', 'u', 'x', or 'X' conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integer promotions, but its value

is converted to **short int** or **unsigned short int** before printing); or that a following 'n' conversion specifier applies to a pointer to a **short int** argument.

'l'

Specifies that a following 'd', 'i', 'o', 'u', 'x', or 'X' conversion specifier applies to a **long int** or **unsigned long int** argument; that a following 'n' conversion specifier applies to a pointer to a **long int** argument; or has no effect on a following 'e', 'E', 'f', 'F', 'g', or 'G' conversion specifier. Some CrossWorks library variants do not support the 'l' length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

'll'

Specifies that a following 'd', 'i', 'o', 'u', 'x', or 'X' conversion specifier applies to a **long long int** or **unsigned long long int** argument; that a following 'n' conversion specifier applies to a pointer to a **long long int** argument. Some CrossWorks library variants do not support the 'll' length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined. Note that the C99 length modifiers 'j', 'z', 't', and 'L' are not supported.

Conversion specifiers

The conversion specifiers and their meanings are:

'd', 'i'

The argument is converted to signed decimal in the style [-]*dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading spaces. The default precision is one. The result of converting a zero value with a precision of zero is no characters.

'o', 'u', 'x', 'X'

The unsigned argument is converted to unsigned octal for 'o', unsigned decimal for 'u', or unsigned hexadecimal notation for 'x' or 'X' in the style *dddd* the letters 'abcdef' are used for 'x' conversion and the letters 'ABCDEF' for 'X' conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading spaces. The default precision is one. The result of converting a zero value with a precision of zero is no characters.

'f', 'F'

A double argument representing a floating-point number is converted to decimal notation in the style [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the '#' flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. A double argument representing an infinity is

converted to ‘inf’. A double argument representing a NaN is converted to ‘nan’. The ‘F’ conversion specifier produces ‘INF’ or ‘NAN’ instead of ‘inf’ or ‘nan’, respectively. Some CrossWorks library variants do not support the ‘f’ and ‘F’ conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Floating Point Support** property of the project if you use these conversion specifiers.

‘e’, ‘E’

A double argument representing a floating-point number is converted in the style [-]*d.ddd*e±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the ‘#’ flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The ‘E’ conversion specifier produces a number with ‘E’ instead of ‘e’ introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero. A double argument representing an infinity is converted to ‘inf’. A double argument representing a NaN is converted to ‘nan’. The ‘E’ conversion specifier produces ‘INF’ or ‘NAN’ instead of ‘inf’ or ‘nan’, respectively. Some CrossWorks library variants do not support the ‘f’ and ‘F’ conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf** Floating Point Support} property of the project if you use these conversion specifiers.

‘g’, ‘G’

A double argument representing a floating-point number is converted in style ‘f’ or ‘e’ (or in style ‘F’ or ‘e’ in the case of a ‘G’ conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as one. The style used depends on the value converted; style ‘e’ (or ‘E’) is used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the ‘#’ flag is specified; a decimal-point character appears only if it is followed by a digit. A double argument representing an infinity is converted to ‘inf’. A double argument representing a NaN is converted to ‘nan’. The ‘G’ conversion specifier produces ‘INF’ or ‘NAN’ instead of ‘inf’ or ‘nan’, respectively. Some CrossWorks library variants do not support the ‘f’ and ‘F’ conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Floating Point Support** property of the project if you use these conversion specifiers.

‘c’

The argument is converted to an **unsigned char**, and the resulting character is written.

‘s’

The argument is be a pointer to the initial element of an array of character type. Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null character.

'p'

The argument is a pointer to **void**. The value of the pointer is converted in the same format as the 'x' conversion specifier with a fixed precision of 2***sizeof**(**void** *).

'n'

The argument is a pointer to a signed integer into which is *written* the number of characters written to the output stream so far by the call to the formatting function. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

'%'

A '%' character is written. No argument is converted.

Note that the C99 width modifier 'l' used in conjunction with the 'c' and 's' conversion specifiers is not supported and nor are the conversion specifiers 'a' and 'A'.

putchar

Synopsis

```
int putchar(int c);
```

Description

putchar writes the character **c** to the standard output stream.

putchar returns the character written. If a write error occurs, **putchar** returns **EOF**.

puts

Synopsis

```
int puts(const char *s);
```

Description

puts writes the string pointed to by **s** to the standard output stream using **putchar** and appends a new-line character to the output. The terminating null character is not written.

puts returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

scanf

Synopsis

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

Description

scanf reads input from the standard input stream under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

scanf returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **scanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Formatted input control strings

The format is composed of zero or more directives: one or more white-space characters, an ordinary character (neither % nor a white-space character), or a conversion specification.

Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- An optional assignment-suppressing character *.
- An optional nonzero decimal integer that specifies the maximum field width (in characters).
- An optional length modifier that specifies the size of the receiving object.
- A conversion specifier character that specifies the type of conversion to be applied.

The formatted input function executes each directive of the format in turn. If a directive fails, the function returns. Failures are described as input failures (because of the occurrence of an encoding error or the unavailability of input characters), or matching failures (because of inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

- Input white-space characters (as specified by the **isspace** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.
- An input item is read from the stream, unless the specification includes an n specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.
- Except in the case of a **%** specifier, the input item (or, in the case of a %n directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

Length modifiers

The length modifiers and their meanings are:

'hh'

Specifies that a following 'd', 'i', 'o', 'u', 'x', 'X', or 'n' conversion specifier applies to an argument with type pointer to **signed char** or pointer to **unsigned char**.

'h'

Specifies that a following 'd', 'i', 'o', 'u', 'x', 'X', or 'n' conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.

'l'

Specifies that a following 'd', 'i', 'o', 'u', 'x', 'X', or 'n' conversion specifier applies to an argument with type pointer to **long int** or **unsigned long int**; that a following 'e', 'E', 'f', 'F', 'g', or 'G' conversion specifier applies to an argument with type pointer to **double**. Some CrossWorks library variants do not support the 'l' length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

'll'

Specifies that a following 'd', 'i', 'o', 'u', 'x', 'X', or 'n' conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**. Some CrossWorks library variants do not support the 'll' length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined. Note that the C99 length modifiers 'j', 'z', 't', and 'L' are not supported.

Conversion specifiers

'd'

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 10 for the **base** argument. The corresponding argument must be a pointer to signed integer.

'i'

Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value zero for the **base** argument. The corresponding argument must be a pointer to signed integer.

'o'

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 18 for the **base** argument. The corresponding argument must be a pointer to signed integer.

'u'

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument must be a pointer to unsigned integer.

'x'

Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 16 for the **base** argument. The corresponding argument must be a pointer to unsigned integer.

'e', 'f', 'g'

Matches an optionally signed floating-point number whose format is the same as expected for the subject sequence of the **strtod** function. The corresponding argument shall be a pointer to floating. Some CrossWorks library variants do not support the 'e', 'f' and 'F' conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Scanf Floating Point Support** property of the project if you use these conversion specifiers.

'c'

Matches a sequence of characters of exactly the number specified by the field width (one if no field width is present in the directive). The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

's'

Matches a sequence of non-white-space characters The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

'['

Matches a nonempty sequence of characters from a set of expected characters (the *scanset*). The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the format string, up to and including the matching right bracket ']'. The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex '^', in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with '[]' or'[^]', the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a '-' character is in the scanlist and is not the first, nor the second where the first character is a '^', nor the last character, it is treated as a member of the scanset. Some CrossWorks library variants do not support the '[' conversion specifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Scanf Classes Supported** property of the project if you use this conversion specifier.

'p'

Reads a sequence output by the corresponding '%p' formatted output conversion. The corresponding argument must be a pointer to a pointer to **void**.

'n'

No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the formatted input function. Execution of a '%n' directive does not increment the assignment count returned at the completion of execution of the fscanf function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

'%'

Matches a single '%' character; no conversion or assignment occurs.

Note that the C99 width modifier 'l' used in conjunction with the 'c', 's', and '[' conversion specifiers is not supported and nor are the conversion specifiers 'a' and 'A'.

snprintf

Synopsis

```
int snprintf(char *s,
             size_t n,
             const char *format,
             ...);
```

Description

snprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output.

If **n** is zero, nothing is written, and **s** can be a null pointer. Otherwise, output characters beyond the **n**–1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

snprintf returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**>.

sprintf

Synopsis

```
int sprintf(char *s,
            const char *format,
            ...);
```

Description

sprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. A null character is written at the end of the characters written; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

sprintf returns number of characters transmitted (not counting the terminating null), or a negative value if an output or encoding error occurred.

sscanf

Synopsis

```
int sscanf(const char *s,
           const char *format,
           ...);
```

Description

sscanf reads input from the string **s** under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

sscanf returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **sscanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

vprintf

Synopsis

```
int vprintf(const char *format,
            __va_list arg);
```

Description

vprintf writes to the standard output stream using **putchar** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. Before calling **vprintf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vprintf** does not invoke the **va_end** macro.

vprintf returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Note

vprintf is equivalent to **printf** with the variable argument list replaced by **arg**.

vscanf

Synopsis

```
int vscanf(const char *format,
           __va_list arg);
```

Description

vscanf reads input from the standard input stream under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. Before calling **vscanf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vscanf** does not invoke the **va_end** macro.

If there are insufficient arguments for the format, the behavior is undefined.

vscanf returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **vscanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Note

vscanf is equivalent to **scanf** with the variable argument list replaced **arg**.

vsnprintf

Synopsis

```
int vsnprintf(char *s,
              size_t n,
              const char *format,
              __va_list arg);
```

Description

vsnprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. Before calling **vsnprintf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vsnprintf** does not invoke the **va_end** macro.

If **n** is zero, nothing is written, and **s** can be a null pointer. Otherwise, output characters beyond the **n**−1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

vsnprintf returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

Note

vsnprintf is equivalent to **snprintf** with the variable argument list replaced by **arg**.

vsprintf

Synopsis

```
int vsprintf(char *s,
             const char *format,
             __va_list arg);
```

Description

vsprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. Before calling **vsprintf**, **arg**> must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vsprintf** does not invoke the **va_end** macro.

A null character is written at the end of the characters written; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

vsprintf returns number of characters transmitted (not counting the terminating null), or a negative value if an output or encoding error occurred.

Note

vsprintf is equivalent to **sprintf** with the variable argument list replaced by **arg**.

vsscanf

Synopsis

```
int vsscanf(const char *s,
            const char *format,
            __va_list arg);
```

Description

vsscanf reads input from the string **s** under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. Before calling **vsscanf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vsscanf** does not invoke the **va_end** macro.

If there are insufficient arguments for the format, the behavior is undefined.

vsscanf returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **vsscanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Note

vsscanf is equivalent to **sscanf** with the variable argument list replaced by **arg**.

<stdlib.h>

API Summary

Macros	
EXIT_FAILURE	EXIT_FAILURE
EXIT_SUCCESS	EXIT_SUCCESS
RAND_MAX	RAND_MAX
Integer arithmetic functions	
abs	Return an integer absolute value
div	Divide two ints returning quotient and remainder
labs	Return a long integer absolute value
ldiv	Divide two longs returning quotient and remainder
llabs	Return a long long integer absolute value
lldiv	Divide two long longs returning quotient and remainder
Memory allocation functions	
calloc	Allocate space for an array of objects and initialize them to zero
free	Frees allocated memory for reuse
malloc	Allocate space for a single object
realloc	Resizes allocated memory space or allocates memory space
String to number conversions	
atof	Convert string to double
atoi	Convert string to int
atol	Convert string to long
atoll	Convert string to long long
strtod	Convert string to double
strtof	Convert string to float
strtol	Convert string to long
strtoll	Convert string to long long
strtoul	Convert string to unsigned long
strtoull	Convert string to unsigned long long
Pseudo-random sequence generation functions	
rand	Return next random number in sequence

srand	Set seed of random number sequence
Search and sort functions	
bsearch	Search a sorted array
qsort	Sort an array
Environment	
atexit	Set function to be execute on exit
exit	Terminates the calling process
Number to string conversions	
itoa	Convert int to string
lltoa	Convert long long to string
ltoa	Convert long to string
ulltoa	Convert unsigned long long to string
ultoa	Convert unsigned long to string
utoa	Convert unsigned to string
Types	
div_t	Structure containing quotient and remainder after division of an int
ldiv_t	Structure containing quotient and remainder after division of a long
lldiv_t	Structure containing quotient and remainder after division of a long long

EXIT_FAILURE

Synopsis

```
#define EXIT_FAILURE    1
```

Description

EXIT_FAILURE pass to **exit** on unsuccessful termination.

EXIT_SUCCESS

Synopsis

```
#define EXIT_SUCCESS    0
```

Description

EXIT_SUCCESS pass to **exit** on successful termination.

RAND_MAX

Synopsis

```
#define RAND_MAX 32767
```

Description

RAND_MAX expands to an integer constant expression that is the maximum value returned by **rand**.

abs

Synopsis

```
int abs(int j);
```

Description

abs returns the absolute value of the integer argument **j**.

atexit

Synopsis

```
int atexit(void (*func)(void));
```

Description

atexit registers **function** to be called when the application has exited. The functions registered with **atexit** are executed in reverse order of their registration. **atexit** returns 0 on success and non-zero on failure.

atof

Synopsis

```
double atof(const char *nptr);
```

Description

atof converts the initial portion of the string pointed to by **nptr** to a **double** representation. **atof** does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atof** is equivalent to `strtod(nptr, (char **)NULL)`.

atof returns the converted value.

See Also

strtod

atoi

Synopsis

```
int atoi(const char *nptr);
```

Description

atoi converts the initial portion of the string pointed to by **nptr** to an **int** representation.

atoi does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atoi** is equivalent to `(int)strtol(nptr, (char **)NULL, 10)`.

atoi returns the converted value.

See Also

strtol

atol

Synopsis

```
long int atol(const char *nptr);
```

Description

atol converts the initial portion of the string pointed to by **nptr to** a **long int** representation.

atol does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atol** is equivalent to `strtol(nptr, (char **)NULL, 10)`.

atol returns the converted value.

See Also

strtol

atoll

Synopsis

```
long long int atoll(const char *nptr);
```

Description

atoll converts the initial portion of the string pointed to by **nptr** to a **long long int** representation.

atoll does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atoll** is equivalent to `strtoll(nptr, (char **)NULL, 10)`. **atoll** returns the converted value.

See Also

strtoll

bsearch

Synopsis

```
void *bsearch(const void *key,
              const void *buf,
              size_t num,
              size_t size,
              int (*compare)(const void *, const void *));
```

Description

bsearch searches the array ***base** for the specified {***key**} and returns a pointer to the first entry that matches or null if no match. The array should have **num** elements of **size** bytes and be sorted by the same algorithm as the **compare** function

The **compare** function should return a negative value if the first parameter is less than second parameter, zero if the parameters are equal, and a positive value if the first parameter is greater than the second parameter.

calloc

Synopsis

```
void *calloc(size_t nobj,
             size_t size);
```

Description

calloc allocates space for an array of **nmemb** objects, each of whose size is **size**. The space is initialized to all zero bits.

calloc returns a null pointer if the space for the array of object cannot be allocated from free memory; if space for the array can be allocated, **calloc** returns a pointer to the start of the allocated space.

div

Synopsis

```
div_t div(int numer,
          int denom);
```

Description

div computes **numer** / **denom** and **numer** % **denom** in a single operation.

div returns a structure of type **div_t** comprising both the quotient and the remainder. The structures contain the members **quot** (the quotient) and **rem** (the remainder), each of which has the same type as the arguments **numer** and **denom**. If either part of the result cannot be represented, the behavior is undefined.

See Also

div_t

div_t

Description

div_t stores the quotient and remainder returned by **div**.

exit

Synopsis

```
void exit(int exit_code);
```

Description

exit returns to the startup code and performs the appropriate cleanup process.

free

Synopsis

```
void free(void *p);
```

Description

free causes the space pointed to by **ptr** to be deallocated, that is, made available for further allocation. If **ptr** is a null pointer, no action occurs.

If **ptr** does not match a pointer earlier returned by **calloc**, **malloc**, or **realloc**, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

itoa

Synopsis

```
char *itoa(int val,
           char *buf,
           int radix);
```

Description

itoa converts **val** to a string in base **radix** and places the result in **buf**.

itoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

If **val** is negative and **radix** is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

See Also

ltoa, lltoa, ultoa, ulltoa, utoa

labs

Synopsis

```
long int labs(long int j);
```

Description

labs returns the absolute value of the long integer argument **j**.

ldiv

Synopsis

```
ldiv_t ldiv(long int numer,
            long int denom);
```

Description

ldiv computes **numer** / **denom** and **numer** % **denom** in a single operation. **ldiv** returns a structure of type ldiv_t comprising both the quotient and the remainder. The structures contain the members **quot** (the quotient) and **rem** (the remainder), each of which has the same type as the arguments **numer** and **denom**. If either part of the result cannot be represented, the behavior is undefined.

See Also

ldiv_t

ldiv_t

Description

ldiv_t stores the quotient and remainder returned by **ldiv**.

llabs

Synopsis

```
long long int llabs(long long int j);
```

Description

llabs returns the absolute value of the long long integer argument **j**.

lldiv

Synopsis

```
lldiv_t lldiv(long long int numer,
              long long int denom);
```

lldiv computes **numer** / **denom** and **numer** % **denom** in a single operation. **lldiv** returns a structure of type lldiv_t comprising both the quotient and the remainder. The structures contain the members **quot** (the quotient) and **rem** (the remainder), each of which has the same type as the arguments **numer** and **denom**. If either part of the result cannot be represented, the behavior is undefined.

See Also

lldiv_t

lldiv_t

Description

lldiv_t stores the quotient and remainder returned by **lldiv**.

lltoa

Synopsis

```
char *lltoa(long long val,
            char *buf,
            int radix);
```

Description

lltoa converts **val** to a string in base **radix** and places the result in **buf**.

lltoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

If **val** is negative and radix is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

See Also

itoa, **ltoa**, **ultoa**, **ulltoa**, **utoa**

ltoa

Synopsis

```
char *ltoa(long val,
           char *buf,
           int radix);
```

Description

ltoa converts **val** to a string in base **radix** and places the result in **buf**.

ltoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

If **val** is negative and radix is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

See Also

itoa, **lltoa**, **ultoa**, **ulltoa**, **utoa**

malloc

Synopsis

```
void *malloc(size_t size);
```

Description

malloc allocates space for an object whose size is specified by 'b size and whose value is indeterminate.

malloc returns a null pointer if the space for the object cannot be allocated from free memory; if space for the object can be allocated, **malloc** returns a pointer to the start of the allocated space.

qsort

Synopsis

```
void qsort(void *buf,
           size_t num,
           size_t size,
           int (*compare)(const void *, const void *));
```

qsort sorts the array ***base** using the **compare** algorithm. The array should have **num** elements of **size** bytes. The **compare** function should return a negative value if the first parameter is less than second parameter, zero if the parameters are equal and a positive value if the first parameter is greater than the second parameter.

rand

Synopsis

```
int rand(void);
```

Description

rand computes a sequence of pseudo-random integers in the range 0 to **RAND_MAX**.

rand returns the computed pseudo-random integer.

realloc

Synopsis

```
void *realloc(void *p,
              size_t size);
```

Description

realloc deallocates the old object pointed to by **ptr** and returns a pointer to a new object that has the size specified by **size**. The contents of the new object is identical to that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

If **ptr** is a null pointer, **realloc** behaves like realloc for the specified size. If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.

realloc returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object could not be allocated.

If **ptr** does not match a pointer earlier returned by **calloc**, **malloc**, or **realloc**, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

srand

Synopsis

```
void srand(unsigned int seed);
```

Description

srand uses the argument **seed** as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand**. If **srand** is called with the same seed value, the same sequence of pseudo-random numbers is generated.

If **rand** is called before any calls to **srand** have been made, a sequence is generated as if **srand** is first called with a seed value of 1.

See Also

rand or 'ref rand_max

strtod

Synopsis

```
double strtod(const char *nptr,
              char **endptr);
```

Description

strtod converts the initial portion of the string pointed to by **nptr** to a **double** representation.

First, **strtod** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling a floating-point constant, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtod** then attempts to convert the subject sequence to a floating-point number, and return the result.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **strtod**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtod returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **HUGE_VAL** is returned according to the sign of the value, if any, and the value of the macro **errno** is stored in **errno**.

strtof

Synopsis

```
float strtof(const char *nptr,
             char **endptr);
```

Description

strtof converts the initial portion of the string pointed to by **nptr** to a **double** representation.

First, **strtof** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling a floating-point constant, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtof** then attempts to convert the subject sequence to a floating-point number, and return the result.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtof returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **HUGE_VALF** is returned according to the sign of the value, if any, and the value of the macro **errno** is stored in **errno**.

strtol

Synopsis

```
long int strtol(const char *nptr,
                char **endptr,
                int base);
```

Description

strtol converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtol** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtol** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters '0x' or '0X' may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtol returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MIN** or **LONG_MAX** is returned according to the sign of the value, if any, and the value of the macro **errno** is stored in **errno**.

strtoll

Synopsis

```
long long int strtoll(const char *nptr,
                      char **endptr,
                      int base);
```

Description

strtoll converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoll** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoll** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters '0x' or '0X' may optionally precede the sequence of letters and digits, following the optional sign. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtoll returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LLONG_MIN** or **LLONG_MAX** is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in **errno**.

strtoul

Synopsis

```
unsigned long int strtoul(const char *nptr,
                          char **endptr,
                          int base);
```

Description

strtoul converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoul** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoul** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant. If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters '0x' or '0X' may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtoul returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MAX** or **ULONG_MAX** is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in **errno**.

strtoull

Synopsis

```
unsigned long long int strtoull(const char *nptr,
                                char **endptr,
                                int base);
```

Description

strtoull converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoull** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoull** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters '0x' or '0X' may optionally precede the sequence of letters and digits, following the optional sign. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtoull returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LLONG_MAX** or **ULLONG_MAX** is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in **errno**.

ulltoa

Synopsis

```
char *ulltoa(unsigned long long val,
             char *buf,
             int radix);
```

Description

ulltoa converts **val** to a string in base **radix** and places the result in **buf**.

ulltoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

See Also

itoa, **ltoa**, **lltoa**, **ultoa**, **utoa**

ultoa

Synopsis

```
char *ultoa(unsigned long val,
            char *buf,
            int radix);
```

Description

ultoa converts **val** to a string in base **radix** and places the result in **buf**.

ultoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

See Also

itoa, **ltoa**, **lltoa**, **ulltoa**, **utoa**

utoa

Synopsis

```
char *utoa(unsigned val,
           char *buf,
           int radix);
```

Description

utoa converts **val** to a string in base **radix** and places the result in **buf**.

utoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

See Also

itoa, **ltoa**, **lltoa**, **ultoa**, **ulltoa**

<string.h>

API Summary

Copying functions	
memccpy	Copy memory with specified terminator (POSIX extension)
memcpy	Copy memory
memmove	Safely copy overlapping memory
mempcpy	Copy memory (GNU extension)
strcat	Concatenate strings
strcpy	Copy string
strdup	Duplicate string (POSIX extension)
strlcat	Copy string up to a maximum length with terminator (BSD extension)
strlcpy	Copy string up to a maximum length with terminator (BSD extension)
strncat	Concatenate strings up to maximum length
strncpy	Copy string up to a maximum length
strndup	Duplicate string (POSIX extension)
Comparison functions	
memcmp	Compare memory
strcasecmp	Compare strings ignoring case (POSIX extension)
strcmp	Compare strings
strncasecmp	Compare strings up to a maximum length ignoring case (POSIX extension)
strncmp	Compare strings up to a maximum length
Search functions	
memchr	Search memory for a character
strcasestr	Find first case-insensitive occurrence of a string within string
strchr	Find character within string
strcspn	Compute size of string not prefixed by a set of characters
strncasestr	Find first case-insensitive occurrence of a string within length-limited string
strnchr	Find character in a length-limited string

strnlen	Calculate length of length-limited string (POSIX extension)
strnstr	Find first occurrence of a string within length-limited string
strpbrk	Find first occurrence of characters within string
strrchr	Find last occurrence of character within string
strsep	Break string into tokens (4.4BSD extension)
strspn	Compute size of string prefixed by a set of characters
strstr	Find first occurrence of a string within string
strtok	Break string into tokens
strtok_r	Break string into tokens, reentrant version (POSIX extension)
Miscellaneous functions	
memset	Set memory to character
strerror	Decode error code
strlen	Calculate length of string

memccpy

Synopsis

```
void *memccpy(void *s1,
              const void *s2,
              int c,
              size_t n);
```

Description

memccpy copies at most **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The copying stops as soon as **n** characters are copied or the character **c** is copied into the destination object pointed to by **s1**. The behavior of **memccpy** is undefined if copying takes place between objects that overlap.

memccpy returns a pointer to the character immediately following **c** in **s1**, or **NULL** if **c** was not found in the first **n** characters of **s2**.

Note

memccpy conforms to POSIX.1-2008.

memchr

Synopsis

```
void *memchr(const void *s,
             int c,
             size_t n);
```

Description

memchr locates the first occurrence of **c** (converted to an **unsigned char**) in the initial **n** characters (each interpreted as **unsigned char**) of the object pointed to by **s**. Unlike **strchr**, **memchr** does *not* terminate a search when a null character is found in the object pointed to by **s**.

memchr returns a pointer to the located character, or a null pointer if **c** does not occur in the object.

memcmp

Synopsis

```
int memcmp(const void *s1,
           const void *s2,
           size_t n);
```

Description

memcmp compares the first **n** characters of the object pointed to by **s1** to the first **n** characters of the object pointed to by **s2**. **memcmp** returns an integer greater than, equal to, or less than zero as the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.

memcpy

Synopsis

```
void *memcpy(void *s1,
             const void *s2,
             size_t n);
```

Description

memcpy copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The behavior of **memcpy** is undefined if copying takes place between objects that overlap.

memcpy returns the value of **s1**.

memmove

Synopsis

```
void *memmove(void *s1,
              const void *s2,
              size_t n);
```

Description

memmove copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1** ensuring that if **s1** and **s2** overlap, the copy works correctly. Copying takes place as if the **n** characters from the object pointed to by **s2** are first copied into a temporary array of **n** characters that does not overlap the objects pointed to by **s1** and **s2**, and then the **n** characters from the temporary array are copied into the object pointed to by **s1**.

memmove returns the value of **s1**.

mempcpy

Synopsis

```
void *mempcpy(void *s1,
              const void *s2,
              size_t n);
```

Description

mempcpy copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The behavior of **mempcpy** is undefined if copying takes place between objects that overlap.

mempcpy returns a pointer to the byte following the last written byte.

Note

This is an extension found in GNU libc.

memset

Synopsis

```
void *memset(void *s,
             int c,
             size_t n);
```

Description

memset copies the value of **c** (converted to an **unsigned char**) into each of the first **n** characters of the object pointed to by **s**.

memset returns the value of **s**.

strcasecmp

Synopsis

```
int strcasecmp(const char *s1,
               const char *s2);
```

Description

strcasecmp compares the string pointed to by **s1** to the string pointed to by **s2** ignoring differences in case. **strcasecmp** returns an integer greater than, equal to, or less than zero if the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.

Note

strcasecmp conforms to POSIX.1-2008.

strcasestr

Synopsis

```
char *strcasestr(const char *s1,
                 const char *s2);
```

Description

strcasestr locates the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2** without regard to character case.

strcasestr returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, **strcasestr** returns **s1**.

Note

strcasestr is an extension commonly found in Linux and BSD C libraries.

strcat

Synopsis

```
char *strcat(char *s1,
             const char *s2);
```

Description

strcat appends a copy of the string pointed to by **s2** (including the terminating null character) to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. The behavior of **strcat** is undefined if copying takes place between objects that overlap.

strcat returns the value of **s1**.

strchr

Synopsis

```
char *strchr(const char *s,
             int c);
```

Description

strchr locates the first occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

strchr returns a pointer to the located character, or a null pointer if **c** does not occur in the string.

strcmp

Synopsis

```
int strcmp(const char *s1,
           const char *s2);
```

Description

strcmp compares the string pointed to by **s1** to the string pointed to by **s2**. **strcmp** returns an integer greater than, equal to, or less than zero if the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.

strcpy

Synopsis

```
char *strcpy(char *s1,
             const char *s2);
```

Description

strcpy copies the string pointed to by **s2** (including the terminating null character) into the array pointed to by **s1**. The behavior of **strcpy** is undefined if copying takes place between objects that overlap.

strcpy returns the value of **s1**.

strcspn

Synopsis

```
size_t strcspn(const char *s1,
               const char *s2);
```

Description

strcspn computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters not from the string pointed to by **s2**.

strcspn returns the length of the segment.

strdup

Synopsis

```
char *strdup(const char *s1);
```

Description

strdup duplicates the string pointed to by **s1** by using **malloc** to allocate memory for a copy of **s** and then copying **s**, including the terminating null, to that memory **strdup** returns a pointer to the new string or a null pointer if the new string cannot be created. The returned pointer can be passed to **free**.

Note

strdup conforms to POSIX.1-2008 and SC22 TR 24731-2.

strerror

Synopsis

```
char *strerror(int num);
```

Description

strerror maps the number in **num** to a message string. Typically, the values for **num** come from **errno**, but **strerror** can map any value of type **int** to a message.

strerror returns a pointer to the message string. The program must not modify the returned message string. The message may be overwritten by a subsequent call to **strerror**.

strlcat

Synopsis

```
size_t strlcat(char *s1,
               const char *s2,
               size_t n);
```

Description

strlcat appends no more than **n**−**strlen**(**dst**)−1 characters pointed to by **s2** into the array pointed to by **s1** and always terminates the result with a null character if **n** is greater than zero. Both the strings **s1** and **s2** must be terminated with a null character on entry to **strlcat** and a byte for the terminating null should be included in **n**. The behavior of **strlcat** is undefined if copying takes place between objects that overlap.

strlcat returns the number of characters it tried to copy, which is the sum of the lengths of the strings **s1** and **s2** or **n**, whichever is smaller.

Note

strlcat is commonly found in OpenBSD libraries.

strlcpy

Synopsis

```
size_t strlcpy(char *s1,
               const char *s2,
               size_t n);
```

Description

strlcpy copies up to **n**−1 characters from the string pointed to by **s2** into the array pointed to by **s1** and always terminates the result with a null character. The behavior of **strlcpy** is undefined if copying takes place between objects that overlap.

strlcpy returns the number of characters it tried to copy, which is the length of the string **s2** or **n**, whichever is smaller.

Note

strlcpy is commonly found in OpenBSD libraries and contrasts with **strncpy** in that the resulting string is always terminated with a null character.

strlen

Synopsis

```
size_t strlen(const char *s);
```

Description

strlen returns the length of the string pointed to by **s**, that is the number of characters that precede the terminating null character.

strncasecmp

Synopsis

```
int strncasecmp(const char *s1,
                const char *s2,
                size_t n);
```

Description

strncasecmp compares not more than **n** characters from the array pointed to by **s1** to the array pointed to by **s2** ignoring differences in case. Characters that follow a null character are not compared.

strncasecmp returns an integer greater than, equal to, or less than zero, if the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the possibly null-terminated array pointed to by **s2**.

Note

strncasecmp conforms to POSIX.1-2008.

strncasestr

Synopsis

```
char *strncasestr(const char *s1,
                  const char *s2,
                  size_t n);
```

Description

strncasestr searches at most **n** characters to locate the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2** without regard to character case.

strncasestr returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, **strncasestr** returns **s1**.

Note

strncasestr is an extension commonly found in Linux and BSD C libraries.

strncat

Synopsis

```
char *strncat(char *s1,
              const char *s2,
              size_t n);
```

Description

strncat appends not more than **n** characters from the array pointed to by **s2** to the end of the string pointed to by **s1**. A null character in **s1** and characters that follow it are not appended. The initial character of **s2** overwrites the null character at the end of **s1**. A terminating null character is always appended to the result. The behavior of **strncat** is undefined if copying takes place between objects that overlap.

strncat returns the value of **s1**.

strnchr

Synopsis

```
char *strnchr(const char *str,
              size_t n,
              int ch);
```

Description

strnchr searches not more than **n** characters to locate the first occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

strnchr returns a pointer to the located character, or a null pointer if **c** does not occur in the string.

strncmp

Synopsis

```
int strncmp(const char *s1,
            const char *s2,
            size_t n);
```

Description

strncmp compares not more than **n** characters from the array pointed to by **s1** to the array pointed to by **s2**. Characters that follow a null character are not compared.

strncmp returns an integer greater than, equal to, or less than zero, if the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the possibly null-terminated array pointed to by **s2**.

strncpy

Synopsis

```
char *strncpy(char *s1,
              const char *s2,
              size_t n);
```

Description

strncpy copies not more than **n** characters from the array pointed to by **s2** to the array pointed to by **s1**. Characters that follow a null character in **s2** are not copied. The behavior of **strncpy** is undefined if copying takes place between objects that overlap. If the array pointed to by **s2** is a string that is shorter than **n** characters, null characters are appended to the copy in the array pointed to by **s1**, until **n** characters in all have been written.

strncpy returns the value of **s1**.

Note

No null character is implicitly appended to the end of **s1**, so **s1** will only be terminated by a null character if the length of the string pointed to by **s2** is less than **n**.

strndup

Synopsis

```
char *strndup(const char *s1,
              size_t n);
```

Description

strndup duplicates at most **n** characters from the the string pointed to by **s1** by using **malloc** to allocate memory for a copy of **s1**.

If the length of string pointed to by **s1** is greater than **n** characters, only **n** characters will be duplicated. If **n** is greater than the length of string pointed to by **s1**, all characters in the string are copied into the allocated array including the terminating null character.

strndup returns a pointer to the new string or a null pointer if the new string cannot be created. The returned pointer can be passed to **free**.

Note

strndup conforms to POSIX.1-2008 and SC22 TR 24731-2.

strnlen

Synopsis

```
size_t strnlen(const char *s,
               size_t n);
```

Description

strnlen returns the length of the string pointed to by **s**, up to a maximum of **n** characters. **strnlen** only examines the first **n** characters of the string **s**.

Note

strnlen conforms to POSIX.1-2008.

strnstr

Synopsis

```
char *strnstr(const char *s1,
              const char *s2,
              size_t n);
```

Description

strnstr searches at most **n** characters to locate the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2**.

strnstr returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, **strnstr** returns **s1**.

Note

strnstr is an extension commonly found in Linux and BSD C libraries.

strpbrk

Synopsis

```
char *strpbrk(const char *s1,
              const char *s2);
```

Description

strpbrk locates the first occurrence in the string pointed to by **s1** of any character from the string pointed to by **s2**.

strpbrk returns a pointer to the character, or a null pointer if no character from **s2** occurs in **s1**.

strrchr

Synopsis

```
char *strrchr(const char *s,
              int c);
```

Description

strrchr locates the last occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

strrchr returns a pointer to the character, or a null pointer if **c** does not occur in the string.

strsep

Synopsis

```
char *strsep(char **stringp,
             const char *delim);
```

Description

strsep locates, in the string referenced by ***stringp**, the first occurrence of any character in the string **delim** (or the terminating null character) and replaces it with a null character. The location of the next character after the delimiter character (or NULL, if the end of the string was reached) is stored in ***stringp**. The original value of ***stringp** is returned.

An empty field (that is, a character in the string **delim** occurs as the first character of ***stringp** can be detected by comparing the location referenced by the returned pointer to the null character.

If ***stringp** is initially null, **strsep** returns null.

Note

strsep is an extension commonly found in Linux and BSD C libraries.

strspn

Synopsis

```
size_t strspn(const char *s1,
              const char *s2);
```

Description

strspn computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

strspn returns the length of the segment.

strstr

Synopsis

```
char *strstr(const char *s1,
             const char *s2);
```

Description

strstr locates the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2**.

strstr returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, **strstr** returns **s1**.

strtok

Synopsis

```
char *strtok(char *s1,
             const char *s2);
```

Description

strtok A sequence of calls to **strtok** breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. The separator string pointed to by **s2** may be different from call to call.

The first call in the sequence searches the string pointed to by **s1** for the first character that is not contained in the current separator string pointed to by **s2**. If no such character is found, then there are no tokens in the string pointed to by **s1** and **strtok** returns a null pointer. If such a character is found, it is the start of the first token.

strtok then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. **strtok** saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Note

strtok maintains static state and is therefore not reentrant and not thread safe. See strtok_r for a thread-safe and reentrant variant.

See Also

strsep, strtok_r.

strtok_r

Synopsis

```
char *strtok_r(char *s1,
               const char *s2,
               char **s3);
```

Description

strtok_r is a reentrant version of the function **strtok** where the state is maintained in the object of type **char** * pointed to by **s3**.

Note

strtok_r conforms to POSIX.1-2008 and is commonly found in Linux and BSD C libraries.

See Also

strtok.

<time.h>

API Summary

Functions	
asctime	Convert a struct tm to a string
asctime_r	Convert a struct tm to a string
ctime	Convert a time_t to a string
ctime_r	Convert a time_t to a string
difftime	Calculates the difference between two times
gmtime	Convert a time_t to a struct tm
gmtime_r	Convert a time_t to a struct tm
localtime	Convert a time_t to a struct tm
localtime_r	Convert a time_t to a struct tm
mktime	Convert a struct tm to time_t
strftime	Format a struct tm to a string
Types	
clock_t	Clock type
time_t	Time type
tm	Time structure

asctime

Synopsis

```
char *asctime(const tm *tp);
```

Description

asctime converts the ***tp** struct to a null terminated string of the form Sun Sep 16 01:03:52 1973. The returned string is held in a static buffer, this function is not re-entrant.

asctime_r

Synopsis

```
char *asctime_r(const tm *tp,
                char *buf);
```

Description

asctime_r converts the ***tp** struct to a null terminated string of the form Sun Sep 16 01:03:52 1973 in **buf** and returns **buf**. The **buf** must point to an array at least 26 bytes in length.

clock_t

Synopsis

```
typedef long clock_t;
```

Description

clock_t is the type returned by the **clock** function.

ctime

Synopsis

```
char *ctime(const time_t *tp);
```

Description

ctime converts the ***tp** to a null terminated string. The returned string is held in a static buffer, this function is not re-entrant.

ctime_r

Synopsis

```
char *ctime_r(const time_t *tp,
              char *buf);
```

Description

ctime_r converts the ***tp** to a null terminated string in **buf** and returns **buf**. The **buf** must point to an array at least 26 bytes in length.

difftime

Synopsis

```
double difftime(time_t time2,
                time_t time1);
```

Description

difftime returns **time1** - **time0** as a double precision number.

gmtime

Synopsis

```
gmtime(const time_t *tp);
```

Description

gmtime converts the ***tp** time format to a **struct** tm time format. The returned value points to a static object - this function is not re-entrant.

gmtime_r

Synopsis

```
gmtime_r(const time_t *tp,
         tm *result);
```

Description

gmtime_r converts the ***tp** time format to a **struct** tm time format in ***result** and returns **result**.

localtime

Synopsis

```
localtime(const time_t *tp);
```

Description

localtime converts the ***tp** time format to a **struct** tm local time format. The returned value points to a static object - this function is not re-entrant.

localtime_r

Synopsis

```
localtime_r(const time_t *tp,
            tm *result);
```

Description

localtime_r converts the ***tp** time format to a **struct** tm local time format in ***result** and returns **result**.

mktime

Synopsis

```
time_t mktime(tm *tp);
```

Description

mktime validates (and updates) the ***tp** struct to ensure that the **tm_sec**, **tm_min**, **tm_hour**, **tm_mon** fields are within the supported integer ranges and the **tm_m_day**, **tm_mon** and **tm_year** fields are consistent. The validated ***tp** struct is converted to the number of seconds since UTC 1/1/70 and returned.

strftime

Synopsis

```
size_t strftime(char *s,
                size_t smax,
                const char *fmt,
                const tm *tp);
```

Description

strftime formats the ***tp** struct to a null terminated string of maximum size **smax**-1 into the array at ***s** based on the **fmt** format string. The format string consists of conversion specifications and ordinary characters. Conversion specifications start with a % character followed by an optional # character. The following conversion specifications are supported:

Specification	Description
%	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for locale
%#c	Long date and time representation appropriate for locale
%d	Day of month [01,31]
%#d	Day of month without leading zero [1,31]
%H	Hour in 24-hour format [00,23]
%#H	Hour in 24-hour format without leading zeros [0,23]
%I	Hour in 12-hour format [01,12]
%#I	Hour in 12-hour format without leading zeros [1,12]
%j	Day of year as a decimal number [001,366]
%#j	Day of year as a decimal number without leading zeros [1,366]
%m	Month as a decimal number [01,12]
%#m	Month as a decimal number without leading zeros [1,12]
%M	Minute as a decimal number [00,59]

%#M	Minute as a decimal number without leading zeros [0,59]
%#p	Locale's a.m or p.m indicator
%S	Second as a decimal number [00,59]
%#S	Second as a decimal number without leading zeros [0,59]
%U	Week number as a decimal number [00,53], Sunday is first day of the week
%#U	Week number as a decimal number without leading zeros [0,53], Sunday is first day of the week
%w	Weekday as a decimal number [0,6], Sunday is 0
%W	Week number as a decimal number [00,53], Monday is first day of the week
%#W	Week number as a decimal number without leading zeros [0,53], Monday is first day of the week
%x	Locale's date representation
%#x	Locale's long date representation
%X	Locale's time representation
%y	Year without century, as a decimal number [00,99]
%#y	Year without century, as a decimal number without leading zeros [0,99]
%z,%Z	Timezone name or abbreviation
%%	%

time_t

Synopsis

```
typedef long time_t;
```

Description

time_t is a long type that represents the time in number of seconds since UTC 1/1/70, negative values indicate time before UTC 1/1/70.

tm

Synopsis

```
typedef struct {
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
  int tm_yday;
  int tm_isdst;
} tm;
```

Description

tm structure has the following fields.

Member	Description
tm_sec	seconds after the minute - [0,59]
tm_min	minutes after the hour - [0,59]
tm_hour	hours since midnight - [0,23]
tm_mday	day of the month - [1,31]
tm_mon	months since January - [0,11]
tm_year	years since 1900
tm_wday	days since Sunday - [0,6]
tm_yday	days since January 1 - [0,365]
tm_isdst	daylight savings time flag

<wchar.h>

API Summary

Character minimum and maximum values	
WCHAR_MAX	Maximum value of a wide character
WCHAR_MIN	Minimum value of a wide character
Copying functions	
wcscat	Concatenate strings
wcscpy	Copy string
wcsncat	Concatenate strings up to maximum length
wcsncpy	Copy string up to a maximum length
wmemccpy	Copy memory with specified terminator (POSIX extension)
wmemcpy	Copy memory
wmemmove	Safely copy overlapping memory
wmempcpy	Copy memory (GNU extension)
Comparison functions	
wcscmp	Compare strings
wcsncmp	Compare strings up to a maximum length
wmemcmp	Compare memory
Search functions	
wcschr	Find character within string
wcscspn	Compute size of string not prefixed by a set of characters
wcsnchr	Find character in a length-limited string
wcsnlen	Calculate length of length-limited string
wcsnstr	Find first occurrence of a string within length-limited string
wcspbrk	Find first occurrence of characters within string
wcsrchr	Find last occurrence of character within string
wcsspn	Compute size of string prefixed by a set of characters
wcsstr	Find first occurrence of a string within string
wcstok	Break string into tokens
wcstok_r	Break string into tokens (reentrant version)
wmemchr	Search memory for a wide character

wstrsep	Break string into tokens
Miscellaneous functions	
wcsdup	Duplicate string
wcslen	Calculate length of string
wmemset	Set memory to wide character
Types	
wint_t	Wide integer type

WCHAR_MAX

Synopsis

```
#define WCHAR_MAX  0xffff
```

Description

WCHAR_MAX is the maximum value for an object of type **wchar_t**.

WCHAR_MIN

Synopsis

```
#define WCHAR_MIN  0x0000
```

Description

WCHAR_MIN is the minimum value for an object of type **wchar_t**.

wcscat

Synopsis

```
wchar_t *wcscat(wchar_t *s1,
                const wchar_t *s2);
```

Description

wcscat appends a copy of the wide string pointed to by **s2** (including the terminating null wide character) to the end of the wide string pointed to by **s1**. The initial character of **s2** overwrites the null wide character at the end of **s1**. The behavior of **wcscat** is undefined if copying takes place between objects that overlap.

wcscat returns the value of **s1**.

wcschr

Synopsis

```
wchar_t *wcschr(const wchar_t *s,
                wchar_t c);
```

Description

wcschr locates the first occurrence of **c** in the wide string pointed to by **s**. The terminating wide null character is considered to be part of the string.

wcschr returns a pointer to the located wide character, or a null pointer if **c** does not occur in the string.

wcscmp

Synopsis

```
int wcscmp(const wchar_t *s1,
           const wchar_t *s2);
```

Description

wcscmp compares the wide string pointed to by **s1** to the wide string pointed to by **s2**. **wcscmp** returns an integer greater than, equal to, or less than zero if the wide string pointed to by **s1** is greater than, equal to, or less than the wide string pointed to by **s2**.

wcscpy

Synopsis

```
wchar_t *wcscpy(wchar_t *s1,
                const wchar_t *s2);
```

Description

wcscpy copies the wide string pointed to by **s2** (including the terminating null wide character) into the array pointed to by **s1**. The behavior of **wcscpy** is undefined if copying takes place between objects that overlap.

wcscpy returns the value of **s1**.

wcscspn

Synopsis

```
size_t wcscspn(const wchar_t *s1,
               const wchar_t *s2);
```

Description

wcscspn computes the length of the maximum initial segment of the wide string pointed to by **s1** which consists entirely of wide characters not from the wide string pointed to by **s2**.

wcscspn returns the length of the segment.

wcsdup

Synopsis

```
wchar_t *wcsdup(const wchar_t *s1);
```

Description

wcsdup duplicates the wide string pointed to by **s1** by using **malloc** to allocate memory for a copy of **s** and then copying **s**, including the terminating wide null character, to that memory. The returned pointer can be passed to **free**. **wcsdup** returns a pointer to the new wide string or a null pointer if the new string cannot be created.

Note

wcsdup is an extension commonly found in Linux and BSD C libraries.

wcslen

Synopsis

```
size_t wcslen(const wchar_t *s);
```

Description

wcslen returns the length of the wide string pointed to by **s**, that is the number of wide characters that precede the terminating null wide character.

wcsncat

Synopsis

```
wchar_t *wcsncat(wchar_t *s1,
                 const wchar_t *s2,
                 size_t n);
```

Description

wcsncat appends not more than **n** wude characters from the array pointed to by **s2** to the end of the wide string pointed to by **s1**. A null wide character in **s1** and wide characters that follow it are not appended. The initial wide character of **s2** overwrites the null wide character at the end of **s1**. A terminating wide null character is always appended to the result. The behavior of **wcsncat** is undefined if copying takes place between objects that overlap.

wcsncat returns the value of **s1**.

wcsnchr

Synopsis

```
wchar_t *wcsnchr(const wchar_t *str,
                 size_t n,
                 wchar_t ch);
```

Description

wcsnchr searches not more than **n** wide characters to locate the first occurrence of **c** in the wide string pointed to by **s**. The terminating wide null character is considered to be part of the wide string.

wcsnchr returns a pointer to the located wide character, or a null pointer if **c** does not occur in the string.

wcsncmp

Synopsis

```
int wcsncmp(const wchar_t *s1,
            const wchar_t *s2,
            size_t n);
```

Description

wcsncmp compares not more than **n** wide characters from the array pointed to by **s1** to the array pointed to by **s2**. Characters that follow a null wide character are not compared.

wcsncmp returns an integer greater than, equal to, or less than zero, if the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the possibly null-terminated array pointed to by **s2**.

wcsncpy

Synopsis

```
wchar_t *wcsncpy(wchar_t *s1,
                 const wchar_t *s2,
                 size_t n);
```

Description

wcsncpy copies not more than **n** wide characters from the array pointed to by **s2** to the array pointed to by **s1**. Wide characters that follow a null wide character in **s2** are not copied. The behavior of **wcsncpy** is undefined if copying takes place between objects that overlap. If the array pointed to by **s2** is a wide string that is shorter than **n** wide characters, null wide characters are appended to the copy in the array pointed to by **s1**, until **n** characters in all have been written.

wcsncpy returns the value of **s1**.

wcsnlen

Synopsis

```
size_t wcsnlen(const wchar_t *s,
               size_t n);
```

Description

this returns the length of the wide string pointed to by **s**, up to a maximum of **n** wide characters. **wcsnlen** only examines the first **n** wide characters of the string **s**.

Note

wcsnlen is an extension commonly found in Linux and BSD C libraries.

wcsnstr

Synopsis

```
wchar_t *wcsnstr(const wchar_t *s1,
                 const wchar_t *s2,
                 size_t n);
```

Description

wcsnstr searches at most **n** wide characters to locate the first occurrence in the wide string pointed to by **s1** of the sequence of wide characters (excluding the terminating null wide character) in the wide string pointed to by **s2**.

wcsnstr returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, **wcsnstr** returns **s1**.

Note

wcsnstr is an extension commonly found in Linux and BSD C libraries.

wcspbrk

Synopsis

```
wchar_t *wcspbrk(const wchar_t *s1,
                 const wchar_t *s2);
```

Description

wcspbrk locates the first occurrence in the wide string pointed to by **s1** of any wide character from the wide string pointed to by **s2**.

wcspbrk returns a pointer to the wide character, or a null pointer if no wide character from **s2** occurs in **s1**.

wcsrchr

Synopsis

```
wchar_t *wcsrchr(const wchar_t *s,
                 wchar_t c);
```

Description

wcsrchr locates the last occurrence of **c** in the wide string pointed to by **s**. The terminating wide null character is considered to be part of the string.

wcsrchr returns a pointer to the wide character, or a null pointer if **c** does not occur in the wide string.

wcsspn

Synopsis

```
size_t wcsspn(const wchar_t *s1,
              const wchar_t *s2);
```

Description

wcsspn computes the length of the maximum initial segment of the wide string pointed to by **s1** which consists entirely of wide characters from the wide string pointed to by **s2**.

wcsspn returns the length of the segment.

wcsstr

Synopsis

```
wchar_t *wcsstr(const wchar_t *s1,
                const wchar_t *s2);
```

Description

wcsstr locates the first occurrence in the wide string pointed to by **s1** of the sequence of wide characters (excluding the terminating null wide character) in the wide string pointed to by **s2**.

wcsstr returns a pointer to the located wide string, or a null pointer if the wide string is not found. If **s2** points to a wide string with zero length, **wcsstr** returns **s1**.

wcstok

Synopsis

```
wchar_t *wcstok(wchar_t *s1,
                const wchar_t *s2);
```

Description

wcstok A sequence of calls to **wcstok** breaks the wide string pointed to by **s1** into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by **s2**. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. The separator wide string pointed to by **s2** may be different from call to call.

The first call in the sequence searches the wide string pointed to by **s1** for the first wide character that is not contained in the current separator wide string pointed to by **s2**. If no such wide character is found, then there are no tokens in the wide string pointed to by **s1** and **wcstok** returns a null pointer. If such a wide character is found, it is the start of the first token.

wcstok then searches from there for a wide character that is contained in the current wide separator string. If no such wide character is found, the current token extends to the end of the wide string pointed to by **s1**, and subsequent searches for a token will return a null pointer. If such a wude character is found, it is overwritten by a wide null character, which terminates the current token. **wcstok** saves a pointer to the following wide character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Note

wcstok maintains static state and is therefore not reentrant and not thread safe. See wcstok_r for a thread-safe and reentrant variant.

wcstok_r

Synopsis

```
wchar_t *wcstok_r(wchar_t *s1,
                  const wchar_t *s2,
                  wchar_t **s3);
```

Description

wcstok_r is a reentrant version of the function **wcstok** where the state is maintained in the object of type **wchar_t** * pointed to by **s3**.

Note

wcstok_r is an extension commonly found in Linux and BSD C libraries.

See Also

wcstok.

wint_t

Synopsis

```
typedef long wint_t;
```

Description

wint_t is an integer type that is unchanged by default argument promotions that can hold any value corresponding to members of the extended character set, as well as at least one value that does not correspond to any member of the extended character set (WEOF).

wmemccpy

Synopsis

```
wchar_t *wmemccpy(wchar_t *s1,
                  const wchar_t *s2,
                  wchar_t c,
                  size_t n);
```

Description

wmemccpy copies at most **n** wide characters from the object pointed to by **s2** into the object pointed to by **s1**. The copying stops as soon as **n** wide characters are copied or the wide character **c** is copied into the destination object pointed to by **s1**. The behavior of **wmemccpy** is undefined if copying takes place between objects that overlap.

wmemccpy returns a pointer to the wide character immediately following **c** in **s1**, or **NULL** if **c** was not found in the first **n** wide characters of **s2**.

Note

wmemccpy conforms to POSIX.1-2008.

wmemchr

Synopsis

```
wchar_t *wmemchr(const wchar_t *s,
                 wchar_t c,
                 size_t n);
```

Description

wmemchr locates the first occurrence of **c** in the initial **n** characters of the object pointed to by **s**. Unlike **wcschr**, **wmemchr** does *not* terminate a search when a null wide character is found in the object pointed to by **s**.

wmemchr returns a pointer to the located wide character, or a null pointer if **c** does not occur in the object.

wmemcmp

Synopsis

```
int wmemcmp(const wchar_t *s1,
            const wchar_t *s2,
            size_t n);
```

Description

wmemcmp compares the first **n** wide characters of the object pointed to by **s1** to the first **n** wide characters of the object pointed to by **s2**. **wmemcmp** returns an integer greater than, equal to, or less than zero as the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.

wmemcpy

Synopsis

```
wchar_t *wmemcpy(wchar_t *s1,
                 const wchar_t *s2,
                 size_t n);
```

Description

wmemcpy copies **n** wide characters from the object pointed to by **s2** into the object pointed to by **s1**. The behavior of **wmemcpy** is undefined if copying takes place between objects that overlap.

wmemcpy returns the value of **s1**.

wmemmove

Synopsis

```
wchar_t *wmemmove(wchar_t *s1,
                  const wchar_t *s2,
                  size_t n);
```

Description

wmemmove copies **n** wide characters from the object pointed to by **s2** into the object pointed to by **s1** ensuring that if **s1** and **s2** overlap, the copy works correctly. Copying takes place as if the **n** wide characters from the object pointed to by **s2** are first copied into a temporary array of **n** wide characters that does not overlap the objects pointed to by **s1** and **s2**, and then the **n** wide characters from the temporary array are copied into the object pointed to by **s1**.

wmemmove returns the value of **s1**.

wmempcpy

Synopsis

```
wchar_t *wmempcpy(wchar_t *s1,
                  const wchar_t *s2,
                  size_t n);
```

Description

wmempcpy copies **n** wide characters from the object pointed to by **s2** into the object pointed to by **s1**. The behavior of **wmempcpy** is undefined if copying takes place between objects that overlap.

wmempcpy returns it returns a pointer to the wide character following the last written wide character.

Note

This is an extension found in GNU libc.

wmemset

Synopsis

```
wchar_t *wmemset(wchar_t *s,
                 wchar_t c,
                 size_t n);
```

Description

wmemset copies the value of **c** into each of the first **n** wide characters of the object pointed to by **s**.

wmemset returns the value of **s**.

wstrsep

Synopsis

```
wchar_t *wstrsep(wchar_t **stringp,
                 const wchar_t *delim);
```

Description

wstrsep locates, in the wide string referenced by ***stringp**, the first occurrence of any wide character in the wide string **delim** (or the terminating wide null character) and replaces it with a wide null character. The location of the next character after the delimiter wide character (or NULL, if the end of the string was reached) is stored in ***stringp**. The original value of ***stringp** is returned.

An empty field (that is, a wide character in the string **delim** occurs as the first wide character of ***stringp** can be detected by comparing the location referenced by the returned pointer to a wide null character.

If ***stringp** is initially null, **wstrsep** returns null.

Note

wstrsep is not an ISO C function, but appears in BSD4.4 and Linux.

C++ Library User Guide

CrossWorks provides a limited C++ library suitable for use in an embedded application.

Standard library

The following C++ standard header files are provided in `$(StudioDir)/include`:

File	Description
<cassert>	C++ wrapper on **assert.h**.
<cctype>	C++ wrapper on **ctype.h**.
<cerrno>	C++ wrapper on **errno.h**.
<cfloat>	C++ wrapper on **float.h**.
<ciso646>	C++ wrapper on **iso646.h**.
<climits>	C++ wrapper on **limits.h**.
<clocale>	C++ wrapper on **locale.h**.
<cmath>	C++ wrapper on **math.h**.
<csetjmp>	C++ wrapper on **setjmp.h**.
<cstdarg>	C++ wrapper on **stdarg.h**.
<cstddef>	C++ wrapper on **stddef.h**.
<cstdio>	C++ wrapper on **stdio.h**.
<cstdlib>	C++ wrapper on **stdlib.h**.
<cstring>	C++ wrapper on **string.h**.
<ctime>	C++ wrapper on **time.h**.
<exception>	Definitions for exceptions. *Note that this file is licensed under the GPL*.
<new>	Types and definitions for placement new and delete.
<exception>	Definitions for RTTI. *Note that this file is licensed under the GPL*.

It's worth mentioning again: to use exceptions or RTTI requires header files and library code to be linked into your application that is licensed under the GPL.

Standard template library

The C++ STL functionality of STLPort 5.1.0 is provided in CrossWorks. To use STLPort you must put `$(StudioDir)/include/stlport` as the first entry in the **User Include Directories** project property. The STLPort is configured to *not* support wide characters, long doubles, and iostreams, and not throw exceptions. The following STLPort header files are supported (not including the above list of standard C++ header files)

<algorithm>	<bitset>	<deque>
<functional>	<hash_map>	<hash_set>
<iterator>	<limits>	<list>
<locale>	<map>	<memory>
<numeric>	<queue>	<set>
<stack>	<stdexcept>	<string>
<utility>	<valarray>	<vector>

Subset API reference

This section contains a subset reference to the CrossWorks C++ library.

<new> - memory allocation

The header file <**new**> defines functions for memory allocation.

Functions	
set_new_handler	Establish a function which is called when memory allocation fails.
Operators	
operator delete	Heap storage deallocators operator.
operator new	Heap storage allocators operator.

operator delete

Synopsis

```
void operator delete(void *ptr) throw();
void operator delete[](void *ptr) throw();
```

Description

operator delete deallocates space of an object.

operator delete will do nothing if **ptr** is null. If **ptr** is not null then it should have been returned from a call to **operator new**.

operator delete[] has the same behaviour as **operator delete** but is used for array deallocation.

Portability

Standard C++.

operator new

Synopsis

```
void *operator new(size_t size) throw();
void *operator new[](size_t size) throw();
```

Description

operator new allocates space for an object whose size is specified by **size** and whose value is indeterminate.

operator new returns a null pointer if the space for the object cannot be allocated from free memory; if space for the object can be allocated, **operator new** returns a pointer to the start of the allocated space.

operator new[] has the same behaviour as **operator new** but is used for array allocation.

Portability

The implementation is not standard. The standard C++ implementation should throw an exception if memory allocation fails.

set_new_handler

Synopsis

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler) throw();
```

Description

set_new_handler establishes a **new_handler** function.

set_new_handler establishes a **new_handler** function that is called when **operator new** fails to allocate the requested memory. If the **new_handler** function returns then **operator new** will attempt to allocate the memory again. The **new_handler** function can throw an exception to implement standard C++ behaviour for memory allocation failure.

Portability

Standard C++.

LIBMEM User Guide

The aim of LIBMEM is to provide a common programming interface for a wide range of different memory types.

LIBMEM consists of a mechanism for installing drivers for the different memories and a set of common memory access and control functions that locate the driver for a particular memory range and call the appropriate memory driver functions for the operation.

The LIBMEM library also includes a set of memory drivers for common memory devices.

Using the LIBMEM library

Probably the best way to demonstrate LIBMEM is to see it in use. The following example demonstrates copying a block of data into FLASH using a LIBMEM common flash interface (CFI) driver.

```
int libmem_example_1(void)
{
  const int flash1_max_geometry_regions = 4;
  libmem_driver_handle_t  flash1_handle;
  libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
  libmem_flash_info_t flash1_info;
  uint8_t *flash1_start = (uint8_t *)0x10000000;
  uint8_t *write_dest = flash1_start + 16;
  const uint8_t write_data[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
  int res;
  // Register the FLASH LIBMEM driver
  res = libmem_register_cfi_driver(&flash1_handle,
                                   flash1_start,
                                   flash1_geometry,
                                   flash1_max_geometry_regions,
                                   &flash1_info);
  if (res != LIBMEM_STATUS_SUCCESS)
    return 0;
  // Unlock the destination memory area.
  res = libmem_unlock(write_dest, sizeof(write_data));
  if (res != LIBMEM_STATUS_SUCCESS)
    return 0;
  // Erase the destination memory area.
  res = libmem_erase(write_dest, sizeof(write_data), 0, 0);
  if (res != LIBMEM_STATUS_SUCCESS)
    return 0;
  // Copy write_data to the destination memory area.
  res = libmem_write(write_dest, write_data, sizeof(write_data));
  if (res != LIBMEM_STATUS_SUCCESS)
    return 0;
  // Complete any outstanding transactions and put FLASH memory back into read mode.
  res = libmem_flush();
  if (res != LIBMEM_STATUS_SUCCESS)
    return 0;
  return 1;
}
```

The following section describes each of the LIBMEM calls in the preceding example in detail.

Before any memory operations can be carried out the LIBMEM drivers that you are going to use must be registered. The following code registers a LIBMEM CFI driver for a FLASH device located at the memory location pointed to by **flash1_start**.

```
// Register the FLASH LIBMEM driver
res = libmem_register_cfi_driver(&flash1_handle,
                                 flash1_start,
                                 flash1_geometry,
                                 flash1_max_geometry_regions,
                                 &flash1_info);
if (res != LIBMEM_STATUS_SUCCESS)
  return 0;
```

This call attempts to detect the type of FLASH and register the correct LIBMEM CFI driver based on the CFI information read out from the FLASH device. Note that using this function will link in all LIBMEM CFI drivers so in your own application you may wish to save memory by using **libmem_cfi_get_info** to get out the FLASH geometry information and registering a specific CFI driver. You may also save further memory and time by not calling **libmem_cfi_get_info** and specifying the FLASH geometry yourself.

For each driver you register you must allocate **libmem_driver_handle_t** structure to act as a handle for the driver. Will the full version of LIBMEM you can register as many drivers as you wish, if you are using the 'light' version of LIBMEM you can only register one driver.

Once you have registered your drivers you can use the general LIBMEM memory functions to access and control your memory. The starting address passed to these functions is used to decide which driver to use for the memory operation, operations cannot span multiple drivers.

The next operation the example code carries out it to unlock the FLASH in preparation for the erase and write operations. Unlocking is not necessary on all memory devices and this operation is not implemented in all LIBMEM drivers.

```
// Unlock the destination memory area.
res = libmem_unlock(write_dest, sizeof(write_data));
if (res != LIBMEM_STATUS_SUCCESS)
  return 0;
```

Once the memory has been unlocked the FLASH memory is erased. Once again erasing is not necessary on all memory devices and this operation may not be implemented in all LIBMEM drivers.

```
// Erase the destination memory area.
res = libmem_erase(write_dest, sizeof(write_data), 0, 0);
if (res != LIBMEM_STATUS_SUCCESS)
  return 0;
```

Parameters three and four of **libmem_erase** are not used in this example, however they provide a mechanism to allow the caller to determine how much memory was actually erased by the erase operation as it may well be more than requested.

Once the FLASH memory has been erased the FLASH can be programmed using the **libmem_write** function.

```
// Copy write_data to the destination memory area.
res = libmem_write(write_dest, write_data, sizeof(write_data));
if (res != LIBMEM_STATUS_SUCCESS)
  return 0;
```

The final step is to call **libmem_flush**. Once again flushing is not necessary on all memory devices, but some LIBMEM drivers do not necessarily carry out operations immediately or they may leave the memory in an unreadable state for performance reasons and calling **libmem_flush** is required to flush outstanding operations and return the device to read mode.

```
// Complete any outstanding transactions and put FLASH memory back into read mode.
res = libmem_flush();
if (res != LIBMEM_STATUS_SUCCESS)
  return 0;
```

Typically you would now access the FLASH memory as you would any other memory and read it directly, LIBMEM does however provide the **libmem_read** function for accessing memory that is not directly accessibly by the CPU.

Light version of LIBMEM

LIBMEM is built in two configurations, the full version and the light version. The only difference between the full and the light versions of LIBMEM is that the light version only supports one installed LIBMEM driver and is compiled with optimization for code size rather than performance. The light version of LIBMEM is therefore useful for situations where code memory is at a premium.

To use the light version of LIBMEM you should link in the light version of the library and also have the preprocessor definition **LIBMEM_LIGHT** defined when including **libmem.h**.

Writing LIBMEM drivers

LIBMEM includes a set of memory drivers for common memory devices which means in most cases you probably won't need to write a LIBMEM driver. If however you wish to use LIBMEM to drive other unsupported memory devices you will need to write your own LIBMEM driver.

It is fairly straightforward to implement a LIBMEM driver, the following example demonstrates the implementation of a minimal LIBMEM driver:

```
#include <libmem.h>
static int
libmem_write_impl(libmem_driver_handle_t *h, uint8_t *dest, const uint8_t *src, size_t siz
{
  // TODO: Implement memory write operation.
  return LIBMEM_STATUS_SUCCESS;
}
static int
libmem_fill_impl(libmem_driver_handle_t *h, uint8_t *dest, uint8_t c, size_t size)
{
  // TODO: Implement memory fill operation.
  return LIBMEM_STATUS_SUCCESS;
}
static int
libmem_erase_impl(libmem_driver_handle_t *h, uint8_t *start, size_t size,
                  uint8_t **erase_start, size_t *erase_size)
{
  // TODO: Implement memory erase operation.
  if (erase_start)
    {
      // TODO: Set erase_start to point to the start of the memory block that
      //       has been erased. For now we'll just return the requested start in
      //       order to keep the caller happy.
 erase_start = start;
    }
  if (erase_size)
    {
      // TODO: Set erase_size to the size of the memory block that has been
      //       erased. For now we'll just return the requested size in order to
      //       keep the caller happy.
 erase_size = size;
    }
  return LIBMEM_STATUS_SUCCESS;
```

```
}
static int
libmem_lock_impl(libmem_driver_handle_t *h, uint8_t *dest, size_t size)
{
  // TODO: Implement memory lock operation
  return LIBMEM_STATUS_SUCCESS;
}
static int
libmem_unlock_impl(libmem_driver_handle_t *h, uint8_t *dest, size_t size)
{
  // TODO: Implement memory unlock operation.
  return LIBMEM_STATUS_SUCCESS;
}
static int
libmem_flush_impl(libmem_driver_handle_t *h)
{
  // TODO: Implement memory flush operation.
  return LIBMEM_STATUS_SUCCESS;
}
static const libmem_driver_functions_t driver_functions =
{
  libmem_write_impl,
  libmem_fill_impl,
  libmem_erase_impl,
  libmem_lock_impl,
  libmem_unlock_impl,
  libmem_flush_impl
};
int
libmem_register_example_driver_1(libmem_driver_handle_t *h, uint8_t *start, size_t size)
{
  libmem_register_driver(h, start, size, 0, 0, &driver_functions, 0);
  return LIBMEM_STATUS_SUCCESS;
}
```

For some types of memory it is necessary to carry out operations on a per-sector basis, in this case it can be useful to register a geometry with the driver and use the geometry helper functions. For example the following code demonstrates how you might implement a driver that can only erase the entire memory or individual sectors.

```
static int
driver_erase_sector(libmem_driver_handle_t *h, libmem_sector_info_t *si)
```

```
{
  // TODO: Implement sector erase for sector starting at si->start
  return LIBMEM_STATUS_SUCCESS;
}
static int
driver_erase_chip(libmem_driver_handle_t *h)
{
  // TODO: Implement chip erase
  return LIBMEM_STATUS_SUCCESS;
}
static int
libmem_erase_impl(libmem_driver_handle_t *h, uint8_t *start, size_t size,
                  uint8_t **erase_start, size_t *erase_size)
{
  int res;
  if (LIBMEM_RANGE_WITHIN_RANGE(h->start, h->start + h-
>size - 1, start, start + size - 1))
    {
      res = driver_erase_chip(h);
      if (erase_start)
 erase_start = h->start;
      if (erase_size)
 erase_size = h->size;
    }
  else
    res = libmem_foreach_sector_in_range(h, start, size, driver_erase_sector, erase_start,
  return res;
}
static const libmem_geometry_t geometry[] =
{
  { 8,  0x00002000 }, // 8 x 8KB sectors
  { 31, 0x00010000 }, // 31 x 64KB sectors
  { 0, 0 }            // NULL terminator
};
int
libmem_register_example_driver_2(libmem_driver_handle_t *h, uint8_t *start, size_t size)
{
  libmem_register_driver(h, start, size, geometry, 0, &driver_functions, 0);
  return LIBMEM_STATUS_SUCCESS;
}
```

There are two sets of driver entry point functions, the standard set that include functions common to most LIBMEM drivers which have been described above and the extended set which provide extra functionality for less common types of driver. The following example demonstrates how you would also register a set of extended LIBMEM driver functions in your driver:

```
static int
libmem_inrange_impl(libmem_driver_handle_t *h, const uint8_t *dest)
{
  // TODO: Implement inrange function (return non-
zero if dest is within range
  //       handled by driver).
  return 0;
}
static int
libmem_read_impl(libmem_driver_handle_t *h, uint8_t *dest, const uint8_t *src, size_t size
{
  // TODO: Implement memory read operation
  return LIBMEM_STATUS_SUCCESS;
}
static uint32_t
libmem_crc32_impl(libmem_driver_handle_t *h, const uint8_t *start, size_t size, uint32_t c
{
  // TODO: Implement CRC-32 operation.
  return crc;
}
int
libmem_register_example_driver_3(libmem_driver_handle_t *h, uint8_t *start, size_t size)
{
  libmem_register_driver(h, start, size, geometry, 0, &driver_functions, &ext_driver_funct
  return LIBMEM_STATUS_SUCCESS;
}
```

LIBMEM loader library

The aim of the LIBMEM loader library is to be an add on to the LIBMEM library that simplifies the writing of loader applications.

To write a loader application all you need to do is register the LIBMEM drivers you require and then call the appropriate loader start function for the communication mechanism you wish to use.

For example, the following code is an example of a LIBMEM loader, it registers one LIBMEM FLASH driver, if the driver is successfully registered it starts up the loader by calling **libmem_rpc_loader_start**. Finally it tells the host that the loader has finished by calling **libmem_rpc_loader_exit**.

```
static unsigned char buffer[256];
int main(void)
{
  uint8_t *flash1_start = (uint8_t *)0x10000000;
  const int flash1_max_geometry_regions = 4;
  libmem_driver_handle_t flash1_handle;
  libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
  libmem_flash_info_t flash1_info;
  int res;
  // Register FLASH driver.
  res = libmem_register_cfi_driver(&flash1_handle,
                                   flash1_start,
                                   flash1_geometry,
                                   flash1_max_geometry_regions,
                                   &flash1_info);
  if (res == LIBMEM_STATUS_SUCCESS)
    {
      // Run the loader
      libmem_rpc_loader_start(buffer, buffer + sizeof(buffer) - 1);
    }

  libmem_rpc_loader_exit(res, NULL);
  return 0;
}
```

Essentially, a LIBMEM loader is just a standard RAM-based application that registers the LIBMEM drivers required by the loader and then calls the appropriate loader start function for the communication mechanism being used.

The LIBMEM loader library currently supports three different loader communication mechanisms.
You select which one to use by calling **libmem_rpc_loader_start**, **libmem_dcc_rpc_loader_start**, or

libmem_dcc_loader_start to start the loader. The documentation for each of these functions describes how each of these communication mechanisms work.

A significant difference between LIBMEM loader applications and regular applications is that once the loader start function is called it is no longer possible to debug the application using the debugger. Therefore if you need to debug your loader application using the debugger you can do it by simply adding calls to the functions you wish to debug in place of the loader start call.

Complete API reference

This section contains a complete reference to the LIBMEM API.

<libmem.h>

API Summary

Utility macros	
LIBMEM_ADDRESS_IN_RANGE	Determine whether an address is within an address range
LIBMEM_ADDRESS_IS_ALIGNED	Determine whether an address is aligned to a specified width
LIBMEM_ALIGNED_ADDRESS	Return an address aligned to a specified width
LIBMEM_KB	Convert kilobytes to bytes
LIBMEM_MB	Convert megabytes to bytes
LIBMEM_RANGE_OCCLUDES_RANGE	Determine whether an address range overlaps another address range or vice versa
LIBMEM_RANGE_OVERLAPS_RANGE	Determine whether an address range overlaps another address range
LIBMEM_RANGE_WITHIN_RANGE	Determine whether an address range is within another address range
Return codes	
LIBMEM_STATUS_CFI_ERROR	Error reading CFI information return code
LIBMEM_STATUS_ERROR	Non-specific error return code
LIBMEM_STATUS_GEOMETRY_REGION_OVERFLOW	No room for geometry information return code
LIBMEM_STATUS_INVALID_DEVICE	Invalid or mismatched device return code
LIBMEM_STATUS_INVALID_PARAMETER	Invalid parameter return code
LIBMEM_STATUS_INVALID_RANGE	Invalid range return code
LIBMEM_STATUS_INVALID_WIDTH	Invalid or unsupported device width return code
LIBMEM_STATUS_LOCKED	Memory locked return code
LIBMEM_STATUS_NOT_IMPLEMENTED	Not implemented return code
LIBMEM_STATUS_NO_DRIVER	No driver for memory range return code
LIBMEM_STATUS_SUCCESS	Successful operation return code
LIBMEM_STATUS_TIMEOUT	Timeout error return code
Command set macros	
LIBMEM_CFI_CMDSET_AMD_EXTENDED	AMD standard command set
LIBMEM_CFI_CMDSET_AMD_STANDARD	AMD standard command set
LIBMEM_CFI_CMDSET_INTEL_EXTENDED	Intel extended command set
LIBMEM_CFI_CMDSET_INTEL_STANDARD	Intel standard command set

LIBMEM_CFI_CMDSET_MITSUBISHI_EXTENDED	Mitsubishi extended command set
LIBMEM_CFI_CMDSET_MITSUBISHI_STANDARD	Mitsubishi standard command set
LIBMEM_CFI_CMDSET_NONE	Invalid CFI command set
LIBMEM_CFI_CMDSET_RESERVED	Reserved command set
LIBMEM_CFI_CMDSET_SST_PAGE_WRITE	SST page write command set
LIBMEM_CFI_CMDSET_WINBOND_STANDARD	Winbond standard command set
Macros	
LIBMEM_VERSION_NUMBER	LIBMEM interface version number
Configuration macros	
LIBMEM_INLINE	Inline definition
Functions	
libmem_cfi_get_info	Return a FLASH memory device's common flash interface (CFI) information
libmem_crc32	Compute CRC-32 checksum
libmem_crc32_direct	Compute CRC-32 checksum of an address range
libmem_enable_timeouts	Enable LIBMEM operation timeouts
libmem_erase	Erase a block of memory using a LIBMEM driver
libmem_erase_all	Erase all memory using LIBMEM drivers
libmem_fill	Fill memory with a specific data value using a LIBMEM driver
libmem_flush	Flush any outstanding memory operations and return memory to read mode if applicable
libmem_foreach_driver	Iterate through all drivers
libmem_foreach_sector	Iterate through all sectors of a driver
libmem_foreach_sector_in_range	Iterate through subset of sectors of a driver
libmem_foreach_sector_in_range_ex	A helper function for iterating through all sectors in a specified geometry that are within a specific address range
libmem_get_driver	Look up driver for address
libmem_get_driver_sector_size	A helper function that locates the driver for a specific address and then returns the sector size for that address using the driver's geometry
libmem_get_geometry_size	A helper function that returns the size of the address range described by a geometry description
libmem_get_number_of_regions	A helper function that returns the number of geometry regions described by a geometry description
libmem_get_number_of_sectors	A helper function that returns the number of sectors described by a geometry description

libmem_get_sector_info	A helper function that returns the sector information for an address within a specified geometry
libmem_get_sector_number	A helper function that returns the sector number of an address within a specified geometry
libmem_get_sector_size	A helper function that returns the sector size for an address within a specified geometry
libmem_get_ticks	Helper function that returns the current timer tick count
libmem_lock	Lock a block of memory using a LIBMEM driver
libmem_lock_all	Lock all memory using LIBMEM drivers
libmem_read	Read a block of data using a LIBMEM driver
libmem_register_driver	Register a LIBMEM driver instance
libmem_set_busy_handler	Specify busy loop function
libmem_unlock	Unlock a block of memory using a LIBMEM driver
libmem_unlock_all	Unlock all memory using LIBMEM drivers
libmem_write	Write a block of data using a LIBMEM driver
Generic FLASH drivers	
libmem_register_cfi_0001_16_driver	Register a 16-bit CFI command set 1 (Intel Extended) LIBMEM driver
libmem_register_cfi_0001_8_driver	Register an 8-bit CFI command set 1 (Intel Extended) LIBMEM driver
libmem_register_cfi_0002_16_driver	Register a 16-bit CFI command set 2 (AMD Standard) LIBMEM driver
libmem_register_cfi_0002_8_driver	Register an 8 bit CFI command set 2 (AMD Standard) LIBMEM driver
libmem_register_cfi_0003_16_driver	Register a 16-bit CFI command set 3 (Intel Standard) LIBMEM driver
libmem_register_cfi_0003_8_driver	Register an 8-bit CFI command set 3 (Intel Standard) LIBMEM driver
libmem_register_cfi_amd_driver	Register a multi-width CFI command set 2 (AMD) LIBMEM driver
libmem_register_cfi_driver	Register a FLASH driver based on detected CFI information
libmem_register_cfi_intel_driver	Register a combined multi-width CFI command set 1 and 3 (Intel) LIBMEM driver
FLASH drivers	
libmem_register_am29f200b_driver	Register a driver for an AMD Am29F200B FLASH chip
libmem_register_am29f200t_driver	Register a driver for an AMD Am29F200T FLASH chip

libmem_register_am29f400bb_driver	Register a driver for an AMD Am29F400BB FLASH chip
libmem_register_am29f400bt_driver	Register a driver for an AMD Am29F400BT FLASH chip
libmem_register_am29fxxx_driver	Register a driver for an AMD Am29Fxxx FLASH chip
libmem_register_am29lv010b_driver	Register a driver for an AMD Am29LV010B FLASH chip
libmem_register_sst39xFx00A_16_driver	Register a driver for a 16-bit SST39xFx00A FLASH chip
libmem_register_st_m28w320cb_driver	Register a driver for an ST M28W320CB FLASH chip
libmem_register_st_m28w320ct_driver	Register a driver for an ST M28W320CT FLASH chip
RAM drivers	
libmem_register_ram_driver	Register a simple driver that directly accesses RAM
Data types	
_libmem_driver_functions_t	Structure containing pointers to a LIBMEM driver's functions
_libmem_driver_handle_t	LIBMEM driver handle structure
_libmem_ext_driver_functions_t	Structure containing pointers to a LIBMEM driver's extended functions
_libmem_flash_info_t	Structure containing information about a specific FLASH chip
_libmem_geometry_t	Structure describing a geometry region
_libmem_sector_info_t	Structure describing a sector
Static data	
libmem_busy_handler_fn	Pointer to a function that should be called each time LIBMEM iterates a busy loop
libmem_drivers	Pointer to the first registered LIBMEM driver
libmem_get_ticks_fn	Pointer to a function that returns the current timer tick count
libmem_ticks_per_second	How fast the tick increments
Function pointers	
libmem_busy_handler_fn_t	A pointer to a function to be called each time LIBMEM iterates a busy loop
libmem_driver_crc32_fn_t	A function pointer to a LIBMEM driver's crc32 extended function
libmem_driver_erase_fn_t	A function pointer to a LIBMEM driver's erase function
libmem_driver_fill_fn_t	A function pointer to a LIBMEM driver's fill function
libmem_driver_flush_fn_t	A function pointer to a LIBMEM driver's flush function
libmem_driver_inrange_fn_t	A function pointer to a LIBMEM driver's inrange extended function
libmem_driver_lock_fn_t	A function pointer to a LIBMEM driver's lock function

libmem_driver_read_fn_t	A function pointer to a LIBMEM driver's read extended function
libmem_driver_unlock_fn_t	A function pointer to a LIBMEM driver's unlock function
libmem_driver_write_fn_t	A function pointer to a LIBMEM driver's write function
libmem_foreach_driver_fn_t	A function pointer to a function handling a libmem_foreach_driver call
libmem_foreach_sector_fn_t	A function pointer to a function handling a libmem_foreach_sector or libmem_foreach_sector_in_range call
libmem_get_ticks_fn_t	A pointer to a function returning the current timer tick count

LIBMEM_ADDRESS_IN_RANGE

Synopsis

```
#define LIBMEM_ADDRESS_IN_RANGE(address, startAddress, endAddress) ((address >= startAddress)
 && (address <= endAddress))
```

Description

LIBMEM_ADDRESS_IN_RANGE is used to determine whether an address is within an address range.

address — The address to check.

startAddress — The start address of the address range.

endAddress — The end address of the address range.

LIBMEM_ADDRESS_IN_RANGE returns — Non-zero if address is within address range.

LIBMEM_ADDRESS_IS_ALIGNED

Synopsis

```
#define LIBMEM_ADDRESS_IS_ALIGNED(address, width) \
   ((((uint32_t)address) & ((width) - 1)) == 0)
```

Description

LIBMEM_ADDRESS_IS_ALIGNED is used to determine whether an address is aligned to a specified width.

address — The address to check alignment of.

width — The alignment width.

LIBMEM_ADDRESS_IS_ALIGNED returns — Non-zero if address is aligned.

LIBMEM_ALIGNED_ADDRESS

Synopsis

```
#define LIBMEM_ALIGNED_ADDRESS(address, width) \
   ((uint8_t *)(((uint32_t)address) & (~((width) - 1))))
```

Description

LIBMEM_ALIGNED_ADDRESS returns an address aligned to a specified width.

address — The address to align.

width — The alignment width.

LIBMEM_ALIGNED_ADDRESS returns — The aligned address.

LIBMEM_CFI_CMDSET_AMD_EXTENDED

Synopsis

```
#define LIBMEM_CFI_CMDSET_AMD_EXTENDED          (0x0004)
```

Description

A definition representing the CFI command set number for the AMD extended command set.

LIBMEM_CFI_CMDSET_AMD_STANDARD

Synopsis

```
#define LIBMEM_CFI_CMDSET_AMD_STANDARD          (0x0002)
```

Description

A definition representing the CFI command set number for the AMD standard command set.

LIBMEM_CFI_CMDSET_INTEL_EXTENDED

Synopsis

```
#define LIBMEM_CFI_CMDSET_INTEL_EXTENDED         (0x0001)
```

Description

A definition representing the CFI command set number for the Intel extended command set.

LIBMEM_CFI_CMDSET_INTEL_STANDARD

Synopsis

```
#define LIBMEM_CFI_CMDSET_INTEL_STANDARD         (0x0003)
```

Description

A definition representing the CFI command set number for the Intel standard command set.

LIBMEM_CFI_CMDSET_MITSUBISHI_EXTENDED

Synopsis

```
#define LIBMEM_CFI_CMDSET_MITSUBISHI_EXTENDED    (0x0101)
```

Description

A definition representing the CFI command set number for the Mitsubishi extended command set.

LIBMEM_CFI_CMDSET_MITSUBISHI_STANDARD

Synopsis

```
#define LIBMEM_CFI_CMDSET_MITSUBISHI_STANDARD    (0x0100)
```

Description

A definition representing the CFI command set number for the Mitsubishi standard command set.

LIBMEM_CFI_CMDSET_NONE

Synopsis

```
#define LIBMEM_CFI_CMDSET_NONE                  (0x0000)
```

Description

A definition representing an invalid CFI command set number.

LIBMEM_CFI_CMDSET_RESERVED

Synopsis

```
#define LIBMEM_CFI_CMDSET_RESERVED              (0xFFFF)
```

Description

A definition representing the reserved CFI command set number.

LIBMEM_CFI_CMDSET_SST_PAGE_WRITE

Synopsis

```
#define LIBMEM_CFI_CMDSET_SST_PAGE_WRITE         (0x0102)
```

Description

A definition representing the CFI command set number for the SST page write command set.

LIBMEM_CFI_CMDSET_WINBOND_STANDARD

Synopsis

```
#define LIBMEM_CFI_CMDSET_WINBOND_STANDARD       (0x0006)
```

Description

A definition representing the CFI command set number for the Winbond standard command set.

LIBMEM_INLINE

Synopsis

```
#define LIBMEM_INLINE inline
```

Description

This definition contains the `inline` keyword if function inlining should be used. This definition is empty for the LIBMEM_LIGHT build.

LIBMEM_KB

Synopsis

```
#define LIBMEM_KB(X) ((X)*1024)
```

Description

LIBMEM_KB converts kilobytes to bytes, e.g. LIBMEM_KB(10) = 10*1024.

LIBMEM_MB

Synopsis

```
#define LIBMEM_MB(X) (LIBMEM_KB(X)*1024)
```

Description

LIBMEM_MB converts megabytes to bytes, e.g. LIBMEM_MB(10) = 10*1024*1024.

LIBMEM_RANGE_OCCLUDES_RANGE

Synopsis

```
#define LIBMEM_RANGE_OCCLUDES_RANGE(r1StartAddress, r1EndAddress, r2StartAddress,
 r2EndAddress) (LIBMEM_RANGE_OVERLAPS_RANGE(r1StartAddress, r1EndAddress, r2StartAddress,
 r2EndAddress) || LIBMEM_RANGE_OVERLAPS_RANGE(r2StartAddress, r2EndAddress, r1StartAddress,
 r1EndAddress))
```

Description

LIBMEM_RANGE_OCCLUDES_RANGE is used to determine whether address range 1 overlaps address range 2 or vice versa.

r1StartAddress — The start address of address range 1.

r1EndAddress — The end address of address range 1.

r2StartAddress — The start address of address range 2.

r2EndAddress — The end address of address range 2.

LIBMEM_RANGE_OCCLUDES_RANGE returns — Non-zero if address range 1 overlaps address range 2 or address range 2 overlaps address range 1.

LIBMEM_RANGE_OVERLAPS_RANGE

Synopsis

```
#define LIBMEM_RANGE_OVERLAPS_RANGE(r1StartAddress, r1EndAddress, r2StartAddress,
 r2EndAddress) (LIBMEM_ADDRESS_IN_RANGE(r1StartAddress, r2StartAddress, r2EndAddress) ||
 LIBMEM_ADDRESS_IN_RANGE(r1EndAddress, r2StartAddress, r2EndAddress))
```

Description

LIBMEM_RANGE_OVERLAPS_RANGE is used to determine whether address range 1 overlaps address range 2.

r1StartAddress — The start address of address range 1.

r1EndAddress — The end address of address range 1.

r2StartAddress — The start address of address range 2.

r2EndAddress — The end address of address range 2.

LIBMEM_RANGE_OVERLAPS_RANGE returns — Non-zero if address range 1 overlaps address range 2.

LIBMEM_RANGE_WITHIN_RANGE

Synopsis

```
#define LIBMEM_RANGE_WITHIN_RANGE(r1StartAddress, r1EndAddress, r2StartAddress,
 r2EndAddress) (LIBMEM_ADDRESS_IN_RANGE(r1StartAddress, r2StartAddress, r2EndAddress) &&
 LIBMEM_ADDRESS_IN_RANGE(r1EndAddress, r2StartAddress, r2EndAddress))
```

Description

LIBMEM_RANGE_WITHIN_RANGE is used to determine whether address range 1 is within address range 2.

r1StartAddress — The start address of address range 1.

r1EndAddress — The end address of address range 1.

r2StartAddress — The start address of address range 2.

r2EndAddress — The end address of address range 2.

LIBMEM_RANGE_WITHIN_RANGE returns — Non-zero if address range 1 is within address range 2.

LIBMEM_STATUS_CFI_ERROR

Synopsis

```
#define LIBMEM_STATUS_CFI_ERROR                 (-6)
```

Description

Status result returned from LIBMEM functions indicating that an error has been detected reading out the CFI information.

LIBMEM_STATUS_ERROR

Synopsis

```
#define LIBMEM_STATUS_ERROR                     (0)
```

Description

Status result returned from LIBMEM functions indicating a non-specific error.

LIBMEM_STATUS_GEOMETRY_REGION_OVERFLOW

Synopsis

```
#define LIBMEM_STATUS_GEOMETRY_REGION_OVERFLOW   (-4)
```

Description

Status result returned from LIBMEM functions indicating that there is not enough room to store all the geometry region information.

LIBMEM_STATUS_INVALID_DEVICE

Synopsis

```
#define LIBMEM_STATUS_INVALID_DEVICE            (-10)
```

Description

Status result returned from LIBMEM functions indicating that the driver has determined that the expected and actual device IDs do not match.

LIBMEM_STATUS_INVALID_PARAMETER

Synopsis

```
#define LIBMEM_STATUS_INVALID_PARAMETER          (-8)
```

Description

Status result returned from LIBMEM functions indicating that an invalid parameter has been passed to the function.

LIBMEM_STATUS_INVALID_RANGE

Synopsis

```
#define LIBMEM_STATUS_INVALID_RANGE             (-7)
```

Description

Status result returned from LIBMEM functions indicating that an invalid address range has been passed to the function.

LIBMEM_STATUS_INVALID_WIDTH

Synopsis

```
#define LIBMEM_STATUS_INVALID_WIDTH             (-9)
```

Description

Status result returned from LIBMEM functions indicating that an invalid or unsupported device width has been passed to the function.

LIBMEM_STATUS_LOCKED

Synopsis

```
#define LIBMEM_STATUS_LOCKED                    (-2)
```

Description

Status result returned from LIBMEM functions indicating that the operation could not be completed because the memory is locked.

LIBMEM_STATUS_NOT_IMPLEMENTED

Synopsis

```
#define LIBMEM_STATUS_NOT_IMPLEMENTED           (-3)
```

Description

Status result returned from LIBMEM functions indicating that the operation being carried out has not been implemented in the LIBMEM driver.

LIBMEM_STATUS_NO_DRIVER

Synopsis

```
#define LIBMEM_STATUS_NO_DRIVER                 (-5)
```

Description

Status result returned from LIBMEM functions indicating that no driver has been installed for the region of memory being used.

LIBMEM_STATUS_SUCCESS

Synopsis

```
#define LIBMEM_STATUS_SUCCESS                    (1)
```

Description

Status result returned from LIBMEM functions indicating success.

LIBMEM_STATUS_TIMEOUT

Synopsis

```
#define LIBMEM_STATUS_TIMEOUT                   (-1)
```

Description

Status result returned from LIBMEM functions indicating that the operation has timed out.

LIBMEM_VERSION_NUMBER

Synopsis

```
#define LIBMEM_VERSION_NUMBER 3
```

Description

The LIBMEM interface version number.

_libmem_driver_functions_t

Synopsis

```
typedef struct {
  libmem_driver_write_fn_t write;
  libmem_driver_fill_fn_t fill;
  libmem_driver_erase_fn_t erase;
  libmem_driver_lock_fn_t lock;
  libmem_driver_unlock_fn_t unlock;
  libmem_driver_flush_fn_t flush;
} _libmem_driver_functions_t;
```

Description

_libmem_driver_functions_t is a structure containing pointers to a LIBMEM driver's functions.

Member	Description
write	A pointer to a LIBMEM driver's write function
fill	A pointer to a LIBMEM driver's fill function
erase	A pointer to a LIBMEM driver's erase function
lock	A pointer to a LIBMEM driver's lock function
unlock	A pointer to a LIBMEM driver's unlock function
flush	A pointer to a LIBMEM driver's flush function

_libmem_driver_handle_t

Synopsis

```
typedef struct {
  libmem_driver_handle_t *next;
  const libmem_driver_functions_t *driver_functions;
  const libmem_ext_driver_functions_t *ext_driver_functions;
  uint8_t *start;
  size_t size;
  const libmem_geometry_t *geometry;
  const libmem_flash_info_t *flash_info;
  uint32_t driver_data;
  uint32_t user_data;
} _libmem_driver_handle_t;
```

Description

_libmem_driver_handle_t contains information on a particular driver's entry point functions, the address range the driver is responsible for and optionally the geometry and device specific information of the memory.

Member	Description
next	The next LIBMEM driver in list of drivers
driver_functions	A pointer to the structure describing the LIBMEM driver's functions
ext_driver_functions	A pointer to the structure describing the LIBMEM driver's extended functions
start	A pointer to the start of the address range handled by the LIBMEM driver
size	The size of address range handled by the LIBMEM driver in bytes
geometry	A pointer to a null-terminated geometry description list
flash_info	A pointer to the FLASH information structure
driver_data	A data word available for storing driver information
user_data	A data word available for storing user information

_libmem_ext_driver_functions_t

Synopsis

```
typedef struct {
  libmem_driver_inrange_fn_t inrange;
  libmem_driver_read_fn_t read;
  libmem_driver_crc32_fn_t crc32;
} _libmem_ext_driver_functions_t;
```

Description

_libmem_ext_driver_functions_t is a structure containing pointers to a LIBMEM driver's extended functions.

Member	Description
inrange	A pointer to a LIBMEM driver's **inrange** function
read	A pointer to a LIBMEM driver's **read** function
crc32	A pointer to a LIBMEM driver's **crc32** function

_libmem_flash_info_t

Synopsis

```
typedef struct {
  uint32_t write_timeout_ticks;
  uint32_t multi_write_timeout_ticks;
  uint32_t erase_sector_timeout_ticks;
  uint32_t erase_chip_timeout_ticks;
  uint32_t max_multi_program_bytes;
  uint16_t primary_cmdset;
  uint8_t width;
  uint8_t pairing;
} _libmem_flash_info_t;
```

Description

_libmem_flash_info_t is a structure containing information about a specific FLASH chip.

Member	Description
write_timeout_ticks	The maximum number of ticks it should take for a write operation to complete
multi_write_timeout_ticks	The maximum number of ticks it should take for a multi-byte write operation to complete
erase_sector_timeout_ticks	The maximum number of ticks it should take for a sector erase operation to complete
erase_chip_timeout_ticks	The maximum number of ticks it should take for a chip erase operation to complete
max_multi_program_bytes	The maximum number of bytes that can be programmed in a multi-program operation
primary_cmdset	The FLASH chip's primary CFI command set
width	The operating width of the FLASH chip in bytes
pairing	Non-zero if using a paired FLASH configuration

_libmem_geometry_t

Synopsis

```
typedef struct {
  unsigned int count;
  size_t size;
} _libmem_geometry_t;
```

Description

_libmem_geometry_t describes a geometry region.

A geometry description can be made up of one or more geometry regions. A geometry region is a collection of equal-size sectors.

Member	Description
count	The number of equal-sized sectors in the geometry region
size	The size of the sector

_libmem_sector_info_t

Synopsis

```
typedef struct {
  int number;
  uint8_t *start;
  size_t size;
} _libmem_sector_info_t;
```

Description

_libmem_sector_info_t is a structure describing a sector.

Member	Description
number	The sector number (sectors in a geometry are numbered in order from zero)
start	The start address of the sector
size	The size of the sector

libmem_busy_handler_fn

Synopsis

```
libmem_busy_handler_fn_t libmem_busy_handler_fn;
```

Description

libmem_busy_handler_fn is a pointer to a function that should be called each time LIBMEM iterates a busy loop.

libmem_busy_handler_fn_t

Synopsis

```
typedef void (*libmem_busy_handler_fn_t)(void);
```

Description

libmem_busy_handler_fn_t is a pointer to a function to be called each time LIBMEM iterates a busy loop.

libmem_cfi_get_info

Synopsis

```
int libmem_cfi_get_info(uint8_t *start,
                        size_t *size,
                        libmem_geometry_t *geometry,
                        int max_geometry_regions,
                        libmem_flash_info_t *flash_info);
```

Description

libmem_cfi_get_info returns a FLASH memory device's common flash interface (CFI) information.

start — The start address of the FLASH memory.

size — A pointer to the memory location to store the size (in bytes) of the FLASH memory.

geometry — A pointer to the memory location to store the geometry description or NULL if not required.

max_geometry_regions — The maximum number of geometry regions that can be stored at the memory pointed to by **geometry**. The geometry description is NULL terminated so **max_geometry_regions** must be at least two regions in size in order to store one geometry region and one terminator entry.

flash_info — A pointer to the memory location to store the remaining FLASH information, or NULL if not required.

libmem_cfi_get_info returns — The LIBMEM status result.

This function attempts to return the FLASH device's type, size, geometry and other FLASH information from only a pointer to the first address the FLASH memory is located at. It uses the common flash memory interface (CFI) to obtain this information and therefore only works on FLASH devices that fully support this interface.

Example:

```
uint8_t *flash1_start = (uint8_t *)0x10000000;
     libmem_flash_info_t flash1_info;
     const int flash1_max_geometry_regions = 4;
     libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
     size_t flash1_size;
     int res;
     res = libmem_cfi_get_info(flash1_start,
                               &flash1_size,
                               flash1_geometry,
                               flash1_max_geometry_regions,
                               &flash1_info);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_cfi_get_info : success\n");
     else
```

```
    printf("libmem_cfi_get_info : failed (%d)\n", res);
```

libmem_crc32

Synopsis

```
uint32_t libmem_crc32(const uint8_t *start,
                      size_t size,
                      uint32_t crc);
```

Description

libmem_crc32 computes the CRC-32 checksum of an address range using a LIBMEM driver.

start — A pointer to the start of the address range.

size — The size of the address range in bytes.

crc — The initial CRC-32 value.

libmem_crc32 returns — The computed CRC-32 value.

This function locates the LIBMEM driver for the address pointed to by **start**, then calls the LIBMEM driver's **crc32** extended function if it has one and returns the result. If the driver hasn't implemented the **crc32** extended function then the **libmem_crc32_direct** function is called which accesses the memory directly. The intention for this function is to allow you to use the LIBMEM library for memory that doesn't appear on the address bus by providing a virtual address range for the device.

Example:

```
uint32_t crc = 0xFFFFFFFF;
     crc = libmem_crc32((uint8_t *)0x10000000, 1024, crc);
```

libmem_crc32_direct

Synopsis

```
uint32_t libmem_crc32_direct(const uint8_t *start,
                             size_t size,
                             uint32_t crc);
```

Description

libmem_crc32_direct computes the CRC-32 checksum of an address range.

start — A pointer to the start of the address range.

size — The size of the address range in bytes.

crc — The initial CRC-32 value.

libmem_crc32_direct returns — The computed CRC-32 value.

This function computes a CRC-32 checksum on a block of data using the standard CRC-32 polynomial (0x04C11DB7). Note that this implementation doesn't reflect the input or the output and the result is not inverted.

Example:

```
uint32_t crc = 0xFFFFFFFF;
     crc = libmem_crc32_direct((uint8_t *)0x10000000, 1024, crc);
```

libmem_driver_crc32_fn_t

Synopsis

```
typedef uint32_t (*libmem_driver_crc32_fn_t)(libmem_driver_handle_t *, const uint8_t *,
 size_t , uint32_t);
```

Description

libmem_driver_crc32_fn_t is a function pointer to a LIBMEM driver's crc32 extended function.

h — A pointer to the handle of the LIBMEM driver.

start — A pointer to the start of the address range.

size — The size of the address range in bytes.

crc — The initial CRC-32 value.

libmem_driver_crc32_fn_t returns — The computed CRC-32 value.

The driver's **crc** function is an optional extended function. It has been provided to allow you to write a driver for memory that is not memory mapped.

Typically memory read operations will be direct memory mapped operations however implementing a driver's **crc** function allows you to carry out a crc32 operation on non-memory mapped memory through the LIBMEM interface.

libmem_driver_erase_fn_t

Synopsis

```
typedef int (*libmem_driver_erase_fn_t)(libmem_driver_handle_t *, uint8_t *, size_t , uint8_t
 ** , size_t *);
```

Description

libmem_driver_erase_fn_t is a function pointer to a LIBMEM driver's erase function.

h — A pointer to the handle of the LIBMEM driver.

start — A pointer to the initial memory address in memory range handled by driver to erase.

size — The number of bytes to erase.

erase_start — A pointer to a location in memory to store a pointer to the start of the memory range that has actually been erased or NULL if not required.

erase_size — A pointer to a location in memory to store the size in bytes of the memory range that has actually been erased or NULL if not required.

libmem_driver_erase_fn_t returns — The LIBMEM status result.

The driver's **erase** function should erase **size** bytes of the memory range handled by the LIBMEM driver pointed to by **start**.

There is no specific module or chip erase driver entry point, it is up to the driver to decide how best to erase the memory based on the supplied address range. If the application needs to know what memory was actually erased it can use the **erase_start** and **erase_size** parameters.

If this operation is not required the function should return **LIBMEM_STATUS_SUCCESS** and if the **erase_start** or **erase_size** parameters are supplied they should be assigned with the values of **start** and **size**.

libmem_driver_fill_fn_t

Synopsis

```
typedef int (*libmem_driver_fill_fn_t)(libmem_driver_handle_t *, uint8_t *, uint8_t ,
 size_t);
```

Description

libmem_driver_fill_fn_t is a function pointer to a LIBMEM driver's fill function.

h — A pointer to the handle of the LIBMEM driver.

dest — A pointer to the memory address in memory range handled by driver to write data to.

c — The data byte to write.

size — The number of bytes to write.

libmem_driver_fill_fn_t returns — The LIBMEM status result.

The driver's **fill** function writes **size** bytes of value `to` the memory address handled by the LIBMEM driver pointed to by **dest**.

If this operation is not required the function should return **LIBMEM_STATUS_SUCCESS**.

libmem_driver_flush_fn_t

Synopsis

```
typedef int (*libmem_driver_flush_fn_t)(libmem_driver_handle_t *);
```

Description

libmem_driver_flush_fn_t is a function pointer to a LIBMEM driver's flush function.

h — A pointer to the handle of the LIBMEM driver.

libmem_driver_flush_fn_t returns — The LIBMEM status result.

The driver's **flush** function should complete any outstanding memory operations (if any) and return the memory to read mode.

If this operation is not required the function should return **LIBMEM_STATUS_SUCCESS**.

libmem_driver_inrange_fn_t

Synopsis

```
typedef int (*libmem_driver_inrange_fn_t)(libmem_driver_handle_t *, const uint8_t *);
```

Description

libmem_driver_inrange_fn_t is a function pointer to a LIBMEM driver's inrange extended function.

h — A pointer to the handle of the LIBMEM driver.

dest — A pointer to then memory location being tested.

libmem_driver_inrange_fn_t returns — The LIBMEM status result.

The driver's **inrange** function is an optional extended function. It has been provided to allow the driver to indicate if it handles a more complex memory range than the single range described by the **start** and **size** **libmem_driver_handle_t** fields, for example if the memory has been aliased over a number of memory ranges.

The function should return non-zero if the address pointed to by **dest** is handled by the driver.

libmem_driver_lock_fn_t

Synopsis

```
typedef int (*libmem_driver_lock_fn_t)(libmem_driver_handle_t *, uint8_t *, size_t);
```

Description

libmem_driver_lock_fn_t is a function pointer to a LIBMEM driver's lock function.

h — A pointer to the handle of the LIBMEM driver.

start — A pointer to the initial memory address in memory range handled by driver to lock.

size — The number of bytes to lock.

libmem_driver_lock_fn_t returns — The LIBMEM status result.

The driver's **lock** function should lock **size** bytes of the memory range handled by the LIBMEM driver pointed to by **start**.

If this operation is not required the function should return **LIBMEM_STATUS_SUCCESS**.

libmem_driver_read_fn_t

Synopsis

```
typedef int (*libmem_driver_read_fn_t)(libmem_driver_handle_t *, uint8_t *, const uint8_t *,
 size_t);
```

Description

libmem_driver_read_fn_t is a function pointer to a LIBMEM driver's read extended function.

h — A pointer to the handle of the LIBMEM driver.

dest — A pointer to the initial memory address to write data to.

src — A pointer to the initial memory address in the memory range handled by the driver to read data from.

size — The number of bytes to write.

libmem_driver_read_fn_t returns — The LIBMEM status result.

The driver's **read** function is an optional extended function. It has been provided to allow you to write a driver for memory that is not memory mapped.

Typically memory read operations will be direct memory mapped operations however implementing a driver's **read** function allows you to access non-memory mapped memory through the LIBMEM interface.

libmem_driver_unlock_fn_t

Synopsis

```
typedef int (*libmem_driver_unlock_fn_t)(libmem_driver_handle_t *, uint8_t *, size_t);
```

Description

libmem_driver_unlock_fn_t is a function pointer to a LIBMEM driver's unlock function.

h — A pointer to the handle of the LIBMEM driver.

start — A pointer to the initial memory address in memory range handled by driver to unlock.

size — The number of bytes to unlock.

libmem_driver_unlock_fn_t returns — The LIBMEM status result.

The driver's **unlock** function should unlock **size** bytes of the memory range handled by the LIBMEM driver pointed to by **start**.

If this operation is not required the function should return **LIBMEM_STATUS_SUCCESS**.

libmem_driver_write_fn_t

Synopsis

```
typedef int (*libmem_driver_write_fn_t)(libmem_driver_handle_t *, uint8_t *, const uint8_t *,
 size_t);
```

Description

libmem_driver_write_fn_t is a function pointer to a LIBMEM driver's write function.

h — A pointer to the handle of the LIBMEM driver.

dest — A pointer to the memory address in memory range handled by driver to write data to.

src — A pointer to the memory address to read data from.

size — The number of bytes to write.

libmem_driver_write_fn_t returns — The LIBMEM status result.

The driver's **write** function copies data from the memory address pointed to by **src** to the memory address handled by the LIBMEM driver pointed to by **dest**.

If this operation is not required the function should return **LIBMEM_STATUS_SUCCESS**.

libmem_drivers

Synopsis

```
libmem_driver_handle_t *libmem_drivers;
```

Description

libmem_drivers is a pointer to the first registered LIBMEM driver.

libmem_enable_timeouts

Synopsis

```
void libmem_enable_timeouts(libmem_get_ticks_fn_t get_ticks_fn,
                            uint32_t ticks_per_second);
```

Description

libmem_enable_timeouts enables LIBMEM operation timeouts.

get_ticks_fn — A pointer to a function that returns an incrementing tick count.

ticks_per_second — The amount the value returned by the **get_ticks_fn** increments per second.

In order for operations to timeout the LIBMEM library needs a function that can supply a timer tick count and also needs to know the frequency the timer increments.

This function should be called prior to registering LIBMEM drivers as the **ticks_per_second** parameter can be used to pre-compute timeout periods when the driver is registered.

libmem_erase

Synopsis

```
int libmem_erase(uint8_t *start,
                 size_t size,
                 uint8_t **erase_start,
                 size_t *erase_size);
```

Description

libmem_erase erases a block of memory using a LIBMEM driver.

start — A pointer to the start address of the memory range to erase.

size — The size of the memory range to erase in bytes.

erase_start — A pointer to a location in memory to store a pointer to the start of the memory range that has actually been erased or NULL if not required.

erase_size — A pointer to a location in memory to store the size in bytes of the memory range that has actually been erased or NULL if not required.

libmem_erase returns — The LIBMEM status result.

This function locates the LIBMEM driver for the address pointed to by **start** and then calls the LIBMEM driver's **erase** function.

Note that the address range being erased cannot span multiple LIBMEM drivers.

Example:

```
uint8_t *erase_start;
     size_t erase_size;
     int res;
     res = libmem_erase((uint8_t *)0x10000000, 1024, &erase_start, &erase_size);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_erase : success (erased %08X - 0x
%08X)\n", erase_start, erase_start + erase_size - 1);
     else
       printf("libmem_erase : failed (%d)\n", res);
```

libmem_erase_all

Synopsis

```
int libmem_erase_all(void);
```

Description

libmem_erase_all erases all memory using LIBMEM drivers.

libmem_erase_all returns — The LIBMEM status result.

This function iterates through all registered LIBMEM drivers calling each driver's **erase** function specifying the drivers entire memory range as its parameters.

The function will terminate if any of the driver's **erase** functions return a result other than **LIBMEM_STATUS_SUCCESS**.

Example:

```
int res;
     res = libmem_erase_all();
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_erase_all : success\n");
     else
       printf("libmem_erase_all : failed (%d)\n", res);
```

libmem_fill

Synopsis

```
int libmem_fill(uint8_t *dest,
                uint8_t c,
                size_t size);
```

Description

libmem_fill fills memory with a specific data value using a LIBMEM driver.

dest — A pointer to the address to write the data.

c — The data value to fill the memory with.

size — The number of bytes to write.

libmem_fill returns — The LIBMEM status result.

This function locates the LIBMEM driver for the address pointed to by **dest** and then calls the LIBMEM driver's **fill** function.

Note that the address range being written to cannot span multiple LIBMEM drivers.

Example:

```
int res;
     res = libmem_fill((uint8_t *)0x10000000, 0xCC, 64);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_fill : success\n");
     else
       printf("libmem_fill : failed (%d)\n", res);
```

libmem_flush

Synopsis

```
int libmem_flush(void);
```

Description

libmem_flush flushes any outstanding memory operations and returns memory to read mode if applicable.

libmem_flush returns — The LIBMEM status result.

LIBMEM drivers do not necessarily carry out operations immediately or they may leave the memory in an unreadable state for performance reasons. You should call **libmem_flush** once you have finished carrying out memory operations in order to complete all outstanding transactions and return the memory to a readable state.

Example:

```
int res;
     res = libmem_flush();
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_flush : success\n");
     else
       printf("libmem_flush : failed (%d)\n", res);
```

libmem_foreach_driver

Synopsis

```
int libmem_foreach_driver(libmem_foreach_driver_fn_t fn);
```

Description

libmem_foreach_driver iterates through all the registered LIBMEM drivers and calls **fn** for each. If any of the calls return a response other than `LIBMEM_STATUS_SUCCESS` this function will terminate and return that response.

fn — The function to call for each driver.

libmem_foreach_driver returns — The LIBMEM status result.

libmem_foreach_driver_fn_t

Synopsis

```
typedef int (*libmem_foreach_driver_fn_t)(libmem_driver_handle_t *);
```

Description

libmem_foreach_driver_fn_t is a function pointer to a function handling a libmem_foreach_driver call.

h — A pointer to the handle of the LIBMEM driver.

libmem_foreach_driver_fn_t returns — The LIBMEM status result. If any value other than LIBMEM_STATUS_SUCCESS is returned from this function the **libmem_foreach_driver** function will terminate and return the response.

libmem_foreach_sector

Synopsis

```
int libmem_foreach_sector(libmem_driver_handle_t *h,
                          libmem_foreach_sector_fn_t fn);
```

Description

libmem_foreach_sector is a helper function for iterating through all sectors handled by a LIBMEM driver.

h — A pointer to the handle of the LIBMEM driver.

fn — The function to call for each sector.

libmem_foreach_sector returns — The LIBMEM status result.

This function iterates through all the sectors handled by a single LIBMEM driver and calls a **libmem_foreach_sector_fn_t** function for each. If any of the calls return a response other than LIBMEM_STATUS_SUCCESS this function will terminate and return the response.

libmem_foreach_sector_fn_t

Synopsis

```
typedef int (*libmem_foreach_sector_fn_t)(libmem_driver_handle_t *, libmem_sector_info_t *);
```

Description

libmem_foreach_sector_fn_t is a function pointer to a function handling a libmem_foreach_sector or libmem_foreach_sector_in_range call.

h — A pointer to the handle of the LIBMEM driver.

sector_info — A pointer to the sector information.

libmem_foreach_sector_fn_t returns — The LIBMEM status result. If any value other than LIBMEM_STATUS_SUCCESS is returned from this function the **libmem_foreach_sector** or **libmem_foreach_sector_in_range** functions will terminate and return the response.

libmem_foreach_sector_in_range

Synopsis

```
int libmem_foreach_sector_in_range(libmem_driver_handle_t *h,
                                   uint8_t *range_start,
                                   size_t range_size,
                                   libmem_foreach_sector_fn_t fn,
                                   uint8_t **actual_range_start,
                                   size_t *actual_range_size);
```

Description

libmem_foreach_sector_in_range is a helper function for iterating through all sectors handled by a driver that are within a specific address range.

h — A pointer to the handle of the LIBMEM driver.

range_start — A pointer to the start of the address range.

range_size — The size of the address range in bytes.

fn — The function to call for each sector.

actual_range_start — A pointer to the start of the first sector that is within the address range.

actual_range_size — The combined size of all the sectors that are within the address range.

libmem_foreach_sector_in_range returns — The LIBMEM status result.

This function iterates through all the sectors handled by a single LIBMEM driver and calls a **libmem_foreach_sector_fn_t** function for each if it is within the specified address range. If any of the calls return a response other than LIBMEM_STATUS_SUCCESS this function will terminate and return the response.

libmem_foreach_sector_in_range_ex

Synopsis

```
int libmem_foreach_sector_in_range_ex(libmem_driver_handle_t *h,
                                      const libmem_geometry_t *geometry,
                                      uint8_t *range_start,
                                      size_t range_size,
                                      libmem_foreach_sector_fn_t fn,
                                      uint8_t **actual_range_start,
                                      size_t *actual_range_size);
```

Description

libmem_foreach_sector_in_range_ex is a helper function for iterating through all sectors in a specified geometry that are within a specific address range.

h — A pointer to the handle of the LIBMEM driver.

geometry — A pointer to the NULL terminated geometry description.

range_start — A pointer to the start of the address range.

range_size — The size of the address range in bytes.

fn — The function to call for each sector.

actual_range_start — A pointer to the start of the first sector that is within the address range.

actual_range_size — The combined size of all the sectors that are within the address range.

libmem_foreach_sector_in_range_ex returns — The LIBMEM status result.

This function iterates through all the sectors in the specified geometry and calls a **libmem_foreach_sector_fn_t** function for each if it is within the specified address range. If any of the calls return a response other than LIBMEM_STATUS_SUCCESS this function will terminate and return the response. This function is essentially the same as **libmem_foreach_sector_in_range** except it allows a different geometry to be specified to that associated with the driver.

libmem_get_driver

Synopsis

```
libmem_driver_handle_t *libmem_get_driver(const uint8_t *p);
```

Description

libmem_get_driver is a helper function that returns the handle of a LIBMEM driver that is responsible for a specific memory location.

p — A pointer to the memory location to get the driver for.

libmem_get_driver returns — The LIBMEM driver handle or NULL if no driver could be found.

libmem_get_driver_sector_size

Synopsis

```
size_t libmem_get_driver_sector_size(const uint8_t *p);
```

Description

libmem_get_driver_sector_size is a helper function that locates the driver for a specific address and then returns the sector size for that address using the driver's geometry.

p — A pointer to the address to determine the sector information of.

libmem_get_driver_sector_size returns — The size of the sector or 0 if the sector cannot be found.

libmem_get_geometry_size

Synopsis

```
size_t libmem_get_geometry_size(const libmem_geometry_t *geometry);
```

Description

libmem_get_geometry_size is a helper function that returns the size of the address range described by a geometry description.

geometry — A pointer to the NULL terminated geometry description.

libmem_get_geometry_size returns — The size of the address range described the by geometry description in bytes.

libmem_get_number_of_regions

Synopsis

```
int libmem_get_number_of_regions(const libmem_geometry_t *geometry);
```

Description

libmem_get_number_of_regions is a helper function that returns the number of geometry regions described by a geometry description.

geometry — A pointer to the NULL terminated geometry description.

libmem_get_number_of_regions returns — The number of geometry regions.

libmem_get_number_of_sectors

Synopsis

```
int libmem_get_number_of_sectors(const libmem_geometry_t *geometry);
```

Description

libmem_get_number_of_sectors is a helper function that returns the number of sectors described by a geometry description.

geometry — A pointer to the NULL terminated geometry description.

libmem_get_number_of_sectors returns — The number of sectors.

libmem_get_sector_info

Synopsis

```
int libmem_get_sector_info(uint8_t *start,
                           const libmem_geometry_t *geometry,
                           const uint8_t *p,
                           libmem_sector_info_t *info);
```

Description

libmem_get_sector_info is a helper function that returns the sector information for an address within a specified geometry.

start — A pointer to the start address of the geometry desribed by **geometry**.

geometry — A pointer to the NULL terminated geometry description.

p — A pointer to the address to determine the sector information of.

info — A pointer to the **libmem_sector_info_t** structure to write the sector information to.

libmem_get_sector_info returns — The LIBMEM status result.

libmem_get_sector_number

Synopsis

```
int libmem_get_sector_number(uint8_t *start,
                             const libmem_geometry_t *geometry,
                             const uint8_t *p);
```

Description

libmem_get_sector_number is a helper function that returns the sector number of an address within a specified geometry.

start — A pointer to the start address of the geometry desribed by **geometry**.

geometry — A pointer to the NULL terminated geometry description.

p — A pointer to the address to determine the sector number of.

libmem_get_sector_number returns — The sector number or -1 if the address is not located within the described geometry.

libmem_get_sector_size

Synopsis

```
size_t libmem_get_sector_size(uint8_t *start,
                              const libmem_geometry_t *geometry,
                              const uint8_t *p);
```

Description

libmem_get_sector_size is a helper function that returns the sector size for an address within a specified geometry.

start — A pointer to the start address of the geometry desribed by **geometry**.

geometry — A pointer to the NULL terminated geometry description.

p — A pointer to the address to determine the sector information of.

libmem_get_sector_size returns — The size of the sector or 0 if the sector cannot be found.

libmem_get_ticks

Synopsis

```
uint32_t libmem_get_ticks(void);
```

Description

libmem_get_ticks is a helper function that returns the current timer tick count.

libmem_get_ticks returns — The current timer tick count as returned by the **libmem_get_ticks_fn** function or 0 if this function has not been defined.

libmem_get_ticks_fn

Synopsis

```
libmem_get_ticks_fn_t libmem_get_ticks_fn;
```

Description

libmem_get_ticks_fn is a pointer to a function that returns the current timer tick count.

libmem_get_ticks_fn_t

Synopsis

```
typedef uint32_t (*libmem_get_ticks_fn_t)(void);
```

Description

libmem_get_ticks_fn_t is a pointer to a function returning the current timer tick count.

libmem_get_ticks_fn_t returns — The current timer tick count.

libmem_lock

Synopsis

```
int libmem_lock(uint8_t *start,
                size_t size);
```

Description

libmem_lock locks a block of memory using a LIBMEM driver.

start — A pointer to the start address of the memory range to lock.

size — The size of the memory range to lock in bytes.

libmem_lock returns — The LIBMEM status result.

This function locates the LIBMEM driver for the address pointed to by **start** and then calls the LIBMEM driver's **lock** function.

Example:

```
int res;
     res = libmem_lock((uint8_t *)0x10000000, 1024);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_lock : success\n");
     else
       printf("libmem_lock : failed (%d)\n", res);
```

libmem_lock_all

Synopsis

```
int libmem_lock_all(void);
```

Description

libmem_lock_all locks all memory using LIBMEM drivers.

libmem_lock_all returns — The LIBMEM status result.

This function iterates through all registered LIBMEM drivers calling each driver's **lock** function specifying the drivers entire memory range as its parameters.

The function will terminate if any of the driver's **lock** functions return a result other than **LIBMEM_STATUS_SUCCESS**.

Example:

```
int res;
     res = libmem_lock_all();
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_lock_all : success\n");
     else
       printf("libmem_lock_all : failed (%d)\n", res);
```

libmem_read

Synopsis

```
int libmem_read(uint8_t *dest,
                const uint8_t *src,
                size_t size);
```

Description

libmem_read reads a block of data using a LIBMEM driver.

dest — A pointer to the address to write the block of data.

src — A pointer to the address to copy the block of data from.

size — The size of the block of data to copy in bytes.

libmem_read returns — The LIBMEM status result.

This function locates the LIBMEM driver for the address pointed to by **src** and then calls the LIBMEM driver's **read** extended function if it has been implemented. If the **read** function has not been implemented then the memory will be read directly using memcpy. The intention for this function is to allow you to use the LIBMEM library for memory that doesn't appear on the address bus by providing a virtual address range for the device.

Note that if the LIBMEM driver's read function is used, the address range being read cannot span multiple LIBMEM drivers.

Example:

```
uint8_t buffer[64];
     int res;
     res = libmem_read(buffer, (uint8_t *)0x10000000, sizeof(buffer));
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_read : success\n");
     else
       printf("libmem_read : failed (%d)\n", res);
```

libmem_register_am29f200b_driver

Synopsis

```
int libmem_register_am29f200b_driver(libmem_driver_handle_t *h,
                                     uint8_t *start);
```

Description

libmem_register_am29f200b_driver registers a driver for an AMD Am29F200B FLASH chip.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

libmem_register_am29f200b_driver returns — The LIBMEM status result.

libmem_register_am29f200t_driver

Synopsis

```
int libmem_register_am29f200t_driver(libmem_driver_handle_t *h,
                                     uint8_t *start);
```

Description

libmem_register_am29f200t_driver registers a driver for an AMD Am29F200T FLASH chip.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

libmem_register_am29f200t_driver returns — The LIBMEM status result.

libmem_register_am29f400bb_driver

Synopsis

```
int libmem_register_am29f400bb_driver(libmem_driver_handle_t *h,
                                      uint8_t *start);
```

Description

libmem_register_am29f400bb_driver registers a driver for an AMD Am29F400BB FLASH chip.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

libmem_register_am29f400bb_driver returns — The LIBMEM status result.

libmem_register_am29f400bt_driver

Synopsis

```
int libmem_register_am29f400bt_driver(libmem_driver_handle_t *h,
                                      uint8_t *start);
```

Description

libmem_register_am29f400bt_driver registers a driver for an AMD Am29F400BT FLASH chip.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

libmem_register_am29f400bt_driver returns — The LIBMEM status result.

libmem_register_am29fxxx_driver

Synopsis

```
int libmem_register_am29fxxx_driver(libmem_driver_handle_t *h,
                                    uint8_t *start,
                                    unsigned size,
                                    const libmem_geometry_t *geometry,
                                    unsigned device_id);
```

Description

libmem_register_am29fxxx_driver registers a driver for an AMD Am29Fxxx FLASH chip.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

size — The size of the address range handled by the LIBMEM driver in bytes.

geometry — A pointer to a null-terminated geometry description list for the device.

device_id — The device ID of the device. The expected device ID is checked against the device ID read from the FLASH. If the device IDs differ this function return LIBMEM_STATUS_INVALID_DEVICE.

libmem_register_am29fxxx_driver returns — The LIBMEM status result.

libmem_register_am29lv010b_driver

Synopsis

```
int libmem_register_am29lv010b_driver(libmem_driver_handle_t *h,
                                      uint8_t *start);
```

Description

libmem_register_am29lv010b_driver registers a driver for an AMD Am29LV010B FLASH chip.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

libmem_register_am29lv010b_driver returns — The LIBMEM status result.

libmem_register_cfi_0001_16_driver

Synopsis

```
int libmem_register_cfi_0001_16_driver(libmem_driver_handle_t *h,
                                       uint8_t *start,
                                       size_t size,
                                       const libmem_geometry_t *geometry,
                                       const libmem_flash_info_t *flash_info);
```

Description

libmem_register_cfi_0001_16_driver registers a 16-bit CFI command set 1 (Intel Extended) LIBMEM driver.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

size — The size of the FLASH memory.

geometry — A NULL terminated description of the FLASH's geometry.

flash_info — A pointer to the FLASH information structure or NULL if not required.

libmem_register_cfi_0001_16_driver returns — The LIBMEM status result.

Example:

```
libmem_driver_handle_t flash1_handle;
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     libmem_geometry_t flash1_geometry[] =
       {
         { 8,  0x00002000 }, // 8 x 8KB sectors
         { 31, 0x00010000 }, // 31 x 64KB sectors
         { 0, 0 },           // NULL terminator
       };
     int res;
     res = libmem_register_cfi_0001_16_driver(&flash1_handle,
                                              flash1_start,
                                              libmem_get_geometry_size(flash1_geometry),
                                              flash1_geometry,
                                              0);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_register_cfi_0001_16_driver : success\n");
     else
       printf("libmem_register_cfi_0001_16_driver : failed (%d)\n", res);
```

libmem_register_cfi_0001_8_driver

Synopsis

```
int libmem_register_cfi_0001_8_driver(libmem_driver_handle_t *h,
                                      uint8_t *start,
                                      size_t size,
                                      const libmem_geometry_t *geometry,
                                      const libmem_flash_info_t *flash_info);
```

Description

libmem_register_cfi_0001_8_driver registers an 8-bit CFI command set 1 (Intel Extended) LIBMEM driver.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

size — The size of the FLASH memory.

geometry — A NULL terminated description of the FLASH's geometry.

flash_info — A pointer to the FLASH information structure or NULL if not required.

libmem_register_cfi_0001_8_driver returns — The LIBMEM status result.

Example:

```
libmem_driver_handle_t flash1_handle;
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     libmem_geometry_t flash1_geometry[] =
       {
         { 8,  0x00002000 }, // 8 x 8KB sectors
         { 31, 0x00010000 }, // 31 x 64KB sectors
         { 0, 0 },           // NULL terminator
       };
     int res;
     res = libmem_register_cfi_0001_8_driver(&flash1_handle,
                                             flash1_start,
                                             libmem_get_geometry_size(flash1_geometry),
                                             flash1_geometry,
                                             0);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_register_cfi_0001_8_driver : success\n");
     else
       printf("libmem_register_cfi_0001_8_driver : failed (%d)\n", res);
```

libmem_register_cfi_0002_16_driver

Synopsis

```
int libmem_register_cfi_0002_16_driver(libmem_driver_handle_t *h,
                                       uint8_t *start,
                                       size_t size,
                                       const libmem_geometry_t *geometry,
                                       const libmem_flash_info_t *flash_info);
```

Description

libmem_register_cfi_0002_16_driver registers a 16-bit CFI command set 2 (AMD Standard) LIBMEM driver.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

size — The size of the FLASH memory.

geometry — A NULL terminated description of the FLASH's geometry.

flash_info — A pointer to the FLASH information structure or NULL if not required.

libmem_register_cfi_0002_16_driver returns — The LIBMEM status result.

Example:

```
libmem_driver_handle_t flash1_handle;
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     libmem_geometry_t flash1_geometry[] =
       {
         { 8,  0x00002000 }, // 8 x 8KB sectors
         { 31, 0x00010000 }, // 31 x 64KB sectors
         { 0, 0 },           // NULL terminator
       };
     int res;
     res = libmem_register_cfi_0002_16_driver(&flash1_handle,
                                              flash1_start,
                                              libmem_get_geometry_size(flash1_geometry),
                                              flash1_geometry,
                                              0);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_register_cfi_0002_16_driver : success\n");
     else
       printf("libmem_register_cfi_0002_16_driver : failed (%d)\n", res);
```

libmem_register_cfi_0002_8_driver

Synopsis

```
int libmem_register_cfi_0002_8_driver(libmem_driver_handle_t *h,
                                      uint8_t *start,
                                      size_t size,
                                      const libmem_geometry_t *geometry,
                                      const libmem_flash_info_t *flash_info);
```

Description

libmem_register_cfi_0002_8_driver registers an 8 bit CFI command set 2 (AMD Standard) LIBMEM driver.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

size — The size of the FLASH memory.

geometry — A NULL terminated description of the FLASH's geometry.

flash_info — A pointer to the FLASH information structure or NULL if not required.

libmem_register_cfi_0002_8_driver returns — The LIBMEM status result.

Example:

```
libmem_driver_handle_t flash1_handle;
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     libmem_geometry_t flash1_geometry[] =
       {
         { 8,  0x00002000 }, // 8 x 8KB sectors
         { 31, 0x00010000 }, // 31 x 64KB sectors
         { 0, 0 },           // NULL terminator
       };
     int res;
     res = libmem_register_cfi_0002_8_driver(&flash1_handle,
                                             flash1_start,
                                             libmem_get_geometry_size(flash1_geometry),
                                             flash1_geometry,
                                             0);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_register_cfi_0002_8_driver : success\n");
     else
       printf("libmem_register_cfi_0002_8_driver : failed (%d)\n", res);
```

libmem_register_cfi_0003_16_driver

Synopsis

```
int libmem_register_cfi_0003_16_driver(libmem_driver_handle_t *h,
                                       uint8_t *start,
                                       size_t size,
                                       const libmem_geometry_t *geometry,
                                       const libmem_flash_info_t *flash_info);
```

Description

libmem_register_cfi_0003_16_driver registers a 16-bit CFI command set 3 (Intel Standard) LIBMEM driver.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

size — The size of the FLASH memory.

geometry — A NULL terminated description of the FLASH's geometry.

flash_info — A pointer to the FLASH information structure or NULL if not required.

libmem_register_cfi_0003_16_driver returns — The LIBMEM status result.

Example:

```
libmem_driver_handle_t flash1_handle;
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     libmem_geometry_t flash1_geometry[] =
       {
         { 8,  0x00002000 }, // 8 x 8KB sectors
         { 31, 0x00010000 }, // 31 x 64KB sectors
         { 0, 0 },           // NULL terminator
       };
     int res;
     res = libmem_register_cfi_0003_16_driver(&flash1_handle,
                                              flash1_start,
                                              libmem_get_geometry_size(flash1_geometry),
                                              flash1_geometry,
                                              0);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_register_cfi_0003_16_driver : success\n");
     else
       printf("libmem_register_cfi_0003_16_driver : failed (%d)\n", res);
```

libmem_register_cfi_0003_8_driver

Synopsis

```
int libmem_register_cfi_0003_8_driver(libmem_driver_handle_t *h,
                                      uint8_t *start,
                                      size_t size,
                                      const libmem_geometry_t *geometry,
                                      const libmem_flash_info_t *flash_info);
```

Description

libmem_register_cfi_0003_8_driver registers an 8-bit CFI command set 3 (Intel Standard) LIBMEM driver.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

size — The size of the FLASH memory.

geometry — A NULL terminated description of the FLASH's geometry.

flash_info — A pointer to the FLASH information structure or NULL if not required.

libmem_register_cfi_0003_8_driver returns — The LIBMEM status result.

Example:

```
libmem_driver_handle_t flash1_handle;
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     libmem_geometry_t flash1_geometry[] =
       {
         { 8,  0x00002000 }, // 8 x 8KB sectors
         { 31, 0x00010000 }, // 31 x 64KB sectors
         { 0, 0 },           // NULL terminator
       };
     int res;
     res = libmem_register_cfi_0003_8_driver(&flash1_handle,
                                             flash1_start,
                                             libmem_get_geometry_size(flash1_geometry),
                                             flash1_geometry,
                                             0);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_register_cfi_0003_8_driver : success\n");
     else
       printf("libmem_register_cfi_0003_8_driver : failed (%d)\n", res);
```

libmem_register_cfi_amd_driver

Synopsis

```
int libmem_register_cfi_amd_driver(libmem_driver_handle_t *h,
                                   uint8_t *start,
                                   size_t size,
                                   const libmem_geometry_t *geometry,
                                   const libmem_flash_info_t *flash_info);
```

Description

libmem_register_cfi_amd_driver registers a multi-width CFI command set 2 (AMD) LIBMEM driver.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

size — The size of the FLASH memory.

geometry — A NULL terminated description of the FLASH's geometry.

flash_info — A pointer to the FLASH information structure.

libmem_register_cfi_amd_driver returns — The LIBMEM status result.

This function registers a multi-width CFI command set 2 (AMD) LIBMEM driver. The advantage of this driver over the individual single width and command set drivers is that one driver will support a range of FLASH chips, the disadvantage is that of increased code size and reduced performance.

Example:

```
const int flash1_max_geometry_regions = 4;
     libmem_driver_handle_t flash1_handle;
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
     libmem_flash_info_t flash1_info;
     size_t flash1_size;
     int res;
     // Detect the type, size and geometry of the Intel FLASH.
     res = libmem_cfi_get_info(flash1_start,
                               &flash1_size,
                               flash1_geometry,
                               flash1_max_geometry_regions,
                               &flash1_info);
     if (res == LIBMEM_STATUS_SUCCESS)
       {
         // Register the driver
         res = libmem_register_cfi_amd_driver(&flash1_handle,
```

```
                                                flash1_start,
                                                flash1_size,
                                                flash1_geometry,
                                                &flash1_info);
      if (res == LIBMEM_STATUS_SUCCESS)
        printf("libmem_register_cfi_amd_driver : success\n");
      else
        printf("libmem_register_cfi_amd_driver : failed (%d)\n", res);
    }
  else
    printf("libmem_cfi_get_info : failed (%d)\n", res);
```

libmem_register_cfi_driver

Synopsis

```
int libmem_register_cfi_driver(libmem_driver_handle_t *h,
                               uint8_t *start,
                               libmem_geometry_t *geometry,
                               int max_geometry_regions,
                               libmem_flash_info_t *flash_info);
```

Description

libmem_register_cfi_driver registers a FLASH driver based on detected CFI information.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

geometry — A pointer to the memory location to store the geometry description.

max_geometry_regions — The maximum number of geometry regions that can be stored at the memory pointed to by **geometry**. The geometry description is NULL terminated so **max_geometry_regions** must be at least two regions in size in order to store one geometry region and one terminator entry.

flash_info — A pointer to the memory location to store the remaining FLASH information.

libmem_register_cfi_driver returns — The LIBMEM status result.

This function calls **libmem_cfi_get_info** to detect the type and geometry of the the FLASH pointed to by **start** and then, if the FLASH memory is supported, registers an appropriate LIBMEM driver for the FLASH.

Use of this function requires all supported CFI LIBMEM drivers to be linked in, therefore if memory is at a premium you should register only the LIBMEM FLASH driver you require instead of using this function.

Example:

```
uint8_t *flash1_start = (uint8_t *)0x10000000;
     libmem_flash_info_t flash1_info;
     const int flash1_max_geometry_regions = 4;
     libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
     libmem_driver_handle_t flash1_handle;
     int res;
     res = libmem_register_cfi_driver(&flash1_handle,
                                      flash1_start,
                                      flash1_geometry,
                                      flash1_max_geometry_regions,
                                      &flash1_info);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_register_cfi_driver : success\n");
```

```
else
  printf("libmem_register_cfi_driver : failed (%d)\n", res);
```

libmem_register_cfi_intel_driver

Synopsis

```
int libmem_register_cfi_intel_driver(libmem_driver_handle_t *h,
                                     uint8_t *start,
                                     size_t size,
                                     const libmem_geometry_t *geometry,
                                     const libmem_flash_info_t *flash_info);
```

Description

libmem_register_cfi_intel_driver registers a combined multi-width CFI command set 1 and 3 (Intel) LIBMEM driver.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

size — The size of the FLASH memory.

geometry — A NULL terminated description of the FLASH's geometry.

flash_info — A pointer to the FLASH information structure.

libmem_register_cfi_intel_driver returns — The LIBMEM status result.

This function registers a combined multi-width CFI command set 1 and 3 (Intel) LIBMEM driver. The advantage of this driver over the individual single width and command set drivers is that one driver will support a range of Intel FLASH chips, the disadvantage is that of increased code size and reduced performance.

Example:

```
const int flash1_max_geometry_regions = 4;
     libmem_driver_handle_t flash1_handle;
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
     libmem_flash_info_t flash1_info;
     size_t flash1_size;
     int res;
     // Detect the type, size and geometry of the Intel FLASH.
     res = libmem_cfi_get_info(flash1_start,
                               &flash1_size,
                               flash1_geometry,
                               flash1_max_geometry_regions,
                               &flash1_info);
     if (res == LIBMEM_STATUS_SUCCESS)
       {
         // Register the driver
```

```
    res = libmem_register_cfi_intel_driver(&flash1_handle,
                                           flash1_start,
                                           flash1_size,
                                           flash1_geometry,
                                           &flash1_info);
    if (res == LIBMEM_STATUS_SUCCESS)
      printf("libmem_register_cfi_intel_driver : success\n");
    else
      printf("libmem_register_cfi_intel_driver : failed (%d)\n", res);
  }
else
  printf("libmem_cfi_get_info : failed (%d)\n", res);
```

libmem_register_driver

Synopsis

```
void libmem_register_driver(libmem_driver_handle_t *h,
                            uint8_t *start,
                            size_t size,
                            const libmem_geometry_t *geometry,
                            const libmem_flash_info_t *flash_info,
                            const libmem_driver_functions_t *driver_functions,
                            const libmem_ext_driver_functions_t *ext_driver_functions);
```

Description

libmem_register_driver registers a LIBMEM driver instance.

h — A pointer to the handle of the LIBMEM driver being registered.

start — A Pointer to the start of the address range handled by the LIBMEM driver.

size — The size of the address range handled by the LIBMEM driver in bytes.

geometry — A pointer to a null-terminated geometry description list or NULL if not required.

flash_info — A pointer to the FLASH information structure or NULL if not required.

driver_functions — A pointer to the structure describing the LIBMEM driver's functions.

ext_driver_functions — A pointer to the structure describing the LIBMEM driver's extended functions, or NULL if not required.

This function adds a LIBMEM driver to the list of LIBMEM drivers currently installed. This function is not normally called directly by an application, it is typically called by a LIBMEM driver's own register function.

libmem_register_ram_driver

Synopsis

```
int libmem_register_ram_driver(libmem_driver_handle_t *h,
                               uint8_t *start,
                               size_t size);
```

Description

libmem_register_ram_driver registers a simple driver that directly accesses RAM.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the RAM.

size — The size of the RAM.

libmem_register_ram_driver returns — The LIBMEM status result.

Example:

```
libmem_driver_handle_t ram1_handle;
     uint8_t *ram1_start = (uint8_t *)0x10000000;
     const size_t ram1_size = 1024;
     int res;
     res = libmem_register_ram_driver(&ram_handle, ram1_start, ram1_size);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_register_ram_driver : success\n");
     else
       printf("libmem_register_ram_driver : failed (%d)\n", res);
```

libmem_register_sst39xFx00A_16_driver

Synopsis

```
int libmem_register_sst39xFx00A_16_driver(libmem_driver_handle_t *h,
                                          uint8_t *start,
                                          size_t size,
                                          const libmem_geometry_t *geometry,
                                          const libmem_flash_info_t *flash_info);
```

Description

libmem_register_sst39xFx00A_16_driver registers a driver for a 16-bit SST39xFx00A FLASH chip.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

size — The size of the FLASH memory.

geometry — A NULL terminated description of the FLASH's geometry.

flash_info — A pointer to the FLASH information structure or NULL if not required.

libmem_register_sst39xFx00A_16_driver returns — The LIBMEM status result.

Example:

```
libmem_driver_handle_t flash1_handle;
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     libmem_flash_info_t flash1_info;
     const int flash1_max_geometry_regions = 4;
     libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
     size_t size;
     int res;
     // Get CFI FLASH information and geometry
     res = libmem_cfi_get_info(flash1_start, &size, flash1_geometry, flash1_max_geometry_r
     if (res == LIBMEM_STATUS_SUCCESS)
       {
         res = libmem_register_sst39xFx00A_16_driver(&flash1_handle, flash1_start, size, f
         if (res == LIBMEM_STATUS_SUCCESS)
           printf("libmem_register_sst39xFx00A_16_driver : success\n");
         else
           printf("libmem_register_sst39xFx00A_16_driver : failed (%d)\n", res);
       }
     else
       printf("libmem_cfi_get_info : failed (%d)\n", res);
```

libmem_register_st_m28w320cb_driver

Synopsis

```
int libmem_register_st_m28w320cb_driver(libmem_driver_handle_t *h,
                                        uint8_t *start);
```

Description

libmem_register_st_m28w320cb_driver registers a driver for an ST M28W320CB FLASH chip.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

libmem_register_st_m28w320cb_driver returns — The LIBMEM status result.

Example:

```
libmem_driver_handle_t flash1_handle;
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     int res;
     res = libmem_register_st_m28w320cb_driver(&flash1_handle, flash1_start);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_register_st_m28w320cb_driver : success\n");
     else
       printf("libmem_register_st_m28w320cb_driver : failed (%d)\n", res);
```

libmem_register_st_m28w320ct_driver

Synopsis

```
int libmem_register_st_m28w320ct_driver(libmem_driver_handle_t *h,
                                        uint8_t *start);
```

Description

libmem_register_st_m28w320ct_driver registers a driver for an ST M28W320CT FLASH chip.

h — A pointer to the LIBMEM handle structure to use for this LIBMEM driver.

start — The start address of the FLASH memory.

libmem_register_st_m28w320ct_driver returns — The LIBMEM status result.

Example:

```
libmem_driver_handle_t flash1_handle;
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     int res;
     res = libmem_register_st_m28w320ct_driver(&flash1_handle, flash1_start);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_register_st_m28w320ct_driver : success\n");
     else
       printf("libmem_register_st_m28w320ct_driver : failed (%d)\n", res);
```

libmem_set_busy_handler

Synopsis

```
libmem_busy_handler_fn_t libmem_set_busy_handler(libmem_busy_handler_fn_t busy_handler_fn);
```

Description

libmem_set_busy_handler specifies a handler function that should be called each time LIBMEM iterates a busy loop.

busy_handler_fn — A pointer to a busy handler function.

libmem_set_busy_handler returns — A pointer to the existing busy handler or NULL if there isn't one.

This function allows a user defined function to be called each time LIBMEM iterates a busy loop. The typical use of this is to keep watchdogs alive while LIBMEM is carrying out blocking operations.

libmem_ticks_per_second

Synopsis

```
uint32_t libmem_ticks_per_second;
```

Description

libmem_ticks_per_second is the amount the value returned by the **libmem_get_ticks_fn** function increments each second.

libmem_unlock

Synopsis

```
int libmem_unlock(uint8_t *start,
                  size_t size);
```

Description

libmem_unlock unlocks a block of memory using a LIBMEM driver.

start — A pointer to the start address of the memory range to unlock.

size — The size of the memory range to unlock in bytes.

libmem_unlock returns — The LIBMEM status result.

This function locates the LIBMEM driver for the address pointed to by **start** and then calls the LIBMEM driver's **unlock** function.

Example:

```
int res;
     res = libmem_unlock((uint8_t *)0x10000000, 1024);
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_unlock : success\n");
     else
       printf("libmem_unlock : failed (%d)\n", res);
```

libmem_unlock_all

Synopsis

```
int libmem_unlock_all(void);
```

Description

libmem_unlock_all unlocks all memory using LIBMEM drivers.

libmem_unlock_all returns — The LIBMEM status result.

This function iterates through all registered LIBMEM drivers calling each driver's **unlock** function specifying the drivers entire memory range as its parameters.

The function will terminate if any of the driver's **unlock** functions return a result other than **LIBMEM_STATUS_SUCCESS**.

Example:

```
int res;
     res = libmem_unlock_all();
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_unlock_all : success\n");
     else
       printf("libmem_unlock_all : failed (%d)\n", res);
```

libmem_write

Synopsis

```
int libmem_write(uint8_t *dest,
                 const uint8_t *src,
                 size_t size);
```

Description

libmem_write writes a block of data using a LIBMEM driver.

dest — A pointer to the address to write the block of data.

src — A pointer to the address to copy the block of data from.

size — The size of the block of data to copy in bytes.

libmem_write returns — The LIBMEM status result.

This function locates the LIBMEM driver for the address pointed to by **start** and then calls the LIBMEM driver's **write** function.

Note that the address range being written to cannot span multiple LIBMEM drivers.

Example:

```
const unsigned char buffer[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
     int res;
     res = libmem_write((uint8_t *)0x10000000, buffer, sizeof(buffer));
     if (res == LIBMEM_STATUS_SUCCESS)
       printf("libmem_write : success\n");
     else
       printf("libmem_write : failed (%d)\n", res);
```

<libmem_loader.h>

API Summary

Macros	
LIBMEM_RPC_LOADER_FLAG_PARAM	Indicates whether the loader parameter has been set.
LIBMEM_RPC_LOADER_FLAG_PRESERVE_STATE	Indicates that a loader should preserve target state.
LIBMEM_RPC_LOADER_MAGIC_NUMBER	Magic number used to identify LIBMEM loader.
Functions	
libmem_dcc_loader_start	Start up a LIBMEM loader that uses the "Comms Channel Loader" protocol.
libmem_dcc_rpc_loader_exit	Exit a DCC RPC loader and return the exit status to the host.
libmem_dcc_rpc_loader_start	Start up a LIBMEM loader that uses remote procedure calls via the ARM's debug comms channel.
libmem_rpc_loader_exit	Exit an RPC loader and return the exit status to the host.
libmem_rpc_loader_start	Start up a LIBMEM loader that uses direct RPC (remote procedure calls).
Types	
libmem_dcc_loader_set_param_fn_t	A function pointer type for a function that handles loader set parameter transactions.

LIBMEM_RPC_LOADER_FLAG_PARAM

Synopsis

```
#define LIBMEM_RPC_LOADER_FLAG_PARAM (1 << 0)
```

Description

LIBMEM loader flag used to indicate whether the loader parameter has been set.

If this flag is set in R0 on entry to an RPC loader then R1 holds the optional loader parameter specified using CrossStudio's "Target | Loader Parameter" project property.

LIBMEM_RPC_LOADER_FLAG_PRESERVE_STATE

Synopsis

```
#define LIBMEM_RPC_LOADER_FLAG_PRESERVE_STATE (1 << 31)
```

Description

LIBMEM loader flag used to indicate that a loader should preserve target state.

If this flag is set in R0 on entry to an RPC loader then the loader should attempt to preserve any existing target state. This is typically set when a loader is used to modify memory while a target program is running which would happen when a software breakpoint is set in ROM during a debug session. If this is functionality is not required then this flag can be ignored.

LIBMEM_RPC_LOADER_MAGIC_NUMBER

Synopsis

```
#define LIBMEM_RPC_LOADER_MAGIC_NUMBER 0x76E9C416
```

Description

Defines the magic number used by host to identify the loader as a LIBMEM loader.

libmem_dcc_loader_set_param_fn_t

Synopsis

```
typedef int (*libmem_dcc_loader_set_param_fn_t)(unsigned , unsigned);
```

Description

parameter — The parameter number to set.

value — The value to set the parameter to.

libmem_dcc_loader_start

Synopsis

```
void libmem_dcc_loader_start(libmem_dcc_loader_set_param_fn_t set_param_fn);
```

Description

set_param_fn — A pointer to the function to call when a loader set parameter message is received or NULL if not required.

Example:

```
int set_param_fn(unsigned int parameter, unsigned int value)
   {
     return 1;
   }
   int main(void)
   {
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     const int flash1_max_geometry_regions = 4;
     libmem_driver_handle_t flash1_handle;
     libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
     libmem_flash_info_t flash1_info;
     // Register FLASH driver.
     libmem_register_cfi_driver(&flash1_handle,
                                flash1_start,
                                flash1_geometry,
                                flash1_max_geometry_regions,
                                &flash1_info);
     // Run the loader
     libmem_dcc_loader_start(set_param_fn);
     return 0;
   }
```

This function starts up a LIBMEM loader that uses the older "Comms Channel Loader" protocol, this function has been provided for backward compatibility with earlier products. This form of loader will only run on ARM7 and ARM9s with the debug comms channel.

To use this form of loader the "Target | Loader File Type" project property must be set to "Comms Channel Loader".

Optional parameters can be passed to the loader using CrossStudio's "Target | Loader Parameter" project property which holds a 32 bit integer value. If a parameter value has been set the function **set_param_fn** will be called when the set parameter message has been received.

libmem_dcc_rpc_loader_exit

Synopsis

```
void libmem_dcc_rpc_loader_exit(int result,
                                const char *error);
```

Description

result — A LIBMEM status result.

error — Pointer to optional error string or NULL if not required.

This function provides a way of signalling to the host that the loader program has completed and also allows the loader to return an exit code and optional error string. Note that this function should only be used in conjunction with libmem_dcc_rpc_loader_start() and that any code located after the call to libmem_dcc_rpc_loader_exit() has been made will not be executed.

The **error** parameter can be used to describe an error not covered by the LIBMEM status results. To use it, set **result** to LIBMEM_STATUS_ERROR and **error** to the error string to be displayed.

The following example demonstrates how to return user defined error messages from the loader and how code can be executed after the loader server has terminated prior to the loader program exiting:

```
static unsigned char buffer[256];
   int initialise()
   {
     ... initialisation code ...
   }
   int deinitialise()
   {
     ... deinitialisation code ...
   }
   int main(void)
   {
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     const int flash1_max_geometry_regions = 4;
     libmem_driver_handle_t flash1_handle;
     libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
     libmem_flash_info_t flash1_info;
     int res;
     const char *error = 0;
     if (initialise())
       {
         // Register FLASH driver.
```

```
        res = libmem_register_cfi_driver(&flash1_handle,
                                         flash1_start,
                                         flash1_geometry,
                                         flash1_max_geometry_regions,
                                         &flash1_info);
        if (res == LIBMEM_STATUS_SUCCESS)
          {
            // Run the loader
            libmem_dcc_rpc_loader_start(buffer, buffer + sizeof(buffer) - 1);
          }
      }
    else
      {
        res = LIBMEM_STATUS_ERROR;
        error = "cannot initialise loader";
      }
    if (!deinitialise() && res == LIBMEM_STATUS_SUCCESS)
      {
        res = LIBMEM_STATUS_ERROR;
        error = "cannot deinitialise loader";
      }
     libmem_dcc_rpc_loader_exit(res, NULL);
     return 0;
   }
```

libmem_dcc_rpc_loader_start

Synopsis

```
void libmem_dcc_rpc_loader_start(void *comm_buffer_start,
                                 void *comm_buffer_end);
```

Description

comm_buffer_start — A pointer to the start of an area of RAM that can be used by the host to store data passed to the remotely called libmem functions.

comm_buffer_end — A pointer to the last byte of the of an area of RAM that can be used by the host to store data passed to the remotely called libmem functions.

Example:

```
static unsigned char buffer[256];
   int main(void)
   {
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     const int flash1_max_geometry_regions = 4;
     libmem_driver_handle_t flash1_handle;
     libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
     libmem_flash_info_t flash1_info;
     // Register FLASH driver.
     libmem_register_cfi_driver(&flash1_handle,
                                flash1_start,
                                flash1_geometry,
                                flash1_max_geometry_regions,
                                &flash1_info);
     // Run the loader
     libmem_dcc_rpc_loader_start(buffer, buffer + sizeof(buffer) - 1);
     return 0;
   }
```

This function starts up a LIBMEM loader that uses remote procedure calls of the LIBMEM library executed via the ARM's debug comms channel. This form of loader will only run on ARM7 and ARM9s with the debug comms channel. It offers some performance advantage over the direct RPC loader.

The advantage of this loader mechanism is that it can be quicker than the direct RPC loader as it uses the ARM debug comms channel for memory access rather than accessing the memory directly reducing significantly the number of JTAG operations required to carry out each operation. It works by the host locating the addresses of the LIBMEM functions in memory by examining the symbols in the loader's binary file and then calling them directly via a simple server handling commands over the ARM debug comms channel.

A communication buffer is required to store the parameters passed to the LIBMEM functions, this buffer is specified using the **comm_buffer_start** and **comm_buffer_end** parameters. The buffer must be at least 8 bytes in length, however you will find the bigger the buffer is, the better the loader performance will be, because less RPC calls will required.

The host loader application adapts how it carries out the download by the set of LIBMEM functions that are linked into the loader application. The only two required functions are **libmem_write**() and **libmem_erase**(), **libmem_unlock**() is only required if the erase or write functions return a **LIBMEM_STATUS_LOCKED** result. Everything else can be done by accessing memory directly however you will get a performance increase when verifying if you link in **libmem_crc32**(). If any of the memory programmed by the LIBMEM drivers cannot be accessed directly you will also need to link in **libmem_read**(), note that if all memory can be accessed directly you should not link this in as using this function will reduce performance.

As the loader application usually makes no direct calls to the LIBMEM functions they will by default be discarded by the linker, you therefore need to make sure the required LIBMEM functions are linked into your loader using the "Linker | Keep Symbols" project property.

To use this form of loader the "Target | Loader File Type" project property must be set to "LIBMEM RPC Loader".

Parameters are passed to an RPC loader by initialising the CPU registers prior to starting the loader. On entry, the register R0 contains the LIBMEM loader flags which can be any of the following:

- **LIBMEM_RPC_LOADER_FLAG_PARAM** - If this flag is set then R1 holds the optional loader parameter specified using CrossStudio's "Target | Loader Parameter" project property.
- **LIBMEM_RPC_LOADER_FLAG_PRESERVE_STATE** - If this flag is set then the loader should attempt to preserve any existing target state. This is typically set when a loader is used to modify memory while a target program is running which would happen when a software breakpoint is set in ROM during a debug session. If this is functionality is not required then this flag can be ignored.

libmem_rpc_loader_exit

Synopsis

```
void libmem_rpc_loader_exit(int result,
                            const char *error);
```

Description

result — A LIBMEM status result.

error — Pointer to optional error string or NULL if not required.

This function provides a way of signalling to the host that the loader program has completed and also allows the loader to return an exit code and optional error string. Note that this function should only be used in conjunction with libmem_rpc_loader_start() and that any code located after the call to libmem_rpc_loader_exit() has been made will not be executed.

The **error** parameter can be used to describe an error not covered by the LIBMEM status results. To use it, set **result** to LIBMEM_STATUS_ERROR and **error** to the error string to be displayed.

The following example demonstrates how to return user defined error messages from the loader and how code can be executed after the loader server has terminated prior to the loader program exiting:

```
static unsigned char buffer[256];
   int initialise()
   {
     ... initialisation code ...
   }
   int deinitialise()
   {
     ... deinitialisation code ...
   }
   int main(void)
   {
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     const int flash1_max_geometry_regions = 4;
     libmem_driver_handle_t flash1_handle;
     libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
     libmem_flash_info_t flash1_info;
     int res;
     const char *error = 0;
     if (initialise())
       {
         // Register FLASH driver.
```

```
        res = libmem_register_cfi_driver(&flash1_handle,
                                         flash1_start,
                                         flash1_geometry,
                                         flash1_max_geometry_regions,
                                         &flash1_info);
        if (res == LIBMEM_STATUS_SUCCESS)
          {
            // Run the loader
            libmem_rpc_loader_start(buffer, buffer + sizeof(buffer) - 1);
          }
      }
    else
      {
        res = LIBMEM_STATUS_ERROR;
        error = "cannot initialise loader";
      }
    if (!deinitialise() && res == LIBMEM_STATUS_SUCCESS)
      {
        res = LIBMEM_STATUS_ERROR;
        error = "cannot deinitialise loader";
      }
     libmem_rpc_loader_exit(res, NULL);
     return 0;
   }
```

libmem_rpc_loader_start

Synopsis

```
int libmem_rpc_loader_start(void *comm_buffer_start,
                            void *comm_buffer_end);
```

Description

comm_buffer_start — A pointer to the start of an area of RAM that can be used by the host to store data passed to the remotely called libmem functions.

comm_buffer_end — A pointer to the last byte of the of an area of RAM that can be used by the host to store data passed to the remotely called libmem functions.

libmem_rpc_loader_start returns — The last error result returned from a LIBMEM function or **LIBMEM_STATUS_SUCCESS** if there has been no error.

Example:

```
static unsigned char buffer[256];
   int main(void)
   {
     uint8_t *flash1_start = (uint8_t *)0x10000000;
     const int flash1_max_geometry_regions = 4;
     libmem_driver_handle_t flash1_handle;
     libmem_geometry_t flash1_geometry[flash1_max_geometry_regions];
     libmem_flash_info_t flash1_info;
     int res;
     // Register FLASH driver.
     res = libmem_register_cfi_driver(&flash1_handle,
                                      flash1_start,
                                      flash1_geometry,
                                      flash1_max_geometry_regions,
                                      &flash1_info);
     if (res == LIBMEM_STATUS_SUCCESS)
       {
         // Run the loader
         libmem_rpc_loader_start(buffer, buffer + sizeof(buffer) - 1);
       }
     libmem_rpc_loader_exit(res, NULL);
     return 0;
   }
```

This function starts up a LIBMEM loader that uses direct remote procedure calls of the LIBMEM library. This form of loader should run on all ARM targets.

The advantage of this loader mechanism is that as it uses direct remote procedure calls and therefore has no server code saving a significant amount of memory. It works by the host locating the addresses of the LIBMEM functions in memory by examining the symbols in the loader's binary file and then calling them directly using the debug interface.

A communication buffer is required to store the parameters passed to the LIBMEM functions, this buffer is specified using the **comm_buffer_start** and **comm_buffer_end** parameters. The buffer must be at least 8 bytes in length, however you will find the bigger the buffer is, the better the loader performance will be, because less RPC calls will required.

The host loader application adapts how it carries out the download by the set of LIBMEM functions that are linked into the loader application. The only two required functions are **libmem_write**() and **libmem_erase**(), **libmem_unlock**() is only required if the erase or write functions return a **LIBMEM_STATUS_LOCKED** result. Everything else can be done by accessing memory directly however you will get a performance increase when verifying if you link in **libmem_crc32**(). If any of the memory programmed by the LIBMEM drivers cannot be accessed directly you will also need to link in **libmem_read**(), note that if all memory can be accessed directly you should not link this in as using this function will reduce performance.

As the loader application usually makes no direct calls to the LIBMEM functions they will by default be discarded by the linker, you therefore need to make sure the required LIBMEM functions are linked into your loader using the "Linker | Keep Symbols" project property.

To use this form of loader the "Target | Loader File Type" project property must be set to "LIBMEM RPC Loader".

Parameters are passed to an RPC loader by initialising the CPU registers prior to starting the loader. On entry, the register R0 contains the LIBMEM loader flags which can be any of the following:

- #LIBMEM_RPC_LOADER_FLAG_PARAM - If this flag is set then R1 holds the optional loader parameter specified using CrossStudio's "Target | Loader Parameter" project property.

- #LIBMEM_RPC_LOADER_FLAG_PRESERVE_STATE - If this flag is set then the loader should attempt to preserve any existing target state. This is typically set when a loader is used to modify memory while a target program is running which would happen when a software breakpoint is set in ROM during a debug session. If this is functionality is not required then this flag can be ignored.

Tasking Library User Guide

This document describes the *CrossWorks Tasking Library* (CTL). The tasking (aka *multitasking*) library provides a multi-priority, preemptive, task switching and synchronization facility. Additionally, it supports a timer, interrupt service routines, and memory-block allocation.

This document is divided into three parts:

- *A whistle-stop tour of the tasking library* introduces the key concepts.
- *Putting the tasking library to use* looks in-depth at the tasking library and how to use it in your applications.
- *Reference information* is a concise reference for each function provided in the tasking library.

Overview

The tasking library enables your application to employ multiple tasks. Tasks are typically used for processing that may suspend execution while other activities occur. For example, you may have a protocol-processing task, a user-interface task, and a data-acquisition task.

Each task has its own *task stack*, which is used to store local variables and function-return information. The task stack is also used to store the CPU execution context when the task isn't executing. The CPU execution context of a task varies between machine architectures; it is typically the subset of the CPU register values that enable a task to be descheduled at any point during its execution.

The process of changing the CPU registers from one task to another is termed *task*switching}. Task switching occurs when a CTL function is called, either from a task or from an interrupt service routine (ISR), and there is a runnable task with higher priority than the executing task. Task switching also occurs when there is a runnable task of the same priority as the executing task, if the executing task has exceeded its time-slice period. If you have more than one runnable task of the same priority, the next task (modulo priority) after the executing task is selected. This is sometimes called *round-robin scheduling*.

There is a single task list and it is kept in priority-sorted order. The task list is updated when tasks are created and deleted, and when their priority changes. The task list is traversed when a CTL function is called that could change the execution state of a task. While the task list is modified or traversed, global interrupts are disabled. Consequently, the length of the interrupt-disable period depends on the number of tasks in the task list, and the priority and type of the task affected by the CTL operation.

If you require a simple, deterministic (sometimes called *real-time*) system, you should ensure that each task has a unique priority. The task switching will always select the highest-priority task that is runnable.

CTL has a pointer to the executing task. There must always be a task executing; if there isn't, a CTL error is signaled. Typically, there will be an idle task that loops and, perhaps, puts the CPU into power-saving mode.

Global interrupts will be enabled when a task switch occurs, so you can safely call tasking library functions while interrupts are disabled.

Task synchronization and resource allocation

The CTL provides several mechanisms to synchronize execution of tasks, to serialize resource access, and to provide high-level communication.

- *Event Sets*: An event set is a word-sized variable, and tasks can wait for its specific bits (representing events) to be set to 1. Events can be used for synchronization and to serialize resource access. Events can be set by interrupt service routines.
- *Semaphores*: A semaphore is a word size variable which tasks can wait for to be non-zero. Semaphores can be used be used for synchronization and to serialize resource access. Semaphores can be signaled by interrupt service routines.

- *Mutexes*: A mutex is a structure that can be used to serialize resource access. Unlike semaphores, mutexes cannot be used by interrupt service routines, but do provide extra features that make mutexes preferable to semaphores for serializing resource access.
- *Message Queues*: A message queue is a structure that enables tasks to post and receive data. Message queues are used to provide a buffered communication mechanism. Messages can be sent by interrupt service routines.
- *Byte Queues*: A byte queue is a specialization of a message queue; i.e., it is a message queue in which the messages are one byte in size. Byte queues can be sent by interrupt service routines.
- *Interrupt Enable and Disable*: The tasking library provides functions that enable and disable the processor's global interrupts. These functions can be used to provide a time-critical, mutual-exclusion facility.

Note that all task synchronization is priority based, i.e., the highest-priority task that is waiting will be scheduled first.

Timer support

If your application can provide a periodic timer interrupt, you can use the timer facility of the CTL. This facility enables time slicing of equal-priority tasks, allows tasks to delay, and provides a timeout capability when waiting for something. The timer is a software counter that is incremented by your timer interrupt. The counter is typically a millisecond counter, but you can change the timer's increment to reduce the interrupt frequency.

Memory allocation support

The CTL provides a simple memory block allocator that can be used in situations for which the standard C **malloc** and **free** functions are either too slow or may block the calling task.

C library support

The CTL provides the functions required of the CrossWorks C library for multi-threading.

Tasks

Each task has a corresponding task structure that contains the following information:

- When the task isn't executing, a pointer to the stack containing the execution context.
- The priority of the task; the lowest priority is 0, the highest is 255.
- The state of the task, runnable or waiting.
- A pointer to the next task.
- If the task is waiting for something, the details of what it is waiting for.
- Thread-specific data such as **errno**.
- A pointer to a null-terminated string that names the task for debugging purposes.

Creating a task

You allocate task structures by declaring them as C variables.

```
CTL_TASK_t mainTask;
```

You create the first task by using ctl_task_init to turn the main program into a task. This function takes a pointer to the task structure that represents the main task, its priority, and a name as parameters.

```
ctl_task_init(&mainTask, 255, "main");
```

This function must be called before any other CrossWorks tasking library calls. The priority (second parameter) must be between 0 (the lowest priority) and 255 (the highest priority). It is advisable to create the first task with the highest priority, which enables it to create other tasks without being descheduled. The name should point to a zero-terminated ASCII string, which is shown in the **Threads** window.

You can create other tasks with the function **ctl_task_run**, which initializes a task structure and may cause a context switch. You supply the same arguments as for **ctl_task_init**, together with the function the task will run and the memory the task will use for its stack.

The function a task will run should take a **void** * parameter and not return any value.

```
void task1Fn(void *parameter)
{
  // task code goes in here
}
```

The **parameter** value is supplied to the function by the **ctl_task_run** call. Note that, when a task function returns, the **ctl_task_die** function is called, terminating the task.

You must allocate the stack for the task as a C array of **unsigned** elements.

```
unsigned task1Stack[64];
```

The stack size you need depends on the CPU (i.e., the number of registers that must be saved), the function calls the task will make, and (again depending on the CPU) the stack used for interrupt service routines. Running out of stack space is a common problem for multitasking systems, and the resulting error behavior is often misleading. Recommended practice is to initialize the stack to known values—that will make it easier to check the stack's contents with the CrossWorks debugger if problems should occur.

```
memset(task1Stack, 0xba, sizeof(task1Stack));
```

Your **ctl_task_run** function call should look something like this:

```
ctl_task_run(&task1Task,
             12,
             task1Fn,
             0,
             "task1",
             sizeof(task1Stack) / sizeof(unsigned),
             task1Stack,
             0);
```

The first parameter is a pointer to the task structure. The second parameter is the priority (in this case 12) at which the task will start executing. The third parameter is a pointer to the function to execute (in this case **task1Fn**). The fourth parameter is the value supplied to the task function (zero, in this case). The fifth parameter is a null-terminated string that names the task for debug purposes. The sixth parameter is the size of the stack, in words. The seventh parameter is the pointer to the stack. The last parameter is for systems that have a separate *call stack*, and its value is the number of words to reserve for that stack.

Changing a task's priority

You can change the priority of a task using **ctl_task_set_priority**. It takes a pointer to a task structure and the new priority as parameters, and returns the old priority.

```
old_priority = ctl_task_set_priority(&mainTask, 255); // lock scheduler
…  your code here
ctl_task_set_priority(old_priority);
```

To enable time slicing, you need to set the **ctl_timeslice_period** variable before any task scheduling occurs.

```
ctl_timeslice_period = 100; // time slice period of 100 ms
```

If you want finer control over the scheduling of tasks, you can call **ctl_task_reschedule**. The following example turns **main** into a task and creates a second task. The main task ultimately will be the lowest-priority task that switches the CPU into a power-saving mode when it is scheduled—this satisfies the requirement of always having a task to execute and enables a simple, power-saving system to be implemented.

```
#include <ctl.h>
```

```
void task1(void *p)
{
  // task code; on return, the task will be terminated.
}
static CTL_TASK_t mainTask, task1Task;
static unsigned task1Stack[64];

int main(void)
{
  // Turn myself into a task running at the highest priority.
  ctl_task_init(&mainTask, 255, "main");
  // Initialize the stack of task1.
  memset(task1Stack, 0xba, sizeof(task1Stack));

  // Prepare another task to run.
  ctl_task_run(&task1Task, 1, task1, 0, "task1",
               sizeof(task1Stack) / sizeof(unsigned),
               task1Stack, 0);
  // Now that all the tasks have been created, go to the lowest priority task.
  ctl_task_set_priority(&mainTask, 0);
  // Main task, if activated because task1 is suspended, just
  // enters low-power mode and waits for task1 to run again
  // (for example, because an interrupt wakes it).
  for (;;)
    {
      // Go into low-power mode
      sleep();
    }
}
```

Note that, initially, the main task is assigned the highest priority while it creates the other tasks; then it changes its priority to the lowest value. This technique can be used, when multiple tasks are to be created, to ensure all the tasks are created before they start to execute.

Note the use of **sizeof** when passing the stack size to **ctl_task_run**.

Event sets

Event sets are a versatile way to communicate between tasks, manage resource allocation, and synchronize tasks.

An event set is a means to synchronize tasks with other tasks and with interrupt service routines. An event set contains a set of events (one per bit), and tasks can wait for one or more of these bits to be set (i.e., to have the value 1). When a task waits on an event set, the events it is waiting for are matched against the current values—if they match, the task can still execute. If they don't match, the task is put on the task list with details about the event set and the events for which the task is waiting.

You allocate an event set by declaring it as C variable:

```
CTL_EVENT_SET_t e1;
```

A **CTL_EVENT_SET_t** is a synonym for an **unsigned** type. Thus, when an **unsigned** is 16 bits wide, an event set will contain 16 events; and when it consists of 32 bits, an event set will contain 32 events.

An event set must be initialized before any tasks can use it. To initialize an event set, use **ctl_events_init**:

```
ctl_events_init(&e1, 0);
```

You can set and clear events in an event set using the **ctl_events_set_clear** function.

```
ctl_events_set_clear(&e1, 1<<0, 1<<15);
```

This example will set the bit-zero event and clear the bit-15 event. If any tasks are waiting on this event set, the events they are waiting on will be matched against the new event set value, which could cause the task to become runnable.

You can wait for events to be set using **ctl_events_wait**. You can wait for any of the events in an event set to be set (**CTL_EVENT_WAIT_ANY_EVENTS**) or all of the events to be set (**CTL_EVENT_WAIT_ALL_EVENTS**). You can also specify that when events have been set and have been matched that they should be automatically reset (**CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR** and **CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR**). You can associate a timeout with a wait for an event set to stop your application blocking indefinitely.

```
ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e1, 1<<15, CTL_TIMEOUT_NONE, 0);
```

This example waits for bit 15 of the event set pointed to by **e1** to become set.

```
if (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &;e1, 1<<15), CTL_TIMEOUT_DELAY, 1000) == 0
  {
    // timeout occurred
  }
```

This example uses a timeout and tests the return result to see if the timeout occurred.

You can use **ctl_events_pulse** to set and immediately clear events. A typical use for this would be to wake up multiple threads and reset the events atomically.

Synchronizing with an ISR

The following example illustrates synchronizing a task with a function called from an ISR.

```
CTL_EVENT_SET_t e1;
CTL_TASK_s t1;
void ISRfn()
{
  // do work, and then...
  ctl_events_set_clear(&e1, 1<<0, 0);
}
void task1(void *p)
{
  for (;;)
    {
      ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e1, 1<<0, CTL_TIMEOUT_NONE, 0);
      // Do whatever needs to be done
      ctl_events_set_clear(&e1, 0, 1<<0);
    }
}
```

Synchronizing with more than one ISR

The following example illustrates synchronizing a task with functions called from two interrupt service routines.

```
CTL_EVENT_SET_t e1;
CTL_TASK_s t1;
void ISRfn1(void)
{
  // do work, and then...
  ctl_events_set_clear(&e1, 1<<0, 0);
}
void ISRfn2(void)
{
  // do work, and then...
  ctl_events_set_clear(&e1, 1<<1, 0);
}
void task1(void *p)
{
```

```
  for (;;)
    {
      unsigned e;
      e = ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR,
                          &;e1,
                          (1<<0) | (1<<1), CTL_TIMEOUT_NONE, 0);
      if (e & (1<<0))
        {
          // ISRfn1 completed
        }
      else if (e & (1<<1))
        {
          // ISRfn2 completed
        }
      else
        {
          // error
        }
    }
}
```

Resource serialization with an event set

The following example illustrates resource serialization of two tasks.

```
CTL_EVENT_SET_t e1;
void task1(void)
{
  for (;;)
    {
      ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR, &e1, 1<<0, CTL_TIMEOUT_NO
      // resource has now been acquired
      ctl_events_set_clear(&e1, 1<<0, 0);
      // resource has now been released
    }
}
void task2(void)
{
  for (;;)
    {
      ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR, &e1, 1<<0, CTL_TIMEOUT_NO
```

```
      // resource has now been acquired
      ctl_events_set_clear(&e1, 1<<0, 0);
      // resource has now been released
    }
}
void main(void)
{
  // Initialize event set
  ctl_events_init(&e1, 1<<0);
  // Create tasks and let them run
}
```

Note that **e1** is initialized with the event set; without this, neither task would acquire the resource.

Semaphores

CTL provides *semaphores* to use for synchronization and resource allocation.

A semaphore is a counter which tasks can wait for to be non-zero. When a semaphore is non-zero and a task waits on it, the semaphore value is decremented and the task continues executing. When a semaphore is zero and a task waits on it, the task will be suspended until the semaphore is signaled. When a semaphore is signaled and no tasks are waiting for it, the semaphore's value is incremented. When a semaphore is signaled and tasks are waiting, one of the tasks is made runnable.

You allocate a semaphore by declaring it as a C variable. For example:

```
CTL_SEMAPHORE_t s1;
```

A **CTL_SEMAPHORE_t** is a synonym for an **unsigned** type, so the maximum value of the counter is dependent upon the word size of the processor (16 or 32 bits).

A semaphore must be initialized done before any tasks can use it. To initialize a semaphore, use **ctl_semaphore_init**:

```
ctl_semaphore_init(&s1, 1);
```

To signal a semaphore, use **ctl_semaphore_signal**:

```
ctl_semaphore_signal(&s1);
```

The highest-priority task waiting on the semaphore pointed at by **s1** will be made runnable by this call. If no tasks are waiting on the semaphore, the semaphore's value is incremented.

To wait for a semaphore with an optional timeout, use **ctl_semaphore_wait**:

```
ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
```

This example will block the task if the semaphore is zero, otherwise it will decrement the semaphore and execution will continue.

```
if (ctl_semaphore_wait(&s1, CTL_TIMEOUT_ABSOLUTE, ctl_get_current_time()+1000) == 0)
  {
    // timeout occurred
  }
```

This example uses a timeout and tests the return result to see if the timeout occurred.

Task synchronization in an interrupt service routine

The following example illustrates synchronizing a task with a function called from an interrupt service routine.

```
CTL_SEMAPHORE_t s1;
void ISRfn()
```

```
{
  // Detected something, signal the waiting task.
  ctl_semaphore_signal(&s1);
}
void task1(void *p)
{
  for (;;)
    {
      // Wait for ISR to signal that an event happened.
      ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
      // Deal with the event.
    }
}
```

Resource serialization with semaphore

The following example illustrates resource serialization of two tasks:

```
CTL_SEMAPHORE_t s1 = 1;
void task1(void)
{
  for (;;)
    {
      // Wait for resource.
      ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
      // Resource has now been acquired, do something with it.
      // Then release it.
      ctl_semaphore_signal(&s1);
      // Resource is now released.
    }
}
void task2(void)
{
  for (;;)
    {
      ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
      // Resource has now been acquired.
      …
      ctl_semaphore_signal(&s1);
      // Resource has now been released.
    }
```

```
}
int main(void)
{
  …
  ctl_semaphore_init(&s1, 1);
  …
}
```

Note that **s1** is initialized to one; without this, neither task would acquire the resource.

Mutexes

A mutex is a means to allocate a resource on a task basis.

A mutex is a structure that can be used to serialize resource access. Tasks can lock and unlock mutexes. Each mutex has a *lock count* that enables a task to recursively lock the mutex. Tasks must ensure that the number of unlocks matches the number of locks. When a mutex has already been locked by another task, a task that wants to lock it must wait until the mutex becomes unlocked. The task that locks a mutex is assigned a higher priority than any other tasks waiting to lock that mutex; this avoids what is often called *priority inversion*, which can prevent some tasks from ever getting access to a required resource. Mutexes cannot be used by interrupt service routines.

You allocate a mutex by declaring it as a C variable. For example:

```
CTL_MUTEX_t mutex;
```

A mutex must be intitialized before any task can use it. To initialize a mutex, use **ctl_mutex_init** as in this example:

```
ctl_mutex_init(&mutex);
```

You can lock a mutex with an optional timeout by using **ctl_mutex_lock**:

```
ctl_mutex_lock(&mutex, CTL_TIMEOUT_NONE, 0);
```

You can unlock a mutex by using **ctl_mutex_unlock**:

```
ctl_mutex_unlock(&mutex);
```

Note: Only the locking task must unlock a successfully-locked mutex.

Resource serialization with mutex

The following example illustrates resource serialization of two tasks.

```
CTL_MUTEX_t mutex;
void fn1(void)
{
  ctl_lock_mutex(&mutex, CTL_TIMEOUT_NONE, 0);
  …
  ctl_unlock_mutex(&mutex);
}
void fn2(void)
{
  ctl_lock_mutex(&mutex, CTL_TIMEOUT_NONE, 0);
```

```
  …
  fn1();
  …
  ctl_unlock_mutex(&mutex);
}
void task1(void)
{
  for (;;)
    {
      fn2()
    }
}
void task2(void)
{
  for (;;)
    {
      fn1();
    }
}
int main(void)
{
  …
  ctl_mutex_init(&mutex);
  …
}
```

Note that **task1** locks the mutex twice by calling **fn2** which then calls **fn1**.

Message queues

Message queues provide buffers between tasks and interrupt service routines.

A *message queue* is a structure that enables tasks to post and receive messages. A *message* is a generic (void) pointer and, as such, can be used to send data that will fit into a pointer type (two or four bytes, depending upon the processor's word size) or to pass a pointer to a block of memory. The message queue uses a buffer to enable a number of posts to be completed without receives occurring. The buffer keeps the posted messages in FIFO order, so the oldest message is received first. When the buffer isn't full, a post will put the message at the back of the queue and the calling task continues execution. When the buffer is full, a post will block the calling task until there is room for the message. When the buffer isn't empty, a receive will return the message from the front of the queue and continue execution of the calling task. When the buffer is empty, a receive will block the calling task until a message is posted.

Initializing a message queue

You allocate a message queue by declaring it as a C variable:

```
CTL_MESSAGE_QUEUE_t m1;
```

A message queue is initialized using **ctl_message_queue_init**:

```
void *queue[20];
...
ctl_message_queue_init(&m1, queue, 20);
```

This example uses a 20-element array for the message queue. The array is a **void** * so pointers to memory or (cast) integers can be communicated via a message queue.

Posting to a message queue

You can post a message to a message queue with an optional timeout by using the **ctl_message_queue_post** function.

```
ctl_message_queue_post(&m1, (void *)45, CTL_TIMEOUT_NONE, 0);
```

This example posts the integer 45 to the message queue.

You can post multiple messages to a message queue with an optional timeout using **ctl_message_queue_post_multi**:

```
if (ctl_message_queue_post_multi(&m1, 4, messages, CTL_TIMEOUT_ABSOLUTE, ctl_get_current_t
= 4)
```

```
  {
    // timeout occurred
  }
```

This example tests the return result to see if the timeout occurred.

If you want to post a message and you cannot afford to block (e.g. inside an interrupt service routine), you can use **ctl_message_queue_post_nb** (or **ctl_message_queue_post_multi_nb** if you want to post multiple messages):

```
if (ctl_message_queue_post_nb(&m1, (void *)45) == 0)
  {
    // queue is full
  }
```

This example tests the return result to see if the post failed.

Receiving from a message queue

You can use **ctl_message_queue_receive** to receive a message with an optional timeout:

```
void *msg;
ctl_message_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
```

This example receives the oldest message in the message queue.

Use **ctl_message_queue_receive_multi** to receive multiple messages from a message queue with an optional timeout:

```
if (ctl_message_queue_multi_receive(&m1, 4, msgs, CTL_TIMEOUT_DELAY, 1000) !
= 4)
  {
    // timeout occurred
  }
```

This example tests the return result to see if the timeout occurred.

If you want to receive a message and you don't want to block (e.g., when executing interrupt service routine), you can use **ctl_message_queue_receive_nb** (or **ctl_message_queue_receive_multi_nb** to receive multiple messages).

```
if (ctl_message_queue_receive_nb(&m1, &msg) == 0)
  {
    // queue is empty
  }
```

Producer-consumer example

The following example uses a message queue to implement the producer-consumer problem.

```
CTL_MESSAGE_QUEUE_t m1;
void *queue[20];
void task1(void)
{
  …
  ctl_message_queue_post(&m1, (void *)i, CTL_TIMEOUT_NONE, 0);
  …
}
void task2(void)
{
  void *msg;
  …
  ctl_message_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
  …
}
int main(void)
{
  …
  ctl_message_queue_init(&m1, queue, 20);
  …
}
```

Advanced use

You can associate event flags with a message queue that are set (and similarly cleared) when the message queue is not full and not empty using the function **ctl_message_queue_setup_events**.

For example, you can use this to wait for messages to arrive from multiple message (or byte) queues:

```
CTL_MESSAGE_QUEUE_t m1, m2;
CTL_EVENT_SET_t e;
ctl_message_queue_setup_events(&m1, &e, 1<<0, 1<<1));
ctl_message_queue_setup_events(&m2, &e, 1<<2, 1<<3));
…
switch (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS,
                        &e, (1<<0) | (1<<2),
                        CTL_TIMEOUT_NONE, 0))
  {
```

```
  case 1<<0:
    ctl_message_queue_receive(&m1, …
    break;
  case 1<<2:
    ctl_message_queue_receive(&m2, …
    break;
  }
```

This example sets up and waits for the not-empty event of message queue **m1** and the not-empty event of message queue **m2**. When the wait completes, it reads from the appropriate message queue. Note that you *should not* use a 'with auto clear' event wait type when waiting for events associated with a message queue.

You can use **ctl_message_queue_num_used** to test how many messages are in a message queue and **ctl_message_queue_num_free** to learn how many free messages are in a message queue. With these functions you can poll the message queue:

```
while (ctl_message_queue_num_free(&m1) < 10)
  ctl_task_timeout_wait(ctl_get_current_time() + 1000);
ctl_message_queue_post_multi(&m1, 10, …
```

This example waits for 10 elements to be free before it posts 10 elements.

Byte queues

Byte queues provide byte-based buffers between tasks and interrupt service routines.

A *byte queue* is a structure that enables tasks to post and receive data bytes. The byte queue has a buffer, which enables a number of posts to be completed without receives occurring. The buffer keeps the posted bytes in FIFO order, so the oldest byte is received first. When the buffer isn't full, a post will put the byte at the back of the queue and the calling task continues execution. When the buffer is full, a post will block the calling task until there is room for the byte. When the buffer isn't empty, a receive will return the byte from the front of the queue and continue execution of the calling task. When the buffer is empty, a receive will block the calling task until a byte is posted.

Initializing a byte queue

You allocate a byte queue by declaring it as a C variable:

```
CTL_BYTE_QUEUE_t m1;
```

A byte queue is initialized using **ctl_byte_queue_init**:

```
unsigned char queue[20];
...
ctl_byte_queue_init(&m1, queue, 20);
```

This example uses an 20-element array for the byte queue.

Posting to a byte queue

You can post a byte to a byte queue with an optional timeout using **ctl_byte_queue_post**:

```
ctl_byte_queue_post(&m1, 45, CTL_TIMEOUT_NONE, 0);
```

This example posts the byte 45 to the byte queue.

You can post multiple bytes to a byte queue with an optional timeout using **ctl_byte_queue_post_multi**:

```
if (ctl_byte_queue_post(&m1, 4, bytes, CTL_TIMEOUT_ABSOLUTE, ctl_get_current_time()+1000)
= 4)
  {
    // timeout occurred
  }
```

This example uses a timeout and tests the return result to see if the timeout occurred.

If you want to post a byte and you don't want to block access (e.g., from an interrupt service routine), you can use **ctl_byte_queue_post_nb** (or **ctl_byte_queue_post_multi_nb** to post multiple bytes).

```
if (ctl_byte_queue_post_nb(&m1, 45) == 0)
  {
    // queue is full
  }
```

This example tests the return result to see if the post failed.

Receiving from a byte queue

You can receive a byte with an optional timeout by using **ctl_byte_queue_receive**:

```
unsigned char msg;
ctl_byte_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
```

This example receives the oldest byte in the byte queue.

You can receive multiple bytes from a byte queue with an optional timeout using **ctl_byte_queue_receive_multi**:

```
if (ctl_byte_queue_receive_multi(&m1, 4, bytes, CTL_TIMEOUT_DELAY, 1000) != 4)
  {
    // timeout occurred
  }
```

This example tests the return result to see if the timeout occurred.

If you want to receive a byte and you don't want to block (e.g., from an interrupt service routine), you can use **ctl_byte_queue_receive_nb** (or **ctl_byte_queue_receive_multi_nb** to receive multiple bytes).

```
if (ctl_byte_queue_receive_nb(&m1, &msg) == 0)
  {
    // queue is empty
  }
```

Producer-consumer example

The following example uses a byte queue to implement the producer-consumer problem.

```
CTL_BYTE_QUEUE_t m1;
void *queue[20];
void task1(void)
```

```
{
  …
  ctl_byte_queue_post(&m1, (void *)i, CTL_TIMEOUT_NONE, 0);
  …
}
void task2(void)
{
  void *msg;
  …
  ctl_byte_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
  …
}
int main(void)
{
  …
  ctl_byte_queue_init(&m1, queue, 20);
  …
}
```

Advanced Use

You can associate event flags with a byte queue that are set (and similarly cleared) when the byte queue is not full and not empty using the function **ctl_byte_queue_setup_events**.

For example, you can use this to wait for messages to arrive from multiple byte (or message) queues.

```
CTL_BYTE_QUEUE_t m1, m2;
CTL_EVENT_SET_t e;
ctl_byte_queue_setup_events(&m1, &e, (1<<0), (1<<1));
ctl_byte_queue_setup_events(&m2, &e, (1<<2), (1<<3));
…
switch (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e, (1<<0)|
(1<<2), CTL_TIMEOUT_NONE, 0))
  {
    case 1<<0:
      ctl_byte_queue_receive(&m1, …
      break;
    case 1<<2:
      ctl_byte_queue_receive(&m2, …
      break;
  }
```

This example sets up and waits for the not-empty event of byte queue **m1** and the not-empty event of byte queue **m2**. When the wait completes, it reads from the appropriate byte queue. Note that you *must not* use a 'with auto clear' event wait type when waiting on events associated with a byte queue.

You can use **ctl_byte_queue_num_used** to test how many bytes are in a byte queue and **ctl_byte_queue_num_free** to learn how many free bytes are in a byte queue. With these functions, you can poll the byte queue:

```
while (ctl_byte_queue_num_free(&m1) < 10)
  ctl_task_timeout_wait(ctl_get_current_time()+1000);
ctl_byte_queue_post_multi(&m1, 10, …
```

This example waits for 10 elements to be free before it posts 10 elements.

Timers and interrupts

This section deals with how CTL timers work.

Global interrupts control

CTL provides functions that enable and disable the global interrupt enables of the processor. CTL uses this mechanism when accessing the task list. It can also be used to provide a fast, mutual-exclusion facility for time-critical uses.

You can disable interrupts by using **ctl_global_interrupts_disable** and you can enable interrupts by using **ctl_global_interrupts_enable**.

If you don't know if interrupts are currently disabled, you can use **ctl_global_interrupts_set**. This will either disable or enable interrupts, depending on the parameter, and will return the previous interrupt-enable state:

```
int en = ctl_global_interrupts_set(0); // disable
...
ctl_global_interrupts_set(en); // set to previous state
```

You can call a tasking library function that causes a task switch with global interrupts disabled. The tasking library will ensure that, when the next task is scheduled, global interrupts are enabled.

Timer support

The current time is held as a 32-bit value in the variable **ctl_current_time**. This variable is incremented by the number held in **ctl_time_increment** each time an ISR calls **ctl_increment_tick_from_isr**.

```
void timerISR{void)
  {
    ctl_increment_tick_from_isr();
    // Your timer code goes here.
  }
int main()
{
  ctl_time_increment = 10;
  // Set up timerISR to be called every 100 ms.
  ..
```

By convention, the timer implements a millisecond counter, but you can set the timer's interrupt-and-increment rate to a value that is appropriate for your application.

You can atomically read **ctl_current_time** by using the **ctl_get_current_time** function on systems whose word size is not 32 bits.

You can use **ctl_timeout_wait** to suspend execution of a task for a fixed period. Note: **ctl_timeout_wait** takes as its parameter the time to resume execution, not the duration: always call this function with **ctl_get_current_time()**+ *duration*.

```
ctl_timeout_wait(ctl_get_current_time()+100);
```

This example suspends execution of the calling task for 100 ticks of the **ctl_current_time** variable.

The counter is implemented as a 32-bit number, so you can delay for a maximum of a 31-bit number.

```
ctl_timeout_wait(ctl_get_current_time() + 0x7fffffff);
```

This example suspends execution of the calling task for the maximum possible time.

Interrupt service routines

Interrupt service routines (ISR) can communicate with CTL tasks using a subset of the CTL programming interface. An ISR should not call any of the CTL functions that can block; if your ISR calls a blocking function, **ctl_handle_error** will be called. To detect whether a task or an ISR has called a function, CTL uses the global variable **ctl_interrupt_count**. Interrupt service routines must increment this variable on entry and decrement it on exit. Any CTL functions called by an ISR that require a task reschedule will set the variable **ctl_reschedule_on_last_isr_exit**.

On exit from an interrupt service routine, **ctl_interrupt_count** is decremented to zero and, if **ctl_reschedule_on_last_isr_exit** is set (after resetting **ctl_reschedule_on_last_isr_exit**), a CTL reschedule operation occurs. The support for writing ISRs differs, depending on the target. In general, on entry to an ISR the following is needed:

```
preserve register state;
ctl_interrupt_count++;
```

…and, on exit from an ISR:

```
ctl_interrupt_count--;
if (ctl_interrupt_count == 0 && ctl_reschedule_on_last_isr_exit)
  {
    ctl_reschedule_on_last_isr_exit = 0;
    reschedule
  }
else
   restore register state
```

Memory areas

Memory areas provide your application with dynamic allocation of fixed-sized memory blocks. Memory areas should be used in preference to the standard C library **malloc** and **free** functions if the calling task cannot block or if memory allocation is done by an ISR.

You allocate a memory area by declaring it as a C variable:

```
CTL_MEMORY_AREA_t m1;
```

Before using a memory area, you must initialize it using **ctl_memory_area_init**:

```
unsigned mem[20];
...
ctl_memory_area_init(&m1, mem, 2, 10);
```

This example uses a 20-element array for the memory area's working storage. The array is split into 10 blocks, each block being two words in size.

To allocate a block from a memory area, use **ctl_memory_area_allocate**. If the memory block cannot be allocated, zero is returned.

```
unsigned *block = ctl_memory_area_allocate(&m1);
if (block)
  // block has been allocated
else
  // no block has been allocated
```

When you have finished with a memory block, use **ctl_memory_area_free** to return it to the memory area *from which it was allocated* so it can be reused:

```
ctl_memory_area_free(&;m1, block);
```

You can associate an event flag with the *block available* state of a memory queue to wait for a memory block to become available:

```
CTL_MEMORY_AREA_t m0, m1, m2;
...
CTL_EVENT_SET_t e;
...
ctl_memory_area_setup_events(&m0, &e, 1<<0);
ctl_memory_area_setup_events(&m1, &e, 1<<1);
ctl_memory_area_setup_events(&m2, &e, 1<<2);
...
switch (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e, (1<<0)|(1<<1)|
(1<<2), 0, 0))
```

```
{
case 1<<0:
  x = ctl_memory_area_allocate(&m0, …
  break;
case 1<<1:
  x = ctl_memory_area_allocate(&m1, …
  break;
case 1<<2:
  x = ctl_memory_area_allocate(&m2, …
  break;
}
```

This example sets up and waits for the block-available events of memory areas **m0**, **m1**, and **m2**. When the wait completes, it attempts to allocate memory from the appropriate memory area. Note that you should *not* use a *with-auto-clear* event wait type when waiting on events associated with a memory area.

Task scheduling example

An example task list could be:

- task1, priority 2, waiting
- task2, priority 1, runnable
- task3, priority 1, executing
- task4, priority 1, runnable
- task5, priority 0, runnable

task2 waits, so task3 is selected to execute:

- task1, priority 2, waiting
- task2, priority 1, waiting
- task3, priority 1, executing
- task4, priority 1, runnable
- task5, priority 0, runnable

An interrupt occurs that makes task1 runnable, which is higher priority than task3 so task1 executes:

- task1, priority 2, executing
- task2, priority 1, waiting
- task3, priority 1, runnable
- task4, priority 1, runnable
- task5, priority 0, runnable

task1 waits, causing task3 to execute:

- task1, priority 2, waiting
- task2, priority 1, waiting
- task3, priority 1, executing
- task4, priority 1, runnable
- task5, priority 0, runnable

An interrupt occurs and task3 has used its timeslice period, so task4 is selected to execute:

- task1, priority 2, waiting
- task2, priority 1, waiting
- task3, priority 1, runnable
- task4, priority 1, executing
- task5, priority 0, runnable

An interrupt occurs and makes task2 runnable, but task4 hasn't used its timeslice period, so it is left to execute:

- task1, priority 2, waiting
- task2, priority 1, runnable

- task3, priority 1, runnable
- task4, priority 1, executing
- task5, priority 0, runnable

A interrupt occurs and task4 has used its timeslice period:

- task1, priority 2, waiting
- task2, priority 1, executing
- task3, priority 1, runnable
- task4, priority 1, runnable
- task5, priority 0, runnable

ARM implementation details

Processor modes

The ARM implementation of CTL uses System and IRQ processor modes. Other processor modes are not used and, therefore, are available for use by the application. In normal execution, tasks run in System mode with IRQ interrupts enabled. When CTL requires exclusive access to variables, for example when traversing the task list, IRQ interrupts are disabled. FIQ interrupts are never disabled by CTL. Co-operative context switching is done by changing to IRQ mode (with IRQ interrupts disabled) and, consequently, uses the IRQ mode stack. Preemptive context switching is done from an IRQ handler, which by definition is running in IRQ mode.

Register save order

When a task is not executing, the ARM register context is saved on the task's stack in the following order:

- PSR
- R15
- R14
- R12
- R3–R0
- R1–R4

…with the **stack_pointer** member of the task structure pointing to the **R4** entry.

IRQ handler

On entry to an IRQ handler, the **ctl_interrupt_count** variable should be incremented. On exit from an IRQ handler, the **ctl_exit_isr** routine should be called with a parameter in **R0** specifying which registers have been saved by the IRQ handler.

If **ctl_exit_isr**(0) is called, the registers should be saved, as in the following example:

```
irq_handler:
  …
  // store the APCS registers
  sub lr, lr, #4
  stmfd sp!, {r0-r3, r12, lr}
  //ctl_interrupt_count++
  ldr r2, =ctl_interrupt_count
  ldrb r3, [r2]
```

```
  add r3, r3, #1
  strb r3, [r2]
  …
  // handle interrupt, possibly re-enabling interrupts
  …
  // ctl_exit_isr(0)
  mov r0, #0
  ldr r1, =ctl_exit_isr
  bx r1
```

If `ctl_exit_isr(!0)` is called, the registers should be saved, as in the following example:

```
irq_handler:
  …
  // store all the registers
  stmfd sp!, {r0-r12, lr}
  mrs r0, spsr
  stmfd sp!, {r0}
  // ctl_interrupt_count++
  ldr r2, =ctl_interrupt_count
  ldrb r3, [r2]
  add r3, r3, #1
  strb r3, [r2]
  …
  // handle interrupt, possibly re-enabling interrupts
  …
  // ctl_exit_isr(!0)
  mov r0, sp
  ldr r1, =ctl_exit_isr
  bx r1
```

The first form (`ctl_exit_isr(0)`) is recommended because it uses less stack space and takes fewer machine cycles. The second form (`ctl_exit_isr(!0)`) is provided for backwards compatibility with earlier releases of CTL.

Cortex-M3 implementation details

Processor modes and interrupts

All CTL threads run in privileged thread mode.

>When CTL requires exclusive access to variables—for example when traversing the task list—interrupts are disabled using calls to **ctl_global_interrupts_disable** and **ctl_global_interrupts_enable**.

The default implementation of global interrupt disable and enable for the Cortex-M3/M4 will set or clear the top bit of the **BASEPRI** register. This enables interrupts that have the highest half of the available priority numbers (lowest priority levels) to use CTL API calls. The lowest half of the available priority numbers (highest priority levels) cannot use CTL API calls but will not be disabled during CTL API calls. If, for example, the device has four priority bits, then priority numbers 8 through 15 can be used for interrupts that make CTL API calls and priorities 0 through 7 can be used for interrupts that cannot make CTL API calls.

The default implementation of global interrupts and enable for the Cortex-M0/M1 will set or clear the **PRIMASK** register.

Exceptions

Context switching is implemented using the PendSV exception handler which should be set to run at the lowest exception priority i.e. the highest exception priority number. The SVCall exception handler is not used.

Stacks

CTL threads can use either the main stack or the process stack. Use of the process stack is recommended so that you don't need to allocate stack space for exceptions in the CTL thread stacks.

Register save order

When a task is not executing, the ARM register context is saved on the task's stack in the following order:

- PSR
- R15
- R14
- R12
- R3–R0
- R1–R4

…with the **stack_pointer** member of the task structure pointing to the **R4** entry.

Interrupt handlers

A Cortex-M interrupt handler that uses CTL services should use the following template code for entry and exit:

```
void
SysTick_ISR()
{
  ctl_enter_isr();
  …
  // handle interrupt
  …
  ctl_exit_isr();
}
```

The call to **ctl_enter_isr** will increment the **ctl_interrupt_count** and the call to **ctl_exit_isr** will decrement the **ctl_interrupt_count** and, if required, trigger the PendSV exception.

>Interrupt handler support code (including **ctl_enter_isr** and **ctl_exit_isr**) is not part of CTL, but there are common definitions that are available in `ctl_api.h` and will be defined by code or libraries supplied by the CPU support package you are using.

System timer

The CPU support package you are using will use the Cortex-M SysTick timer to implement the CTL timer. Typically, the timer will be programmed to interrupt at 10 millisecond intervals and increment the CTL timer by 10 to create the millisecond CTL timer.

The ISR will run on a dedicated stack which avoids having to allocate stack space for interrupt service routines on task stacks. You must specify the dedicated stack size using the the linker symbol **CTL_IRQ_STACK_SIZE**. For example, using `CTL_IRQ_STACK_SIZE=128` will allocate 128 bytes of stack. The size of the stack must be an positive, even integer.

CTL Revisions

CTL has been supplied with various 1.x releases of CrossWorks, and its revision history is available in the corresponding release notes—such releases are termed CTL V1. CTL is supplied in 2.x releases of CrossWorks and is termed CTL V2. This document explains the differences between CTL V1 and CTL V2.

Scheduling implementation

In CTL V1, the executing task was removed from the task list and then put back on when it was descheduled. In CTL V2, the executing task isn't moved from the task list. This change enables CTL to run on the Cortex-M3, results in smaller code with faster context switching, and reduces the interrupt-lockout period.

In CTL V1, rescheduling would happen on exit from the last nested interrupt service routine. In CTL V2, a reschedule only occurs on exit from the last nested interrupt service routine if the run state of a task has changed.

Mutexes

POSIX thread-style mutexes have been added.

Task restore

The new function **ctl_task_restore** allows tasks that have been removed from the task list (using **ctl_task_remove**) to be returned to the task list again.

Suspended task state

The new task state **suspended** has been added. It can be used instead of removing and restoring a task from the task list.

Thread-specific data pointer

The new member **data** has been added to the task structure. It can be used to store thread-specific data.

Task execution time

The new global variable **ctl_last_schedule_time** has been added, and a new field **execution_time** has been added to the task structure to hold the cumulative number of timer ticks since the task began executing.

Change to global interrupt functions

The functions **ctl_global_interrupts_disable** and **ctl_global_interrupts_enable** no longer return the previous interrupt-enables state. If you need that, use **ctl_global_interrupts_set** instead.

Other API changes

The function **ctl_task_set_priority** returns the old task priority.

Header file changes

In CTL V1, the file **ctl/include/ctl_api.h** contained CTL declarations and board-support declarations. In CTL V2, the file **ctl/include/ctl.h** contains CTL declarations. In CTL V2, the file **ctl/include/ctl_api.h** includes **ctl/source/ctl.h** and has board-support declarations for backwards compatibility.

Removed support for interrupt re-enabling

The following functions have been removed from CTL V2:

- **ctl_global_interrupts_re_enable_from_isr**
- **ctl_global_interrupts_un_re_enable_from_isr**

These are defined in **ctl_api.h** to use their libarm equivalents. Rather than use these functions, it is recommended to re-enable interrupts in the **irq_handler**.

Removed support for programmable interrupt handler

The following functions have been removed from CTL V2:

- **ctl_set_isr**
- **ctl_unmask_isr**
- **ctl_mask_isr**

These functions are now declared in **ctl_api.h** and are implemented in board- and CPU-support packages.

Removed CPU-specific timer functions

The following functions have been removed from CTL V2:

- **ctl_start_timer**
- **ctl_get_ticks_per_second**

These functions are now declared in **ctl_api.h** and are implemented in support packages for the board and CPU.

Removed board-specific functions

The following functions have been removed from CTL V2:

- **ctl_board_init**
- **ctl_board_set_leds**
- **ctl_board_on_button_pressed**

These functions are now declared in **ctl_api.h** and are implemented in board-support packages.

Moved libc mutex

The declaration of the event set **ctl_libc_mutex** has been moved into the implementation of the libc multi-threading helper functions.

Byte-queue and message-queue additions

Functions to post and receive multiple bytes/messages.

Functions to query the state of byte/message queues.

Function to associate events that are set when byte or message queues are not empty or not full.

Usage of ctl_global_interrupts_set

In CTL V2.1, the usage of **ctl_global_interrupts_set** has been replaced with usage of **ctl_global_interrupts_disable** and **ctl_global_interrupts_enable**. These functions are now implemented using compiler intrinsics in the default CTL build. You can rebuild with the C preprocessor symbol **__NO_USE_INTRINSICS__** defined if your application requires **ctl_global_interrupts_set** to be used.

Complete API reference

This section contains a complete reference to the CrossWorks Tasking Library (CTL) API.

<ctl.h>

API Summary

Tasks	
ctl_task_die	Terminate the executing task
ctl_task_init	Create the initial task
ctl_task_remove	Remove a task from the task list
ctl_task_reschedule	Cause a reschedule
ctl_task_restore	Put back a task on to the task list
ctl_task_run	Start a task
ctl_task_set_priority	Set the priority of a task
Timer	
ctl_get_current_time	Atomically return the current time
ctl_increment_tick_from_isr	Increment tick timer
ctl_timeout_wait	Wait until timeout has occurred
Event sets	
ctl_events_init	Initialize an event set
ctl_events_pulse	Pulse events in an event set
ctl_events_set_clear	Set and clear events in an event set
ctl_events_wait	Wait for events in an event set
Semaphores	
ctl_semaphore_init	Initialize a semaphore
ctl_semaphore_signal	Signal a semaphore
ctl_semaphore_wait	Wait for a semaphore
Message queues	
ctl_message_queue_init	Initialize a message queue
ctl_message_queue_num_free	Return number of free elements in a message queue
ctl_message_queue_num_used	Return number of used elements in a message queue
ctl_message_queue_post	Post message to a message queue
ctl_message_queue_post_multi	Post messages to a message queue
ctl_message_queue_post_multi_nb	Post messages to a message queue without blocking
ctl_message_queue_post_nb	Post message to a message queue without blocking
ctl_message_queue_receive	Receive message from a message queue
ctl_message_queue_receive_multi	Receive messages from a message queue

ctl_message_queue_receive_multi_nb	Receive messages from a message queue without blocking
ctl_message_queue_receive_nb	Receive message from a message queue without blocking
ctl_message_queue_setup_events	Associate events with the not-full and not-empty state of a message queue
Bytes queues	
ctl_byte_queue_init	Initialize a byte queue
ctl_byte_queue_num_free	Return number of free bytes in a byte queue
ctl_byte_queue_num_used	Return number of used bytes in a byte queue
ctl_byte_queue_post	Post byte to a byte queue
ctl_byte_queue_post_multi	Post bytes to a byte queue
ctl_byte_queue_post_multi_nb	Post bytes to a byte queue without blocking
ctl_byte_queue_post_nb	Post byte to a byte queue without blocking
ctl_byte_queue_receive	Receive a byte from a byte queue
ctl_byte_queue_receive_multi	Receive multiple bytes from a byte queue
ctl_byte_queue_receive_multi_nb	Receive multiple bytes from a byte queue without blocking
ctl_byte_queue_receive_nb	Receive a byte from a byte queue without blocking
ctl_byte_queue_setup_events	Associate events with the not-full and not-empty state of a byte queue
Mutexes	
ctl_mutex_init	Initialize a mutex
ctl_mutex_lock	Lock a mutex
ctl_mutex_unlock	Unlock a mutex
Interrupts	
ctl_global_interrupts_disable	Disable global interrupts
ctl_global_interrupts_enable	Enable global interrupts
ctl_global_interrupts_set	Enable/disable interrupts
Error handling	
ctl_handle_error	Handle a CTL error condition
Memory areas	
ctl_memory_area_allocate	Allocate a block from a memory area
ctl_memory_area_free	Free a memory area block
ctl_memory_area_init	Initialize a memory area
ctl_memory_area_setup_events	Set memory area events

Types	
CTL_BYTE_QUEUE_t	Byte queue struct definition
CTL_ERROR_CODE_t	Error cause
CTL_EVENT_SET_t	Event set definition
CTL_EVENT_WAIT_TYPE_t	Event set wait types
CTL_MEMORY_AREA_t	Memory area struct definition
CTL_MESSAGE_QUEUE_t	Message queue struct definition
CTL_MUTEX_t	Mutex struct definition
CTL_SEMAPHORE_t	Semaphore definition
CTL_STATE_t	Task states
CTL_TASK_t	Task struct definition
CTL_TIMEOUT_t	Type of wait
CTL_TIME_t	Time definition
System state variables	
ctl_current_time	The current time in ticks
ctl_interrupt_count	Nested interrupt count
ctl_last_schedule_time	The time (in ticks) of the last task schedule
ctl_reschedule_on_last_isr_exit	Reschedule is required on last ISR exit
ctl_task_executing	The task that is currently executing
ctl_task_list	List of tasks sorted by priority
ctl_task_switch_callout	A function pointer called on a task switch
ctl_time_increment	Current time tick increment
ctl_timeslice_period	Time slice period in ticks

CTL_BYTE_QUEUE_t

Synopsis

```
typedef struct {
  unsigned char *q;
  unsigned s;
  unsigned front;
  unsigned n;
  CTL_EVENT_SET_t *e;
  CTL_EVENT_SET_t notempty;
  CTL_EVENT_SET_t notfull;
} CTL_BYTE_QUEUE_t;
```

Description

CTL_BYTE_QUEUE_t defines the byte queue structure. The byte queue structure contains:

Member	Description
q	pointer to the array of bytes
s	size of the array of bytes
front	the next byte to leave the byte queue
n	the number of elements in the byte queue
e	the event set to use for the not empty and not full events
notempty	the event number for a not empty event
notfull	the event number for a not full event

CTL_ERROR_CODE_t

Synopsis

```
typedef enum {
  CTL_ERROR_NO_TASKS_TO_RUN,
  CTL_UNSUPPORTED_CALL_FROM_ISR,
  CTL_MUTEX_UNLOCK_CALL_ERROR,
  CTL_UNSPECIFIED_ERROR
} CTL_ERROR_CODE_t;
```

Description

CTL_ERROR_CODE_t defines the set of errors that are detected by the CrossWorks tasking library; the errors are reported by a call to **ctl_handle_error**.

Constant	Description
CTL_ERROR_NO_TASKS_TO_RUN	A reschedule has occurred but there are no tasks which are runnable.
CTL_UNSUPPORTED_CALL_FROM_ISR	An interrupt service routine has called a tasking library function that could block or is otherwise unsupported when called from inside an interrupt service routine.
CTL_MUTEX_UNLOCK_CALL_ERROR	A task called **ctl_mutex_unlock** passing a mutex which it has not locked, or which a different task holds a lock on. Only the task that successfully acquired a lock on a mutex can unlock that mutex.
CTL_UNSPECIFIED_ERROR	An unspecified error has occurred.

CTL_EVENT_SET_t

Synopsis

```
typedef unsigned CTL_EVENT_SET_t;
```

Description

CTL_EVENT_SET_t defines an event set. Event sets are word sized 16 or 32 depending on the machine.

CTL_EVENT_WAIT_TYPE_t

Synopsis

```
typedef enum {
  CTL_EVENT_WAIT_ANY_EVENTS,
  CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR,
  CTL_EVENT_WAIT_ALL_EVENTS,
  CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR
} CTL_EVENT_WAIT_TYPE_t;
```

Description

CTL_EVENT_WAIT_TYPE_t defines how to wait for an event set.

Constant	Description
CTL_EVENT_WAIT_ANY_EVENTS	Wait for any of the specified events to be set in the event set.
CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR	Wait for any of the specified events to be set in the event set and reset (clear) them.
CTL_EVENT_WAIT_ALL_EVENTS	Wait for all of the specified events to be set in the event set.
CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR	Wait for all of the specified events to be set in the event set and reset (clear) them.

See Also

ctl_events_wait

CTL_MEMORY_AREA_t

Synopsis

```
typedef struct {
  unsigned *head;
  CTL_EVENT_SET_t *e;
  CTL_EVENT_SET_t blockavailable;
} CTL_MEMORY_AREA_t;
```

Description

CTL_MEMORY_AREA_t defines the memory area structure. The memory area structure contains:

Member	Description
head	the next free memory block
e	the event set containing the blockavailable event
blockavailable	the blockavailable event

CTL_MESSAGE_QUEUE_t

Synopsis

```
typedef struct {
  void ** q;
  unsigned s;
  unsigned front;
  unsigned n;
  CTL_EVENT_SET_t *e;
  CTL_EVENT_SET_t notempty;
  CTL_EVENT_SET_t notfull;
} CTL_MESSAGE_QUEUE_t;
```

Description

CTL_MESSAGE_QUEUE_t defines the message queue structure. The message queue structure contains:

Member	Description
q	pointer to the array of message queue objects
s	size of the array of message queue objects
front	the next element to leave the message queue
n	the number of elements in the message queue
e	the event set to use for the not empty and not full events
notempty	the event number for a not empty event
notfull	the event number for a not full event

CTL_MUTEX_t

Synopsis

```
typedef struct {
  unsigned lock_count;
  CTL_TASK_t *locking_task;
  unsigned locking_task_priority;
} CTL_MUTEX_t;
```

Description

CTL_MUTEX_t defines the mutex structure. The mutex structure contains:

Member	Description
lock_count	number of times the mutex has been locked
locking_task	the task that has locked the mutex
locking_task_priority	the priority of the task at the time it locked the mutex

CTL_SEMAPHORE_t

Synopsis

```
typedef unsigned CTL_SEMAPHORE_t;
```

Description

CTL_SEMAPHORE_t defines the semaphore type. Semaphores are held in one word, 16 or 32 bits depending on the machine.

CTL_STATE_t

Synopsis

```
typedef enum {
  CTL_STATE_RUNNABLE,
  CTL_STATE_TIMER_WAIT,
  CTL_STATE_EVENT_WAIT_ALL,
  CTL_STATE_EVENT_WAIT_ALL_AC,
  CTL_STATE_EVENT_WAIT_ANY,
  CTL_STATE_EVENT_WAIT_ANY_AC,
  CTL_STATE_SEMAPHORE_WAIT,
  CTL_STATE_MESSAGE_QUEUE_POST_WAIT,
  CTL_STATE_MESSAGE_QUEUE_RECEIVE_WAIT,
  CTL_STATE_MUTEX_WAIT,
  CTL_STATE_SUSPENDED
} CTL_STATE_t;
```

Description

CTL_STATE_t defines the states the task can be on.

Constant	Description
CTL_STATE_RUNNABLE	Task can run.
CTL_STATE_TIMER_WAIT	Waiting for a time value.
CTL_STATE_EVENT_WAIT_ALL	Waiting for all events to be set.
CTL_STATE_EVENT_WAIT_ALL_AC	Waiting for all events to be set with auto clear.
CTL_STATE_EVENT_WAIT_ANY	Waiting for any events to be set.
CTL_STATE_EVENT_WAIT_ANY_AC	Waiting for any events to be set with auto clear.
CTL_STATE_SEMAPHORE_WAIT	Task is waiting for a semaphore.
CTL_STATE_MESSAGE_QUEUE_POST_WAIT	Task is waiting to post to a message queue.
CTL_STATE_MESSAGE_QUEUE_RECEIVE_WAIT	Task is waiting to receive from a message queue.
CTL_STATE_MUTEX_WAIT	Task is waiting for a mutex.
CTL_STATE_SUSPENDED	Task cannot run.

CTL_TASK_t

Synopsis

```
typedef struct {
  unsigned *stack_pointer;
  unsigned char priority;
  unsigned char state;
  unsigned char timeout_occured;
  CTL_TIME_t timeout;
  void *wait_object;
  CTL_EVENT_SET_t wait_events;
  int thread_errno;
  void *data;
  CTL_TIME_t execution_time;
  unsigned *stack_start;
  char *name;
} CTL_TASK_t;
```

Description

CTL_TASK_t defines the task structure. The task structure contains:

Member	Description
stack_pointer	the saved register state of the task when it is not scheduled
priority	the priority of the task
state	the state of task CTL_STATE_RUNNABLE or (CTL_STATE_*_WAIT_* | CTL_STATE_TIMER_WAIT) or CTL_STATE_SUSPENDED
timeout_occured	1 if a wait timed out otherwise 0 - when state is CTL_RUNNABLE
next	next pointer for wait queue
timeout	wait timeout value or time slice value when the task is executing
wait_object	the event set, semaphore, message queue or mutex to wait on
wait_events	the events to wait for
thread_errno	thread specific errno
data	task specific data pointer
execution_time	number of ticks the task has executed for
stack_start	the start (lowest address) of the stack

name	task name

CTL_TIMEOUT_t

Synopsis

```
typedef enum {
  CTL_TIMEOUT_NONE,
  CTL_TIMEOUT_INFINITE,
  CTL_TIMEOUT_ABSOLUTE,
  CTL_TIMEOUT_DELAY,
  CTL_TIMEOUT_NOW
} CTL_TIMEOUT_t;
```

Description

CTL_TIMEOUT_t defines the type of timeout for a blocking function call.

Constant	Description
CTL_TIMEOUT_NONE	No timeout — block indefinitely.
CTL_TIMEOUT_INFINITE	Identical to **CTL_TIMEOUT_NONE**.
CTL_TIMEOUT_ABSOLUTE	The timeout is an absolute time.
CTL_TIMEOUT_DELAY	The timeout is relative to the current time.
CTL_TIMEOUT_NOW	The timeout happens immediately — no rescheduling occurs.

CTL_TIME_t

Synopsis

```
typedef unsigned long CTL_TIME_t;
```

Description

CTL_TIME_t defines the base type for times that CTL uses.

ctl_byte_queue_init

Synopsis

```
void ctl_byte_queue_init(CTL_BYTE_QUEUE_t *q,
                         unsigned char *queue,
                         unsigned queue_size);
```

Description

ctl_byte_queue_init is given a pointer to the byte queue to initialize in **q**. The array that will be used to implement the byte queue pointed to by **queue** and its size in **queue_size** are also supplied.

ctl_byte_queue_num_free

Synopsis

```
unsigned ctl_byte_queue_num_free(CTL_BYTE_QUEUE_t *q);
```

Description

ctl_byte_queue_num_free returns the number of free bytes in the byte queue **q**.

ctl_byte_queue_num_used

Synopsis

```
unsigned ctl_byte_queue_num_used(CTL_BYTE_QUEUE_t *q);
```

Description

ctl_byte_queue_num_used returns the number of used elements in the byte queue **q**.

ctl_byte_queue_post

Synopsis

```
unsigned ctl_byte_queue_post(CTL_BYTE_QUEUE_t *q,
                             unsigned char b,
                             CTL_TIMEOUT_t t,
                             CTL_TIME_t timeout);
```

Description

ctl_byte_queue_post posts **b** to the byte queue pointed to by **q**. If the byte queue is full then the caller will block until the byte can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value. **ctl_byte_queue_post** returns zero if the timeout occurred otherwise it returns one.

Note

ctl_byte_queue_post must not be called from an interrupt service routine.

ctl_byte_queue_post_multi

Synopsis

```
unsigned ctl_byte_queue_post_multi(CTL_BYTE_QUEUE_t *q,
                                   unsigned n,
                                   unsigned char *b,
                                   CTL_TIMEOUT_t t,
                                   CTL_TIME_t timeout);
```

Description

ctl_byte_queue_post_multi posts **n** bytes to the byte queue pointed to by **q**. The caller will block until the bytes can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value. **ctl_byte_queue_post_multi** returns the number of bytes that were posted.

Note

ctl_byte_queue_post_multi must not be called from an interrupt service routine.

ctl_byte_queue_post_multi does not guarantee that the bytes will be all be posted to the byte queue atomically. If you have multiple tasks posting (multiple bytes) to the same byte queue then you may get unexpected results.

ctl_byte_queue_post_multi_nb

Synopsis

```
unsigned ctl_byte_queue_post_multi_nb(CTL_BYTE_QUEUE_t *q,
                                      unsigned n,
                                      unsigned char *b);
```

Description

ctl_byte_queue_post_multi_nb posts **n** bytes to the byte queue pointed to by **q**.

ctl_byte_queue_post_multi_nb returns the number of bytes that were posted.

ctl_byte_queue_post_nb

Synopsis

```
unsigned ctl_byte_queue_post_nb(CTL_BYTE_QUEUE_t *q,
                                unsigned char b);
```

Description

ctl_byte_queue_post_nb posts **b** to the byte queue pointed to by **q**. If the byte queue is full then the function will return zero otherwise it will return one.

ctl_byte_queue_receive

Synopsis

```
unsigned ctl_byte_queue_receive(CTL_BYTE_QUEUE_t *q,
                                unsigned char *b,
                                CTL_TIMEOUT_t t,
                                CTL_TIME_t timeout);
```

Description

ctl_byte_queue_receive pops the oldest byte in the byte queue pointed to by **q** into the memory pointed to by **b**. **ctl_byte_queue_receive** will block if no bytes are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_byte_queue_receive returns zero if a timeout occurs otherwise 1.

Note

ctl_byte_queue_receive must not be called from an interrupt service routine.

ctl_byte_queue_receive_multi

Synopsis

```
unsigned ctl_byte_queue_receive_multi(CTL_BYTE_QUEUE_t *q,
                                      unsigned n,
                                      unsigned char *b,
                                      CTL_TIMEOUT_t t,
                                      CTL_TIME_t timeout);
```

Description

ctl_byte_queue_receive_multi pops the oldest **n** bytes in the byte queue pointed to by **q** into the memory pointed at by **b**. **ctl_byte_queue_receive_multi** will block until the number of bytes are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_byte_queue_receive_multi returns the number of bytes that have been received.

Note

ctl_byte_queue_receive_multi must not be called from an interrupt service routine.

ctl_byte_queue_receive_multi_nb

Synopsis

```
unsigned ctl_byte_queue_receive_multi_nb(CTL_BYTE_QUEUE_t *q,
                                         unsigned n,
                                         unsigned char *b);
```

Description

ctl_byte_queue_receive_multi_nb pops the oldest **n** bytes in the byte queue pointed to by **q** into the memory pointed to by **b**. **ctl_byte_queue_receive_multi_nb** returns the number of bytes that have been received.

ctl_byte_queue_receive_nb

Synopsis

```
unsigned ctl_byte_queue_receive_nb(CTL_BYTE_QUEUE_t *q,
                                   unsigned char *b);
```

Description

ctl_byte_queue_receive_nb pops the oldest byte in the byte queue pointed to by **m** into the memory pointed to by **b**. If no bytes are available the function returns zero otherwise it returns 1.

ctl_byte_queue_setup_events

Synopsis

```
void ctl_byte_queue_setup_events(CTL_BYTE_QUEUE_t *q,
                                 CTL_EVENT_SET_t *e,
                                 CTL_EVENT_SET_t notempty,
                                 CTL_EVENT_SET_t notfull);
```

Description

ctl_byte_queue_setup_events registers events in the event set **e** that are set when the byte queue **q** becomes **notempty** or becomes **notfull**. No scheduling will occur with this operation, you need to do this before waiting for events.

ctl_current_time

Synopsis

```
CTL_TIME_t ctl_current_time;
```

Description

ctl_current_time holds the current time in ticks. **ctl_current_time** is incremented by **ctl_increment_ticks_from_isr**.

Note

For portable programs without race conditions you should not read this variable directly, you should use **ctl_get_current_time** instead. As this variable is changed by an interrupt, it cannot be read atomically on processors whose word size is less than 32 bits without first disabling interrupts. That said, you can read this variable directly in your interrupt handler as long as interrupts are still disabled.

ctl_events_init

Synopsis

```
void ctl_events_init(CTL_EVENT_SET_t *e,
                     CTL_EVENT_SET_t set);
```

Description

ctl_events_init initializes the event set **e** with the **set** values.

ctl_events_pulse

Synopsis

```
void ctl_events_pulse(CTL_EVENT_SET_t *e,
                      CTL_EVENT_SET_t set_then_clear);
```

Description

ctl_events_pulse will set the events defined by **set_then_clear** in the event set pointed to by **e**.

ctl_events_pulse will then search the task list, matching tasks that are waiting on the event set **e**, and make them runnable if the match is successful. The events defined by **set_then_clear** are then cleared.

See Also

ctl_events_set_clear.

ctl_events_set_clear

Synopsis

```
void ctl_events_set_clear(CTL_EVENT_SET_t *e,
                          CTL_EVENT_SET_t set,
                          CTL_EVENT_SET_t clear);
```

Description

ctl_events_set_clear sets the events defined by **set** and clears the events defined by **clear** of the event set pointed to by **e**. **ctl_events_set_clear** will then search the task list, matching tasks that are waiting on the event set **e** and make them runnable if the match is successful.

See Also

ctl_events_pulse.

ctl_events_wait

Synopsis

```
unsigned ctl_events_wait(CTL_EVENT_WAIT_TYPE_t type,
                         CTL_EVENT_SET_t *eventSet,
                         CTL_EVENT_SET_t events,
                         CTL_TIMEOUT_t t,
                         CTL_TIME_t timeout);
```

Description

ctl_events_wait waits for **events** to be set (value 1) in the event set pointed to by **eventSet** with an optional **timeout** applied if **timeoutType** is non-zero.

The **waitType** can be one of:

- **CTL_EVENT_WAIT_ANY_EVENTS** — wait for any of **events** in **eventSet** to be set.
- **CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR** — wait for any of **events** in **eventSet** to be set and reset (clear) them.
- **CTL_EVENT_WAIT_ALL_EVENTS** — wait for all **events** in ***eventSet** to be set.
- **CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR** — wait for all **events** in **eventSet** to be set and reset (clear) them.

ctl_events_wait returns the value pointed to by **eventSet** before any auto-clearing occurred or zero if the **timeout** occurred.

Note

ctl_events_wait must not be called from an interrupt service routine.

ctl_get_current_time

Synopsis

```
CTL_TIME_t ctl_get_current_time(void);
```

Description

ctl_get_current_time atomically reads the value of **ctl_current_time**.

ctl_global_interrupts_disable

Synopsis

```
int ctl_global_interrupts_disable(void);
```

Description

ctl_global_interrupts_disable disables global interrupts. If ctl_global_interrupts_disable is called and interrupts are already disabled then it will return 0. If ctl_global_interrupts_disable is called and interrupts are enabled then it will return non-zero which may or may not represent the true interrupt disabled state. ctl_global_interrupts_disable is used to provide exclusive access to CTL data structures the implementation of it may or may not disable global interrupts.

ctl_global_interrupts_enable

Synopsis

```
void ctl_global_interrupts_enable(void);
```

Description

ctl_global_interrupts_enable enables global interrupts. ctl_global_interrupts_enable is used to provide exclusive access to CTL data structures the implementation of it may or may not disable global interrupts.

ctl_global_interrupts_set

Synopsis

```
int ctl_global_interrupts_set(int enable);
```

Description

ctl_global_interrupts_set disables or enables global interrupts according to the state **enable**. If **enable** is zero, interrupts are disabled and if **enable** is non-zero, interrupts are enabled. If ctl_global_interrupts_set is called and interrupts are already disabled then it will return 0. If ctl_global_interrupts_set is called and interrupts are enabled then it will return non-zero which may or may not represent the true interrupt disabled state. ctl_global_interrupts_set is used to provide exclusive access to CTL data structures the implementation of it may or may not disable global interrupts.

ctl_handle_error

Synopsis

```
void ctl_handle_error(CTL_ERROR_CODE_t e);
```

Description

ctl_handle_error is a function that you must supply in your application that handles errors detected by the CrossWorks tasking library.

The errors that can be reported in **e** are are described in CTL_ERROR_CODE_t.

ctl_increment_tick_from_isr

Synopsis

```
void ctl_increment_tick_from_isr(void);
```

Description

ctl_increment_tick_from_isr increments **ctl_current_time** by the number held in **ctl_time_increment** and does rescheduling. This function should be called from a periodic interrupt service routine.

Note

ctl_increment_tick_from_isr must only be invoked by an interrupt service routine.

ctl_interrupt_count

Synopsis

```
unsigned char ctl_interrupt_count;
```

Description

ctl_interrupt_count contains a count of the interrupt nesting level. This variable must be incremented immediately on entry to an interrupt service routine and decremented immediately before return from the interrupt service routine.

ctl_last_schedule_time

Synopsis

```
CTL_TIME_t ctl_last_schedule_time;
```

Description

ctl_last_schedule_time contains the time (in ticks) of the last task schedule.

Description

ctl_last_schedule_time contains the time of the last reschedule in ticks.

ctl_memory_area_allocate

Synopsis

```
unsigned *ctl_memory_area_allocate(CTL_MEMORY_AREA_t *memory_area);
```

Description

ctl_memory_area_allocate allocates a block from the initialized memory area **memory_area**. **ctl_memory_area_allocate** returns a block of the size specified in the call to ctl_memory_area_init or zero if no blocks are available.

ctl_memory_area_allocate executes in constant time and is very fast. You can call **ctl_memory_area_allocate** from an interrupt service routine, from a task, or from initialization code.

ctl_memory_area_free

Synopsis

```
void ctl_memory_area_free(CTL_MEMORY_AREA_t *memory_area,
                          unsigned *block);
```

Description

ctl_memory_area_free is given a pointer to a memory area **memory_area** which has been initialized and a **block** that has been returned by **ctl_memory_area_allocate**. The block is returned to the memory area so that it can be allocated again.

ctl_memory_area_init

Synopsis

```
void ctl_memory_area_init(CTL_MEMORY_AREA_t *memory_area,
                          unsigned *memory,
                          unsigned block_size_in_words,
                          unsigned num_blocks);
```

Description

ctl_memory_area_init is given a pointer to the memory area to initialize in **memory_area**. The array that is used to implement the memory area is pointed to by **memory**. The size of a memory block is given supplied in **block_size_in_words** and the number of block is supplied in **num_blocks**.

Note

memory must point to a block of memory that is at least **block_size_in_words** × **num_blocks** words long.

ctl_memory_area_setup_events

Synopsis

```
void ctl_memory_area_setup_events(CTL_MEMORY_AREA_t *m,
                                  CTL_EVENT_SET_t *e,
                                  CTL_EVENT_SET_t blockavailable);
```

Description

ctl_memory_area_setup_events registers the events **blockavailable** in the event set **e** that are set when a block becomes available in the the memory area **m**.

ctl_message_queue_init

Synopsis

```
void ctl_message_queue_init(CTL_MESSAGE_QUEUE_t *q,
                            void **queue,
                            unsigned queue_size);
```

Description

ctl_message_queue_init is given a pointer to the message queue to initialize in **q**. The array that will be used to implement the message queue pointed to by **queue** and its size in **queue_size** are also supplied.

ctl_message_queue_num_free

Synopsis

```
unsigned ctl_message_queue_num_free(CTL_MESSAGE_QUEUE_t *q);
```

Description

ctl_message_queue_num_free returns the number of free elements in the message queue **q**.

ctl_message_queue_num_used

Synopsis

```
unsigned ctl_message_queue_num_used(CTL_MESSAGE_QUEUE_t *q);
```

Description

ctl_message_queue_num_used returns the number of used elements in the message queue **q**.

ctl_message_queue_post

Synopsis

```
unsigned ctl_message_queue_post(CTL_MESSAGE_QUEUE_t *q,
                                void *message,
                                CTL_TIMEOUT_t t,
                                CTL_TIME_t timeout);
```

Description

ctl_message_queue_post posts **message** to the message queue pointed to by **q**. If the message queue is full then the caller will block until the message can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value. **ctl_message_queue_post** returns zero if the timeout occurred otherwise it returns one.

Note

ctl_message_queue_post must not be called from an interrupt service routine.

ctl_message_queue_post_multi

Synopsis

```
unsigned ctl_message_queue_post_multi(CTL_MESSAGE_QUEUE_t *q,
                                      unsigned n,
                                      void **messages,
                                      CTL_TIMEOUT_t t,
                                      CTL_TIME_t timeout);
```

Description

ctl_message_queue_post_multi posts **n messages** to the message queue pointed to by **q**. The caller will block until the messages can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value. **ctl_message_queue_post_multi** returns the number of messages that were posted.

Note

ctl_message_queue_post_multi must not be called from an interrupt service routine.

ctl_message_queue_post_multi function does not guarantee that the messages will be all be posted to the message queue atomically. If you have multiple tasks posting (multiple messages) to the same message queue then you may get unexpected results.

ctl_message_queue_post_multi_nb

Synopsis

```
unsigned ctl_message_queue_post_multi_nb(CTL_MESSAGE_QUEUE_t *q,
                                         unsigned n,
                                         void **messages);
```

Description

ctl_message_queue_post_multi_nb posts **n messages** to the message queue pointed to by **m**.

ctl_message_queue_post_multi_nb returns the number of messages that were posted.

ctl_message_queue_post_nb

Synopsis

```
unsigned ctl_message_queue_post_nb(CTL_MESSAGE_QUEUE_t *q,
                                   void *message);
```

Description

ctl_message_queue_post_nb posts **message** to the message queue pointed to by **q**. If the message queue is full then the function will return zero otherwise it will return one.

ctl_message_queue_receive

Synopsis

```
unsigned ctl_message_queue_receive(CTL_MESSAGE_QUEUE_t *q,
                                   void **message,
                                   CTL_TIMEOUT_t t,
                                   CTL_TIME_t timeout);
```

Description

ctl_message_queue_receive pops the oldest message in the message queue pointed to by **q** into the memory pointed to by **message**. **ctl_message_queue_receive** will block if no messages are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_message_queue_receive returns zero if a timeout occurs otherwise 1.

Note

ctl_message_queue_receive must not be called from an interrupt service routine.

ctl_message_queue_receive_multi

Synopsis

```
unsigned ctl_message_queue_receive_multi(CTL_MESSAGE_QUEUE_t *q,
                                         unsigned n,
                                         void **messages,
                                         CTL_TIMEOUT_t t,
                                         CTL_TIME_t timeout);
```

Description

ctl_message_queue_receive_multi pops the oldest **n** messages in the message queue pointed to by **q** into the memory pointed to by **message**. **ctl_message_queue_receive_multi** will block until all the messages are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_message_queue_receive_multi returns the number of messages that were received.

Note

ctl_message_queue_receive_multi must not be called from an interrupt service routine.

ctl_message_queue_receive_multi_nb

Synopsis

```
unsigned ctl_message_queue_receive_multi_nb(CTL_MESSAGE_QUEUE_t *q,
                                            unsigned n,
                                            void **messages);
```

Description

ctl_message_queue_receive_multi_nb pops the oldest **n** messages in the message queue pointed to by **q** into the memory pointed to by **message**.

ctl_message_queue_receive_multi_nb returns the number of messages that were received.

ctl_message_queue_receive_nb

Synopsis

```
unsigned ctl_message_queue_receive_nb(CTL_MESSAGE_QUEUE_t *q,
                                      void **message);
```

Description

ctl_message_queue_receive_nb pops the oldest message in the message queue pointed to by **q** into the memory pointed to by **message**. If no messages are available the function returns zero otherwise it returns 1.

ctl_message_queue_setup_events

Synopsis

```
void ctl_message_queue_setup_events(CTL_MESSAGE_QUEUE_t *q,
                                    CTL_EVENT_SET_t *e,
                                    CTL_EVENT_SET_t notempty,
                                    CTL_EVENT_SET_t notfull);
```

Description

ctl_message_queue_setup_events registers events in the event set **e** that are set when the message queue **q** becomes **notempty** or becomes **notfull**. No scheduling will occur with this operation, you need to do this before waiting for events.

ctl_mutex_init

Synopsis

```
void ctl_mutex_init(CTL_MUTEX_t *m);
```

Description

ctl_mutex_init initializes the mutex pointed to by **m**.

ctl_mutex_lock

Synopsis

```
unsigned ctl_mutex_lock(CTL_MUTEX_t *m,
                        CTL_TIMEOUT_t t,
                        CTL_TIME_t timeout);
```

Description

ctl_mutex_lock locks the mutex pointed to by **m** to the calling task. If the mutex is already locked by the calling task then the mutex lock count is incremented. If the mutex is already locked by a different task then the caller will block until the mutex is unlocked. In this case, if the priority of the task that has locked the mutex is less than that of the caller the priority of the task that has locked the mutex is raised to that of the caller whilst the mutex is locked. If **timeoutType** is non-zero and the current time reaches the **timeout** value before the lock is acquired the function returns zero otherwise it returns one.

Note

ctl_mutex_lock must not be called from an interrupt service routine.

ctl_mutex_unlock

Synopsis

```
void ctl_mutex_unlock(CTL_MUTEX_t *m);
```

Description

ctl_mutex_unlock unlocks the mutex pointed to by **m**. The mutex must have previously been locked by the calling task. If the calling task's priority has been raised (by another task calling **ctl_mutex_unlock** whilst the mutex was locked), then the calling tasks priority will be restored.

Note

ctl_mutex_unlock must not be called from an interrupt service routine.

ctl_reschedule_on_last_isr_exit

Synopsis

```
unsigned char ctl_reschedule_on_last_isr_exit;
```

Description

ctl_reschedule_on_last_isr_exit is set to a non-zero value if a CTL call is made from an interrupt service routine that requires a task reschedule. This variable is checked and reset on exit from the last nested interrupt service routine.

ctl_semaphore_init

Synopsis

```
void ctl_semaphore_init(CTL_SEMAPHORE_t *s,
                        unsigned value);
```

Description

ctl_semaphore_init initializes the semaphore pointed to by **s** to **value**.

ctl_semaphore_signal

Synopsis

```
void ctl_semaphore_signal(CTL_SEMAPHORE_t *s);
```

Description

ctl_semaphore_signal signals the semaphore pointed to by **s**. If tasks are waiting for the semaphore then the highest priority task will be made runnable. If no tasks are waiting for the semaphore then the semaphore value will be incremented.

ctl_semaphore_wait

Synopsis

```
unsigned ctl_semaphore_wait(CTL_SEMAPHORE_t *s,
                            CTL_TIMEOUT_t t,
                            CTL_TIME_t timeout);
```

Description

ctl_semaphore_wait waits for the semaphore pointed to by **s** to be non-zero. If the semaphore is zero then the caller will block unless **timeoutType** is non-zero and the current time reaches the **timeout** value. If the timeout occurred **ctl_semaphore_wait** returns zero otherwise it returns one.

Note

ctl_semaphore_wait must not be called from an interrupt service routine.

ctl_task_die

Synopsis

```
void ctl_task_die(void);
```

Description

ctl_task_die terminates the currently executing task and schedules the next runnable task.

ctl_task_executing

Synopsis

```
CTL_TASK_t *ctl_task_executing;
```

Description

ctl_task_executing points to the **CTL_TASK_t** structure of the currently executing task. The **priority** field is the only field in the **CTL_TASK_t** structure that is defined for the task that is executing. It is an error if **ctl_task_executing** is **NULL**.

ctl_task_init

Synopsis

```
void ctl_task_init(CTL_TASK_t *task,
                   unsigned char priority,
                   const char *name);
```

Description

ctl_task_init turns the main program into a task. This function takes a pointer in **task** to the **CTL_TASK_t** structure that represents the main task, its **priority** (0 is the lowest priority, 255 the highest), and a zero-terminated string pointed by **name**. On return from this function global interrupts will be enabled.

The function must be called before any other CrossWorks tasking library calls are made.

ctl_task_list

Synopsis

```
CTL_TASK_t *ctl_task_list;
```

Description

ctl_task_list points to the **CTL_TASK_t** structure of the highest priority task that is not executing. It is an error if **ctl_task_list** is **NULL**.

ctl_task_remove

Synopsis

```
void ctl_task_remove(CTL_TASK_t *task);
```

Description

ctl_task_remove removes the task **task** from the waiting task list. Once you you have removed a task the only way to re-introduce it to the system is to call **ctl_task_restore**.

ctl_task_reschedule

Synopsis

```
void ctl_task_reschedule(void);
```

Description

ctl_task_reschedule causes a reschedule to occur. This can be used by tasks of the same priority to share the CPU without using timeslicing.

ctl_task_restore

Synopsis

```
void ctl_task_restore(CTL_TASK_t *task);
```

Description

ctl_task_restore adds a task **task** that was removed (using **ctl_task_remove**) onto the task list and do scheduling.

ctl_task_run

Synopsis

```
void ctl_task_run(CTL_TASK_t *task,
                  unsigned char priority,
                  void (*entrypoint)(void *),
                  void *parameter,
                  const char *name,
                  unsigned stack_size_in_words,
                  unsigned *stack,
                  unsigned call_size_in_words);
```

Description

ctl_task_run takes a pointer in **task** to the **CTL_TASK_t** structure that represents the task. The **priority** can be zero for the lowest priority up to 255 which is the highest. The **entrypoint** parameter is the function that the task will execute which has the **parameter** passed to it.

name is a pointer to a zero-terminated string used for debug purposes.

The start of the memory used to hold the stack that the task will execute in is **stack** and the size of the memory is supplied in **stack_size_in_words**. On systems that have two stacks (e.g. Atmel AVR) then the **call_size_in_words** parameter must be set to specify the number of stack elements to use for the call stack.

ctl_task_set_priority

Synopsis

```
unsigned char ctl_task_set_priority(CTL_TASK_t *task,
                                    unsigned char priority);
```

Description

ctl_task_set_priority changes the priority of **task** to **priority**. The priority can be 0, the lowest priority, to 255, which is the highest priority.

You can change the priority of the currently executing task by passing **ctl_task_executing** as the **task** parameter. **ctl_task_set_priority** returns the previous priority of the task.

ctl_task_switch_callout

Synopsis

```
void (*ctl_task_switch_callout)(CTL_TASK_t *);
```

Description

ctl_task_switch_callout contains a pointer to a function that is called (if it is set) when a task schedule occurs. The task that will be scheduled is supplied as a parameter to the function (**ctl_task_executing** will point to the currently scheduled task).

Note that the callout function is called from the CTL scheduler and as such any use of CTL services whilst executing the callout function has undefined behavior.

Note

Because this function pointer is called in an interrupt service routine, you should assign it before interrupts are started or with interrupts turned off.

ctl_time_increment

Synopsis

```
unsigned ctl_time_increment;
```

Description

ctl_time_increment contains the value that **ctl_current_time** is incremented when **ctl_increment_tick_from_isr** is called.

ctl_timeout_wait

Synopsis

```
void ctl_timeout_wait(CTL_TIME_t timeout);
```

Description

ctl_timeout_wait takes the **timeout** (not the duration) as a parameter and suspends the calling task until the current time reaches the timeout.

Note

ctl_timeout_wait must not be called from an interrupt service routine.

ctl_timeslice_period

Synopsis

```
CTL_TIME_t ctl_timeslice_period;
```

Description

ctl_timeslice_period contains the number of ticks to allow a task to run before it will be preemptively rescheduled by a task of the same priority. The variable is set to zero by default so that only higher priority tasks will be preemptively scheduled.

Compiler driver

This section describes the switches accepted by the compiler driver, **cc**. The compiler driver is capable of controlling compilation by all supported language compilers and the final link by the linker. It can also construct libraries automatically.

In contrast to many compilation and assembly language development systems, with you don't invoke the assembler or compiler directly. Instead you'll normally use the compiler driver **cc** as it provides an easy way to get files compiled, assembled, and linked. This section will introduce you to using the compiler driver to convert your source files to object files, executables, or other formats.

We recommend that you use the compiler driver rather than use the assembler or compiler directly because there the driver can assemble multiple files using one command line and can invoke the linker for you too. There is no reason why you should not invoke the assembler or compiler directly yourself, but you'll find that typing in all the required options is quite tedious-and why do that when **cc** will provide them for you automatically?

File naming conventions

The compiler driver uses file extensions to distinguish the language the source file is written in. The compiler driver recognizes the extension **.c** as C source files, **.cpp**, **.cc** or **.cxx** as C++ source files, **.s** and **.asm** as assembly code files.

The compiler driver recognizes the extension **.o** as object files, **.a** as library files, **.ld** as linker script files and **.xml** as special-purpose XML files.

We strongly recommend that you adopt these extensions for your source files and object files because you'll find that using the tools is much easier if you do.

C language files

When the compiler driver finds a file with a **.c** extension, it runs the C compiler to convert it to object code.

C++ language files

When the compiler driver finds a file with a **.cpp** extension, it runs the C++ compiler to convert it to object code.

Assembly language files

When the compiler driver finds a file with a **.s** or **.asm** extension, it runs the C preprocessor and then the assembler to convert it to object code.

Object code files

When the compiler driver finds a file with a **.o** or **.a** extension, it passes it to the linker to include it in the final application.

Command-line options

This section describes the command-line options accepted by the CrossWorks compiler driver.

-ansi (Warn about potential ANSI problems)

Syntax

-ansi

Description

Warn about potential problems that conflict with the relevant ANSI or ISO standard for the files that are compiled.

-ar (Archive output)

Syntax

-ar

Description

This switch instructs the compiler driver to archive all output files into a library. Using **-ar** implies **-c.**

Example

The following command compiles **file1.c**, **file2.asm**, and **file3.c** to object code and archives them into the library file **libfunc.a** together with the object file **file4.o**.

```
cc -ar file1.c file2.asm file3.c file4.o -o libfunc.a
```

-arch (Set ARM architecture)

Syntax

-arch = *a*

Description

Specifies the version of the instruction set to generate code for. The options are:

- **-arch=v4T** — ARM7TDMI and ARM920T
- **-arch=v5TE** — ARM9E, Feroceon and XScale
- **-arch=v6** — ARM11
- **-arch=v6M** — Cortex-M0 and Cortex-M1
- **-arch=v7A** — Cortex-A8 and Cortex-A9
- **-arch=v7M** — Cortex-M3
- **-arch=v7EM** — Cortex-M4
- **-arch=v7R** — Cortex-R4

Example

To force compilation for V7A architecture you would use:

```
cc -arch=v7A …
```

-be (Big Endian)

Syntax

-be

Description

Generate code for a big endian target.

-c (Compile to object code, do not link)

Syntax

-c

Description

All named files are compiled to object code modules, but are not linked. You can use the **-o** option to name the output if you just supply one input filename.

Example

The following command compiles **file1.c** and **file4.c** to produce the object files **file1.o** and **file4.o**.

```
cc -c file1.c file4.c
```

The following command compiles **file1.c** and produces the object file **obj/file1.o**.

```
cc -c file.c -o obj/file1.o
```

-d (Define linker symbol)

Syntax

-d *name* = *value*

Description

You can define linker symbols using the **-d** option. The symbol definitions are passed to linker.

Example

The following defines the symbol, **STACK_SIZE** with a value of 512.

```
-dSTACK_SIZE=512
```

-D (Define macro symbol)

Syntax

-D *name*
-D *name* = *value*

Description

You can define preprocessor macros using the **-D** option. The macro definitions are passed on to the respective language compiler which is responsible for interpreting the definitions and providing them to the programmer within the language.

The first form above defines the macro *name* but without an associated replacement value, and the second defines the same macro with the replacement value *value*.

Example

The following defines two macros, **SUPPORT_FLOAT** with a value of 1 and **LITTLE_ENDIAN** with no replacement value.

```
-DSUPPORT_FLOAT=1 -DLITTLE_ENDIAN
```

-e (Set entry point symbol)

Syntax

-e *name*

Description

Linker option to set the entry point symbol to be *name*. The debugger will start execution from this symbol.

-E (Preprocess)

Syntax

-E

Description

This option preprocesses the supplied file and outputs the result to the standard output.

Example

The following preprocesses the file **file.c** supplying the macros, **SUPPORT_FLOAT** with a value of 1 and **LITTLE_ENDIAN**.

```
-E -DSUPPORT_FLOAT=1 -DLITTLE_ENDIAN file.c
```

-exceptions (Enable C++ Exception Support)

Syntax

-exceptions

Description

Enables C++ exceptions to be compiled.

-fabi (Floating Point Code Generation)

Syntax

-fabi = *a*

Description

Specifies the type of floating point code generation. The options are:

- **-fabi=SoftFP** — FPU instructions are generated, CPU registers are used for floating point parameters.
- **-fabi=Hard** — FPU instructions are generated, FPU registers are used for floating point parameters.

-fpu (Set ARM FPU)

Syntax

-fpu = *a*

Description

Specifies the floating point unit to generate code for when the **fpabi** option has been supplied. The options are:

- **-fpu=VFP** — generate FPU instructions for ARM9 and ARM11
- **-fpu=VFPv3-D32** — generate FPU instructions for CortexA
- **-fpu=VFPv3-D16** — generate FPU instructions for CortexR
- **-fpu=FPv4-SP-D16** — generate FPU instructions for CortexM4

-F (Set output format)

Syntax

-F *fmt*

Description

The **-F** option instructs the compiler driver to generate an additional output file in the format *fmt*. The compiler driver supports the following formats:

- **-Fbin** — Create a .bin file
- **-Fhex** — Create a .hex file
- **-Fsrec** — Create a .srec file

The compiler driver will always output a **.elf** file as specified with the **-o** option. The name of the additional output file is the same as the **.elf** file with the file extension changed.

For example

```
cc file.c -o file.elf -Fbin
```

will generate the files **file.elf** and **file.bin**.

-g (Generate debugging information)

Syntax

-g

Description

The **-g** option instructs the compiler and assembler to generate source level debugging information for the debugger to use.

-g1 (Generate minimal debugging information)

Syntax

-g1

Description

The **-g1** option instructs the compiler to generate debugging information that enables the debugger to be able to backtrace only.

-help (Display help information)

Syntax

-help

Description

Displays a short summary of the options accepted by the compiler driver.

-io (Select I/O library implementation)

Syntax

-io= *i*

Description

This option specifies the I/O library implementation that is included in the linked image. The options are:

- **-io=d** — I/O library is implemented using debugIO e.g calls to **printf** will call **debug_printf**.
- **-io=t** — I/O library is implemented on the target, debugIO is not used.
- **-io=d+t** — I/O library is implemented on the target, debugIO is not used but debugIO is enabled.

-I (Define user include directories)

Syntax

-I *directory*

Description

In order to find include files the compiler driver arranges for the compilers to search a number of standard directories. You can add directories to the search path using the **-I** switch which is passed on to each of the language processors.

You can specify more than one include directory by separating each directory component with either a comma or semicolon.

-I- (Exclude standard include directories)

Syntax

-I-

Description

Usually the compiler and assembler search for include files in the standard include directory created when the product is installed. If for some reason you wish to exclude these system locations from being searched when compiling a file, the **-I-** option will do this for you.

-J (Define system include directories)

Syntax

-J *directory*

Description

The **-J** option adds *directory* to the end of the list of directories to search for source files included (using triangular brackets) by the `#include` preprocessor command.

You can specify more than one include directory by separating each directory component with either a comma or semicolon in the property

-K (Keep linker symbol)

Syntax

-K *name*

Description

The linker removes unused code and data from the output file. This process is called *deadstripping*. To prevent the linker from deadstripping unreferenced code and data you wish to keep, you must use the **-K** command line option to force inclusion of symbols.

Example

If you have a C function, **contextSwitch** that must be kept in the output file (and which the linker will normally remove), you can force its inclusion using:

```
-KcontextSwitch
```

-L (Set library directory path)

Syntax

-L *dir*

Description

Sets the library directory to *dir*. If **-L** is not specified on the command line, the default location to search for libraries is set to **$(InstallDir)/lib**.

-I- (Do not link standard libraries)

Syntax

-I-

Description

The **-I** option instructs the compiler driver not to link standard libraries. If you use this option you must supply your own library functions or libraries.

-make (Make-style build)

Syntax

-make

Description

The **-make** option avoids build steps based on the modification date of the output file and modification date of the input file and its dependencies.

-M (Display linkage map)

Syntax

-M

Description

The **-M** option prints a linkage map named the same as the linker output file with the **.map** file extension.

-n (Dry run, no execution)

Syntax

-n

Description

When **-n** is specified, the compiler driver processes options as usual, but does not execute any subprocesses to compile, assemble, archive or link applications.

-nostderr (No stderr output)

Syntax

-nostderr

Description

When **-nostderr** is specified, any stderr output of subprocesses is redirected to stdout.

-o (Set output file name)

Syntax

-o *filename*

Description

The **-o** option instructs the compiler driver to write linker or archiver output to *filename*.

-oabi (Use oabi compiler)

Syntax

-oabi

Description

The **-oabi** option instructs the compiler driver to generate code and link libraries for the legacy GCC ARM ABI.

-O (Optimize output)

Syntax

-O *x*

Description

Pass the optimization option **-O** *x* to the compiler and select library variant. The following options are supported:

- **-O0** — No optimization, use -O1 built libraries
- **-O1** — Level 1 optimization, use -O1 built libraries
- **-O2** — Level 2 optimization, use -O1 built libraries
- **-O3** — Level 3 optimization, use -O1 built libraries
- **-Os** — Optimize for size, use optimize for size built libraries

-printf (Select printf capability)

Syntax

-**printf**= *c*

Description

The **-printf** option selects the printf capability for the linked executable. The options are:

- **-printf=i** — integer is supported
- **-printf=li** — long integer is supported
- **-printf=ll** — long long integer is supported
- **-printf=f** — floating point is supported
- **-printf=wp** — width and precision is supported

-rtti (Enable C++ RTTI Support)

Syntax

-rtti

Description

Enables C++ run-time type information to be compiled.

-R (Set section name)

Syntax

-R *x name*

Description

These options name the default name of the sections generated by the compiler/assembler to be *name*. The options are:

- **-Rc** *name* — change the default name of the code section
- **-Rd** *name* — change the default name of the data section
- **-Rk** *name* — change the default name of the const section
- **-Rz** *name* — change the default name of the bss section

-scanf (Select scanf capability)

Syntax

-scanf= *c*

Description

The **-scanf** option selects the scanf capability for the linked executable. The options are:

- **-scanf=i** — integer is supported
- **-scanf=li** — long integer is supported
- **-scanf=ll** — long long integer is supported
- **-scanf=f** — floating point is supported
- **-scanf=wp** — %[...] and %[^...] character class is supported

-sd (Treat double as float)

Syntax

-sd

Description

The **-sd** option instructs the compiler to compile double as float and selects the appropriate library for linking.

-Thumb (Generate Thumb code)

Syntax

-Thumb

Description

The **-Thumb** option instructs the compiler to generate Thumb code rather than ARM code and link in Thumb libraries. This option is NOT needed for Cortex-M architectures.

-v (Verbose execution)

Syntax

-v

Description

The **-v** switch displays command lines executed by the compiler driver.

-w (Suppress warnings)

Syntax

-w

Description

This option instructs the compiler, assembler, and linker not to issue any warnings.

-we (Treat warnings as errors)

Syntax

-we

Description

This option directs the compiler, assembler, and linker to treat all warnings as errors.

-Wa (Pass option to tool)

Syntax

-W *tool option*

Description

The **-W** command-line option passes *option* directly to the specified *tool*. Supported tools are

- **-Wa** — pass option to assembler
- **-Wc** — pass option to compiler
- **-Wl** — pass option to linker

Example

The following example passes the (compiler specific) -version option to the compiler

```
cc … -Wc-version
```

-x (Specify file types)

Syntax

-x *type*

Description

The **-x** option causes the compiler driver to treat subsequent files to be of the following file type

- **-xa** — archives/libraries
- **-xasm** — assembly code files
- **-xc** — C code files
- **-xc++** — C++ code files
- **-xld** — linker script files
- **-xo** — object code files

Example

The following command line enables an assembly code file with the extension **.arm** to be assembled.

```
cc -xasm a.arm
```

-y (Use project template)

Syntax

-y *t*

Description

If required this option must be the first option on the command line. It instantiates a project template type from the installed packages. The files and common project properties of the project template are used by the compiler driver. Project configurations are not supported by the compiler driver, use crossbuild if you require project configurations.

Example

The following command builds an executable based on the STM32_EXE project template.

```
cc -ySTM32_EXE -zTarget=STM32F100C4 file.c -o file.elf
```

-z (Set project property)

Syntax

-z *p* = *v*

Description

Sets the value of the project property *p* to the value *v*.

Example

The following command compiles the file arguments and puts the resulting object files into the directory **objects**.

```
cc -c file1.c file2.c -zbuild_output_directory=objects
```

CrossBuild

CrossBuild is a program used to build your software from the command line without using **CrossStudio**. You can, for example, use **CrossBuild** for nightly (automated) builds, production builds, and batch builds.

Building with a CrossStudio project file

You can specify a CrossStudio project file:

Syntax

crossbuild [*options …*] *project-file*

You must specify a configuration to build using **-config.** For instance:

```
crossbuild -config "V5T Thumb LE Release" arm.hzp
```

The above example uses the configuration **V5T Thumb LE Release** to build all projects in the solution contained in **arm.hzp**.

To build a specific project that is in a solution, you can specify it using the **-project** option. For example:

```
crossbuild -config "V5T Thumb LE Release" -project "libm" libc.hzp
```

This example will use the configuration **V5T Thumb LE Release** to build the project **libm** that is contained in **libc.hzp**.

If your project file imports other project files (using the <import…> mechanism), when denoting projects you must specify the solution names as a comma-separated list in parentheses after the project name:

```
crossbuild -config "V5T Thumb LE Release" -project "libc(C Library)" arm.hzp
```

libc(C Library) specifies the **libc** project in the **C Library** solution that has been imported by the project file **arm.hzp**.

To build a specific solution that has been imported from other project files, you can use the **-solution** option. This option takes the solution names as a comma-separated list. For example:

```
crossbuild -config "ARM Debug" -solution "ARM Targets,EB55" arm.hzp
```

In this example, **ARM Targets,EB55** specifies the **EB55** solution imported by the **ARM Targets** solution, which was itself imported by the project file **arm.hzp**.

You can do a batch build using the **-batch** option:

```
crossbuild -config "ARM Debug" -batch libc.hzp
```

This will build the projects in **libc.hzp** that are marked for batch build in the configuration **ARM Debug**.

By default, a *make-style* build will be done—i.e., the dates of input files are checked against the dates of output files, and the build is avoided if the output is up to date. You can force a complete build by using the **-rebuild** option. Alternatively, to remove all output files, use the **-clean** option.

To see the commands being used in the build, use the **-echo** option. To also see why commands are being executed, use the **-verbose** option. You can see what commands will be executed, without executing them, by using the **-show** option.

Building without a CrossStudio project file

To use **CrossBuild** without a CrossStudio project, specify the name of an installed project template, the name of the project, and the files to build. For example:

```
crossbuild -config … -template LM3S_EXE -project myproject -file main.c
```

Or, instead of a template, you can specify a project type:

```
crossbuild -config … -type "Library" -project myproject -file main.c
```

You can specify project properties with the **-property** option:

```
crossbuild … -property Target=LM3S811
```

Command-line options

This section describes the command-line options accepted by CrossBuild.

-batch (Batch build)

Syntax

-batch

Description

Perform a batch build.

-config (Select build configuration)

Syntax

-config *name*

Description

Specify the configuration for a build. If the configuration *name* can't be found, CrossBuild will list the available configurations.

-clean (Remove output files)

Syntax

-clean

Description

Remove all output files resulting from the build process.

-define (Define macro)

Syntax

-D *macro=value*

Description

Define a CrossWorks macro value for the build process.

-echo (Show command lines)

Syntax

-echo

Description

Show the command lines as they are executed.

-file (Build a named file)

Syntax

-file *name*

Description

Build the file *name*. Use with **-template** or **-type.**

-packagesdir (Specify packages directory)

Syntax

-**packagesdir** *dir*

Description

Override the default value of the **$(PackagesDir)** macro.

-project (Specify project to build)

Syntax

-**project** *name*

Description

Specify the name of the project to build. When used with a project file, if CrossBuild can't find the specified project, the names of available projects are listed.

-property (Set project property)

Syntax

-**project** *name=value*

Description

Specify the value of a project property — use with **-template** or **-type.** If CrossBuild cannot find the specified property, a list of the properties is shown.

-rebuild (Always rebuild)

Syntax

-rebuild

Description

Always execute the build commands.

-show (Dry run, don't execute)

Syntax

-show

Description

Show the command lines that would be executed, but do not execute them.

-solution (Specify solution to build)

Syntax

-solution *name*

Description

Specify the name of the solution to build. If CrossBuild cannot find the given solution, the valid solution names are listed.

-studiodir (Specify CrossStudio directory)

Syntax

-studiodir *name*

Description

Override the default value of the **$(StudioDir)** macro.

-template (Specify project template)

Syntax

-template *name*

Description

Specify the project template to use. If CrossBuild cannot find the specified template then a list of template names is shown.

-type (Specify project type)

Syntax

-type *name*

Description

Specify the project type to use. If CrossBuild cannot find the specified project type then a list of project type names is shown.

-verbose (Show build information)

Syntax

-verbose

Description

Show extra information relating to the build process.

CrossLoad

The **CrossLoad** program can be used to download and, optionally, debug applications without using CrossStudio.

In order to carry out a download or verify, **CrossLoad** needs to know what target interface to use. The supported target interfaces vary between operating systems; to list the supported target interfaces, use the **-listtargets** option:

```
crossload -listtargets
```

This command will produce a list of target interface names and descriptions, such as:

```
usb                  USB CrossConnect
parport              Parallel Port Interface
sim                  Simulator
```

Use the **-target** option followed by the desired target interface's name to specify which interface to use:

```
crossload -target usb ...
```

CrossLoad normally is used to download and/or verify projects created and built with CrossStudio. To do this, you must specify the target interface you want to use, the CrossStudio solution file, the project name, and the build configuration. The following command line will download and verify the debug version of the project **MyProject** contained within the **MySolution.hzp** solution file, using a USB CrossConnect:

```
crossload -target usb -solution MySolution.hzp -project MyProject -
config Debug
```

In some cases, it is useful to download a program that was not created with CrossStudio by using the settings from an existing CrossStudio project. You might want to do this if your existing project describes specific loaders or scripts required in order to download the application. To do this, you simply add the name of the file you want to download to the command line. For example, the following command line will download the Intel hex file **ExternalApp.hex** using the release settings of the project **MyProject** connecting, using a USB CrossConnect:

```
crossload -target usb -solution MySolution.hzp -project MyProject -
config Release ExternalApp.hex
```

CrossLoad can download and verify a range of file types. The supported file types vary between systems; to list the file types supported on your system, use the **-listfiletypes** option:

```
crossload -listfiletypes
```

This produces a list of the supported file types. For example:

```
hzx                  CrossStudio Executable File
bin                  Binary File
ihex                 Intel Hex File
```

```
hex                 Hex File
tihex               TI Hex File
srec                Motorola S-Record File
```

CrossLoad will attempt to determine the type of any load file given to it. If it cannot do this, you may specify the file type using the **-filetype** option:

```
crossload -target usb -solution MySolution.hzp -project MyProject -
config Release ExternalApp.txt -filetype tihex
```

It is possible, with some targets, to download without specifying a CrossStudio project. In such cases, you only need to specify the target interface and the load file. For example, the following will download **myapp.s19** using a USB CrossConnect:

```
crossload -target usb myapp.s19
```

Each target interface has a range of configurable properties allowing you to customize the default behaviour. To list the target properties and their current values, use the **-listprops** option:

```
crossload -target parport -listprops
```

This command will list the **parport** target-interfaces properties, a description of what the properties are, and their current values:

```
Name:        JTAG Clock Divider
Description: The amount to divide the JTAG clock frequency.
Value      : 1
Name:        Parallel Port
Description: The parallel port connection to use to connect to target.
Value      : Lpt1
Name:        Parallel Port Sharing
Description: Specifies whether sharing of the parallel port with other device drivers or p
Value      : No
```

You can modify a target property using the **-setprop** option. For example, the following command line would set the parallel port interfaced used to **lpt2**:

```
crossload -target parport -setprop "Parallel Port"="Ltp2" …
```

Command line debugging

You can instruct CrossLoad to start a command-line debugging session by using **-debug** and optional **-break** and **-script** options. For example:

```
crossload -target sim -solution mysolution.hzp -project myproject -
config "ARM RAM Debug" -debug -break main
```

This will load the executable created with the **ARM RAM Debug** configuration for **myproject** onto the simulator and run it until its **main** function is called.

A command prompt is then shown that will accept JavaScript statements. The debugger functionality is accessed using the built-in JavaScript object **Debug**, so all debugger commands are be entered using the form **Debug.** *command*().

Managing breakpoints

You can set breakpoints on global symbols using the **Debug.breakexpr("expr")** method. The type of the symbol will determine the breakpoint that is set. For example…

```
Debug.breakexpr("fn1")
```

…will set a breakpoint on entry to the **fn1** function, and…

```
Debug.breakexpr("var1")
```

…will set a breakpoint when the variable **var1** is written. This method can also be used set breakpoints on addresses. For example…

```
Debug.breakexpr("0x248")
```

…will cause a breakpoint when the address **0x248** is executed, and…

```
Debug.breakexpr("(unsigned[1])0xec8")
```

…will cause a breakpoint when the word at the address **0xec8** is written.

You can use the **Debug.breakline("filename", linenumber)** method to set breakpoints on specific lines of code. For example, to set a breakpoint at line number 4 of **c:/directory/file.c**, you can use:

```
Debug.breakline("c:/directory/file.c", 4)
```

Note the use of forward slashes when specifying filenames.

To refer to the current file (the one where the debugger is located), you can use the **Debug.getfilename()** method. Similarly, the current line number is accessed using the **Debug.getlinenumber()** method. Using these functions, you can set a breakpoint at a line-offset from the current position. For example…

```
Debug.breakline(Debug.getfilename(), Debug.getlinenumber()+4)
```

…will break at 4 lines after the current line.

You can use the **Debug.breakdata("expr", value, readNotWrite)** method to set a breakpoint for when a value is written to a global variable. For example…

```
Debug.breakdata("var1", 4, false)
```

…will cause a breakpoint when the value 4 is written to variable **var**. The third parameter, **readNotWrite** specifies whether a breakpoint is set on reading (true) or writing (false) the data.

Each method of setting a breakpoint accepts three optional arguments: *temporary*, *counter*, and *hardware*.

A *temporary breakpoint* is removed the next time it occurs. For example…

```
Debug.breakexpr("fn1()", true)
```

…will break on entry to **fn1** unless another breakpoint occurs before this one.

Counted breakpoints are ignored for the specified number of hits. For example…

```
Debug.breakexpr("fn1()", false, 9)
```

…will break the 10th time **fn1** is called.

The **hardware** argument specifies whether the debugger should use a hardware breakpoint in preference to a software breakpoint. This can be used to set breakpoints on code that is copied to RAM prior to the copying.

The **breakexpr** and **breakline** methods return a positive breakpoint number that can be used to delete the breakpoint using the **Debug.deletebreak(number)** method. For example:

```
fn1bkpt=Debug.breakexpr("fn1")
...
Debug.deletebreak(fn1bkpt)
```

To delete all breakpoints, supply zero to the **deletebreak** method. Note that temporary breakpoints do not have breakpoint numbers.

The **Debug.showbreak(number)** method displays information about a breakpoint.

To show all breakpoints, supply zero to the **showbreak** method.

Some targets support *exception breakpoints*, which can be listed using the **Debug.showexceptions()** method. For example, on an ARM9 or XScale target:

```
>Debug.showexceptions()
Reset disabled
Undef enabled
SWI disabled
P_Abort enabled
D_Abort enabled
IRQ disabled
FIQ disabled
```

You can enable or disable an exception with the **Debug.enableexception("exception", enable)** method. For example…

```
Debug.enableexception("IRQ", true)
```

…will enable breakpoints when the IRQ exception occurs.

Some targets support *breakpoint chaining*. This enables breakpoints to be paired, with one breakpoint enabling another one. For example:

```
>first=Debug.breakdata("count",3)
>second=Debug.breakexpr("fn1")
>Debug.chainbreak(first, second)
```

When **count** is written with the value 3, the breakpoint at **fn1** is enabled; so when **fn1** is subsequently called, if ever, the breakpoint occurs. To remove breakpoint chaining, specify 0 as the second argument. For example:

```
Debug.chainbreak(first, 0)
```

Deleting either of the chained breakpoints will break the chain.

Displaying state

You can display the register state of the current context using the **Debug.printregisters** method, the local variables of the current context using the **Debug.printlocals()** method and the global variables by using the **Debug.printglobals()** method. To display single variables, use the **Debug.print("expr"[,"format"])** method. For example, where **int i = -1**:

```
>Debug.print("i")
0xffffffff
>Debug.print("i", "d")
-1
>Debug.print("i, "u")
4294967295
```

You can change the default radix, used when printing numbers, with the **Debug.setprintradix(radix)** method. For example:

```
>Debug.setprintradix(10)
>Debug.print("i")
-1
>Debug.setprintradix(8)
>Debug.print("i)
037777777777
```

The **Debug.print** method is used to access registers…

```
>Debug.print("@pc")
0x000002ac
```

…and memory, too:

```
>Debug.print("((unsigned[2])0x0)")
[0xeaffffe, 0xe59ff018]
```

You can use the print method to update variables, registers, and memory using assignment operators:

```
>Debug.print("x=45")
0x0000002d
>Debug.print("x+=45")
0x0000005a
```

You can change whether character pointers are displayed as null-terminated strings using the **Debug.setprintstring(bool)** method. For example, where **const char *string = "hello"**:

```
>Debug.print("string")
hello
```

```
>Debug.print("string", "p")
0x00000770
>Debug.setprintstring(false)
>Debug.print("string")
0x00000770
>Debug.print("string", "s")
hello
```

To change the maximum number of array elements that will be displayed, use the **Debug.setprintarray(n)** method. For example, where **unsigned array[4] = {1, 2, 3, 4 }:**

```
>Debug.print("array","d")
[1, 2, 3, 4]
>Debug.setprintarray(2)
>Debug.print("array","d")
[1, 2]
```

You can use the **Debug.evaluate(expr)** method to return the value of variables rather than displaying them. For example…

```
>x=Debug.evaluate("x")
>if (x==-1) Debug.echo("x is 45")
x is 45
```

…where the method **Debug.echo(str)** outputs its string argument.

Locating the current context

You can use the **Debug.where()** method to display a backtrace of the functions that have been called. Each entry in the backtrace has its own *framenumber* which can be supplied to the **Debug.locate(framenumber)** method. Framenumbers start at zero and are incremented for each function call. So framenumber zero is the current location, framenumber one is the caller of the current location, and so on. For example…

```
>Debug.where()
0) int debug_printf(const char* fmt=5)  C:\svn\shared\target\libc
\debug_printf.c:6
1) int main()   C:\tmp\try\main.c:17
2) ???  C:\svn\arm\arm\source\crt0.s:237
```

…then…

```
>Debug.locate(1)
```

…will locate the debugger context at **main** and…

```
>Debug.locate(0)
```

…will change the debugger location back to **debug_printf**.

When the debugger locates (either because `locate` has been called or it has stopped), the corresponding source line is displayed. You can display source lines around the located line by using the **Debug.list(before, after)** method, which specifies the number of lines to display before and after the located line.

You can set the debugger to locate (and step) to machine instructions using the method **Debug.setmode(mode)**. Setting the mode to 1 selects interleaved mode (source code interleaved with assembly code). Setting the mode to 2 selects assembly mode (disassembly with source code annotation). Setting the mode to 0 selects source mode. For example:

```
>Debug.setmode(2)
0000031C  E1A0C00D  mov r12, sp
>Debug.stepinto()
00000320  E92DD800  stmfd sp!, {r11-r12, lr-pc}
>Debug.setmode(0)
```

You can locate the debugger at a specified program counter by using the **Debug.locatepc(pc)** method. For example, you can disassemble from specific address:

```
>Debug.setmode(2)
>Debug.locatepc(0x2f4)
000002F4        E59F30D0   ldr r3, [pc, #+0x0D0]
>Debug.list(0,1)
000002F4        E59F30D0   ldr r3, [pc, #+0x0D0]
```

```
000002F8        E50B3020   str r3, [r11, #-0x020]
```

You can locate the debugger to a full register context using the **Debug.locateregisters(registers)** method. This method takes an array that specifies each register value, typically in ascending register number order. You can use the
{Debug.printregisters()} method to see the the order. For example, for an ARM7, ARM9, or XScale:

```
var a = new Array();
a[0] = 0 // r0 value
...
a[15] = 0x2f4 // pc value
a[16] = 0x10  // cspr value
Debug.locateregisters(a)
```

You can put the debugger context back at the stopped state by calling **Debug.locate** without any parameters:

```
>Debug.locate()
```

Controlling execution

To continue execution from a breakpoint, use the **Debug.go()** method. You can single step into function calls with **Debug.stepinto()**. You can single step over function calls by using the **Debug.stepover()** method. To complete execution of the current function, use the **Debug.stepout()** method.

You will get the debugger prompt immediately when the **go**, **stepinto**, **stepover** or **stepout** methods are called. If you want to wait for the target to stop (for example in a script), you need to use the **Debug.wait(mstimeout)** method, which returns 0 if the millisecond timeout occurred or 1 if execution has stopped. For example…

```
>Debug.go(); Debug.wait(1000)
```

…will wait for one second or until a breakpoint occurs. If a breakpoint isn't reached, you can use the method **Debug.breaknow()** to stop execution. You can end the debug session with the **Debug.quit()** method.

Command-line options

This section describes the command-line options accepted by CrossLoad.

Usage

crossload [*options* …] [*files* …]

ARM Usage

crossload [*options* …] [*files* …] **-serve** [*arguments* …]

-break (Stop execution at symbol)

Syntax

-break *symbol*

Description

When used with the **-debug** option, this will stop execution at *symbol*.

-config (Specify build configuration)

Syntax

-config *name*

Description

Specify the build configuration to use.

-debug (Enter command line debugging)

Syntax

-debug

Description

Enable command-line debugging. A command prompt is displayed at which debugger commands can be entered. The command prompt has a simple history and editing mechanism.

-filetype (Specify load file type)

Syntax

-filetype *filetype*

Description

Specify the type of the file to download. By default, **CrossLoad** will attempt to detect the file type, you should use this option if **CrossLoad** cannot determine the file type or to override the detection and force the type to a specific value. Use the **-listfiletypes** option to list the supported file types.

-help (Display help)

Syntax

-help

Description

Display the command-line options **CrossLoad** accepts.

-listfiletypes (Display supported load file types)

Syntax

-listfiletypes

Description

Lists all the supported file types.

-listprops (Display target properties)

Syntax

-listprops

Description

List the target properties of the target specified by the **-target** option.

-listtargets (Display supported target interfaces)

Syntax

-listtargets

Description

List all the supported target interfaces.

-loadaddress (Set load address)

Syntax

-loadaddress *address*

Description

When downloading a load file that doesn't contain any address information, such a binary file, this option specifies the base address to which the file should be downloaded.

-nodisconnect (Inhibit target disconnection)

Syntax

-nodisconnect

Description

Do not disconnect the target interface when finished.

-nodownload (Inhibit download)

Syntax

-nodownload

Description

Do not download, just verify.

-noverify (Inhibit verification)

Syntax

-noverify

Description

Do not verify the downloaded application.

-packagesdir (Specify package directory)

Syntax

-**packagesdir** *directory*

Description

Set **$(PackagesDir)** to *directory*.

-project (Specify project name)

Syntax

-**project** *name*

Description

Specify the name of the desired project.

-quiet (Be silent)

Syntax

-quiet

Description

Do not output any progress messages.

-script (Execute debug script)

Syntax

-script *file*

Description

When used with the **-debug** option, this will execute the debug commands in *file*.

-serve (Run semihosting server)

Syntax

-serve

Description

Serve CrossStudio debug I/O operations. Any command-line arguments following this option will be passed to the target application. The application can access them either by calling **debug_getargs** or by compiling the startup code in **crt0.s** or **crt0.asm** with the **FULL_LIBRARY** C preprocessor symbol defined so that **argc** and **argv** are passed to **main**.

-setprop (Set target interface property)

Syntax

-setprop *property=value*

Description

Set the target interface property *property* to *value*.

-solution (Specify solution file)

Syntax

-solution *file*

Description

Specify the CrossWorks solution file to use.

-studiodir (Specify Studio directory)

Syntax

-studiodir *directory*

Description

Set **$(StudioDir)** to *directory*.

-target (Specify target interface)

Syntax

-target *name*

Description

Specify the target interface to use. Use the **-listtargets** option to list the supported target interfaces.

-verbose (Display additional status)

Syntax

-verbose

Description

Produce verbose output.

CrossScript

CrossScript is a program that allows you to run CrossStudio's JavaScript (ECMAScript) interpreter from the command line.

The primary purpose of **CrossScript** is to facilitate the creation of platform-independent build scripts.

Syntax

crossscript [*options*] *file* …

Command-line options

This section describes the command-line options accepted by CrossBuild.

-define (Define global variable)

Syntax

-define *variable* = *value*

Description

-help (Show usage)

Syntax

-help

Description

Display usage information and command line options.

-load (Load script file)

Syntax

-load *path*

Description

Loads the script file *path*.

-define (Verbose output)

Syntax

-verbose

Description

Produces verbose output.

CrossScript classes

CrossScript provides the following predefined classes:

- CWSys
- ElfFile
- WScript

Example uses

The following example demonstrates using **CrossScript** to increment a build number:

First, add a JavaScript file to your project called `incbuild.js` containing the following code:

```
function incbuild()
{
  var file = "buildnum.h"
  var text = "#define BUILDNUMBER "
  var s = CWSys.readStringFromFile(file);
  var n;
  if (s == undefined)
    n = 1;
  else
    n = eval(s.substring(text.length)) + 1;
  CWSys.writeStringToFile(file, text + n);
}
incbuild();
```

Add a file called `getbuildnum.h` to your project containing the following code:

```
#ifndef GETBUILDNUM_H
#define GETBUILDNUM_H
unsigned getBuildNumber();
#endif
```

Add a file called `getbuildnum.c` to your project containing the following code:

```
#include "getbuildnum.h"
#include "buildnum.h"
unsigned getBuildNumber()
{
  return BUILDNUMBER;
}
```

Now, to combine these:

- Set the **Build Options** > **Always Rebuild** project property of `getbuildnum.c` to **Yes**.
- Set the **User Build Step Options** > **Pre-Compile Command** project property of `getbuildnum.c` to **"$(StudioDir)/bin/crossscript" -load "$(ProjectDir)/incbuild.js"**.

Header file generator

The command line program **mkhdr** generates a C or C++ header file from a CrossWorks memory map file.

Using the header generator

For each register definition in the memory map file a corresponding **#define** is generated in the header file. The **#define** is named the same as the register name and is defined as a volatile pointer to the address.

The type of the pointer is derived from the size of the register. A four-byte register generates an unsigned long pointer. A two-byte register generates an unsigned short pointer. A one-byte register will generates an unsigned char pointer.

If a register definition in the memory map file has bitfields then preprocessor symbols are generated for each bitfield. Each bitfield will have two preprocessor symbols generated, one representing the mask and one defining the start bit position. The bitfield preprocessor symbol names are formed by prepending the register name to the bitfield name. The mask definition has **_MASK** appended to it and the start definition has **_BIT** appended to it.

For example consider the following definitions in the the file **memorymap.xml**.

```
<RegisterGroup start="0xFFFFF000" name="AIC" >
  <Register start="+0x00" size="4" name="AIC_SMR0">
    <BitField size="3" name="PRIOR" start="0" />
    <BitField size="2" name="SRCTYPE" start="5" />
  </Register>
  ...
```

We can generate the header file associated with this file using:

```
mkhdr memorymap.xml memorymap.h
```

This generates the following definitions in the file **memorymap.h**.

```
#define AIC_SMR0 (*(volatile unsigned long *)0xFFFFF000)
#define AIC_SMR0_PRIOR_MASK 0x7
#define AIC_SMR0_PRIOR_BIT 0
#define AIC_SMR0_SRCTYPE_MASK 0x60
#define AIC_SMR0_SRCTYPE_BIT 5
```

These definitions can be used in the following way in a C/C++ program:

Reading a register

```
unsigned r = AIC_SMR0;
```

Writing a register

```
AIC_SMR0 = (priority << AIC_SMR0_PRIOR_BIT) | (srctype << AIC_SMR0_SRCTYPE_BIT);
```

Reading a bitfield

```
unsigned srctype = (AIC_SMR0 & AIC_SMR0_SRCTYPE_MASK) >> AIC_SMR0_SRCTYPE_BIT;
```

Writing a bitfield

```
AIC_SMR0 = (AIC_SMR0 & ~AIC_SMR0_SRCTYPE_MASK) | ((srctype & AIC_SMR0_SRCTYPE_MASK) << AIC
```

Command line options

This section describes the command line options accepted by the header file generator.

Syntax

mkhdr *inputfile outputfile targetname* [*option* …]

inputfile is the name of the source CrossWorks memory map file. **outputfile** is the the name of the file to write.

-regbaseoffsets (Use offsets from peripheral base)

Syntax

-regbaseoffsets

Description

Instructs the header generator to include offsets of registers from the peripheral base.

-nobitfields (Inhibit bitfield macros)

Syntax

-nobitfields

Description

Instructs the header generator not to generate any definitions for bitfields.

Package generator

To create a package the program **mkpkg** can be used. The set of files to put into the package should be in the desired location in the `$(PackagesDir)` directory. The **mkpkg** command should be run with `$(PackagesDir)` as the working directory and all files to go into the package must be referred to using relative paths. A package must have a package description file that is placed in the `$(PackagesDir)/packages` directory. The package description file name must end with `_package.xml`. If a package is to create entries in the new project wizard then it must have a file name `project_templates.xml`.

For example, a package for the mythical FX150 processor would supply the following files:

- A project template file called `targets/FX150/project_templates.xml`. The format of the project templates file is described in **Project Templates file format**.
- The `$(PackagesDir)`-relative files that define the functionality of the package.
- A package description file called `packages/FX150_package.xml`. The format of the package description file is described in **Package Description file format**.

The package file `FX150.hzq` would be created using the following command line:

```
mkpkg -c packages/FX150.hzq targets/FX150/project_templates.xml … packages/FX150_package.xml
```

You can list the contents of the package using the **-t** option:

```
mkpkg -t packages/FX150.hzq
```

You can remove an entry from a package using the **-d** option:

```
mkpkg -d packages/FX150.hzq -d fileName
```

You can add or replace a file into an existing package using the **-r** option:

```
mkpkg -r packages/FX150.hzq -r fileName
```

You can extract files from an existing package using the **-x** option:

```
mkpkg -x packages/FX150.hzq outputDirectory
```

You can automate the package creation process using a **Combining** project type.

- Using the new project wizard create a combining project in the directory `$(PackagesDir)`.
- Set the **Output File Path** property to be `$(PackagesDir)/packages/mypackage.hzq`.
- Set the **Combine command** property to **$(StudioDir)/bin/mkpkg -c $(CombiningOutputFilePath) $(CombiningRelInputPaths)**.
- Add the files you want to go into the package into the project using the Project Explorer.
- Right-click the project node in the Project Explorer and choose **Build**.

When a package is installed, the files in the package are copied into the desired `$(PackagesDir)`-relative locations. When a file is copied into the `$(PackagesDir)/packages` directory and its filename ends with

_package.xml the file $(PackagesDir)/packages/installed_packages.xml is updated with an entry:

```
<include filename="FX150_package.xml" />
```

During development of a package you can manually edit this file. The same applies to the file $(PackagesDir)/targets/project_templates.xml which will contain a reference to your project_templates.xml file.

Usage:

mkpkg [*options*] *packageFileName file1 file2 …*

Option	Description
-c	Create a new package.
-d	Remove files from a package.
-r	Replace files in a package.
-t	List the contents of a package.
-v	Be chatty.
-V	Show version information.

File formats

This section describes the file formats CrossWorks uses:

Memory Map file format

Describes the memory map file format that defines memory regions and registers in a microcontroller.

Section Placement file format

Describes the section placement file format that maps program sections to memory areas in the target microcontroller.

Project file format

Describes the format of CrossStudio project files.

Project Templates file format

Describes the format of project template files used by the **New Project** wizard.

Property Groups file format

Describes the format of the property groups file you can use to define 'meta-properties'.

Package Description file format

Describes the format of the package description files you use to create packages other users can install in CrossStudio.

Memory Map file format

CrossStudio memory-map files are structured using XML syntax for its simple construction and parsing.

The first entry of the project file defines the XML document type used to validate the file format.

```
<!DOCTYPE Board_Memory_Definition_File>
```

The next entry is the `Root` element. There can only be one `Root` element in a memory map file:

```
<Root name="My Board" >
```

A `Root` element has a `name` attribute — every element in a memory map file has a `name` attribute. Names should be unique within a hierarchy level. Within a `Root` element, there are `MemorySegment` elements that represent regions within the memory map.

```
<Root name="My Board" >
  <MemorySegment name="Flash" start="0x1000" size="0x200" access="ReadOnly" >
```

`MemorySegment` elements have the following attributes:

- *start*: The start address of the memory segment. A simple expression, usually a hexadecimal number with a 0x prefix.
- *size*: The size of the memory segment. A simple expression, usually a hexadecimal number with a 0x prefix.
- *access*: The permissible access types of the memory segment. One of `ReadOnly`, `Read/Write`, `WriteOnly`, or `None`.
- *address_symbol*: A symbolic name for the start address of the memory segment.
- *size_symbol*: A symbolic name for the size of the memory segment.
- *address_symbol*: A symbolic name for the end address of the memory segment.

`RegisterGroup` elements are used to organize registers into groups. `Register` elements are used to define peripheral registers:

```
<Root name="My Board" >
  <MemorySegment name="System" start="0x2000" size="0x200" >
    <RegisterGroup name="Peripheral1" start="0x2100" size="0x10" >
      <Register name="Register1" start="+0x8" size="4" >
```

`RegisterGroup` elements have the same attributes as `MemorySegment` elements. `Register` elements have the following attributes:

- *name*: Register names should be valid C/C++ identifier names, i.e., alphanumeric characters and underscores are allowed but names cannot start with a number.
- *start*: The start address of the memory segment. Either a C-style hexadecimal number or, if given a + prefix, an offset from the enclosing element's start address.
- *size*: The size of the register in bytes, either 1, 2, or 4.

- *access*: The same as the `access` attribute of the `MemorySegment` element.
- *address_symbol*: The same as the `address_symbol` attribute of the `MemorySegment` element.

A `Register` element can contain `BitField` elements that represent the bits in a peripheral register:

```
<Root name="My Board" >
  <MemorySegment name="System" start="0x2000" size="0x200" >
    <RegisterGroup name="Peripheral1" start="0x2100" size="0x10" >
      <Register name="Register1" start="+0x8" size="4" >
        <BitField name="Bits_0_to_3" start="0" size="4" />
```

`BitField` elements have the following attributes:

- *name*: The same as the `name` attribute of the `RegisterGroup` element.
- *start*: The starting bit position, 0–31.
- *size*: The total number of bits, 1–32.

A `Bitfield` element can contain `Enum` elements:

```
<Root name="My Board" >
  <RegisterGroup name="Peripheral1" start="0x2100" size="0x10" >
    <Register name="Register1" start="+0x8" size="4" >
      <BitField name="Bits_0_to_3" start="0" size="4" />
        <Enum name="Enum3" start="3" />
        <Enum name="Enum5" start="5" />
```

You can import CMSIS SVD files (see **http://www.onarm.com/**) into a memory map using the `ImportSVD` element:

```
<ImportSVD filename="$(TargetsDir)/targets/Manufacturer1/Processor1.svd.xml" >
```

The `filename` attribute is an absolute filename which is macro-expanded using CrossWorks system macros.

When a memory map file is loaded either for the memory map viewer or to be used for linking or debugging, it is preprocessed using the (as yet undocumented) CrossWorks XML preprocessor.

Section Placement file format

CrossStudio section-placement files are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type used to validate the file format:

```
<!DOCTYPE Linker_Placement_File>
```

The next entry is the `Root` element. There can only be one `Root` element in a memory map file:

```
<Root name="Flash Placement" >
```

A `Root` element has a `name` attribute. Every element in a section-placement file has a `name` attribute. Each name should be unique within its hierarchy level. Within a `Root` element, there are `MemorySegment` elements. These correspond to memory regions defined in a memory map file that will be used in conjunction with the section-placement file when linking a program. For example:

```
<Root name="Flash Placement" >
  <MemorySegment name="FLASH" >
```

A `MemorySegment` contains `ProgramSection` elements that represent program sections created by the C/C++ compiler and assembler. The order of `ProgramSection` elements within a `MemorySegment` element represents the order in which the sections will be placed when linking a program. The first `ProgramSection` will be placed first and the last one will be placed last.

```
<Root name="My Board" >
  <MemorySegment name="FLASH" >
    <ProgramSection name=".text" ... >
```

`ProgramSection` elements have the following attributes:

- *alignment*: The required alignment of the program section; a decimal number specifying the byte alignment.
- *inputsections*: An expression describing the input sections to be placed in this section. If you omit this (recommended) and the section name isn't one of **.text**, **.dtors**, **.ctors**, **.data**, **.rodata**, or **.bss**, then the equivalent input section of ***(.name .name.***) is supplied to the linker.
- *load*: If **Yes**, the section is loaded. If **No**, the section isn't loaded.
- *runin*: This specifies the name of the section to copy this section to.
- *start*: The optional start address of the program section, a hexadecimal number with a 0x prefix.
- *size*: The optional size of the program section in bytes, a hexadecimal number with a 0x prefix.
- *address_symbol*: A symbolic name for the start address of the section.
- *end_symbol*: A symbolic name for the end address of the section.
- *size_symbol*: A symbolic name for the size of the section.
- *fill*: The optional value used to fill unspecified regions of memory, a hexadecimal number with a 0x prefix.

When a section placement file is used for linking it is preprocessed using the (as yet undocumented) CrossWorks XML preprocessor.

Project file format

CrossStudio project files are held in text files with the `.hzp` extension. Because you may want to edit project files, and perhaps generate them, they are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type used to validate the file format:

```
<!DOCTYPE CrossStudio_Project_File>
```

The next entry is the `solution` element; there can only be one `solution` element in a project file. This specifies the solution name displayed in the **Project Explorer** and has a version attribute that defines the file-format version of the project file. Solutions can contain projects, projects can contain folders and files, and folders can contain folders and files. This hierarchy is reflected in the XML nesting—for example:

```
<solution version="1" Name="solutionname">
  <project Name="projectname">
    <file Name="filename"/>
    <folder Name="foldername">
      <file Name="filename2"/>
    </folder>
  </project>
</solution>
```

Note that each entry has a `Name` attribute. Names of `project` elements must be unique to the solution, and names of `folder` elements must be unique to the project, but names of files do not need to unique.

Each `file` element must have a `file_name` attribute that is unique to the project. Ideally, the `file_name` is a file path relative to the project (or solution directory), but you can also specify a full file path, if you want to. File paths are case-sensitive and use "/" as the directory separator. They may contain macro instantiations, so file paths cannot contain the "$" character. For example…

```
<file file_name="$(StudioDir)/source/crt0.s" Name="crt0.s" />
```

…will be expanded using the value of the $(StudioDir) when the file is referenced from CrossStudio.

Project properties are held in configuration elements with the `Name` attribute of the configuration element corresponding to the configuration name, e.g., "Debug". At a given project level (i.e., solution, project, folder), there can only be one named configuration element—i.e., all properties defined for a configuration are in single configuration element.

```
<project Name="projectname">
  ...
  <configuration project_type="Library" Name="Common" />
  <configuration Name="Release" build_debug_information="No" />
  ...
```

```
</project>
```

You can use the `import` element to link projects:

```
<import file_name="target/libc.hzp" />
```

Project Templates file format

The CrossStudio **New Project** dialog works from a file called `project_templates.xml` in the `targets` subdirectory of the CrossStudio installation directory. Because you may want to add your own new project types, they are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type used to validate the file format:

```
<!DOCTYPE Project_Templates_File>
```

The next entry is the `projects` element, which is used to group a set of new project entries into an XML hierarchy.

```
<projects>
  <project…
</projects>
```

Each entry has a `project` element that contains the class of the project (attribute `caption`), the name of the project (attribute `name`), its type (attribute `type`) and a description (attribute `description`). For example:

```
<project caption="ARM Evaluator7T" name="Executable" description="An executable for an ARM
>
```

The project type can be one of these:

- *Executable*: — a fully linked executable.
- *Library*: — a static library.
- *Object file*: — an object file.
- *Staging*: — a staging project.
- *Combining*: — a combining project.
- *Externally Built Executable*: — an externally built executable.

The configurations to be created for the project are defined using the `configuration` element, which must have a `name` attribute:

```
<configuration name="ARM RAM Release"/>
```

The property values to be created for the project are defined using the `property` element. If you have a defined value, you can specify this using the `value` attribute and, optionally, set the property in a defined `configuration`, such as:

```
<property name="target_reset_script" configuration="RAM" value="Evaluator7T_ResetWithRamAt
>
```

Alternatively, you can include a property that will be shown to the user, prompting them to supply a value as part of the new-project process.

```
<property name="linker_output_format"/>
```

The folders to be created are defined using the `folder` element. The `folder` element must have a `name` attribute and can also have a `filter` attribute. For example:

```
<folder name="Source Files" filter="c;cpp;cxx;cc;h;s;asm;inc"/>
```

The files to be in the project are specified using the `file` element. You can use build-system macros (see **Project macros**) to specify files located in the CrossStudio installation directory. Files will be copied to the project directory or just left as references, depending on the value of the `expand` attribute:

```
<file name="$(StudioDir)/source/crt0.s" expand="no"/>
```

You can define the set of configurations that can be referred to in the top-level `configurations` element:

```
<configurations>
  <configuration…
</configurations>
```

This contains the set of all configurations that can be created when a project is created. Each configuration is defined using a `configuration` element, which can define the property values for that configuration. For example:

```
<configuration name="Debug">
  <property name="build_debug_information" value="Yes">
```

Property Groups file format

The CrossStudio project system provides a means to create new properties that change a number of project property settings and can also set C pre-processor definitions when selected. Such properties are called *property groups* and are defined in a property-groups file. The property-group file to use for a project is defined by the **Property Groups File** property. These files usually define target-specific properties and are structured using XML syntax to enable simple construction and parsing.

The first entry of the property groups file defines the XML document type, which is used to validate the file format:

```
<!DOCTYPE CrossStudio_Group_Values>
```

The next entry is the `propertyGroups` element, which is used to group a set of property groups entries into an XML hierarchy:

```
<propertyGroups>
  <group...
  ...
  <group...
</propertyGroups>
```

Each group has the name of the group (attribute `name`), the name of the options category (attribute `group`), short (attribute `short`) and long (attribute `long`) help descriptions, and a default value (attribute `default`). For example:

```
<group short="Target Processor" group="Build Options" short="Target Processor" long="Selec
>
```

Each group has a number of `groupEntry` elements that define the enumerations of the group.

```
<group...>
  <groupEntry...
  ...
  <groupEntry...
</group>
```

Each `groupEntry` has the name of the entry (attribute `name`), e.g.:

```
<groupEntry name="STR910FW32">
```

A `groupEntry` has the property values and C pre-processor definitions that are set when the `groupEntry` is selected; they are specified with `property` and `cdefine` elements. For example:

```
<groupEntry...>
  <property...
  <cdefine...
```

```
  <property…
</groupEntry>
```

A `property` element has the property's name (attribute `name`), its value (attribute `value`), and an optional configuration (attribute `configuration`):

```
<property name="linker_memory_map_file" value="$(StudioDir)/targets/ST_STR91x/
ST_STR910FM32_MemoryMap.xml"/>
```

A `cdefine` element has the C preprocessor name (attribute `name`) and its value (attribute `value`):

```
<cdefine value="STR910FM32" name="TARGET_PROCESSOR">
```

Package Description file format

Package-description files are XML files used by CrossStudio to describe a support package, its contents, and any dependencies it has on other packages.

Each package file must contain one `package` element that describes the package. Optionally, the `package` element can contain a collection of `file`, `history`, and `documentation` elements to be used by CrossStudio for documentation purposes.

The filename of the package-description file should match that of the package and end in "*_package.xml*".

Below is an example of two package-description files. The first is for a base chip-support package for the LPC2000; the second is for a board-support package dependent on the first:

Philips_LPC2000_package.xml

```
<!DOCTYPE CrossStudio_Package_Description_File>
<package cpu_manufacturer="Philips" cpu_family="LPC2000" version="1.1" crossstudio_version
  <file file_name="$(TargetsDir)/Philips_LPC210X/
arm_target_Philips_LPC210X.htm" title="LPC2000 Support Package Documentation" /
>
  <file file_name="$(TargetsDir)/Philips_LPC210X/
Loader.hzp" title="LPC2000 Loader Application Solution" />
  <group title="System Files">
    <file file_name="$(TargetsDir)/Philips_LPC210X/
Philips_LPC210X_Startup.s" title="LPC2000 Startup Code" />
    <file file_name="$(TargetsDir)/Philips_LPC210X/
Philips_LPC210X_Target.js" title="LPC2000 Target Script" />
  </group>
  <history>
    <version name="1.1" >
      <description>Corrected LPC21xx header files and memory maps to include GPIO ports 2
description>
      <description>Modified loader memory map so that .libmem sections will be placed corr
description>
    </version>
    <version name="1.0" >
      <description>Initial Release.</description>
    </version>
  </history>
  <documentation>
    <section name="Supported Targets">
      <p>This CPU support package supports the following LPC2000 targets:
```

```
        <ul>
          <li>LPC2103</li>
          <li>LPC2104</li>
          <li>LPC2105</li>
          <li>LPC2106</li>
          <li>LPC2131</li>
          <li>LPC2132</li>
          <li>LPC2134</li>
          <li>LPC2136</li>
          <li>LPC2138</li>
        </ul>
      </p>
    </section>
  </documentation>
</package>
```

CrossFire_LPC2138_package.xml

```
<!DOCTYPE CrossStudio_Package_Description_File>
<package cpu_manufacturer="Philips" cpu_family="LPC2000" cpu_name="LPC2138" board_manufact
  <file file_name="$(SamplesDir)/CrossFire_LPC2138/
CrossFire_LPC2138.hzp" title="CrossFire LPC2138 Samples Solution" />
  <file file_name="$(SamplesDir)/CrossFire_LPC2138/ctl/
ctl.hzp" title="CrossFire LPC2138 CTL Samples Solution" />
</package>
```

Package elements

The `package` element describes the support package, its contents, and any dependencies it has on other packages. Valid attributes for this element are:

Attribute	Description
`author`	The author of the package.
`board_manufacturer`	The manufacturer of the board supported by the package *(if omitted, CPU manufacturer will be used)*.
`board_name`	The name of the specific board supported by the package *(only required for board-support packages)*.
`cpu_family`	The family name of the CPU supported by the package *(optional)*.

`cpu_manufacturer`	The manufacturer of the CPU supported by the package.
`cpu_name`	The name of the specific CPU supported by the package *(may be omitted if the CPU family is specified)*.
`crossstudio_versions`	A string describing which version of CrossStudio supports the package *(optional)*. The format of the string is *target_id_number:version_range_string*.
`description`	A description of the package *(optional)*.
`dependencies`	A semicolon-separated list of packages the package requires to be installed in order to work.
`installation_directory`	The directory in which the package should be installed *(optional\--if undefined, defaults to "$(PackagesDir)")*.
`title`	A short description of the package *(optional)*.
`version`	The package version number.

File elements

The `file` element is used by CrossStudio for documentation purposes by adding links to files of interest within the package such as example project files and documentation.

Attribute	Description		
`file_name`	The file path of the file.	title	A description of the file.

Optionally, `file` elements can be grouped into categories using the `group` element.

Group elements

The `group` element is used for categorizing files described by `file` elements into a particular group.

Attribute	Description
title	Title of the group.

History elements

The `history` element is used to hold a description of the package's version history.

The `history` element should contain a collection of `version` elements.

Version element

The `version` element is used to hold the description of a particular version of the package.

Attribute	Description
name	The name of the version being described.

The `version` element should contain a collection of `description` elements.

Description elements

Each `description` element contains text that describes a feature of the package version.

Documentation elements

The `documentation` element is used to provide arbitrary documentation for the package.

The `documentation` element should contain a collection of one or more `section` elements.

Section elements

The `section` element contains package documentation in XHTML format.

Attribute	Description
name	The title of the documentation section.

target_id_number

The following table lists the possible target ID numbers:

Target	ID

AVR	4
ARM	8
MSP430	9
MAXQ20	18
MAXQ30	19

version_range_string

The `version_range_string` can be any of the following:

- *version_number*: The package will only work on *version_number*.
- *version_number-*: The package will work on *version_number* or any future version.
- *-version_number*: The package will work on *version_number* or any earlier version.
- *low_version_number-high_version_number*: The package will work on *low_version_number*, *high_version_number* or any version in between.

General Build Properties

Build Options

Property	Description
Always Rebuild `build_always_rebuild` – Boolean	Specifies whether or not to always rebuild the project/folder/file.
Build Quietly `build_quietly` – Boolean	Suppress the display of startup banners and information messages.
Enable Unused Symbol Removal `build_remove_unused_symbols` – Boolean	Enable the removal of unused symbols from the executable.
Exclude From Build `build_exclude_from_build` – Boolean	Specifies whether or not to exclude the project/folder/file from the build.
Include Debug Information `build_debug_information` – Boolean	Specifies whether symbolic debug information is generated.
Intermediate Directory `build_intermediate_directory` – FileName	Specifies a relative path to the intermediate file directory. This property will have macro expansion applied to it. The macro $(IntDir) is set to this value.
Memory Map File `linker_memory_map_file` – ProjFileName	The name of the file containing the memory map description.
Memory Map Macros `linker_memory_map_macros` – StringList	Macro values to substitue in memory map nodes. Each macro is defined as **name=value** and are seperated by **;**.
Output Directory `build_output_directory` – FileName	Specifies a relative path to the output file directory. This property will have macro expansion applied to it. The macro $(OutDir) is set to this value. The macro $(RootRelativeOutDir) is set relative to the **Root Output Directory** if specified.
Project Dependencies `project_dependencies` – StringList	Specifies the projects the current project depends upon.
Project Directory `project_directory` – String	Path of the project directory relative to the directory containing the project file. The macro $(ProjectDir) is set to the absolute path of this property.
Project Macros `macros` – StringList	Specifies macro values which are expanded in project properties. Each macro is defined as **name=value** and are seperated by **;**.
Project Type `project_type` – Enumeration	Specifies the type of project to build. The options are **Executable, Library, Object file, Staging, Combining, Externally Built Executable**.

Property Groups File `property_groups_file_path` – ProjFileName	The file containing the property groups for this project. This is applicable to **Executable** and **Externally Built Executable** project types only.
Root Output Directory `build_root_output_directory` – FileName	Allows a common root output directory to be specified that can be referenced using the $(RootOutDir) macro.
Suppress Warnings `build_suppress_warnings` – Boolean	Don't report warnings.
Treat Warnings as Errors `build_treat_warnings_as_errors` – Boolean	Treat all warnings as errors.

General Options

Property	**Description**
Batch Build Configurations `batch_build_configurations` – StringList	The set of configurations to batch build.
Inherited Configurations `inherited_configurations` – StringList	The list of configurations that are inherited by this configuration.

Combining Project Properties

Combining Options

Property	Description
Combine Command `combine_command` – String	The command to execute. This property will have macro expansion applied to it with the macro **$(CombiningOutputFilePath)** set to the output filepath of the combine command and the macro **$(CombiningRelInputPaths)** is set to the (project relative) names of all of the files in the project.
Combine Command Working Directory `combine_command_wd` – String	The working directory in which the combine command is run. This property will have macro expansion applied to it.
Output File Path `combine_output_filepath` – String	The output file path the stage command will create. This property will have macro expansion applied to it.
Set To Read-only `combine_set_readonly` – Boolean	Set the output file to read only or read/write.

Compilation Properties

Assembler Options

Property	Description
Additional Assembler Options `asm_additional_options` – StringList	Enables additional options to be supplied to the assembler. This property will have macro expansion applied to it.
Additional Assembler Options From File `asm_additional_options_from_file` – ProjFileName	Enables additional options to be supplied to the assembler from a file. This property will have macro expansion applied to it.

Code Generation Options

Property	Description
ARM Architecture `arm_architecture` – Enumeration	Specifies the version of the instruction set to generate code for. The options are: • v4T - ARM7TDMI and ARM920T processors • v5TE - ARM9E, Feroceon and XScale processors • v6 - ARM11 processors • v6M - Cortex-M0 and Cortex-M1 processors • v7M - Cortex-M3 processors • v7EM - Cortex-M4 processors • v7R - Cortex-A processors • v7A - Cortex-R processors The corresponding preprocessor definitions: • __ARM_ARCH_4T__ • __ARM_ARCH_5TE__ • __ARM_ARCH_6__ • __ARM_ARCH_6M__ • __ARM_ARCH_7M__ • __ARM_ARCH_7EM__ • __ARM_ARCH_7R__ • __ARM_ARCH_7A__ are defined.

ARM Core Type `arm_core_type` – Enumeration	Specifies the core to generate code for. The options are: • ARM7TDMI • ARM7TDMI-S • ARM720T • ARM920T • ARM946E-S • ARM966E-S • ARM968E-S • ARM926EJ-S • ARM1136J-S • ARM1136JF-S • Cortex-M0 • Cortex-M1 • Cortex-M3 • Cortex-M4 • Cortex-R4 • Cortex-R4F • Cortex-A8 • Cortex-A9 • XScale • None If this property is set to None then the architecture value is used
ARM FP ABI Type `arm_fp_abi` – Enumeration	Specifies the FP ABI type to generate code for. The options are: • **Soft** generate calls to the CrossWorks C library to implement floating point operations. • **SoftFP** generate VFP code to implement floating point operations. • **Hard** generate VFP code to implement floating point operations and use VFP registers to pass floating point parameters on function calls. • **None** will not specify the FP ABI or the FPU.

ARM FPU Type `arm_fpu_type` – Enumeration	Specifies the FPU type to generate code for. The options are: • VFP - ARM9/ARM11 based processors • VFP9 - the same as VFP • VFPv3-D32 - Cortex-A/Cortex-R based processors • VFPv3-D16 - Cortex-A/Cortex-R based processors • FPv4-SP-D16 - Cortex-M4 processors The corresponding preprocessor definitions: • __ARM_ARCH_VFP__ • __ARM_ARCH_VFPv3_D32__ • __ARM_ARCH_VFPv3_D16__ • __ARM_ARCH_FPv4_SP_D16__ are defined.
ARM/Thumb Interworking `arm_interwork` – Boolean	Specifies that ARM/Thumb interworking code should be generated. Setting this property to **No** may result in smaller code sizes when compiling for architecture **v4T**.
Byte Order `arm_endian` – Enumeration	Specify the byte order of the target processor.
Debugging Level `gcc_debugging_level` – Enumeration	Specifies the level of debugging information to generate.
Emit Assembler CFI `gcc_emit_assembler_cfi` – Boolean	Emit DWARF 2 unwind info using GAS .cfi_* directives rather than a compiler generated .eh_frame section.
Enable Exception Support `cpp_enable_exceptions` – Boolean	Specifies whether exception support is enabled for C++ programs.
Enable RTTI Support `cpp_enable_rtti` – Boolean	Specifies whether RTTI support is enabled for C++ programs.
GCC Target `arm_gcc_target` – Enumeration	Specifies which GCC compiler target to use.
Instruction Set `arm_instruction_set` – Enumeration	Specifies the instruction set to generate code for.
Long Calls `arm_long_calls` – Boolean	Specifies whether function calls are made using absolute addresses.
No COMMON `gcc_no_common` – Boolean	Don't put globals
Omit Frame Pointer `gcc_omit_frame_pointer` – Boolean	Specifies whether a frame pointer register is omitted if not required.
Optimization Level `gcc_optimization_level` – Enumeration	Specifies the optimization level to use.
Treat 'double' as 'float' `double_is_float` – Boolean	Forces the compiler to make 'double' equivalent to 'float'.

Compiler Options

Property	Description
Additional C Compiler Only Options `c_only_additional_options` – StringList	Enables additional options to be supplied to the C compiler only. This property will have macro expansion applied to it.
Additional C Compiler Only Options From File `c_only_additional_options_from_file` – ProjFileName	Enables additional options to be supplied to the C compiler only from a file. This property will have macro expansion applied to it.
Additional C++ Compiler Only Options `cpp_only_additional_options` – StringList	Enables additional options to be supplied to the C++ compiler only. This property will have macro expansion applied to it.
Additional C++ Compiler Only Options From File `cpp_only_additional_options_from_file` – ProjFileName	Enables additional options to be supplied to the C++ compiler only from a file. This property will have macro expansion applied to it.
Additional C/C++ Compiler Options `c_additional_options` – StringList	Enables additional options to be supplied to the C/C++ compiler. This property will have macro expansion applied to it.
Additional C/C++ Compiler Options From File `c_additional_options_from_file` – ProjFileName	Enables additional options to be supplied to the C/C++ compiler from a file. This property will have macro expansion applied to it.
Enforce ANSI Checking `c_enforce_ansi_checking` – Boolean	Ensure programs conform to the ANSI-C/C++ standard.
Keep Assembly Source `arm_keep_assembly` – Boolean	Specifies whether assembly code generated by the compiler is kept.
Object File Name `build_object_file_name` – FileName	Specifies a name to override the default object file name.
Supply Absolute File Path `arm_supply_absolute_file_path` – Boolean	Specifies whether absolute file paths are supplied to the compiler.

Preprocessor Options

Property	Description
Ignore Includes `c_ignore_includes` – Boolean	Ignore the include directories properties.
Preprocessor Definitions `c_preprocessor_definitions` – StringList	Specifies one or more preprocessor definitions. This property will have macro expansion applied to it.
Preprocessor Undefinitions `c_preprocessor_undefinitions` – StringList	Specifies one or more preprocessor undefinitions. This property will have macro expansion applied to it.
System Include Directories `c_system_include_directories` – StringList	Specifies the system include path. This property will have macro expansion applied to it.

Undefine All Preprocessor Definitions `c_undefine_all_preprocessor_definitions` – Boolean	Does not define any standard preprocessor definitions.
User Include Directories `c_user_include_directories` – StringList	Specifies the user include path. This property will have macro expansion applied to it.

Section Options

Property	Description
Code Section Name `default_code_section` – String	Specifies the default name to use for the program code section.
Constant Section Name `default_const_section` – String	Specifies the default name to use for the read-only constant section.
Data Section Name `default_data_section` – String	Specifies the default name to use for the initialized, writable data section.
Vector Section Name `default_vector_section` – String	Specifies the default name to use for the interrupt vector section.
Zeroed Section Name `default_zeroed_section` – String	Specifies the default name to use for the zero-initialized, writable data section.

User Build Step Options

Property	Description
Post-Compile Command `compile_post_build_command` – String	A command to run after the compile command has completed. This property will have macro expansion applied to it with the additional **$(TargetPath)** macro set to the output filepath of the compiler command.
Post-Compile Working Directory `compile_post_build_command_wd` – String	The working directory where the post-compile command is run. This property will have macro expansion applied to it.
Pre-Compile Command `compile_pre_build_command` – String	A command to run before the compile command. This property will have macro expansion applied to it.
Pre-Compile Command Output File Path `compile_pre_build_command_output_file_name` – String	The pre-compile generated file name. This property will have macro expansion applied to it.
Pre-Compile Working Directory `compile_pre_build_command_wd` – String	The working directory where the pre-compile command is run. This property will have macro expansion applied to it.

Debugging Properties

Debugger Options

Property	Description
ARM Floating Point Format `arm_fp_format` – Enumeration	Specifies how to display double precision floating point numbers.This can be used with **Externally Built Executable** projects whose executables have been built with FPA floating point format.
Additional Load File `debug_additional_load_file` – ProjFileName	Additional file to load on debug load. This property will have macro expansion applied to it.
Additional Load File Address `debug_additional_load_file_address` – String	The address to load the additional load file.
Additional Load File Type `debug_additional_load_file_type` – Enumeration	The file type of the additional load file. The options are **Detect, elf, bin, ihex, hex, tihex, srec**.
Command Arguments `debug_command_arguments` – String	The command arguments passed to the executable. This property will have macro expansion applied to it.
Entry Point Symbol `debug_entry_point_symbol` – String	Debugger will start execution at symbol if defined.
Load Offset `debug_load_file_offset` – String	The offset to add to the load address of the load file.This offset is added to any absolute relocations of symbols (whose address is less than Load Offset Symbol Limit) if the load file contains relocation sections.
Load Offset Symbol Limit `debug_load_file_limit` – String	If set apply the Load Offset logic to only those symbols that have addresses less than the specified limit.
Read-only Software Breakpoints `arm_target_read_only_software_breakpoints` – Enumeration	Specifies how software breakpoints set in read-only (Flash) memory are handled. Options are • **Disabled** - no software breakpoints used. • **Permanent** - software breakpoints are set permanently on download. • **Dynamic** - software breakpoints are set and cleared as required.
Read-write Software Breakpoints `arm_target_read_write_software_breakpoints` – Enumeration	Specifies software breakpoints set in read-write memory are handled. Options are • **Disabled** - no software breakpoints used. • **Permanent** - software breakpoints are set permanently on download. • **Dynamic** - software breakpoints are set and cleared as required.

Startup Completion Point `debug_startup_completion_point` – String	Specifies the point in the program where startup is complete. Software breakpoints and debugIO will be enabled after this point has been reached.
Working Directory `debug_working_directory` – DirPath	The working directory for a debug session.

JTAG Chain Options

Property	Description
JTAG Data Bits After `arm_linker_jtag_pad_post_dr` – IntegerRange	Specifies the number of bits to pad the JTAG data register after the target.
JTAG Data Bits Before `arm_linker_jtag_pad_pre_dr` – IntegerRange	Specifies the number of bits to pad the JTAG data register before the target.
JTAG Instruction Bits After `arm_linker_jtag_pad_post_ir` – IntegerRange	Specifies the number of bits to pad the JTAG instruction register with the BYPASS instruction after the target.
JTAG Instruction Bits Before `arm_linker_jtag_pad_pre_ir` – IntegerRange	Specifies the number of bits to pad the JTAG instruction register with the BYPASS instruction before the target.

Simulator Options

Property	Description
Memory Simulation Filename `arm_simulator_memory_simulation_filename` – ProjFileName	Specifies the dll that simulates the memory system. This property will have macro expansion applied to it.
Memory Simulation Parameter `arm_simulator_memory_simulation_parameter` – String	Parameter passed to the memory simulation on creation.
Memory Simulation Parameter Macros `arm_simulator_memory_simulation_parameter_macros` – StringList	Macros to apply to the parameter passed to the memory simulation on creation.
Run Target Loader `arm_simulator_run_target_loader` – Boolean	Run the target loader.

Target Control Options

Property	Description

ARM Debug Interface `arm_target_debug_interface_type` – Enumeration	Specifies the type of debug interface the target has.
Coprocessor Instruction Execution Address `arm_target_coprocessor_execute_address` – String	Specifies the address of read/write memory that the debugger can use to execute coprocessor instructions.
Debug Handler File Path `arm_target_debug_handler_file_path` – ProjFileName	The file path to the debug handler to use, this entry should be blank if no debug handler is required. This property will have macro expansion applied to it.
Debug Handler Load Address `arm_target_debug_handler_load_address` – String	The address to load the debug handler.
Do Not Use bkpt Instruction `arm_target_do_not_use_bkpt` – Boolean	Specifies that the bkpt instructions should not be used when setting software breakpoints on ARM architectures that support the instruction.
Monitor Mode Debug `arm_target_monitor_mode_debug` – Boolean	Specifies whether the debug handler is a monitor mode debug handler.
Monitor Mode Memory `arm_target_monitor_mode_memory` – Boolean	Specifies whether to use monitor mode memory accesses.
Stop CPU Using DBGRQ `arm_target_stop_cpu_using_dbgrq` – Boolean	Specifies whether the CPU should be stopped by asserting DBGRQ rather than by using breakpoints.
Target Interface Type `arm_target_interface_type` – Enumeration	Specifies the type of interface the target has.
Trace Interface `arm_target_trace_interface_type` – Enumeration	Specifies the type of trace interface the target has.
Use Debug Handler `arm_target_use_debug_handler` – Enumeration	Specifies whether to use a debug handler.

Target Loader Options

Property	Description
Applicable Loaders `arm_target_loader_applicable_loaders` – StringList	The set of target loaders that are applicable to the configuration
Can Erase All `arm_target_loader_can_erase_all` – Boolean	Loader can erase all of memory
Can Erase Range `arm_target_loader_can_erase_range` – Boolean	Loader can erase a range of memory
Can Lock All `arm_target_loader_can_lock_all` – Boolean	Loader can lock all of memory

Can Lock Range `arm_target_loader_can_lock_range` – Boolean	Loader can lock a range of memory
Can Only Download After Erase `arm_target_loader_can_only_download_after_erase` – Boolean	Loader can only download after erase
Can Only Verify With Download `arm_target_loader_can_only_verify_with_download` – Boolean	Loader can only verify with download
Can Peek `arm_target_loader_can_peek` – Boolean	Loader can peek memory
Can UnLock All `arm_target_loader_can_unlock_all` – Boolean	Loader can unlock all of memory
Can UnLock Range `arm_target_loader_can_unlock_range` – Boolean	Loader can unlock a range of memory
Default Loader `arm_target_loader_default_loader` – String	The default target loader to use for the configuration
First Loader Program Section `arm_target_loader_first_program_section` – String	The loader's first program section. This parameter is only required if the program being downloaded overwrites the loader.
Last Loader Program Section `arm_target_loader_last_program_section` – String	The loader's last program section. This parameter is only required if the program being downloaded overwrites the loader.
Loader File Path `arm_target_flash_loader_file_path` – ProjFileName	The file path to the loader, this entry should be blank if no loader program is required. This property will have macro expansion applied to it.
Loader File Type `arm_target_flash_loader_type` – Enumeration	Talk to the loader via the comms channel or via RAM.
Loader Parameter `arm_target_loader_parameter` – String	The parameter to pass to the loader on startup.
Reset After Download `arm_target_loader_reset_after_download` – Boolean	Specifies whether the target should be reset after a program has been downloaded by a loader.

Target Script Options

Property	**Description**
Attach Script `target_attach_script` – String	The script that is executed when the target is attached to.
Connect Script `target_connect_script` – String	The script that is executed when the target is connected to.

Debug Begin Script `target_debug_begin_script` – String	The script that is executed when the debugger begins a debug session.
Debug End Script `target_debug_end_script` – String	The script that is executed when the debugger ends a debug session.
Debug Interface Reset Script `target_debug_interface_reset_script` – String	The script that is executed to reset the debug interface. If not specified the default debug interface reset will be carried out instead.
Disconnect Script `target_disconnect_script` – String	The script that is executed when the target is disconnected from.
Get Part Name Script `target_get_partname_script` – String	The script that returns the part name of the connected target.
Loader Reset Script `target_loader_reset_script` – String	The script that is executed when the target is reset prior to downloading a loader program. If not specified "Reset Script" will be used instead.
Match Part Name Script `target_match_partname_script` – String	The script that matches the part name of the connected target prior to start debugging. The macro $(TARGET) is substituted with the Target project property group value.
Reset Script `target_reset_script` – String	The script that is executed when the target is reset.
Run Script `target_go_script` – String	The script that is executed when the target is run.
Stop Script `target_stop_script` – String	The script that is executed when the target is stopped.
TAP Reset Script `target_TAP_reset_script` – String	The script that is executed when the TAP is reset.
Target Extras Script `target_extras_script` – String	The script that is executed to supply extra menu entries in the targets window context menu.
Trace Initialize Script `target_trace_initialize_script` – String	The script that is executed to initialise the target trace hardware.

Externally Built Executable Project Properties

External Build Options

Property	Description
Build Command `external_build_command` – String	The command line to build the executable.
Clean Command `external_clean_command` – String	The command line to clean the executable.
Debug Symbols File `external_debug_symbols_file_name` – ProjFileName	The name of the debug symbols file. This property will have macro expansion applied to it.
Debug Symbols Load Address `external_debug_symbols_load_address` – String	The (code) address to be added to the debug symbol (code) addresses.
Executable File `external_build_file_name` – ProjFileName	The name of the externally built executable. This property will have macro expansion applied to it.
Load Address `external_load_address` – String	The address to load the externally built executable.
Load File Type `external_load_file_type` – Enumeration	The file type of the externally built executable. The options are **Detect, elf, bin, ihex, hex, tihex, srec**.
Start Address `external_start_address` – String	The address to start the externally built executable running from.

File and Folder Properties

(Information)

Property	Description
File Name `file_name` – String	The name of the file. This property will have global macro expansion applied to it. The following macros are set based on the value: $(InputDir) relative directory of file, $(InputName) file name without directory or extension, $(InputFileName) file name, $(InputExt) file name extension, $(InputPath) absolute path to the file name, $(RelInputPath) relative path from project directory to the file name.
Name `Name` – String	Names the item. The macro $(ProjectNodeName) is set to this value.
Platform `Platform` – String	Specifies the platform for the project. The macro $(Platform) is set to this value.

File Options

Property	Description
File Encoding `file_codec` – Enumeration	Specifies the encoding to use when reading and writing the file.
File Open Action `file_open_with` – Enumeration	Specifies how to open the file when it is double clicked.
File Type `file_type` – Enumeration	The type of file. Default setting uses the file extension to determine file type.
Flag `file_flag` – Enumeration	Flag which you can use to draw attention to important files in your project.

Folder Options

Property	Description
Dynamic Folder Directory `path` – DirPath	Dynamic folder directory specification.
Dynamic Folder Filter `filter` – String	Dynamic folder directory specification.
Dynamic Folder Recurse `recurse` – Boolean	Dynamic folder recurse into subdirectories.

Filter `filter` – StringList	A filter used when adding new files to the project.

Library Project Properties

Library Options

Property	Description
Library File Name `build_output_file_name` – FileName	Specifies a name to override the default library file name.

Executable Project Properties

Library Options

Property	Description
I/O Library Name `link_IOLibraryName` – Enumeration	Specifies the IO library (printf etc) to use.
Include Standard Libraries `link_include_standard_libraries` – Boolean	Specifies whether the standard libraries should be linked into your application.
Library Instruction Set `arm_library_instruction_set` – Enumeration	Specifies the instruction set variant of the libraries to link with.
Library Optimization `arm_library_optimization` – Enumeration	Specifies whether to link with libraries optimized for speed or size.
Standard Libraries Directory `link_standard_libraries_directory` – String	Specifies where to find the standard libraries
Use GCC Libraries `arm_use_gcc_libraries` – Boolean	Use GCC exception and RTTI libraries.
Use Multi Threaded Libraries `link_use_multi_threaded_libraries` – Boolean	Specifies whether to use thread safe standard libraries.

Linker Options

Property	Description
Additional Input Files `linker_additional_files` – StringList	Enables additional object and library files to be supplied to the linker.
Additional Linker Options `linker_additional_options` – StringList	Enables additional options to be supplied to the linker.
Additional Linker Options From File `linker_additional_options_from_file` – ProjFileName	Enables additional options to be supplied to the linker from a file.
Additional Output Format `linker_output_format` – Enumeration	The format used when creating an additional linked output file.
Check For Memory Segment Overflow `arm_library_check_memory_segment_overflow` – Boolean	Specifies whether the linker should check whether program sections fit in their memory segments.

DebugIO Implementation `arm_link_debugio_type` – Enumeration	Specifies which DebugIO mechanism to link in. Options are • **Breakpoint** - hardware breakpoint instruction (not available on v4t architecture) and memory locations are used. • **DCC** - ARM debug communication channel is used. • **Memory Poll** - memory locations are polled.
DebugIO Supported `linker_DebugIO_enabled` – Boolean	Is DebugIO supported.
Emit Relocations `arm_linker_emit_relocations` – Boolean	Output relocation information into the executable.
Entry Point `gcc_entry_point` – String	Specifies the entry point of the program.
Executable File Name `build_output_file_name` – FileName	Specifies a name to override the default executable file name.
Gap Fill Value `arm_linker_gap_fill` – IntegerHex	The value to fill gaps between sections.
Generate Map File `linker_map_file` – Boolean	Specifies whether to generate a linkage map file.
Keep Symbols `linker_keep_symbols` – StringList	Specifies the symbols that should be kept by the linker even if they are not reachable.
Linker `arm_linker_variant` – Enumeration	Specifies which linker to use.
Linker Symbol Definitions `link_symbol_definitions` – StringList	Specifies one or more linker symbol definitions.
Section Placement File `linker_section_placement_file` – ProjFileName	The name of the file containing section placement description.
Section Placement Macros `linker_section_placement_macros` – StringList	Macro values to substitue in section placement nodes - MACRO1=value1;MACRO2=value2.
Strip Symbols `gcc_strip_symbols` – Boolean	Specifies whether symbols should be stripped.
Suppress Warning on Mismatch `arm_linker_no_warn_on_mismatch` – Boolean	No warning on mismatched object files/libraries.
Treat Linker Warnings as Errors `arm_linker_treat_warnings_as_errors` – Boolean	Treat linker warnings as errors.

Printf/Scanf Options

Property	Description

Printf Floating Point Supported `linker_printf_fp_enabled` – Boolean	Are floating point numbers supported by the printf function group.
Printf Integer Support `linker_printf_fmt_level` – Enumeration	The largest integer type supported by the printf function group.
Printf Supported `linker_printf_enabled` – Boolean	Is printf supported.
Printf Width/Precision Supported `linker_printf_width_precision_supported` – Boolean	Enables support for width and precision specification in the printf function group.
Scanf Classes Supported `linker_scanf_character_group_matching_enabled` – Boolean	Enables support for %[...] and %[^...] character class matching in the scanf functions.
Scanf Floating Point Supported `linker_scanf_fp_enabled` – Boolean	Are floating point numbers supported by the scanf function group.
Scanf Integer Support `linker_scanf_fmt_level` – Enumeration	The largest integer type supported by the scanf function group.
Scanf Supported `linker_scanf_enabled` – Boolean	Is scanf supported.

Runtime Memory Area Options

Property	Description
Heap Size `arm_linker_heap_size` – IntegerRange	The size of the heap in bytes.
Main Stack Size `arm_linker_stack_size` – IntegerRange	The size of the main stack in bytes.
Process Stack Size `arm_linker_process_stack_size` – IntegerRange	The size of the process stack in bytes.
Stack Size (Abort Mode) `arm_linker_abt_stack_size` – IntegerRange	The size of the Abort mode stack in bytes.
Stack Size (FIQ Mode) `arm_linker_fiq_stack_size` – IntegerRange	The size of the FIQ mode stack in bytes.
Stack Size (IRQ Mode) `arm_linker_irq_stack_size` – IntegerRange	The size of the IRQ mode stack in bytes.
Stack Size (Supervisor Mode) `arm_linker_svc_stack_size` – IntegerRange	The size of the Supervisor mode stack in bytes.
Stack Size (Undefined Mode) `arm_linker_und_stack_size` – IntegerRange	The size of the Undefined mode stack in bytes.
Stack Size (User/System Mode) `arm_linker_stack_size` – IntegerRange	The size of the User/System mode stack in bytes.

User Build Step Options

Property	Description
Link Patch Command `linker_patch_build_command` – String	A command to run after the link but prior to additional binary file generation. This property will have macro expansion applied to it with the additional **$(TargetPath)** macro set to the output filepath of the linker command.
Link Patch Working Directory `linker_patch_build_command_wd` – String	The working directory where the link patch command is run. This property will have macro expansion applied to it.
Post-Link Command `linker_post_build_command` – String	A command to run after the link command has completed. This property will have macro expansion applied to it with the additional **$(TargetPath)** macro set to the output filepath of the linker command.
Post-Link Working Directory `linker_post_build_command_wd` – String	The working directory where the post-link command is run. This property will have macro expansion applied to it.
Pre-Link Command `linker_pre_build_command` – String	A command to run before the link command. This property will have macro expansion applied to it.
Pre-Link Working Directory `linker_pre_build_command_wd` – String	The working directory where the pre-link command is run. This property will have macro expansion applied to it.

Staging Project Properties

Staging Options

Property	Description
Output File Path `stage_output_filepath` – String	The output file path the stage command will create. This property will have macro expansion applied to it.
Set To Read-only `stage_set_readonly` – Boolean	Set the output file to read only or read/write.
Stage Command `stage_command` – String	The command to execute. This property will have macro expansion applied to it with the additional **$(StageOutputFilePath)** macro set to the output filepath of the stage command.
Stage Command Working Directory `stage_command_wd` – String	The working directory in which the stage command is run. This property will have macro expansion applied to it.
Stage Post-Build Command `stage_post_build_command` – String	The command to execute after staging commands have executed. This property will have macro expansion applied to it.
Stage Post-Build Command Working Directory `stage_post_build_command_wd` – String	The working directory where the post build command runs. This property will have macro expansion applied to it.

System Macros

System Macro Values

Property	Description
$(Date) `$(Date)` – String	Day Month Year e.g. 21 June 2011.
$(DateDay) `$(DateDay)` – String	Year e.g. 2011.
$(DateMonth) `$(DateMonth)` – String	Month e.g. June.
$(DateYear) `$(DateYear)` – String	Day e.g. 21.
$(DesktopDir) `$(DesktopDir)` – String	Path to users desktop directory e.g. c:/users/mpj/Desktop.
$(DocumentsDir) `$(DocumentsDir)` – String	Path to users documents directory e.g. c:/users/mpj/Documents.
$(HomeDir) `$(HomeDir)` – String	Path to users home directory e.g. c:/users/mpj.
$(HostArch) `$(HostArch)` – String	The CPU architecture that CrossStudio is running on e.g. x86.
$(HostDLL) `$(HostDLL)` – String	The file extension for dynamic link libraries on the CPU that CrossStudio is running on e.g. .dll.
$(HostEXE) `$(HostEXE)` – String	The file extension for executables on the CPU that CrossStudio is running on e.g. .exe.
$(HostOS) `$(HostOS)` – String	The name of the operating system that CrossStudio is running on e.g. win.
$(Micro) `$(Micro)` – String	The CrossWorks target e.g. ARM.
$(PackagesDir) `$(PackagesDir)` – String	Path to the users packages directory e.g. c:/users/mpj/AppData/Local/Rowley Associates Limited/CrossWorks for ARM/packages.
$(Platform) `$(Platform)` – String	The CrossWorks target e.g. arm
$(SamplesDir) `$(SamplesDir)` – String	Path to the samples subdirectory of the packages directory e.g. c:/users/mpj/AppData/Local/Rowley Associates Limited/CrossWorks for ARM/packages/samples.

$(StudioDir) `$(StudioDir)` – String	The install directory of CrossStudio e.g. c:/Program Files(x86)/Rowley Associates Limited/CrossWorks for ARM 2.0.
$(StudioMajorVersion) `$(StudioMajorVersion)` – String	The major release version of CrossStudio e.g. 2.
$(StudioMinorVersion) `$(StudioMinorVersion)` – String	The minor release version of CrossStudio e.g. 0.
$(StudioName) `$(StudioName)` – String	The name of CrossStudio e.g. CrossStudio for ARM.
$(StudioRevision) `$(StudioRevision)` – String	The release revision of CrossStudio e.g. 11.
$(StudioUserDir) `$(StudioUserDir)` – String	The directory containing the packages directory e.g. c:/users/mpj/AppData/Local/Rowley Associates Limited/CrossWorks for ARM.
$(TargetID) `$(TargetID)` – String	ID number representing the CrossWorks target.
$(TargetsDir) `$(TargetsDir)` – String	Path to the targets subdirectory of the packages directory e.g. c:/users/mpj/AppData/Local/Rowley Associates Limited/CrossWorks for ARM/packages/targets.
$(Time) `$(Time)` – String	Hour:Minutes:Seconds e.g. 15:34:03.
$(TimeHour) `$(TimeHour)` – String	Hour e.g. 15.
$(TimeMinute) `$(TimeMinute)` – String	Hour e.g. 34.
$(TimeSecond) `$(TimeSecond)` – String	Hour e.g. 03.

Build Macros

Build Macro Values

Property	Description
$(CombiningOutputFilePath) `$(CombiningOutputFilePath)` – String	The full path of the output file of the combining command.
$(CombiningRelInputPaths) `$(CombiningRelInputPaths)` – String	The relative inputs to the combining command.
$(Configuration) `$(Configuration)` – String	The build configuration e.g. ARM Flash Debug.
$(EXE) `$(EXE)` – String	The default file extension for an executable file including the dot e.g. .elf.
$(GCCTarget) `$(GCCTarget)` – String	The value of the **GCC Target** project property.
$(InputDir) `$(InputDir)` – String	The absolute directory of the input file.
$(InputExt) `$(InputExt)` – String	The extension of an input file not including the dot e.g cpp.
$(InputFileName) `$(InputFileName)` – String	The name of an input file relative to the project directory.
$(InputName) `$(InputName)` – String	The name of an input file relative to the project directory without the extension.
$(InputPath) `$(InputPath)` – String	The absolute name of an input file including the extension.
$(IntDir) `$(IntDir)` – String	The macro-expanded value of the **Intermediate Directory** project property.
$(LIB) `$(LIB)` – String	The default file extension for a library file including the dot e.g. .lib.
$(LibExt) `$(LibExt)` – String	The architecture and build specific library extension.
$(LoaderType) `$(LoaderType)` – String	The loader type specific extension, **dcc**, **ram** or **rpc**.
$(OBJ) `$(OBJ)` – String	The default file extension for an object file including the dot e.g. .o.
$(OutDir) `$(OutDir)` – String	The macro-expanded value of the **Output Directory** project property.
$(PackageExt) `$(PackageExt)` – String	The file extension of a CrossWorks package file e.g. hzq.

$(ProjectDir) `$(ProjectDir)` – String	The absolute value of the **Project Directory** project property of the current project. If this isn't set then the directory containing the solution file.
$(ProjectName) `$(ProjectName)` – String	The project name of the current project.
$(ProjectNodeName) `$(ProjectNodeName)` – String	The name of the selected project node.
$(RelInputPath) `$(RelInputPath)` – String	The relative path of the input file to the project directory.
$(RootOutDir) `$(RootOutDir)` – String	The macro-expanded value of the **Root Output Directory** project property.
$(RootRelativeOutDir) `$(RootRelativeOutDir)` – String	The relative path to get from the path specified by the **Output Directory** project property to the path specified by the **Root Output Directory** project property.
$(SolutionDir) `$(SolutionDir)` – String	The absolute path of the directory containing the solution file.
$(SolutionExt) `$(SolutionExt)` – String	The extension of the solution file without the dot.
$(SolutionFileName) `$(SolutionFileName)` – String	The filename of the solution file.
$(SolutionName) `$(SolutionName)` – String	The basename of the solution file.
$(SolutionPath) `$(SolutionPath)` – String	The absolute path of the solution file.
$(StageOutputFilePath) `$(StageOutputFilePath)` – String	The full path of the output file of the stage command.
$(TargetPath) `$(TargetPath)` – String	The full path of the output file of the link or compile command.

CWSys

The following table lists the CWSys object's member functions.

CWSys.copyFile(srcPath, destPath) copies file *srcPath* to *destPath*.
CWSys.crc32(array) returns the CRC-32 checksum of the byte array *array*. This function computes a CRC-32 checksum on a block of data using the standard CRC-32 polynomial (0x04C11DB7) with an initial value of 0xFFFFFFFF. Note that this implementation doesn't reflect the input or the output and the result is inverted.
CWSys.fileExists(path) returns true if file *path* exists.
CWSys.fileSize(path) return the number of bytes in file *path*.
CWSys.getRunStderr() returns the stderr output from the last *CWSys.run()* call.
CWSys.getRunStdout() returns the stdout output from the last *CWSys.run()* call.
CWSys.makeDirectory(path) create the directory *path*.
CWSys.packU32(array, offset, number, le) packs *number* into the *array* at *offset*.
CWSys.readByteArrayFromFile(path) returns the byte array contained in the file *path*.
CWSys.readStringFromFile(path) returns the string contained in the file *path*.
CWSys.removeDirectory(path) remove the directory *path*.
CWSys.removeFile(path) deletes file *path*.
CWSys.renameFile(oldPath, newPath) renames file *oldPath* to be *newPath*.
CWSys.run(cmd, wait) runs command line *cmd* optionally waits for it to complete if *wait* is true.
CWSys.unpackU32(array, offset, le) returns the number unpacked from the *array* at *offset*.
CWSys.writeByteArrayToFile(path, array) creates a file *path* containing the byte array *array*.
CWSys.writeStringToFile(path, string) creates a file *path* containing *string*.

Debug

The following table lists the Debug object's member functions.

Debug.breakexpr(expression, count, hardware) set a breakpoint on *expression*, with optional ignore *count* and use *hardware* parameters. Return the, none zero, allocated breakpoint number.
Debug.breakline(filename, linenumber, temporary, count, hardware) set a breakpoint on *filename* and *linenumber*, with optional *temporary*, ignore *count* and use *hardware* parameters. Return the, none zero, allocated breakpoint number.
Debug.breaknow() break execution now.
Debug.deletebreak(number) delete the specified breakpoint or all breakpoints if zero is supplied.
Debug.disassembly(source, labels, before, after) set debugger mode to disassembly mode. Optionally specify *source* and *labels* to be displayed and the number of bytes to disassemble *before* and *after* the located program counter.
Debug.echo(s) display string.
Debug.enableexception(exception, enable) *enable* break on *exception*.
Debug.evaluate(expression) evaluates debug *expression* and returns it as a JavaScript value.
Debug.getfilename() return located filename.
Debug.getlineumber() return located linenumber.
Debug.go() continue execution.
Debug.locate(frame) locate the debugger to the optional *frame* context.
Debug.locatepc(pc) locate the debugger to the specified *pc*.
Debug.locateregisters(registers) locate the debugger to the specified *register* context.
Debug.print(expression, fmt) evaluate and display debug*expression* using optional *fmt*. Supported formats are *b* binary, *c* character, *d* decimal, *e* scientific float, *f* decimal float, *g* scientific or decimal float, *i* signed decimal, *o* octal, *p* pointer value, *s* null terminated string, *u* unsigned decimal, *x* hexadecimal.
Debug.printglobals() display global variables.
Debug.printlocals() display local variables.
Debug.quit() stop debugging.
Debug.setprintarray(elements) set the maximum number of array elements for printing variables.
Debug.setprintradix(radix) set the default radix for printing variables.
Debug.setprintstring(c) set the default to print character pointers as strings.
Debug.showbreak(number) show information on the specified breakpoint or all breakpoints if zero is supplied.
Debug.showexceptions() show the exceptions.
Debug.source(before, after) set debugger mode to source mode. Optionally specify the number of source lines to display *before* and *after* the location.

Debug.stepinto() step an instruction or a statement.
Debug.stepout() continue execution and break on return from current function.
Debug.stepover() step an instruction or a statement stepping over function calls.
Debug.stopped() return stopped state.
Debug.wait(ms) wait *ms* millseconds for a breakpoint and return the number of the breakpoint that hit.
Debug.where() display call stack.

ElfFile

The following table lists the ElfFile object's member functions.

ElfFile.crc32(address, length, virtualNotPhysical, padding) returns the CRC-32 checksum of an address range *length* bytes long, located at *address*. If *virtualNotPhysical* is true or undefined, *address* is a virtual address otherwise it is a physical address. If *padding* is defined, it specifies the byte value used to fill gaps in the program. This function computes a CRC-32 checksum on a block of data using the standard CRC-32 polynomial (0x04C11DB7) with an initial value of 0xFFFFFFFF. Note that this implementation doesn't reflect the input or the output and the result is inverted.
ElfFile.findProgram(address) returns an object with *start*, the *data* and the *size* to allocate of the Elf program that contains *address*.
ElfFile.getEntryPoint() returns the entry point in the ELF file.
ElfFile.getSection(name) returns an object with *start* and the *data* of the Elf section corresponding to the *name*.
ElfFile.isLittleEndian() returns true if the Elf file has numbers encoded as little endian.
ElfFile.load(path) loads Elf file from *path*.
ElfFile.peekBytes(address, length, virtualNotPhysical, padding) returns byte array containing *length* bytes peeked from *address*. If *virtualNotPhysical* is true or undefined, *address* is a virtual address otherwise it is a physical address. If *padding* is defined, it specifies the byte value used to fill gaps in the program.
ElfFile.peekUint32(address, virtualNotPhysical) returns a 32-bit word peeked from *address*. If *virtualNotPhysical* is true or undefined, *address* is a virtual address otherwise it is a physical address.
ElfFile.pokeBytes(address, byteArray, virtualNotPhysical) poke byte array *byteArray* to *address*. If *virtualNotPhysical* is true or undefined, *address* is a virtual address otherwise it is a physical address.
ElfFile.pokeUint32(address, value, virtualNotPhysical) poke a *value* to 32-bit word located at *address*. If *virtualNotPhysical* is true or undefined, *address* is a virtual address otherwise it is a physical address.
ElfFile.save(path) saves Elf file to *path*.
ElfFile.symbolValue(symbol) returns the value of *symbol* in Elf file.

TargetInterface

The following table lists the TargetInterface object's member functions.

TargetInterface.beginDebugAccess() puts the target into debug state if it is not already in order to carry out a number of debug operations. The idea behind beginDebugAccess and endDebugAccess is to minimize the number of times the target enters and exits debug state when carrying out a number of debug operations. Target interface functions that require the target to be in debug state (such as peek and poke) also use beginDebugAccess and endDebugAccess to get the target into the correct state. A nesting count is maintained, incremented by beginDebugAccess and decremented by endDebugAccess. The initial processor state is recorded on the first nested call to beginDebugAccess and this state is restored when the final endDebugAccess is called causing the count to return to it initial state.
TargetInterface.commReadWord() returns a word from the ARM7/ARM9 debug comms channel.
TargetInterface.commWriteWord(word) writes a word to the ARM7/ARM9 debug comms channel.
TargetInterface.cycleTCK(n) provide n TCK clock cycles.
TargetInterface.delay(ms) waits for ms milliseconds
TargetInterface.downloadDebugHandler() downloads the debug handler as specified by the Debug Handler File Path/Load Address project properties and uses the debug handler for the target connection.
TargetInterface.endDebugAccess(alwaysRun) restores the target run state recorded at the first nested call to beginDebugAccess. See beginDebugAccess for more information. If alwaysRun is non-zero the processor will exit debug state on the last nested call to endDebugAccess.
TargetInterface.eraseBytes(address,length) erases a length block of target memory starting at address.
TargetInterface.error(message) terminates execution of the script and outputs a target interface error message to the target log.
TargetInterface.executeFunction(address, parameter, timeout) calls a function at address with the parameter and returns the function result. The timeout is in milliseconds.
TargetInterface.executeMCR(opcode) interprets/executes the opcode assuming it to be an MRC instruction and returns the value of the specified coprocessor register.
TargetInterface.executeMCR(opcode, value) interprets/executes the opcode assuming it to be an MCR instruction that writes value to the specified coprocessor register.
TargetInterface.fillScanChain(bool, lsb, msb) sets bits from lsb (least significant bit) to msb (most significant bit) in internal buffer to bool value.
TargetInterface.getDebugRegister(address) returns the value of the ADIv5 debug register denoted by address. Address has the nibble sized access point number starting at bit 24 and the register number in the bottom byte.
TargetInterface.getICEBreakerRegister(r) returns the value of the ARM7/ARM9/ARM11/CortexA/CortexR debug register r.
TargetInterface.getProjectProperty(savename) returns the value of the savename project property.

TargetInterface.getRegister(registername) returns the value of the register, register is a string specifying the register to get and must be one of r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, sp, lr, pc, cpsr, r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq, r13_fiq, r14_fiq, spsr_fiq, r13_svc, r14_svc, spsr_svc, r13_abt, r14_abt, spsr_abt, r13_irq, r14_irq, spsr_irq, r13_und, r14_und, spsr_und.
TargetInterface.getTDO() return the TDO signal.
TargetInterface.getTargetProperty(savename) returns the value of the savename target property.
TargetInterface.go() allows the target to run.
TargetInterface.idcode() returns the JTAG idcode of the target.
TargetInterface.implementation() returns a string defining the target interface implementation.
TargetInterface.isStopped() returns true if the target is stopped.
TargetInterface.message(message) outputs a target interface message to the target log.
TargetInterface.packScanChain(data, lsb, msb) packs data from lsb (least significant bit) to msb (most significant bit) into internal buffer.
TargetInterface.peekBinary(address, length, filename) reads a block of bytes from target memory starting at address for length bytes and writes them to filename.
TargetInterface.peekByte(address) reads a byte of target memory from address and returns it.
TargetInterface.peekBytes(address, length) reads a block of bytes from target memory starting at address for length bytes and returns the result as an array containing the bytes read.
TargetInterface.peekMultUint16(address, length) reads length unsigned 16-bit integers from target memory starting at address and returns them as an array.
TargetInterface.peekMultUint32(address, length) reads length unsigned 32-bit integers from target memory starting at address and returns them as an array.
TargetInterface.peekUint16(address) reads a 16-bit unsigned integer from target memory from address and returns it.
TargetInterface.peekUint32(address) reads a 32-bit unsigned integer from target memory from address and returns it.
TargetInterface.peekWord(address) reads a word as an unsigned integer from target memory from address and returns it.
TargetInterface.pokeBinary(address, filename) reads a block of bytes from filename and writes them to target memory starting at address.
TargetInterface.pokeByte(address, data) writes the byte data to address in target memory.
TargetInterface.pokeBytes(address, data) writes the array data containing 8-bit data to target memory at address.
TargetInterface.pokeMultUint16(address, data) writes the array data containing 16-bit data to target memory at address.
TargetInterface.pokeMultUint32(address, data) writes the array data containing 32-bit data to target memory at address.
TargetInterface.pokeUint16(address, data) writes data as a 16-bit value to address in target memory.

TargetInterface.pokeUint32(address, data) writes data as a 32-bit value to address in target memory.
TargetInterface.pokeWord(address, data) writes data as a word value to address in target memory.
TargetInterface.readBinary(filename) reads a block of bytes from filename and returns them in an array.
TargetInterface.reset() resets the target, optionally executes the reset script and lets the target run.
TargetInterface.resetAndStop(delay) resets the target by cycling nSRST and then stops the target. delay is the number of milliseconds to hold the target in reset.
TargetInterface.resetAndStopAtZero(delay) sets a breakpoint on the instruction at address zero execution, resets the target by cycling nSRST and waits for the breakpoint to be hit. delay is the number of milliseconds to hold the target in reset.
TargetInterface.resetDebugInterface() resets the target interface (not the target).
TargetInterface.runFromAddress(address, timeout) start the target executing at address and waits for a breakpoint to be hit. The timeout is in milliseconds.
TargetInterface.runTestIdle() moves the target JTAG state machine into Run-Test/Idle state
TargetInterface.runToAddress(address, timeout) sets a breakpoint at address, starts the target executing and waits for the breakpoint to be hit. The timeout is in milliseconds.
TargetInterface.scanDR(length, count) scans length bits from the internal buffer into the data register and puts the result into the internal buffer (count specifies the number of times the function is done).
TargetInterface.scanIR(length, count) scans length bits from the internal buffer into the instruction register and puts the result into the internal buffer (count specifies the number of times the function is done).
TargetInterface.selectDevice(irPre, irPost, drPre, drPost) sets the instruction and data register (number of devices) pre and post bits.
TargetInterface.setDBGRQ(v) sets/clears the DBGRQ bit of the ARM7/ARM9 debug control register.
TargetInterface.setDebugInterfaceProperty("reset_delay", N) set the XScale reset delay property to N. **TargetInterface.setDebugInterfaceProperty**("post_reset_delay", N) set the XScale post reset delay property to N. **TargetInterface.setDebugInterfaceProperty**("post_reset_cycles", N) set the XScale post reset cycles property to N. **TargetInterface.setDebugInterfaceProperty**("post_ldic_cycles", N) set the XScale ldic cycles property to N. **TargetInterface.setDebugInterfaceProperty**("sync_exception_vectors", bool) turn on/off the XScale sync exception vectors property. **TargetInterface.setDebugInterfaceProperty**("peek_flash_workaround", bool) turn on/off the ARMv6M/ ARMv7M peek flash memory workaround debug property. **TargetInterface.setDebugInterfaceProperty**("adiv5_fast_delay_cycles", N) set the ADIv5 fast delay cycles property to N (FTDI2232 target interfaces only). **TargetInterface.setDebugInterfaceProperty**("use_adiv5_AHB", bool) set the ARMv7A/ARMv7R debug property to turn on/off usage of the ADIv5 AHB MEM-AP. **TargetInterface.setDebugInterfaceProperty**("max_ap_num", N) set the ADIv5 debug property to limit the number of AP's to detect to N. **TargetInterface.setDebugInterfaceProperty**("component_base", N) set the ARMv7A/ARMv7R debug property that specifies the base address N of the CoreSight debug component.

TargetInterface.setDebugRegister(address, value) set the value of the ADIv5 debug register denoted by address. Address has the nibble sized access point number starting at bit 24 and the register number in the bottom byte.
TargetInterface.setDeviceTypeProperty(type) sets the target interface's Device Type property string to type. This would typically be used by a Connect Script to override the default Device Type property and provide a custom description of the connected target.
TargetInterface.setICEBreakerBreakpoint(n, address, addressMask, data, dataMask, control, controlMask) sets the ARM7/ARM9 watchpoint n registers.
TargetInterface.setICEBreakerRegister(r, value) set the value of the ARM7/ARM9/ARM11/CortexA/CortexR debug register r.
TargetInterface.setMaximumJTAGFrequency(hz) allows the maximum TCK frequency of the currently connected JTAG interface to be set dynamically. The speed setting will only apply for the current connection session, if you reconnect the setting will revert to the speed specfied by the target interface properties. Calls to this function will be ignored if adaptive clocking is being used.
TargetInterface.setNSRST(v) sets/clears the NSRST signal.
TargetInterface.setNTRST(v) sets/clears the NTRST signal.
TargetInterface.setRegister(registername, value) sets the register to the value, register is a string specifying the register to get and must be one of r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, sp, lr, pc, cpsr, r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq, r13_fiq, r14_fiq, spsr_fiq, r13_svc, r14_svc, spsr_svc, r13_abt, r14_abt, spsr_abt, r13_irq, r14_irq, spsr_irq, r13_und, r14_und, spsr_und.
TargetInterface.setTDI(v) clear/set TDI signal.
TargetInterface.setTMS(v) clear/set TMS signal.
TargetInterface.setTargetProperty(savename) set the value of the savename target property.
TargetInterface.stop() stops the target.
TargetInterface.stopAndReset(delay) sets a breakpoint on any instruction execution, resets the target by cycling nSRST and waits for the breakpoint to be hit. delay is the number of milliseconds to hold the device in reset.
TargetInterface.trst() resets the target interface (not the target).
TargetInterface.type() returns a string defining the target interface type.
TargetInterface.unpackScanChain(lsb, msb) unpacks data from lsb (least significant bit) to msb (most significant bit) from internal buffer and returns the result.
TargetInterface.waitForDebugState(timeout) waits for the target to stop or the timeout in milliseconds.
TargetInterface.writeBinary(array, filename) write the bytes in array to filename.

WScript

The following table lists the WScript object's member functions.

WScript.Echo(s) echos string *s* to the output terminal.