



CrossWorks for ARM Reference Manual

Version: 5.0.0.2024031200.56020



Contents

Introduction	33
What is CrossWorks?	34
What we don't tell you	36
Activating your product	37
Text conventions	39
Additional resources	41
Highlights	42
Release notes	44
CrossStudio Tutorial	45
Activating CrossWorks	47
Managing support packages	49
Creating a project	52
Managing files in a project	58
Setting project options	62
Building projects	64
Exploring projects	67
Using the debugger	78
Low-level debugging	84
Debugging externally built applications	88
CrossStudio User Guide	93
CrossStudio standard layout	94
Menu bar	95
Title bar	96

Status bar	97
Editing workspace	99
Docking windows	100
Dashboard	101
CrossStudio help and assistance	102
Creating and managing projects	104
Solutions and projects	105
Creating a project	108
Adding existing files to a project	109
Adding new files to a project	110
Removing a file, folder, project, or project link	111
Building your application	112
Creating variants using configurations	114
Project properties	116
Configurations and property values	118
Project macros	120
Dependencies and build order	122
Precompile Header File support	123
Linking and section placement	124
Using source control	127
Source control capabilities	128
Configuring source-control providers	129
Connecting to the source-control system	130
File source-control status	131
Source-control operations	132
Adding files to source control	133
Updating files	134
Committing files	135
Reverting files	136
Locking files	137
Unlocking files	138
Removing files from source control	139
Showing differences between files	140
Source-control properties	141
Subversion provider	142
CVS provider	144
Package management	146
Exploring your application	150
Project explorer	151
Source navigator window	156
References window	158

Symbol browser window	159
Stack usage window	164
Memory usage window	165
Bookmarks window	168
Code Outline Window	169
Analyzing Source Code	170
Editing your code	171
Basic editing	172
Moving the insertion point	173
Adding text	175
Deleting text	176
Using the clipboard	177
Undo and redo	178
Drag and drop	179
Searching	180
Advanced editing	181
Indenting source code	182
Commenting out sections of code	184
Adjusting letter case	185
Using bookmarks	186
Find and Replace window	188
Clipboard Ring window	190
Mouse-click accelerators	192
Regular expressions	194
Debugging windows	196
Locals window	196
Globals window	198
Watch window	200
Register window	203
Memory window	206
Breakpoints window	210
Call Stack window	214
Threads window	217
Execution Profile window	221
Execution Trace window	222
Debug file search editor	223
Debug Terminal window	225
Debug Immediate window	226
Breakpoint expressions	227
Debug expressions	228
Utility windows	229

Output window	229
Properties window	230
Targets window	231
Terminal emulator window	235
Script Console window	236
Downloads window	237
Latest News window	238
Command-line options	239
-D (Define macro)	240
-noclang (Disable Clang support)	241
-noload (Disable loading of last project)	242
-packagesdir (Specify packages directory)	243
-permit-multiple-studio-instances (Permit multiple studio instances)	244
-rootuserdir (Set the root user data directory)	245
-save-settings-off (Disable saving of environment settings)	246
-set-setting (Set environment setting)	247
-templatesfile (Set project templates path)	248
Uninstalling CrossWorks for ARM	249
Uninstalling from Windows	249
Uninstalling from macOS	250
Uninstalling from Linux	251
ARM target support	253
Target startup code	255
Startup code	257
Section Placement	260
Project configurations	262
Target script file	265
Program loading	269
Debug Capabilities	270
Trace Capabilities	273
Target interfaces	277
ARM Simulator target interface	279
Amontec JTAGkey Target Interface	280
CMSIS-DAP Target Interface	282
CrossConnect Target Interface	284
Generic FT2232 Target Interface	286
Generic Target Interface	289
Olimex ARM-USB-OCD Target Interface	290
Kinetis OSJTAG Target Interface	292
P&E UNIT Interface DLL Target Interface	293
Segger J-Link Target Interface	294

Stellaris ICDI Target Interface	296
ST-LINK Target Interface	297
ST-LINK/V2 Target Interface	298
Macraigor Wiggler (20 and 14 pin) Target Interface	299
Using an external ARM GCC toolchain	301
C Library User Guide	303
Floating point	304
Multithreading	305
Thread safety in the CrossWorks library	306
Implementing mutual exclusion in the C library	307
Input and output	309
Customizing putchar	310
Locales	315
Unicode, ISO 10646, and wide characters	316
Multi-byte characters	317
The standard C and POSIX locales	318
Additional locales in source form	319
Installing a locale	320
Setting a locale directly	322
Complete API reference	323
<assert.h>	325
__assert	326
assert	327
<complex.h>	328
cabs	330
cabsf	331
cacos	332
cacosf	333
cacosh	334
cacoshf	335
carg	336
cargf	337
casin	338
casinf	339
casinh	340
casinhf	341
catan	342
catanf	343
catanh	344
catanhf	345
ccos	346

ccosf	347
ccosh	348
ccoshf	349
cexp	350
cexpf	351
cimag	352
cimagf	353
clog	354
clogf	355
conj	356
conjf	357
cpow	358
cpowf	359
cproj	360
cprojf	361
creal	362
crealf	363
csin	364
csinf	365
csinh	366
csinhf	367
csqrt	368
csqrtf	369
ctan	370
ctanf	371
ctanh	372
ctanhf	373
<ctype.h>	374
isalnum	376
isalnum_l	377
isalpha	378
isalpha_l	379
isblank	380
isblank_l	381
iscntrl	382
iscntrl_l	383
isdigit	384
isdigit_l	385
isgraph	386
isgraph_l	387
islower	388

islower_l	389
isprint	390
isprint_l	391
ispunct	392
ispunct_l	393
isspace	394
isspace_l	395
isupper	396
isupper_l	397
isxdigit	398
isxdigit_l	399
tolower	400
tolower_l	401
toupper	402
toupper_l	403
<debugio.h>	404
debug_abort	407
debug_break	408
debug_clearerr	409
debug_clock	410
debug_enabled	411
debug_evaluate	412
debug_exit	413
debug_fclose	414
debug_feof	415
debug_ferror	416
debug_fflush	417
debug_fgetc	418
debug_fgetpos	419
debug_fgets	420
debug_filesize	421
debug_fopen	422
debug_fprintf	423
debug_fprintf_c	424
debug_fputc	425
debug_fputs	426
debug_fread	427
debug_freopen	428
debug_fscanf	429
debug_fscanf_c	430
debug_fseek	431

debug_fsetpos	432
debug_ftell	433
debug_fwrite	434
debug_getargs	435
debug_getch	436
debug_getchar	437
debug_getd	438
debug_getenv	439
debug_getf	440
debug_geti	441
debug_getl	442
debug_getll	443
debug_gets	444
debug_getu	445
debug_getul	446
debug_getull	447
debug_kbhit	448
debug_loadsymbols	449
debug_perror	450
debug_printf	451
debug_printf_c	452
debug_putchar	453
debug_puts	454
debug_remove	455
debug_rename	456
debug_rewind	457
debug_runtime_error	458
debug_scanf	459
debug_scanf_c	460
debug_system	461
debug_time	462
debug_tmpfile	463
debug_tmpnam	464
debug_ungetc	465
debug_unloadsymbols	466
debug_vfprintf	467
debug_vfscanf	468
debug_vprintf	469
debug_vscanf	470
<errno.h>	471
EDOM	472

EILSEQ	473
EINVAL	474
ENOMEM	475
ERANGE	476
errno	477
<float.h>	478
DBL_DIG	479
DBL_EPSILON	480
DBL_MANT_DIG	481
DBL_MAX	482
DBL_MAX_10_EXP	483
DBL_MAX_EXP	484
DBL_MIN	485
DBL_MIN_10_EXP	486
DBL_MIN_EXP	487
DECIMAL_DIG	488
FLT_DIG	489
FLT_EPSILON	490
FLT_EVAL_METHOD	491
FLT_MANT_DIG	492
FLT_MAX	493
FLT_MAX_10_EXP	494
FLT_MAX_EXP	495
FLT_MIN	496
FLT_MIN_10_EXP	497
FLT_MIN_EXP	498
FLT_RADIX	499
FLT_ROUNDS	500
<intrinsics.h>	501
__breakpoint	507
__cdp	508
__cdp2	509
__clrex	510
__clz	511
__dbg	512
__disable_fiq	513
__disable_interrupt	514
__disable_irq	515
__dmb	516
__dsb	517
__enable_fiq	518

__enable_interrupt	519
__enable_irq	520
__fabs	521
__fabsf	522
__fma	523
__fmaf	524
__get_APSR	525
__get_BASEPRI	526
__get_CONTROL	527
__get_CPSR	528
__get_FAULTMASK	529
__get_PRIMASK	530
__isb	531
__ldc	532
__ldc2	533
__ldc2_noidx	534
__ldc2l	535
__ldc2l_noidx	536
__ldc_noidx	537
__ldcl	538
__ldcl_noidx	539
__ldrbt	540
__ldrex	541
__ldrex_b	542
__ldrex_d	543
__ldrex_h	544
__ldrht	545
__ldrsbt	546
__ldrsht	547
__ldrt	548
__mcr	549
__mcr2	550
__mcrr	551
__mcrr2	552
__mrc	553
__mrc2	554
__mrrc	555
__mrrc2	556
__nop	557
__pld	558
__pli	559

__qadd	560
__qadd16	561
__qadd8	562
__qasx	563
__qdadd	564
__qdbl	565
__qdsb	566
__qflag	567
__qsax	568
__qsub	569
__qsub16	570
__qsub8	571
__rbit	572
__rev	573
__rev16	574
__revsh	575
__rintn	576
__rintnf	577
__sadd16	578
__sadd8	579
__sasx	580
__sel	581
__set_APSR	582
__set_BASEPRI	583
__set_CONTROL	584
__set_CPSR	585
__set_FAULTMASK	586
__set_PRIMASK	587
__sev	588
__shadd16	589
__shadd8	590
__shasx	591
__shsax	592
__shsub16	593
__shsub8	594
__smlabb	595
__smlabt	596
__smlad	597
__smladx	598
__smlalbb	599
__smlalbt	600

__smlald	601
__smlaldx	602
__smlaltb	603
__smlaltt	604
__smlatb	605
__smlatt	606
__smlawb	607
__smlawt	608
__smlsd	609
__smlsdx	610
__smlsld	611
__smlsldx	612
__smuad	613
__smuadx	614
__smulbb	615
__smulbt	616
__smultb	617
__smultt	618
__smulwb	619
__smulwt	620
__smusd	621
__smusdx	622
__sqrt	623
__sqrtf	624
__ssat	625
__ssat16	626
__ssax	627
__ssub16	628
__ssub8	629
__stc	630
__stc2	631
__stc2l	632
__stc_noidx	633
__stcl	634
__strbt	635
__strex	636
__strexb	637
__strexh	638
__strexh	639
__strht	640
__strt	641

__swp	642
__swpb	643
__sxtab16	644
__sxtb16	645
__uadd16	646
__uadd8	647
__uasx	648
__uhadd16	649
__uhadd8	650
__uhasx	651
__uhsax	652
__uhsub16	653
__uhsub8	654
__uqadd16	655
__uqadd8	656
__uqasx	657
__uqsax	658
__uqsub16	659
__uqsub8	660
__usad8	661
__usad8a	662
__usat	663
__usat16	664
__usax	665
__usub8	666
__uxtab16	667
__uxtb16	668
__wfe	669
__wfi	670
__yield	671
<iso646.h>	672
and	673
and_eq	674
bitand	675
bitor	676
compl	677
not	678
not_eq	679
or	680
or_eq	681
xor	682

xor_eq	683
<itm.h>	684
ITM_base	685
ITM_channel_enabled	686
ITM_send_byte	687
ITM_send_half_word	688
ITM_send_pc	689
ITM_send_word	690
<libarm.h>	691
libarm_dcc_read	692
libarm_dcc_write	693
libarm_disable_fiq	694
libarm_disable_irq	695
libarm_disable_irq_fiq	696
libarm_enable_fiq	697
libarm_enable_irq	698
libarm_enable_irq_fiq	699
libarm_get_cpsr	700
libarm_isr_disable_irq	701
libarm_isr_enable_irq	702
libarm_mmu_flat_initialise_level_1_table	703
libarm_mmu_flat_initialise_level_2_small_page_table	704
libarm_mmu_flat_set_level_1_cacheable_region	705
libarm_mmu_flat_set_level_2_small_page_cacheable_region	706
libarm_restore_irq_fiq	707
libarm_run_dcc_port_server	708
libarm_set_cpsr	709
libarm_set_fiq	710
libarm_set_irq	711
<limits.h>	712
CHAR_BIT	713
CHAR_MAX	714
CHAR_MIN	715
INT_MAX	716
INT_MIN	717
LLONG_MAX	718
LLONG_MIN	719
LONG_MAX	720
LONG_MIN	721
MB_LEN_MAX	722
SCHAR_MAX	723

SCHAR_MIN	724
SHRT_MAX	725
SHRT_MIN	726
UCHAR_MAX	727
UINT_MAX	728
ULLONG_MAX	729
ULONG_MAX	730
USHRT_MAX	731
<locale.h>	732
lconv	733
localeconv	735
setlocale	736
<math.h>	737
acos	741
acosf	742
acosh	743
acoshf	744
asin	745
asinf	746
asinh	747
asinhf	748
atan	749
atan2	750
atan2f	751
atanf	752
atanh	753
atanhf	754
cbrt	755
cbrtf	756
ceil	757
ceilf	758
copysign	759
copysignf	760
cos	761
cosf	762
cosh	763
coshf	764
erf	765
erfc	766
erfcf	767
erff	768

exp	769
exp2	770
exp2f	771
expf	772
expm1	773
expm1f	774
fabs	775
fabsf	776
fdim	777
fdimf	778
floor	779
floorf	780
fma	781
fmaf	782
fmax	783
fmaxf	784
fmin	785
fminf	786
fmod	787
fmodf	788
fpclassify	789
frexp	790
frexpf	791
hypot	792
hypotf	793
ilogb	794
ilogbf	795
isfinite	796
isgreater	797
isgreaterequal	798
isinf	799
isless	800
islessequal	801
islessgreater	802
isnan	803
isnormal	804
isunordered	805
ldexp	806
ldexpf	807
lgamma	808
lgammaf	809

llrint	810
llrintf	811
llround	812
llroundf	813
log	814
log10	815
log10f	816
log1p	817
log1pf	818
log2	819
log2f	820
logb	821
logbf	822
logf	823
lrint	824
lrintf	825
lround	826
lroundf	827
modf	828
modff	829
nearbyint	830
nearbyintf	831
nextafter	832
nextafterf	833
pow	834
powf	835
remainder	836
remainderf	837
remquo	838
remquof	839
rint	840
rintf	841
round	842
roundf	843
scalbn	844
scalbnf	845
scalbn	846
scalbnf	847
signbit	848
sin	849
sinf	850

sinh	851
sinhf	852
sqrt	853
sqrtf	854
tan	855
tanf	856
tanh	857
tanhf	858
tgamma	859
tgammaf	860
trunc	861
truncf	862
<setjmp.h>	863
longjmp	864
setjmp	865
<stdarg.h>	866
va_arg	867
va_copy	868
va_end	869
va_start	870
<stddef.h>	871
NULL	872
max_align_t	873
offsetof	874
ptrdiff_t	875
size_t	876
<stdio.h>	877
getchar	878
gets	879
printf	880
putchar	885
puts	886
scanf	887
snprintf	891
sprintf	892
sscanf	893
vprintf	894
vscanf	895
vsnprintf	896
vsprintf	897
vsscanf	898

<stdlib.h>	899
EXIT_FAILURE	901
EXIT_SUCCESS	902
MB_CUR_MAX	903
RAND_MAX	904
abs	905
atexit	906
atof	907
atoi	908
atol	909
atoll	910
bsearch	911
calloc	912
div	913
div_t	914
exit	915
free	916
itoa	917
labs	918
ldiv	919
ldiv_t	920
llabs	921
lldiv	922
lldiv_t	923
ltoa	924
ltoa	925
malloc	926
mblen	927
mblen_l	928
mbstowcs	929
mbstowcs_l	930
mbtowc	931
mbtowc_l	932
qsort	933
rand	934
realloc	935
srand	936
strtod	937
strtof	938
strtol	939
strtoll	941

strtol	943
strtol	945
ulltoa	947
ultoa	948
utoa	949
<string.h>	950
memccpy	952
memchr	953
memcmp	954
memcpy	955
memcpy_fast	956
memmove	957
memcpy	958
memset	959
strcasestr	960
strcasestr	961
strcat	962
strchr	963
strcmp	964
strcpy	965
strcspn	966
strdup	967
strerror	968
strlcat	969
strncpy	970
strlen	971
strncat	972
strncat	973
strncat	974
strnchr	975
strncmp	976
strncpy	977
strndup	978
strnlen	979
strnstr	980
strpbrk	981
strrchr	982
strsep	983
strspn	984
strstr	985
strtok	986

strtok_r	987
<time.h>	988
asctime	989
asctime_r	990
clock_t	991
ctime	992
ctime_r	993
difftime	994
gmtime	995
gmtime_r	996
localtime	997
localtime_r	998
mktime	999
strftime	1000
time_t	1002
tm	1003
<wchar.h>	1004
WCHAR_MAX	1006
WCHAR_MIN	1007
WEOF	1008
btowc	1009
btowc_l	1010
mbrlen	1011
mbrlen_l	1012
mbrtowc	1013
mbrtowc_l	1014
mbsrtowcs	1015
mbsrtowcs_l	1016
msbinit	1017
wchar_t	1018
wrtomb	1019
wrtomb_l	1020
wcscat	1021
wcschr	1022
wcscmp	1023
wcscpy	1024
wcscspn	1025
wcsdup	1026
wcslen	1027
wcsncat	1028
wcsnchr	1029

wcsncmp	1030
wcsncpy	1031
wcsnlen	1032
wcsnstr	1033
wcspbrk	1034
wcsrchr	1035
wcsspn	1036
wcsstr	1037
wcstok	1038
wcstok_r	1039
wctob	1040
wctob_l	1041
wint_t	1042
wmemccpy	1043
wmemchr	1044
wmemcmp	1045
wmemcpy	1046
wmemmove	1047
wmempcpy	1048
wmemset	1049
wstrsep	1050
<wctype.h>	1051
iswalnum	1053
iswalnum_l	1054
iswalpha	1055
iswalpha_l	1056
iswblank	1057
iswblank_l	1058
iswcntrl	1059
iswcntrl_l	1060
iswctype	1061
iswctype_l	1062
iswdigit	1063
iswdigit_l	1064
iswgraph	1065
iswgraph_l	1066
iswlower	1067
iswlower_l	1068
iswprint	1069
iswprint_l	1070
iswpunct	1071

iswpunct_l	1072
iswspace	1073
iswspace_l	1074
iswupper	1075
iswupper_l	1076
iswxdigit	1077
iswxdigit_l	1078
towctrans	1079
towctrans_l	1080
towlower	1081
towlower_l	1082
towupper	1083
towupper_l	1084
wctrans	1085
wctrans_l	1086
wctype	1087
<xlocale.h>	1088
duplocale	1089
freelocale	1090
localeconv_l	1091
newlocale	1092
C++ Library User Guide	1093
Standard template library	1095
Subset API reference	1096
<new> - memory allocation	1097
operator delete	1098
operator new	1099
set_new_handler	1100
LIBMEM User Guide	1101
Using the LIBMEM library	1102
Light version of LIBMEM	1105
Writing LIBMEM drivers	1106
LIBMEM loader library	1110
Complete API reference	1111
<libmem.h>	1112
LIBMEM_ADDRESS_IN_RANGE	1117
LIBMEM_ADDRESS_IS_ALIGNED	1118
LIBMEM_ALIGNED_ADDRESS	1119
LIBMEM_CFI_CMDSET_AMD_EXTENDED	1120
LIBMEM_CFI_CMDSET_AMD_STANDARD	1121
LIBMEM_CFI_CMDSET_INTEL_EXTENDED	1122

LIBMEM_CFI_CMDSET_INTEL_STANDARD	1123
LIBMEM_CFI_CMDSET_MITSUBISHI_EXTENDED	1124
LIBMEM_CFI_CMDSET_MITSUBISHI_STANDARD	1125
LIBMEM_CFI_CMDSET_NONE	1126
LIBMEM_CFI_CMDSET_RESERVED	1127
LIBMEM_CFI_CMDSET_SST_PAGE_WRITE	1128
LIBMEM_CFI_CMDSET_WINBOND_STANDARD	1129
LIBMEM_DRIVER_PAGED_WRITE_OPTION_DISABLE_DIRECT_WRITES	1130
LIBMEM_DRIVER_PAGED_WRITE_OPTION_DISABLE_PAGE_PRELOAD	1131
LIBMEM_INLINE	1132
LIBMEM_KB	1133
LIBMEM_MB	1134
LIBMEM_RANGE_OCCLUDES_RANGE	1135
LIBMEM_RANGE_OVERLAPS_RANGE	1136
LIBMEM_RANGE_WITHIN_RANGE	1137
LIBMEM_STATUS_CFI_ERROR	1138
LIBMEM_STATUS_ERROR	1139
LIBMEM_STATUS_GEOMETRY_REGION_OVERFLOW	1140
LIBMEM_STATUS_INVALID_DEVICE	1141
LIBMEM_STATUS_INVALID_PARAMETER	1142
LIBMEM_STATUS_INVALID_RANGE	1143
LIBMEM_STATUS_INVALID_WIDTH	1144
LIBMEM_STATUS_LOCKED	1145
LIBMEM_STATUS_NOT_IMPLEMENTED	1146
LIBMEM_STATUS_NO_DRIVER	1147
LIBMEM_STATUS_SUCCESS	1148
LIBMEM_STATUS_TIMEOUT	1149
LIBMEM_VERSION_NUMBER	1150
_libmem_driver_functions_t	1151
_libmem_driver_handle_t	1152
_libmem_driver_paged_write_ctrlblk_t	1153
_libmem_ext_driver_functions_t	1154
_libmem_flash_info_t	1155
_libmem_geometry_t	1156
_libmem_sector_info_t	1157
libmem_busy_handler_fn	1158
libmem_busy_handler_fn_t	1159
libmem_cfi_get_info	1160
libmem_crc32	1161
libmem_crc32_direct	1162
libmem_driver_crc32_fn_t	1163

libmem_driver_erase_fn_t	1164
libmem_driver_fill_fn_t	1165
libmem_driver_flush_fn_t	1166
libmem_driver_inrange_fn_t	1167
libmem_driver_lock_fn_t	1168
libmem_driver_page_write_fn_t	1169
libmem_driver_paged_write	1170
libmem_driver_paged_write_fill	1171
libmem_driver_paged_write_flush	1172
libmem_driver_paged_write_init	1173
libmem_driver_read_fn_t	1174
libmem_driver_unlock_fn_t	1175
libmem_driver_write_fn_t	1176
libmem_drivers	1177
libmem_enable_timeouts	1178
libmem_erase	1179
libmem_erase_all	1180
libmem_fill	1181
libmem_flush	1182
libmem_foreach_driver	1183
libmem_foreach_driver_fn_t	1184
libmem_foreach_sector	1185
libmem_foreach_sector_fn_t	1186
libmem_foreach_sector_in_range	1187
libmem_foreach_sector_in_range_ex	1188
libmem_get_driver	1189
libmem_get_driver_sector_size	1190
libmem_get_geometry_size	1191
libmem_get_number_of_regions	1192
libmem_get_number_of_sectors	1193
libmem_get_sector_info	1194
libmem_get_sector_number	1195
libmem_get_sector_size	1196
libmem_get_ticks	1197
libmem_get_ticks_fn	1198
libmem_get_ticks_fn_t	1199
libmem_lock	1200
libmem_lock_all	1201
libmem_read	1202
libmem_register_am29f200b_driver	1203
libmem_register_am29f200t_driver	1204

libmem_register_am29f400bb_driver	1205
libmem_register_am29f400bt_driver	1206
libmem_register_am29fxxx_driver	1207
libmem_register_am29lv010b_driver	1208
libmem_register_cfi_0001_16_driver	1209
libmem_register_cfi_0001_8_driver	1210
libmem_register_cfi_0002_16_driver	1211
libmem_register_cfi_0002_8_driver	1212
libmem_register_cfi_0003_16_driver	1213
libmem_register_cfi_0003_8_driver	1214
libmem_register_cfi_amd_driver	1215
libmem_register_cfi_driver	1217
libmem_register_cfi_intel_driver	1218
libmem_register_driver	1220
libmem_register_ram_driver	1221
libmem_register_sst39xFx00A_16_driver	1222
libmem_register_st_m28w320cb_driver	1223
libmem_register_st_m28w320ct_driver	1224
libmem_set_busy_handler	1225
libmem_ticks_per_second	1226
libmem_unlock	1227
libmem_unlock_all	1228
libmem_write	1229
<libmem_loader.h>	1230
LIBMEM_LOADER_VERSION_NUMBER	1231
LIBMEM_RPC_LOADER_FLAG_PARAM	1232
LIBMEM_RPC_LOADER_FLAG_PRESERVE_STATE	1233
LIBMEM_RPC_LOADER_MAGIC_NUMBER	1234
LIBMEM_RPC_LOADER_OPTION_HOST_ERASE	1235
LIBMEM_RPC_LOADER_OPTION_HOST_WRITE	1236
libmem_rpc_loader_exit	1237
libmem_rpc_loader_start	1239
libmem_rpc_loader_start_ex	1241
Utilities Reference	1243
Command-Line Compiler	1244
File Naming	1244
Compilation	1244
Linking	1245
Target Selection	1246
Advanced	1246
Options	1246

Command-Line Project Builder	1253
Building with a CrossStudio project file	1253
Building without a CrossStudio project file	1254
Options	1254
Command-Line Simulator	1256
Example	1256
Usage	1257
Command-Line Project Download and Debug	1259
Command line debugging	1261
Managing breakpoints	1262
Displaying state	1265
Locating the current context	1267
Controlling execution	1269
Support packages	1270
Command-line options	1271
-break (Stop execution at symbol)	1272
-config (Specify build configuration)	1273
-connection (Specify connection)	1274
-debug (Enter command line debugging)	1275
-eraseall (Erase all flash memory)	1276
-filetype (Specify load file type)	1277
-help (Display help)	1278
-listfiletypes (Display supported load file types)	1279
-listprojectprops (Display all project properties)	1280
-listprops (Display target properties)	1281
-listtargets (Display supported target interfaces)	1282
-loadaddress (Set load address)	1283
-loader (Specify loader configuration)	1284
-nodifferential (Inhibit differential download)	1285
-nodisconnect (Inhibit target disconnection)	1286
-nodownload (Inhibit download)	1287
-noverify (Inhibit verification)	1288
-packagesdir (Specify package directory)	1289
-project (Specify project name)	1290
-quiet (Be silent)	1291
-reset (Reset only)	1292
-script (Execute debug script)	1293
-serve (Run semihosting server)	1294
-setprop (Set target interface property)	1295
-solution (Specify solution file)	1296
-studiudir (Specify Studio directory)	1297

-target (Specify target interface)	1298
-verbose (Display additional status)	1299
Command-Line Scripting	1300
Command-line options	1301
-define (Define global variable)	1302
-help (Show usage)	1303
-load (Load script file)	1304
-define (Verbose output)	1305
CrossScript classes	1306
Example uses	1307
Embed	1308
Header file generator	1309
Using the header generator	1310
Command line options	1311
-regbaseoffsets (Use offsets from peripheral base)	1312
-nobitfields (Inhibit bitfield macros)	1313
Linker script file generator	1314
Command-line options	1315
-check-section-overflow	1316
-check-segment-overflow	1317
-disable-missing-runin-error	1318
-memory-map-macros	1319
-no-check-unplaced-sections	1320
-no-ctors	1321
-no-dtors	1322
-section-placement-file	1323
-section-placement-macros	1324
-symbols	1325
Package generator	1326
Package manager	1328
Appendices	1331
Technical	1332
File formats	1332
Memory Map file format	1333
Section Placement file format	1335
Project file format	1337
Project Templates file format	1338
Property Groups file format	1340
Package Description file format	1342
External Tools file format	1346
Debugger Type Interpretation file format	1349

Environment Options	1351
Building Environment Options	1351
Debugging Environment Options	1353
IDE Environment Options	1356
Programming Language Environment Options	1362
Source Control Environment Options	1366
Text Editor Environment Options	1368
Windows Environment Options	1380
Project Options	1393
Code Options	1393
Debug Options	1422
Macros	1433
System Macros	1433
Build Macros	1436
Script classes	1441
BinaryFile	1441
CWSys	1442
Debug	1443
ElfFile	1445
TargetInterface	1446
WScript	1451



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 product.

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.

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 ARM and Cortex, microcontrollers and microprocessors that runs on Windows, Mac OS and Linux.

C/C++ Compiler

CrossWorks comes with pre-built versions of both GCC and Clang/LLVM C and C++ compilers and assemblers. The GNU linker and librarian are also supplied 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.

CrossWorks C++ Library

CrossWorks for ARM supplies a C++ library that implements STL containers, exceptions and RTTI.

CrossStudio IDE

CrossStudio 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.

CrossWorks Tools

CrossWorks for ARM supplies command line tools that enable you to build your application on the command line and flash it to the target board using the same project file that the IDE uses.

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 ANSI 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 go to:

<http://gcc.gnu.org/>.

LLVM/Clang

For the latest LLVM/Clang documentation to to:

<http://www.llvm.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**.
- Click "Request Activation After Purchasing". 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 and Linux 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 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:\code\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 suggestion forum:

<https://rowley.zendesk.com/forums/171704-Suggestions>

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.

Highlights

The development of CrossWorks 3 has taken longer than we ever expected. During that period, we visited each part of the software to evaluate, polish, improve, and perhaps completely rewrite it. The changes in CrossStudio range from subtle (changing a few icons here and there, improving performance) to extensive (threaded source indexer, parallel build system, slick source control, new trace support). Here are some of the highlights in CrossWorks 3

Parallel and Unity Building

Quad core processor are now standard in desktops and laptops, and CrossStudio can take full advantage of multi-core processors when building your applications by scheduling projects to build in parallel. To partner parallel building, CrossStudio also introduces support for *unity builds* where a set of source files are compiled as a single unit.

To illustrate the advantages of these new features, here are the build times for rebuilding the example HTTP server included in many CrossWorks board support packages, with the exception that all projects are source code rather than object code libraries:

Cores	Unity?	Time	Speedup	Comments
1	No	21s	1x	Baseline
2	No	15s	1.4x	
4	No	10s	2.1x	
8	No	9.4s	2.2x	Hyperthreading doesn't really help
1	Yes	6.3s	3.3x	
2	Yes	4.5s	4.6x	
4	Yes	3.1s	6.8x	
8	Yes	3.2s	6.5x	Hyperthreading isn't an advantage

And, building a set of sensor example projects, again in many board support packages, in a single solution:

Cores	Unity?	Time	Speedup	Comments
1	No	65s	1x	Baseline
2	No	41s	1.6x	
4	No	24s	2.7x	
8	No	20s	3.3x	Hyperthreading does help
1	Yes	39s	1.6x	
2	Yes	23s	2.8x	

4	Yes	14s	4.6x	
8	Yes	11s	5.9x	Hyperthreading does help

These timings were taken on Windows 7 running under Parallels 9 on a Retina MacBook Pro with a 4-core 2.3 GHz Intel Core i7 and 8 GB of memory allocated to the virtual machine. The effect of parallel building will depend upon the way you structure your project and the performance of your hardware.

Source indexer

The source indexer is completely reworked to be much more precise. Indexing takes place in the background, using threads to index your code quickly. You can change the number of threads launched to index your project, choosing between performance and responsiveness when indexing.

Hand in hand with the indexer, the code editor is improved with *code completion* where appropriate suggestions pop up as you type. Because the indexer is very accurate, code completion is also accurate, increasing your productivity as a programmer.

To complement the indexer, CrossStudio adds a **Find References** capability that fill search your application for references to items. As you would expect, Find References runs in parallel, is configurable, and is a great way to find the uses of functions, variables, types, and members.

Source control

Source-control integration is now significantly faster in CrossStudio 3. We're added source-control annotations to the project explorer, but kept the ability to show the source-control column from CrossStudio 2.

We've changed the source-control model that CrossStudio 3 uses from the check-out/lock/check-in model (as used by Visual Source Safe and RCS) to the widely used update/merge/commit model (as used by CVS and Subversion).

In addition, we've added the popular **Pending Changes** window that succinctly shows you the changes you've made and the overall state of the items in your project. We've also added a source-control state filter to the project explorer, if you're more comfortable working in that window.

Source-control state updates progress in the background and much more efficient than the CrossStudio 2 implementation, making CrossStudio a real pleasure to use.

Release notes

Version 5.0.0

What's New

- Improved code completion and navigation.
- In-place error and warning diagnostics as you type.
- Quick fix suggestions.

What's Changed

CrossConnect and FTDI FTx232 based debug interfaces now have a **Speed** target property rather than a **JTAG Clock Divider** target property to control the speed of the JTAG/SWD clock.

Build

- Updated the GCC/BINUTILS tools build to use the Arm GNU Toolchain 13.2.Rel1 source release.
- Updated the LLVM/Clang tools build to use the 17.0.6 source release.



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 CrossStudio help **Browser** window, you may find it more convenient to view using an external web browser so you can still see the entire CrossStudio window. To do so, simply right-click on the help content in the CrossStudio **Browser** and choose **Open With 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.

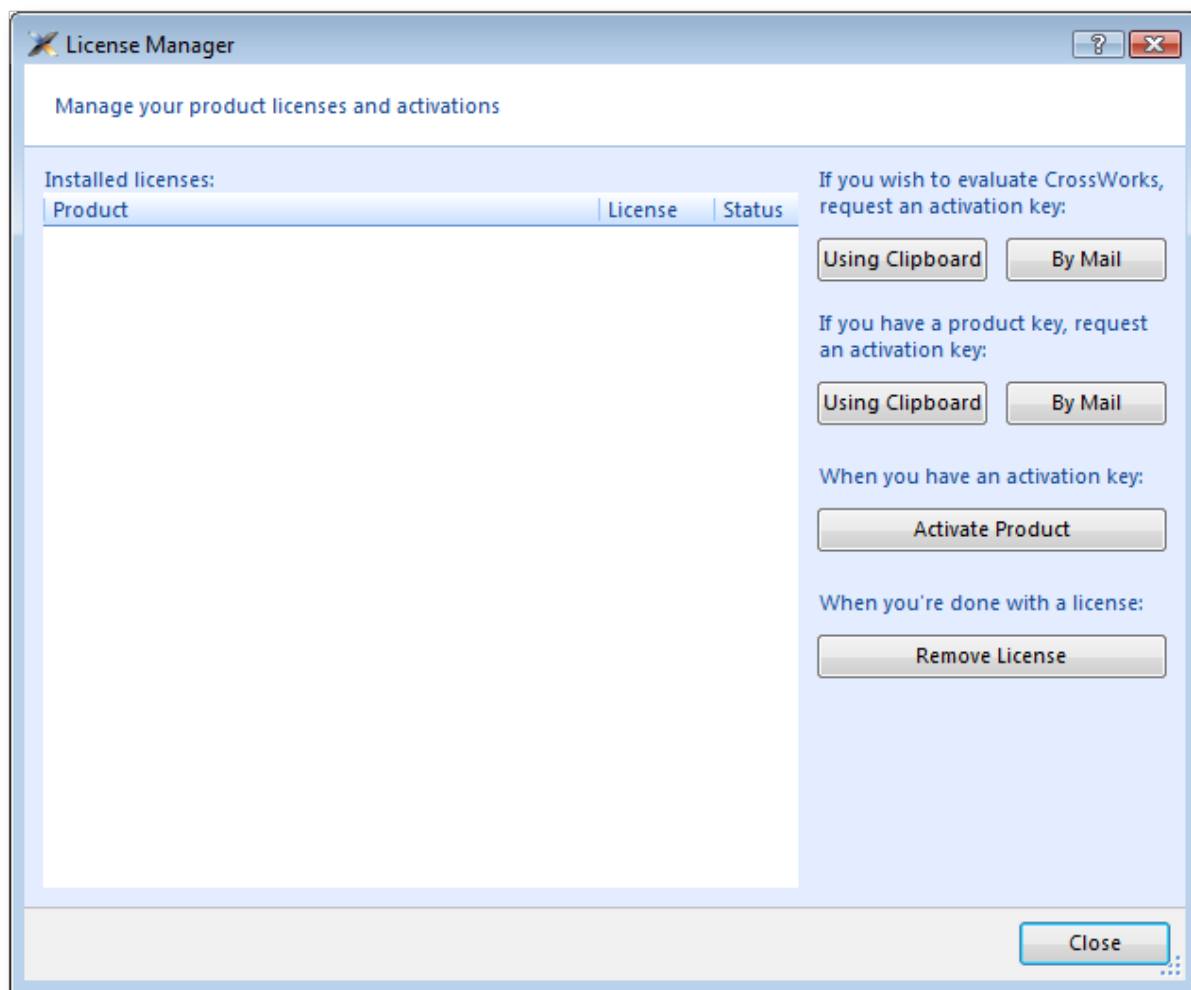
Debugging externally built applications

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 CrossWorks'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.

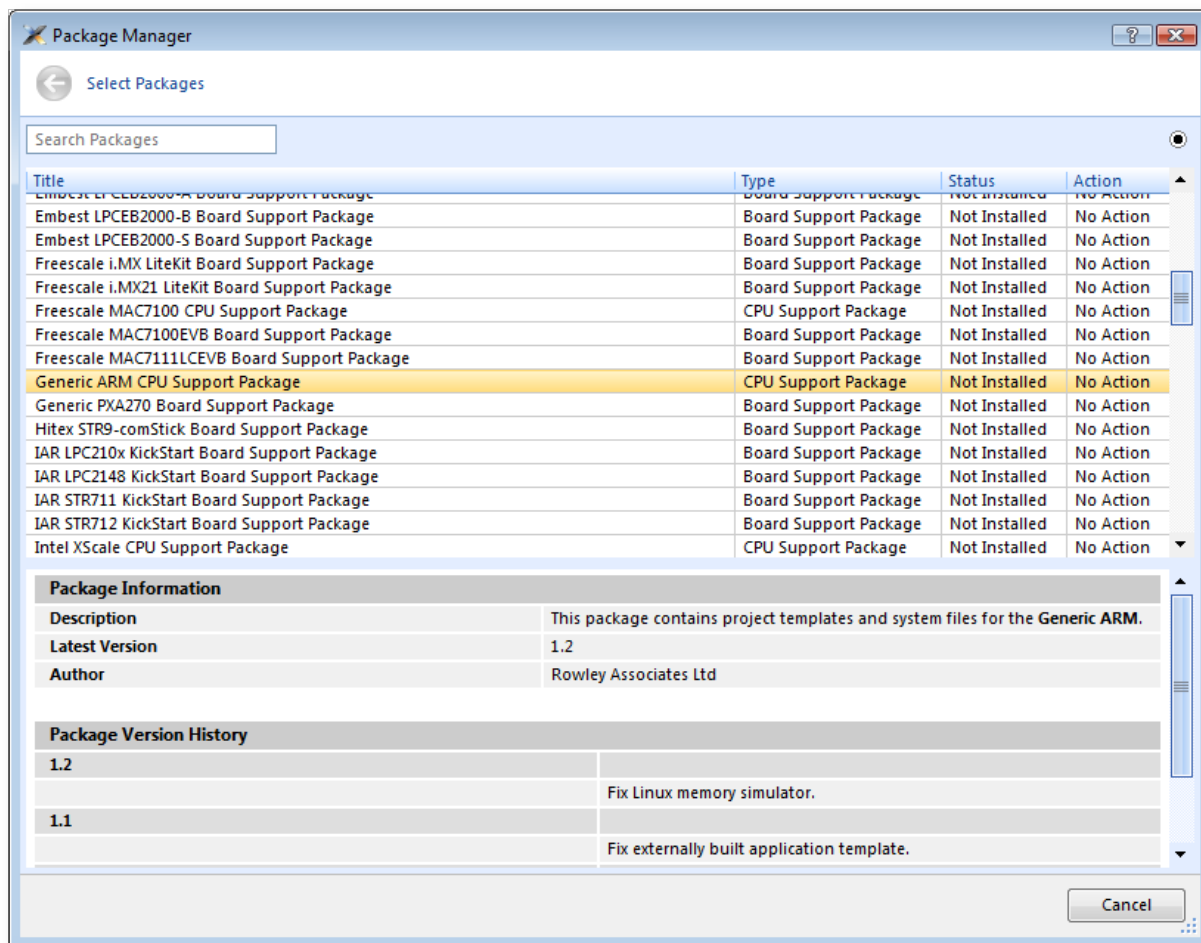
Downloading and installing a support package

To download and install a support package:

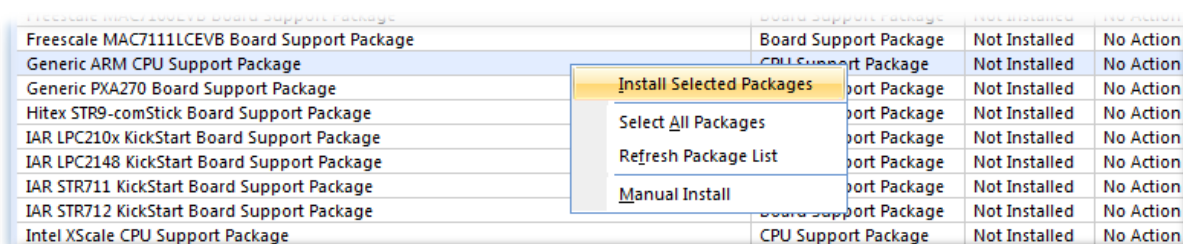
- Choose **Tools > Manage Packages**.

- 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:

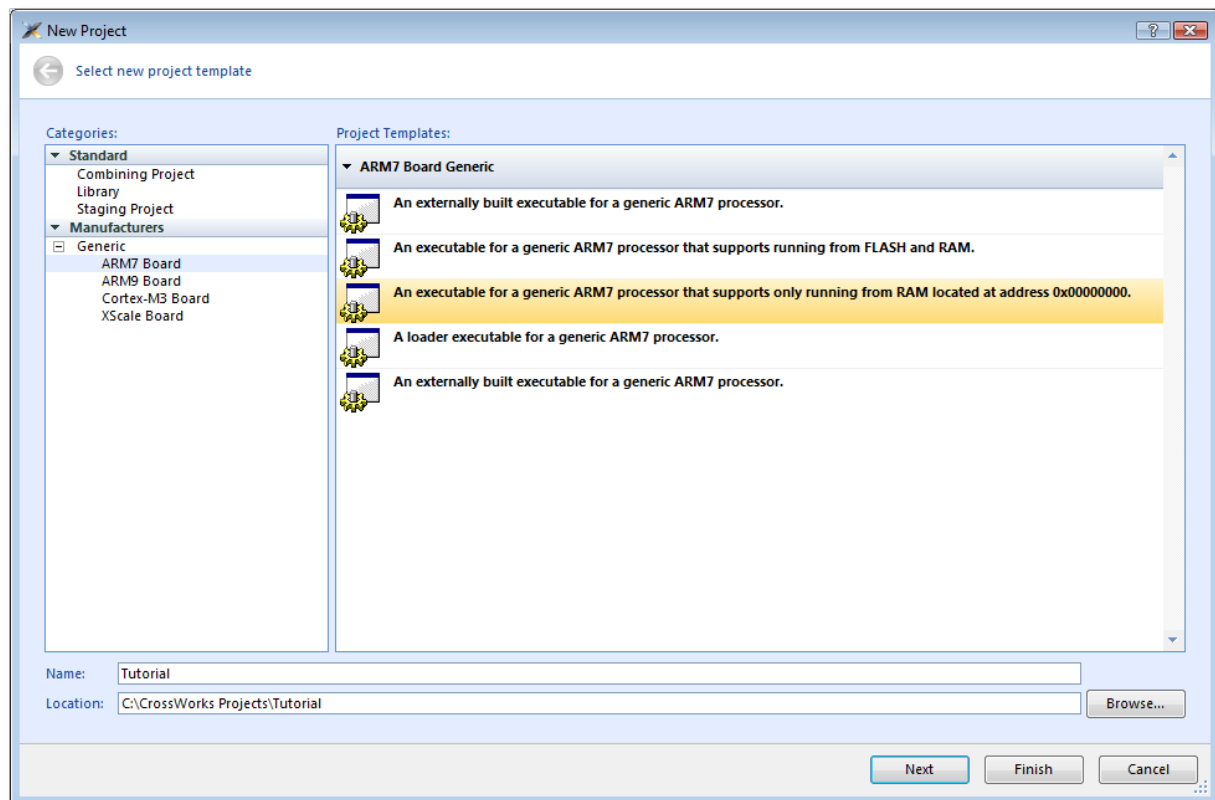
Choose **Tools > Show Installed Packages** to list the support packages you have installed on your system. You should see the name of the **Generic ARM CPU Support Package** you just installed. Click **Generic ARM CPU Support Package** to view the support package page in the CrossWorks **Browser** window. 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:

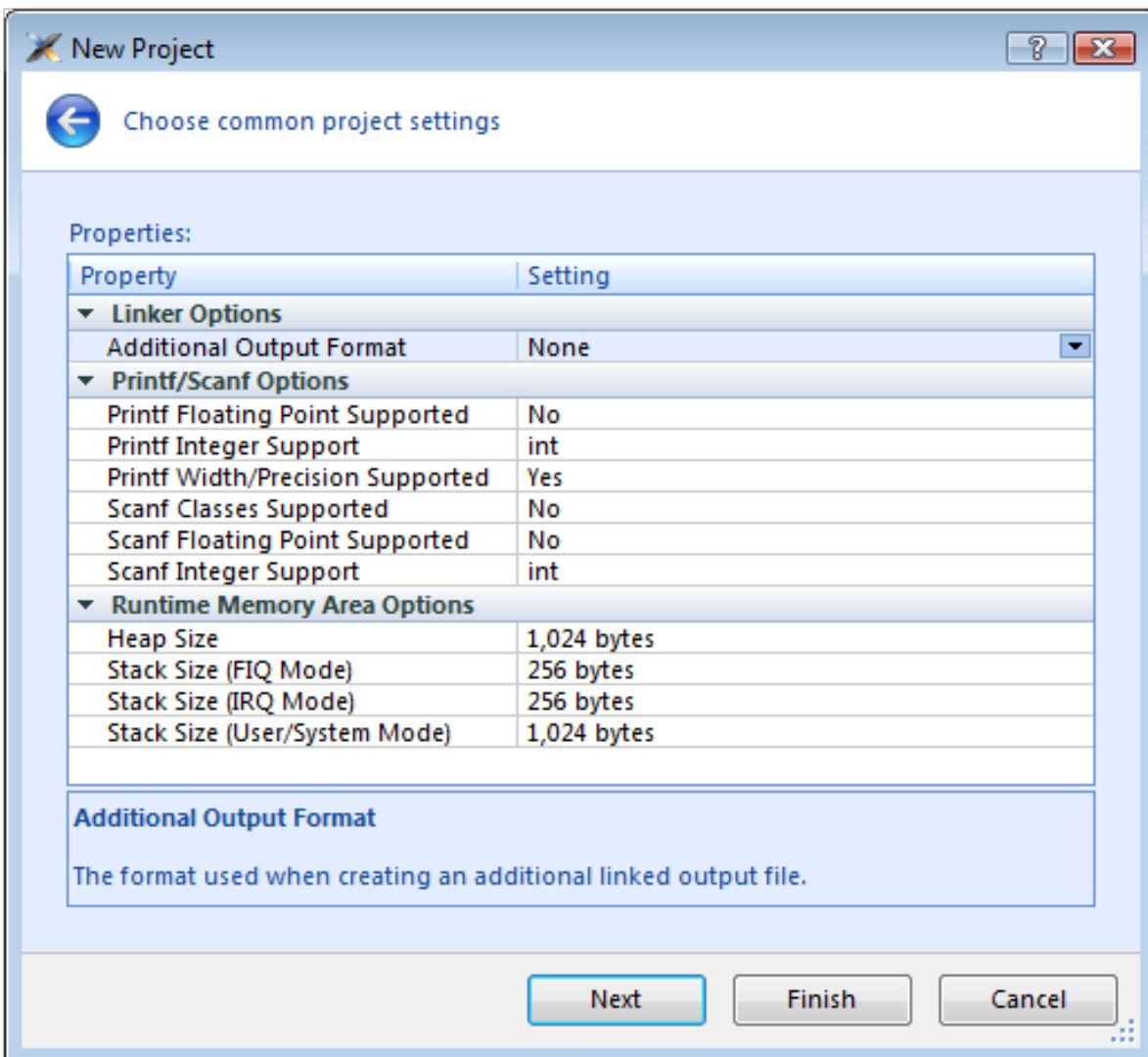
Choose **File > New Project** or press **Ctrl+Shift+N**

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



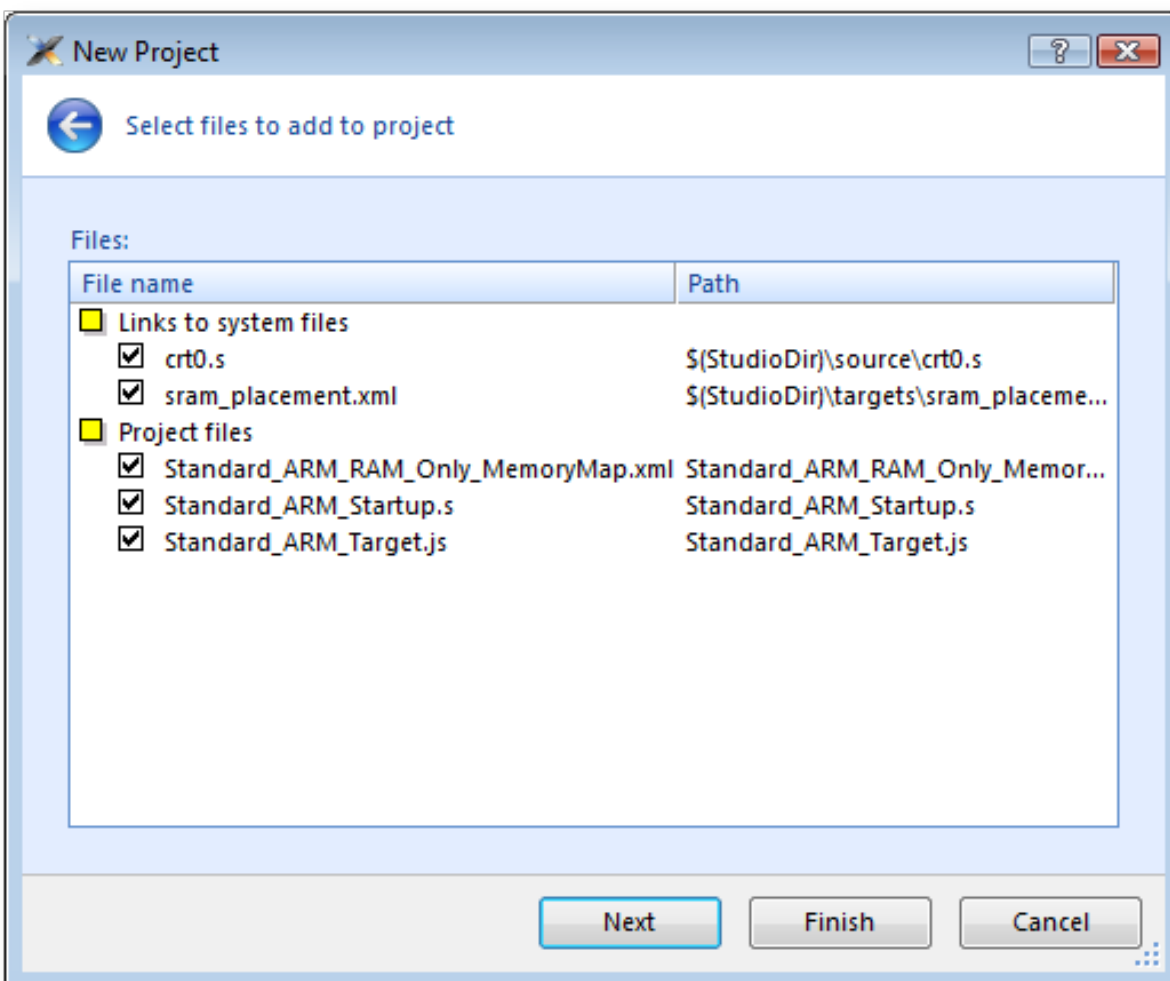
We'll create a project to develop our application in C:

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 CrossWorks will add to this project by default. You can uncheck any file you plan to add manually or that you know will not be needed.

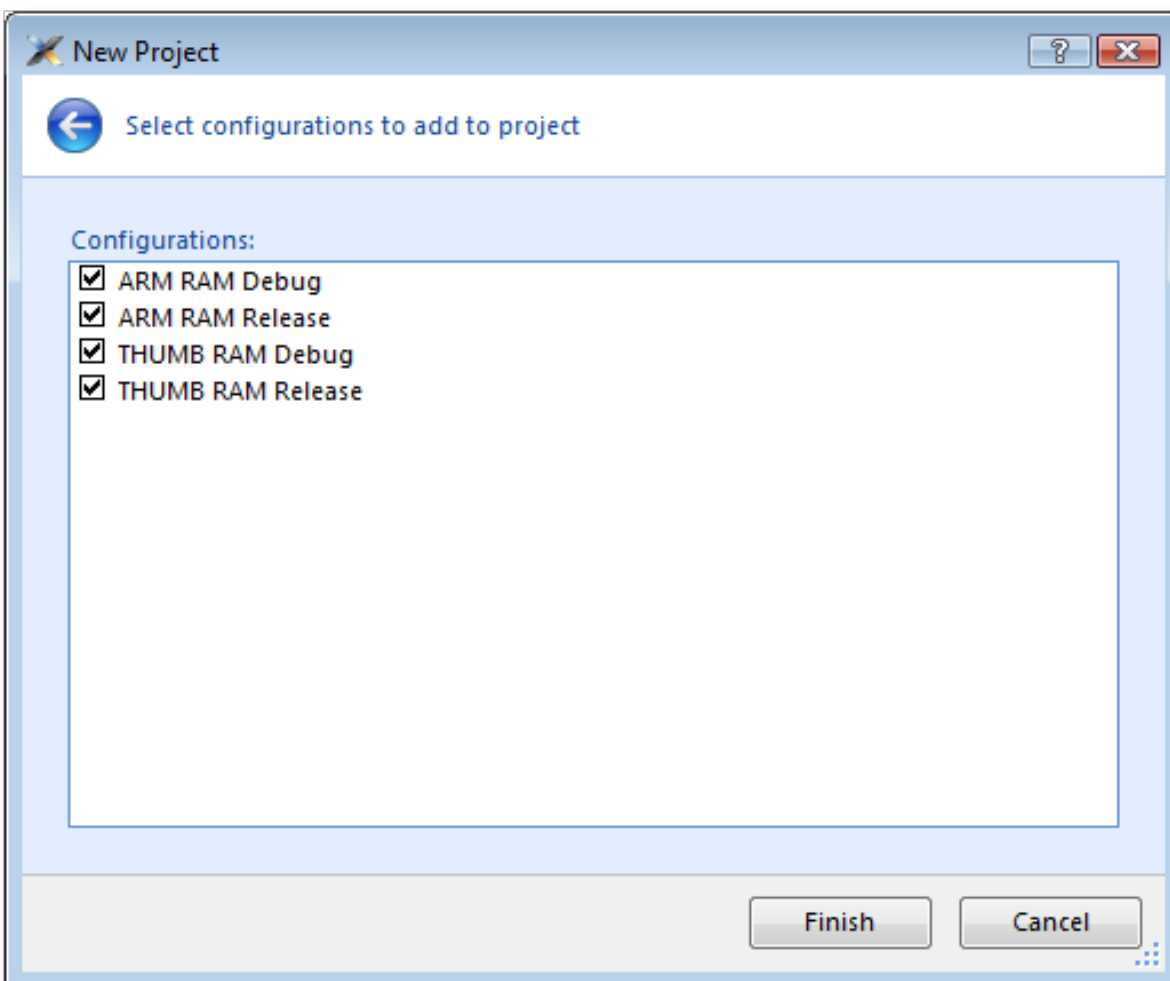


The **Links to system files** group shows the links to CrossWorks system files that will be created in the project. Because these files are links, the default behavior is that they will be shared with other projects so modifying one will affect all projects containing similar links. To prevent accidental 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.

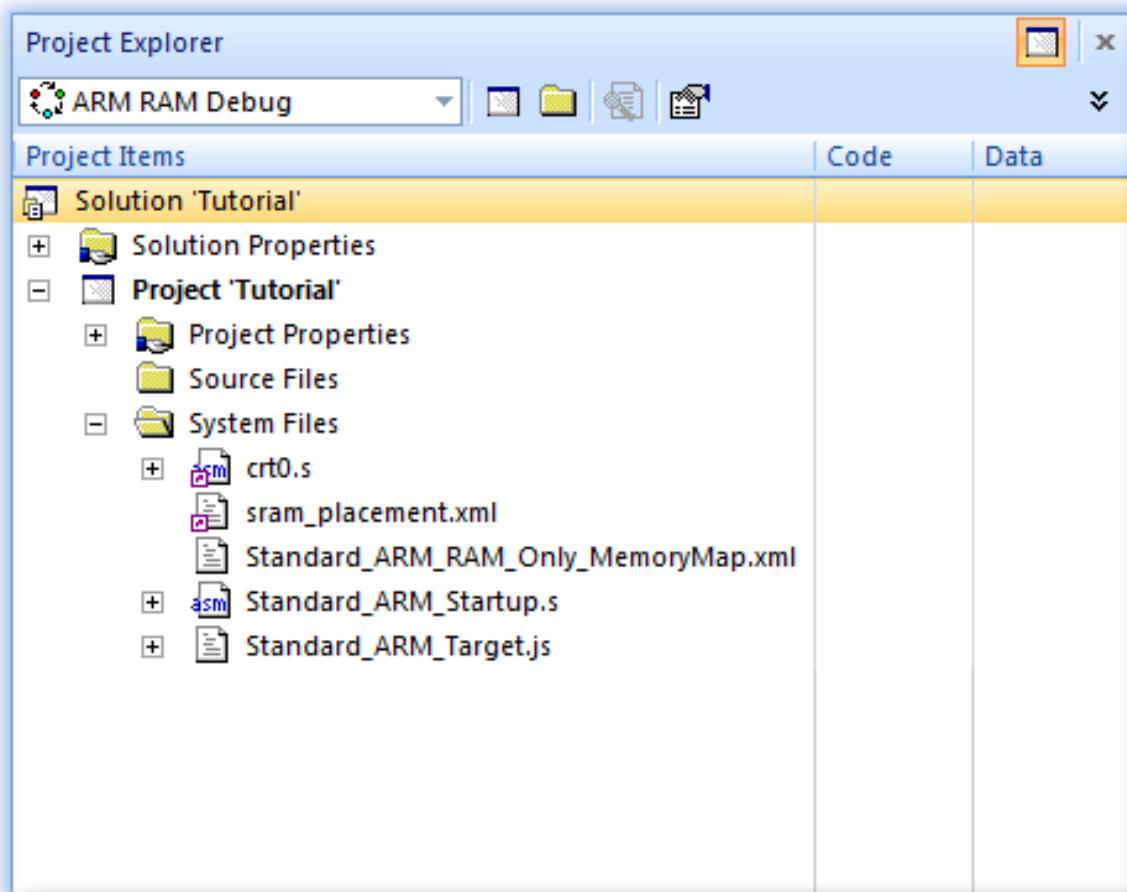
The **Project Explorer** shows the overall structure of your project. To invoke it, do one of the following:

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 CrossWorks 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:

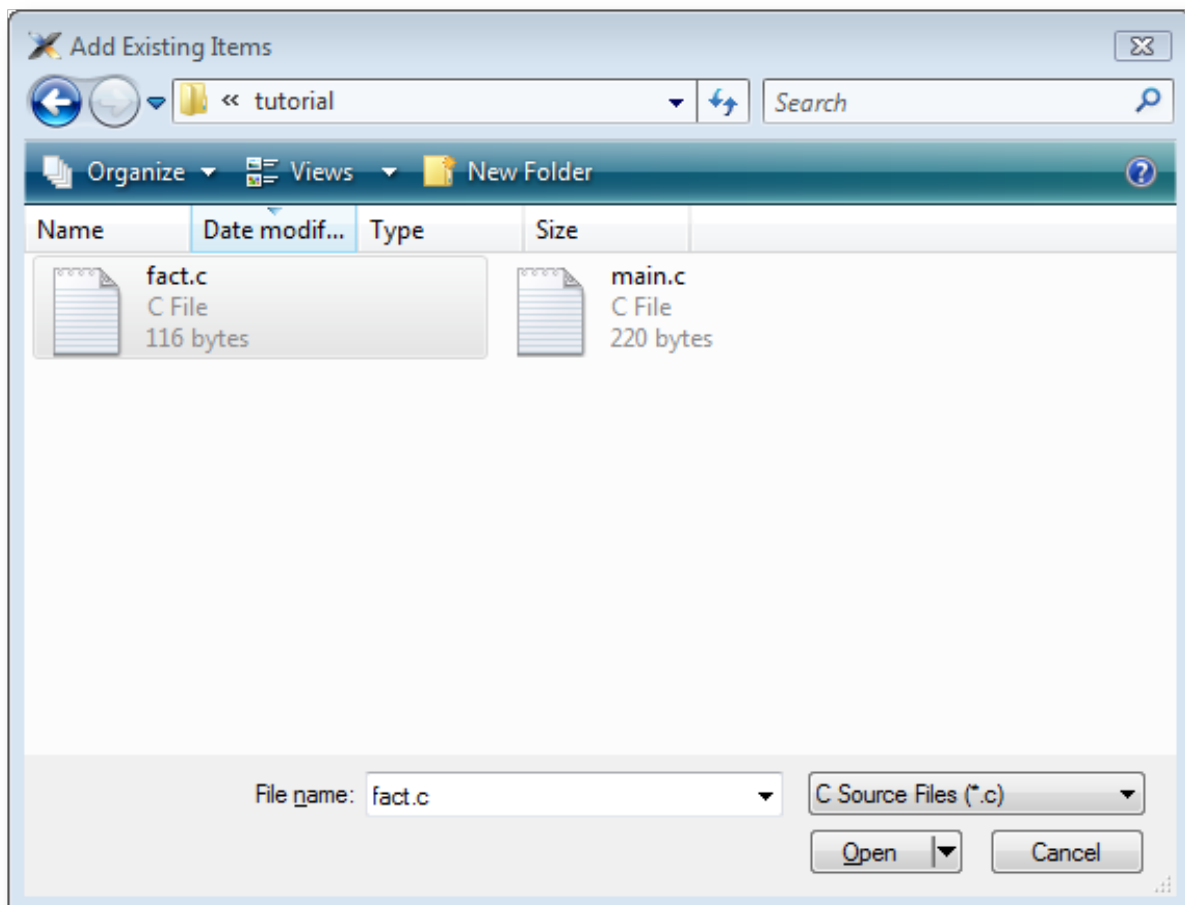
Choose **Project > Add Existing File** or press **Ctrl+P, A**.

or

In the **Project Explorer**, right-click the Tutorial project node.

Choose **Add Existing File** from the shortcut menu.

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



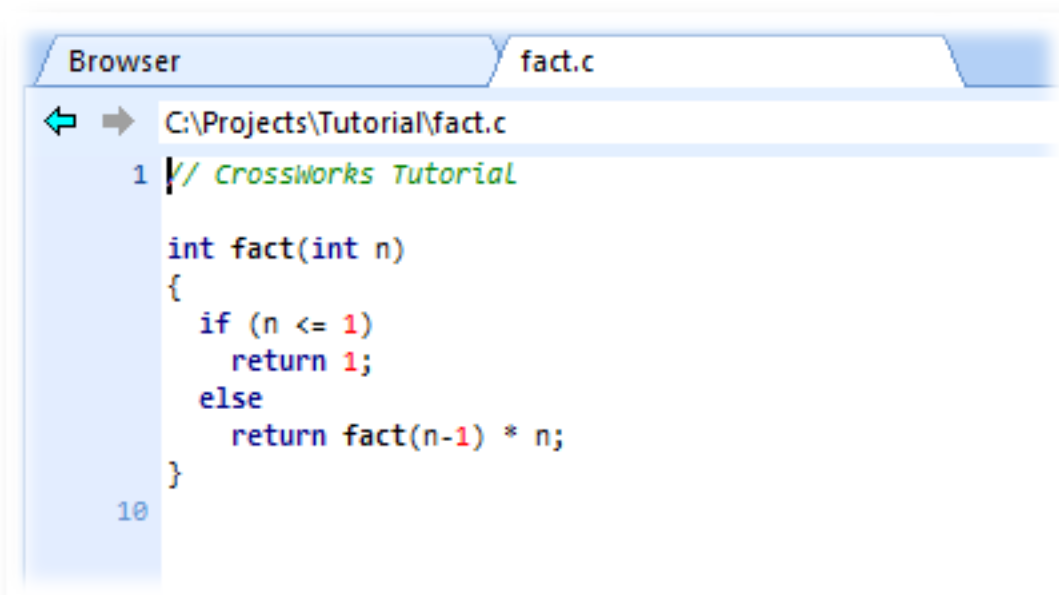
Click **Open** to add the file to the project. The **Project Explorer** will list `fact.c` in the **Project Items' Source Files** folder, with a shortcut arrow because the file is not in the project's home directory. Rather than edit the file in the tutorial directory, we'll put a copy of it into the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

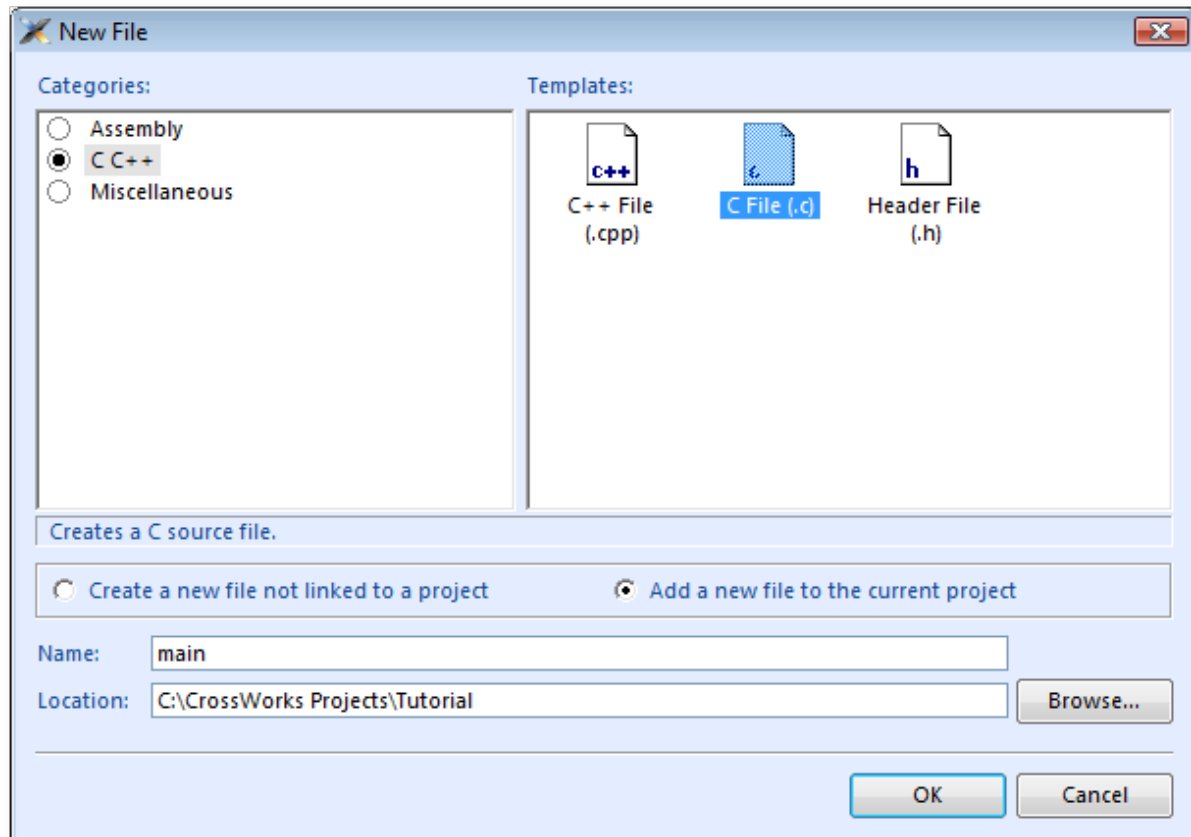
The **New File** dialog appears.

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.

CrossWorks 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:

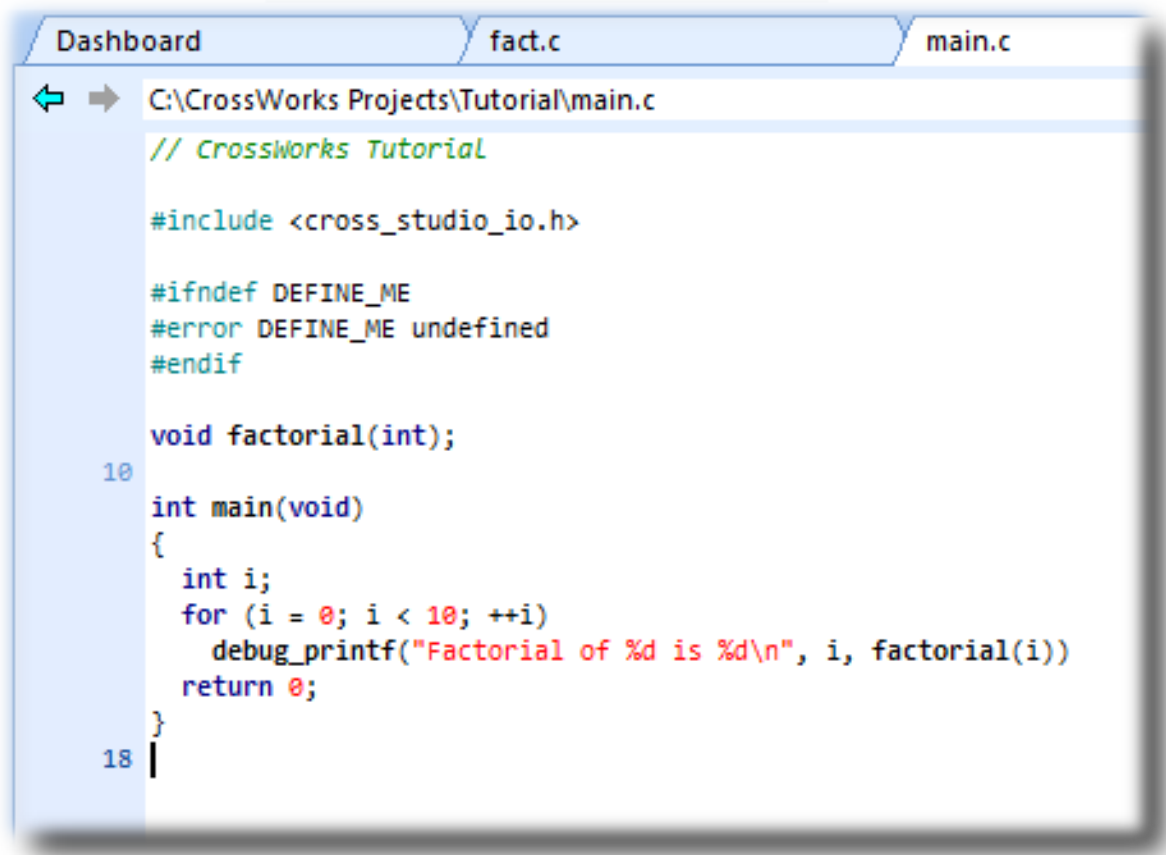
Choose **Edit > Others > Insert File** or press **Ctrl+K, Ctrl+I**.

Using the 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:

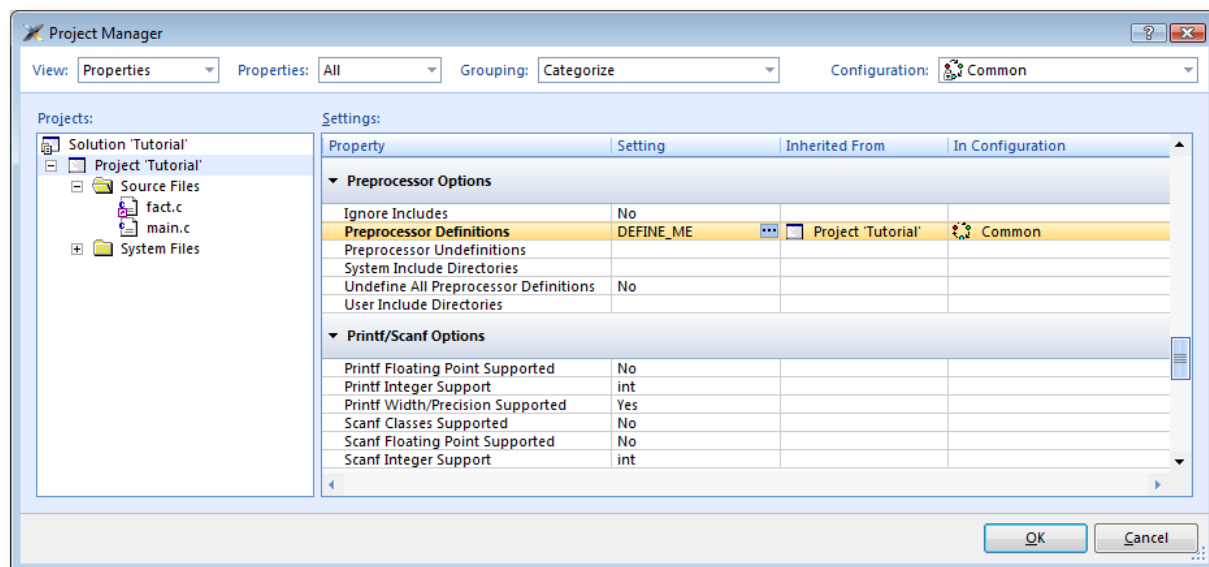
Right-click the Tutorial project in the **Project Explorer** and select **Properties** from the menu the **Project Manager** dialog appears.

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

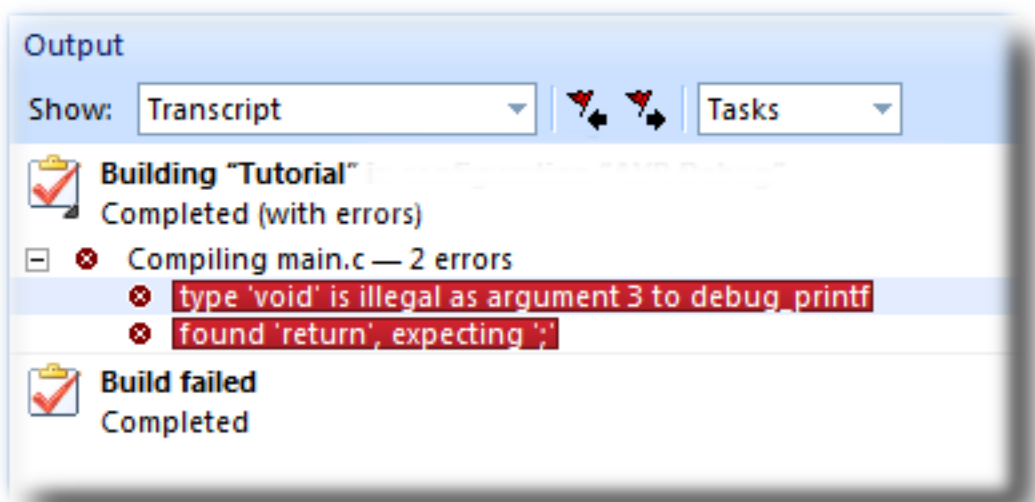
Type **F7**.

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

In the **Project Explorer**, right-click the Tutorial project node.

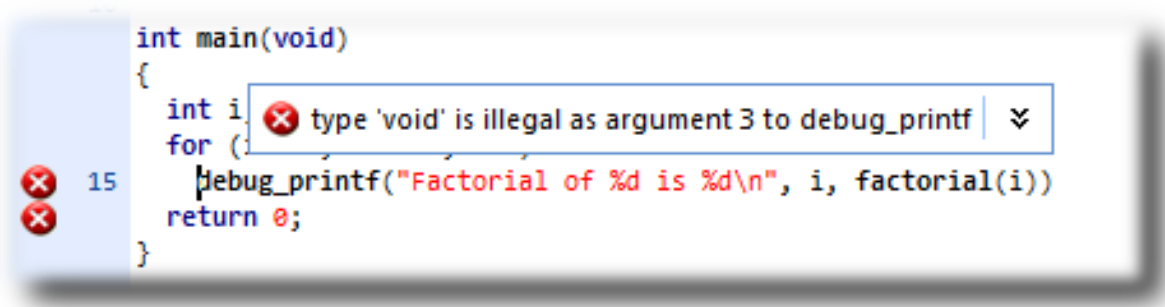
Select **Build** from the shortcut menu.

CrossWorks 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

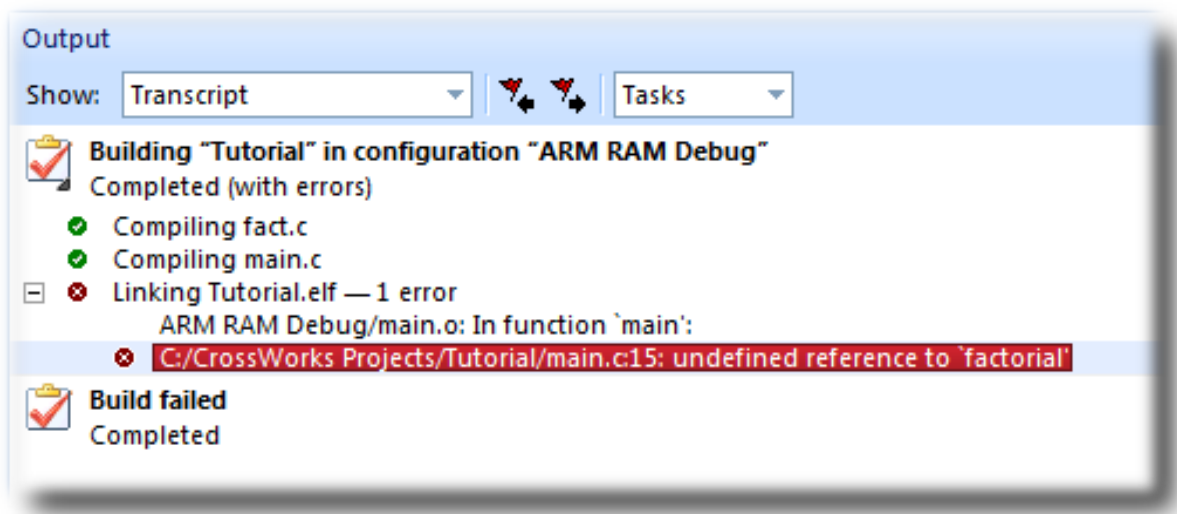
The file `main.c` contains two errors. After compilation, CrossWorks moves the cursor to the line containing the first reported error and displays an error message in the **Output** window. (You can change this behavior by modifying the **Text Editor > Editing Options > Enable Popup Diagnostics** environment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



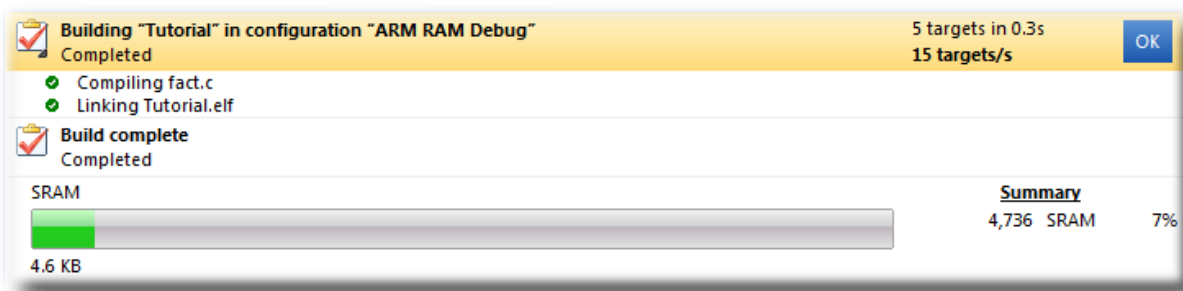
To correct the error, change the return type of `factorial` from `void` to `int` in its prototype.

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

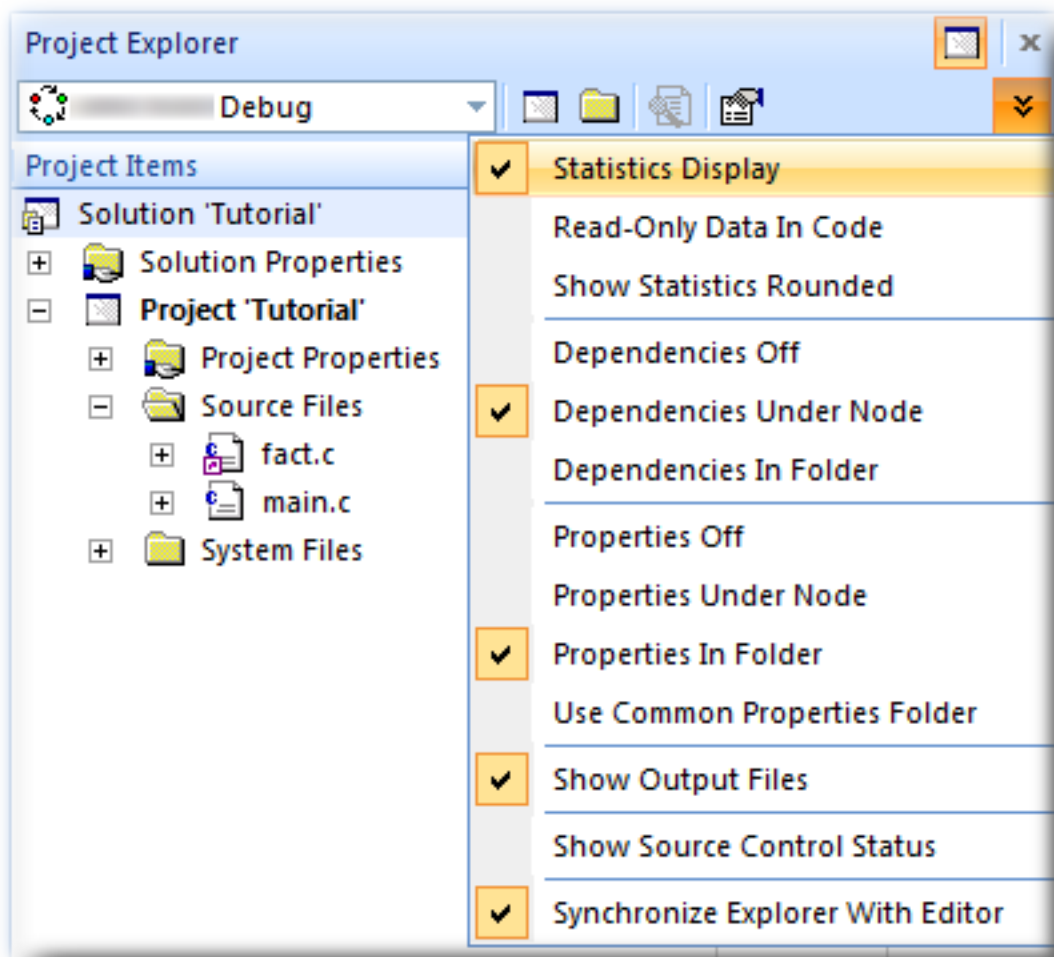
Now that the project has no errors and builds correctly, we can turn our attention to uncovering exactly how our application fits in memory and how to navigate around it.

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 CrossWorks 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**.

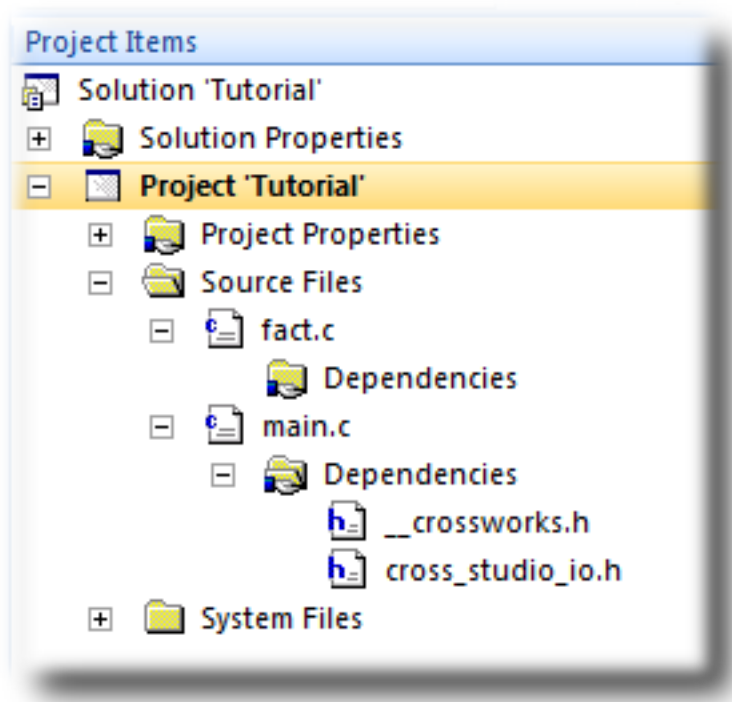
Project Items	Code	Data
 Solution 'Tutorial'		
 Solution Properties		
 Project 'Tutorial'	2,148	28
 Project Properties		
 Source Files		
 fact.c	80	
 main.c	100	24
 System Files		

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, CrossWorks finds and records the relationships between files—that is, it finds which files depend upon other files. CrossWorks uses these known relationships when it builds the project again, to minimize the amount of work required to bring the project up to date.

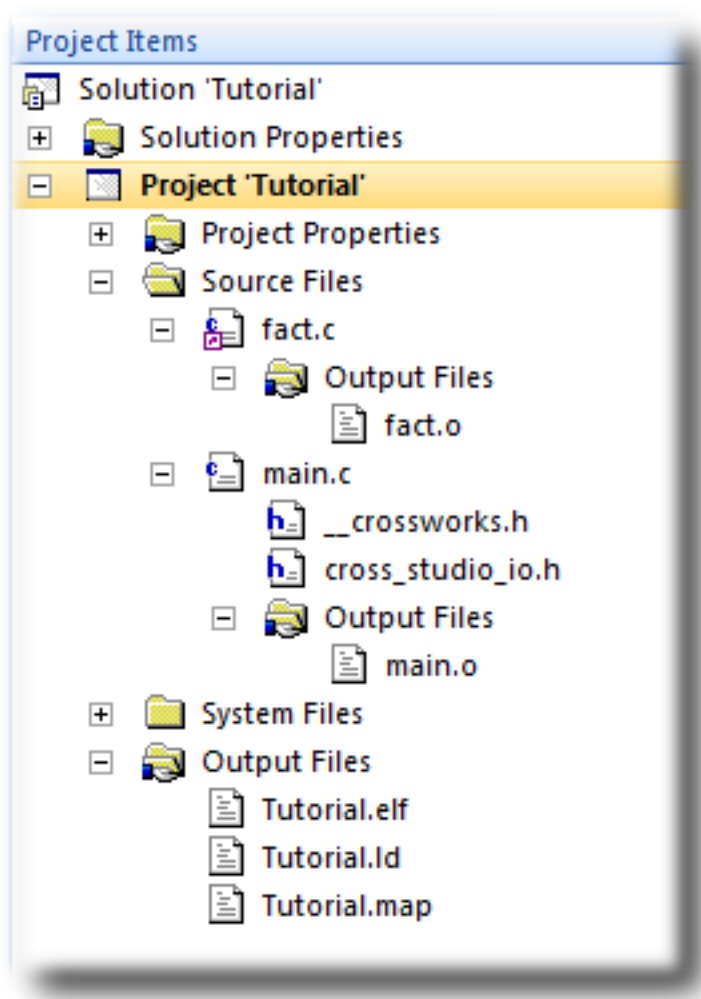
To show the dependencies for a project, use the **Options** button on the **Project Explorer** tool bar to ensure that either **Dependencies Under Node** or **Dependencies In Folder** is checked. Once checked, dependent files are shown as sub-nodes of the file that depends on them.



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 CrossWorks 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**.

CrossWorks 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.

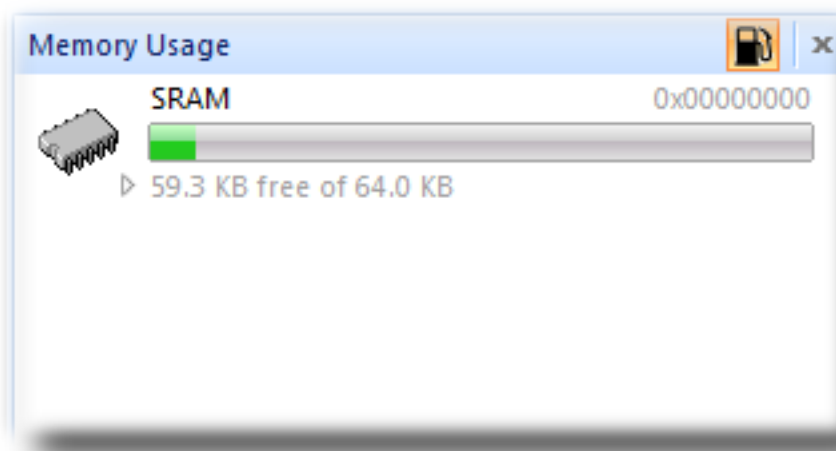
Using Memory Usage Window features

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:



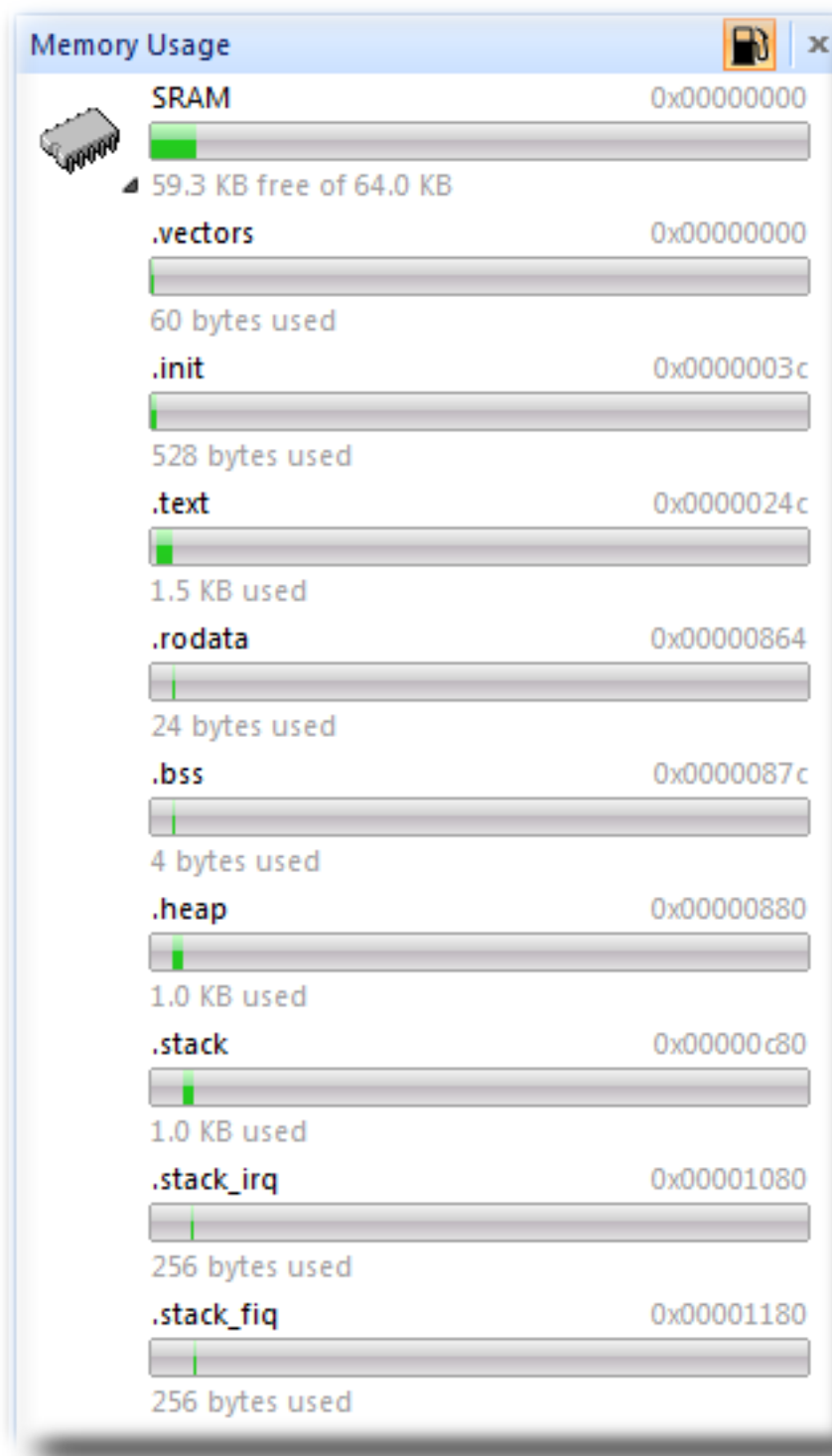
From this, you can see:

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, CrossWorks will display all the program sections contained within the segment:



Using Symbol Browser features

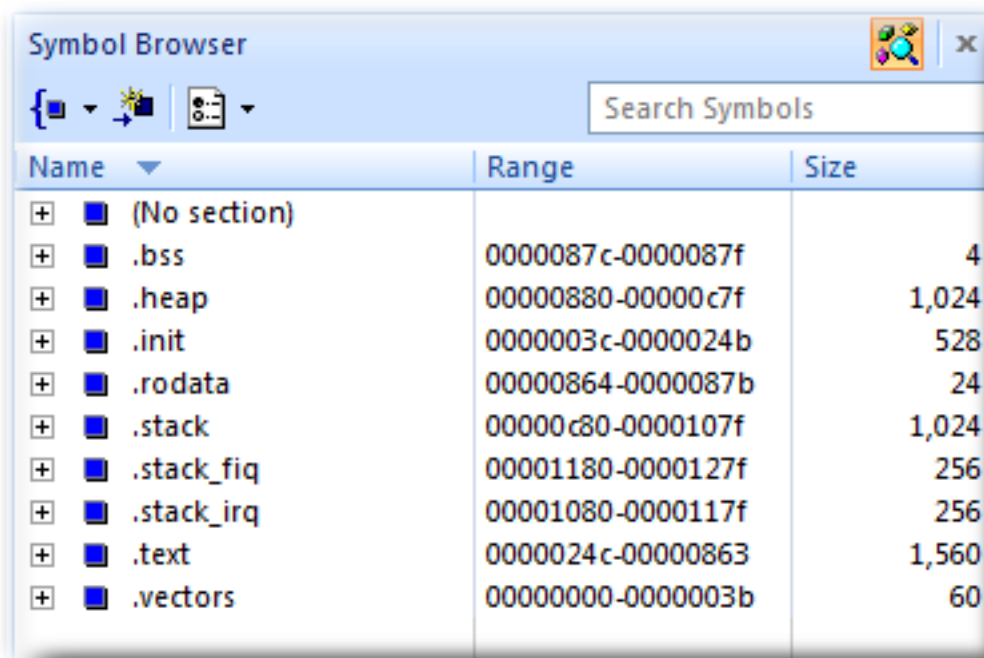
For a more-detailed view of how your application is laid out in memory than the **Memory Usage** window provides, you can use the **Symbol Browser**. It allows you to navigate your application, see which data objects and functions have been linked into your application, what their sizes are, which section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

To activate the Symbol Browser:

Choose **Navigate > 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: CrossWorks displays the individual functions that have been placed in memory and their sizes:

Symbol Browser		
<div> <div> <div></div> <div></div> <div></div> </div> <div> <div></div> <div></div> <div></div> </div> </div> <div>Search Symbols</div>		
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_oper	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, CrossWorks 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. CrossWorks 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 CrossWorks 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 CrossWorks to a target. For this tutorial, you'll be debugging on the simulator, not hardware, to simplify matters.

To connect to the simulator:

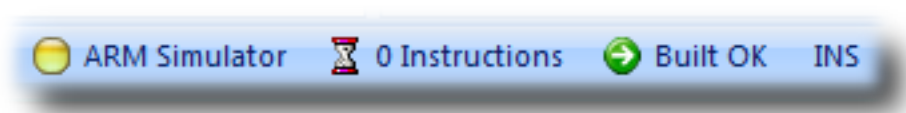
Choose **Target > Connect > ARM Simulator**.

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 CrossWorks and the target are doing:

White No target is connected.

Yellow Target is connected.

Solid green Target is free running, not under control of CrossWorks 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 CrossWorks is programming the application into the target.

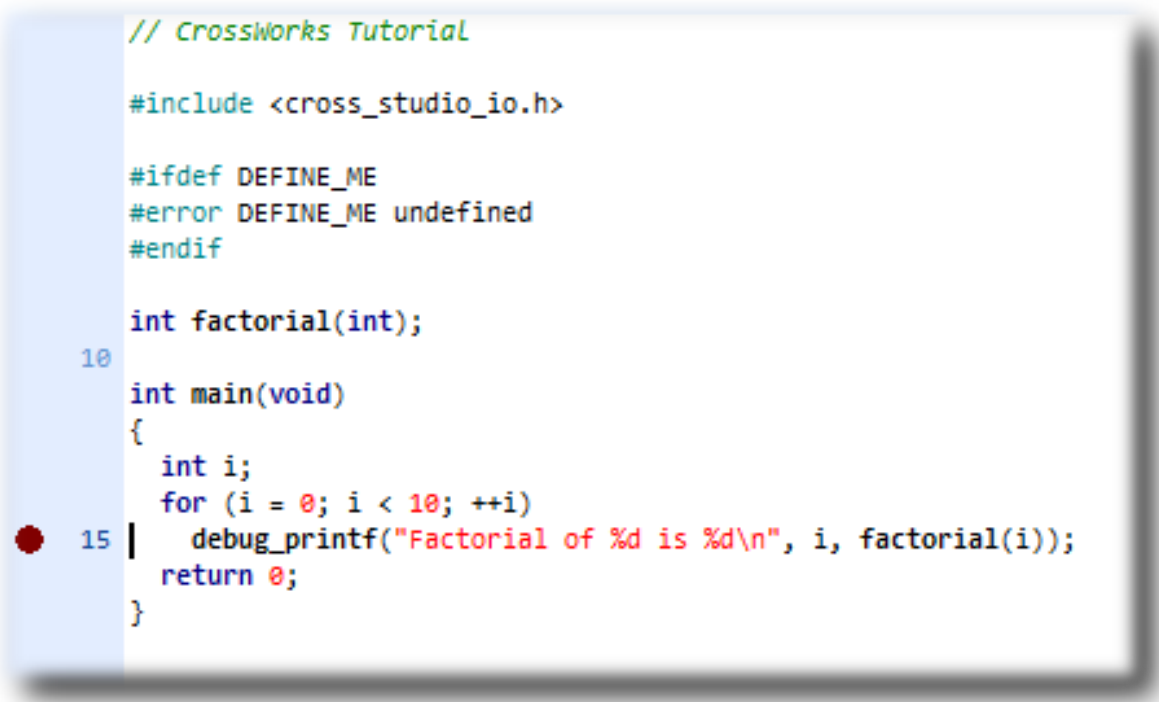
Double-clicking the **Target Status** will show the **Targets** window, if it is not already visible.

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

CrossWorks 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.

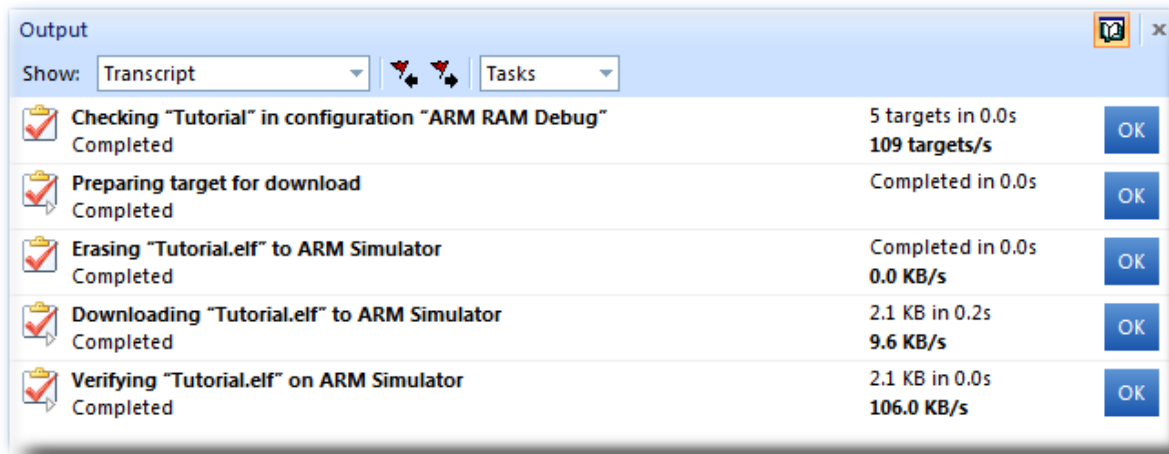


The gutter displays an icon on lines where breakpoints are set. The **Breakpoints** window updates to show where each breakpoint is set and whether it's set, disabled, or invalid. You can find more detailed information in the [Breakpoints window](#) section. The breakpoints you set are stored in a session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

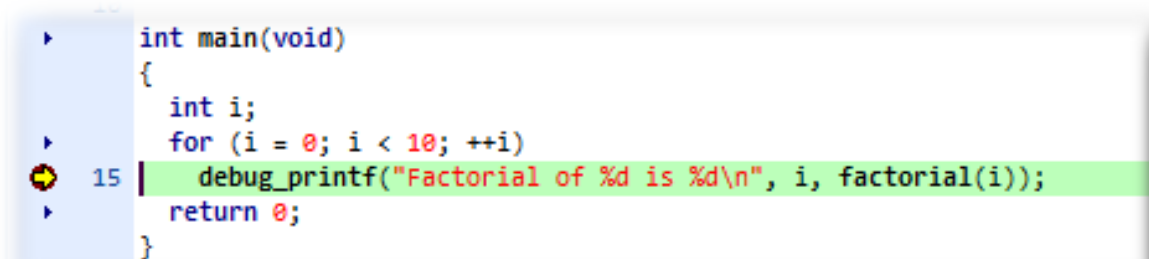
Starting the application

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. CrossWorks 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:

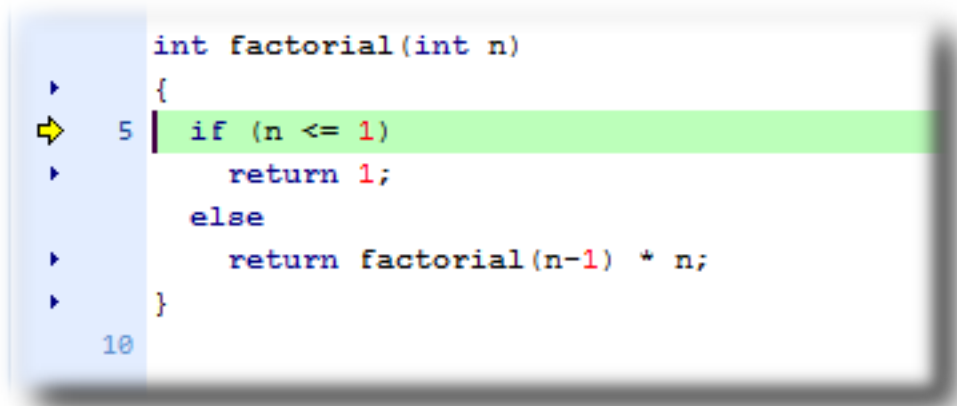


The program stops at our breakpoint and a yellow arrow in the gutter indicates where the program is paused.



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.

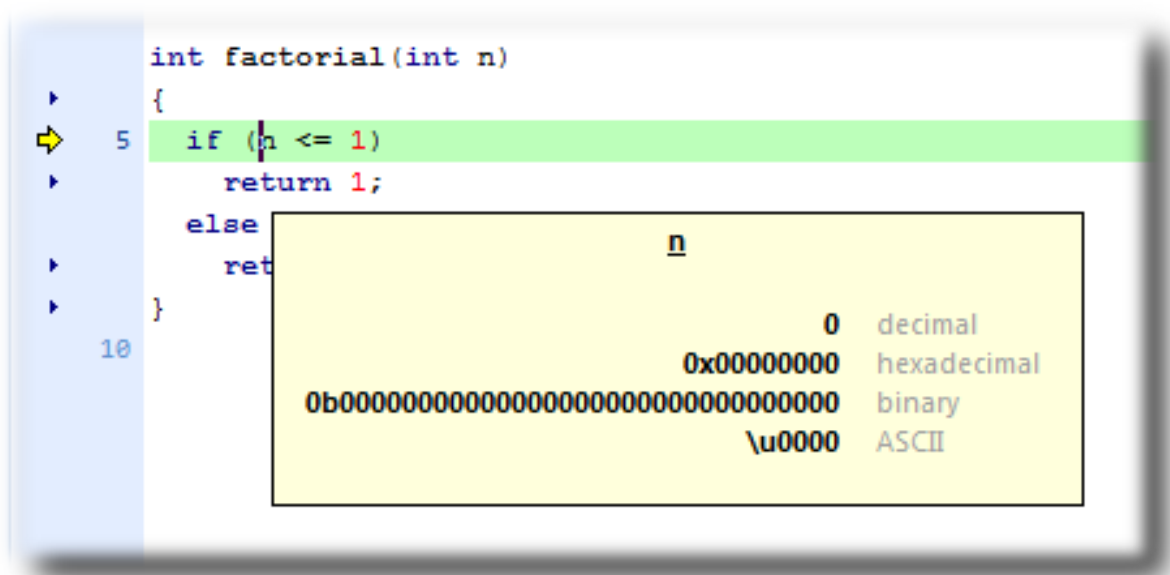


You can step out of a function by choosing **Debug > Step Out**, by typing **Shift+F11**, or by clicking the **Step Out** button on the **Debug** tool bar. You can also step to a specific statement by choosing **Debug > Run To Cursor**. To allow your application to run to the next breakpoint, choose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

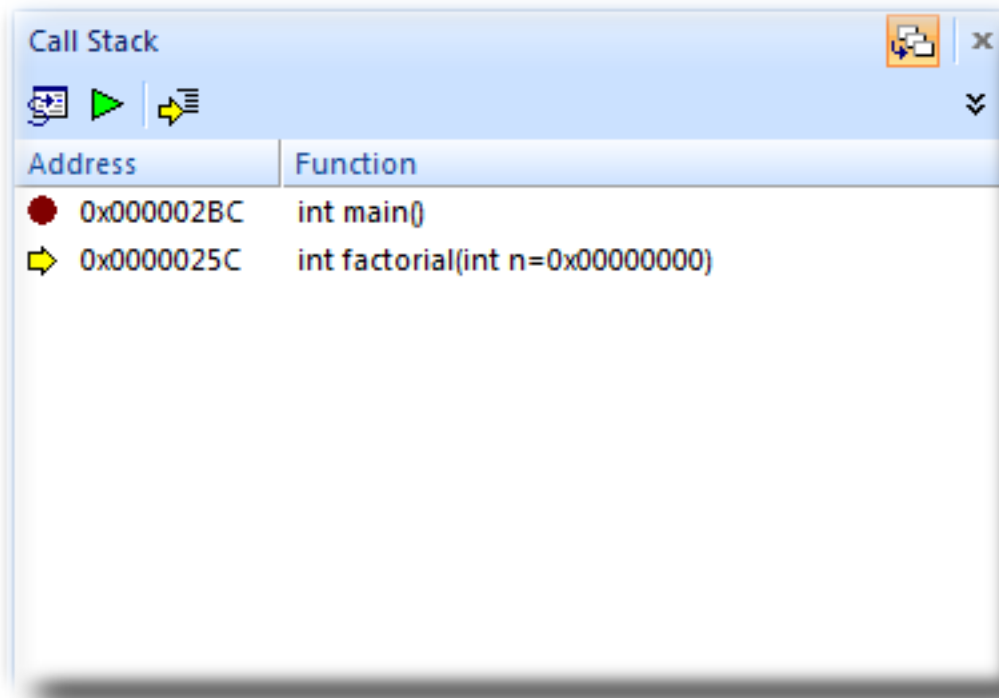


You can configure CrossWorks to display data tips in a variety of formats at the same time using the **Environment Options** dialog. You can also use the **Autos**, **Locals**, **Globals**, **Watch**, and **Memory** windows to view variables and memory. These windows are described in [CrossStudio User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

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. CrossWorks provides the tools you need to do so.

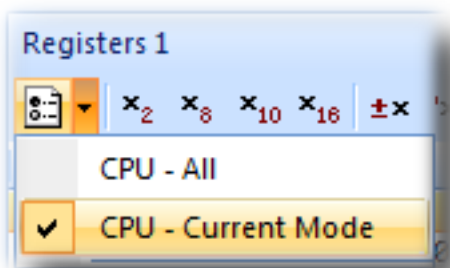
Setting up again

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:

Registers 1

x₂

x₈

x₁₀

x₁₆

±x

'x'

→

Name

Value

▼ CPU - Current Mode

r0

0x00000000

r1

0x00000000

r2

0x00000291

r3

0x00000000

r4

0xcdcdcdcd

r5

0xcdcdcdcd

r6

0xcdcdcdcd

r7

0x00000e60

r8

0xcdcdcdcd

r9

0xcdcdcdcd

r10

0xcdcdcdcd

r11

0xcdcdcdcd

r12

0xcdcdcdcd

sp(r13)

0x00000e60

lr(r14)

0x00000164

pc(r15)

0x0000029c

[-]

cpsr

0x800000ff

+

M

31

☑

T

1

☑

F

1

☑

I

1

☐

V

0

☐

C

0

☐

Z

0

☑

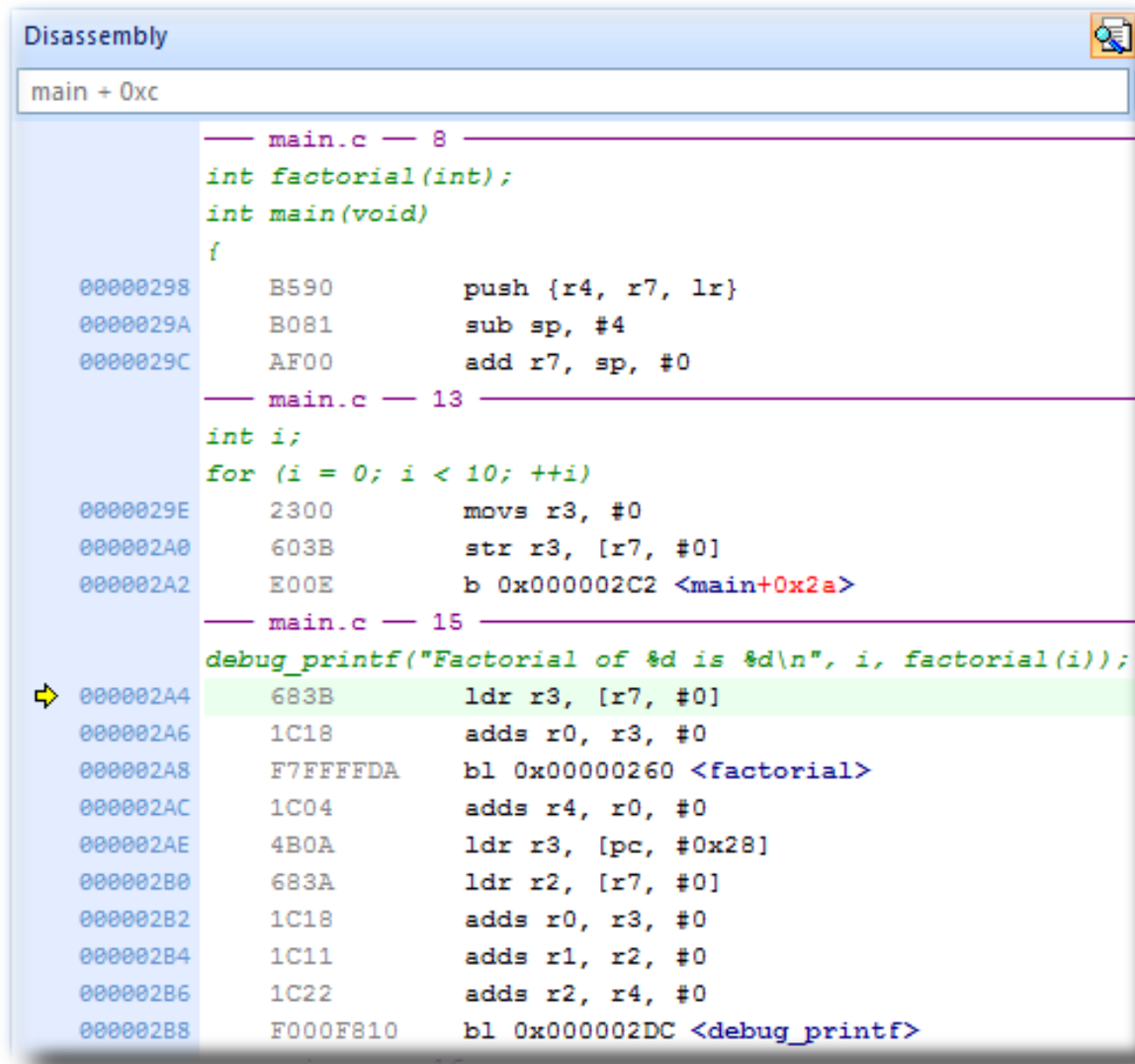
N

1

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**.

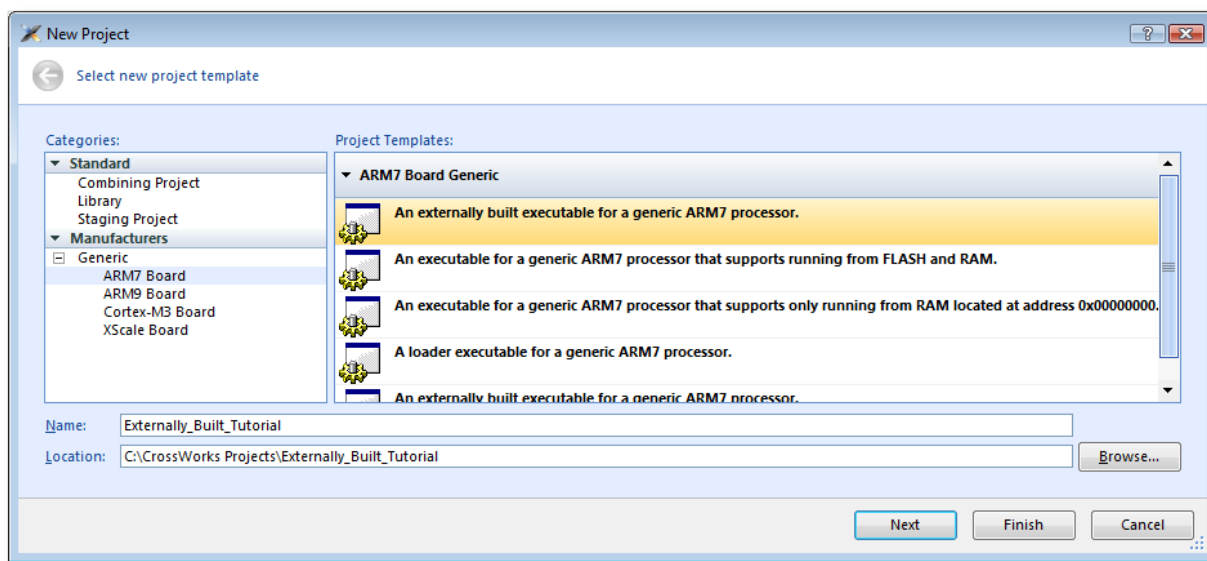
Debugging externally built applications

This section describes how to debug applications that were not built by CrossWorks. 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.

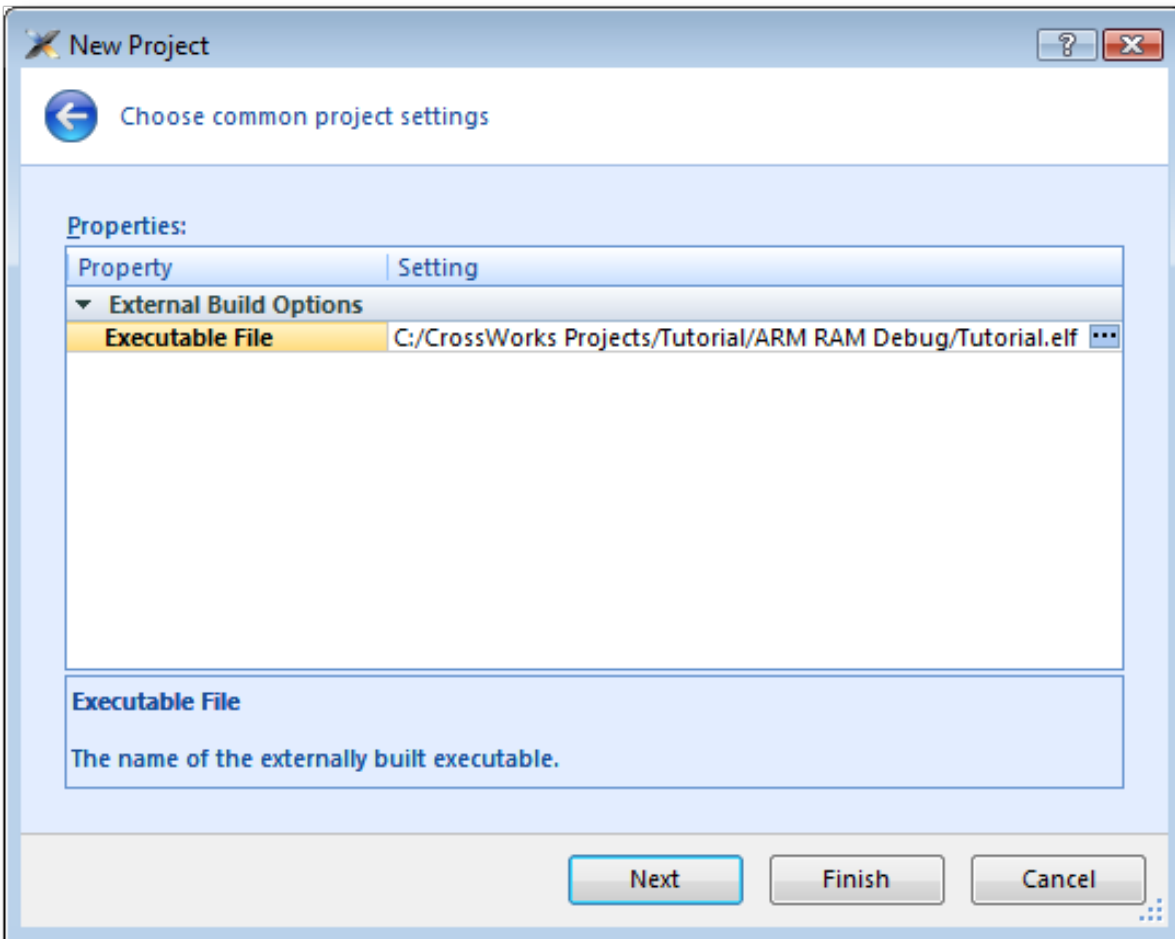
In the **Project Templates** pane, select the **An externally built executable for a generic ARM7 processor** icon, which selects the type of project to add.

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.

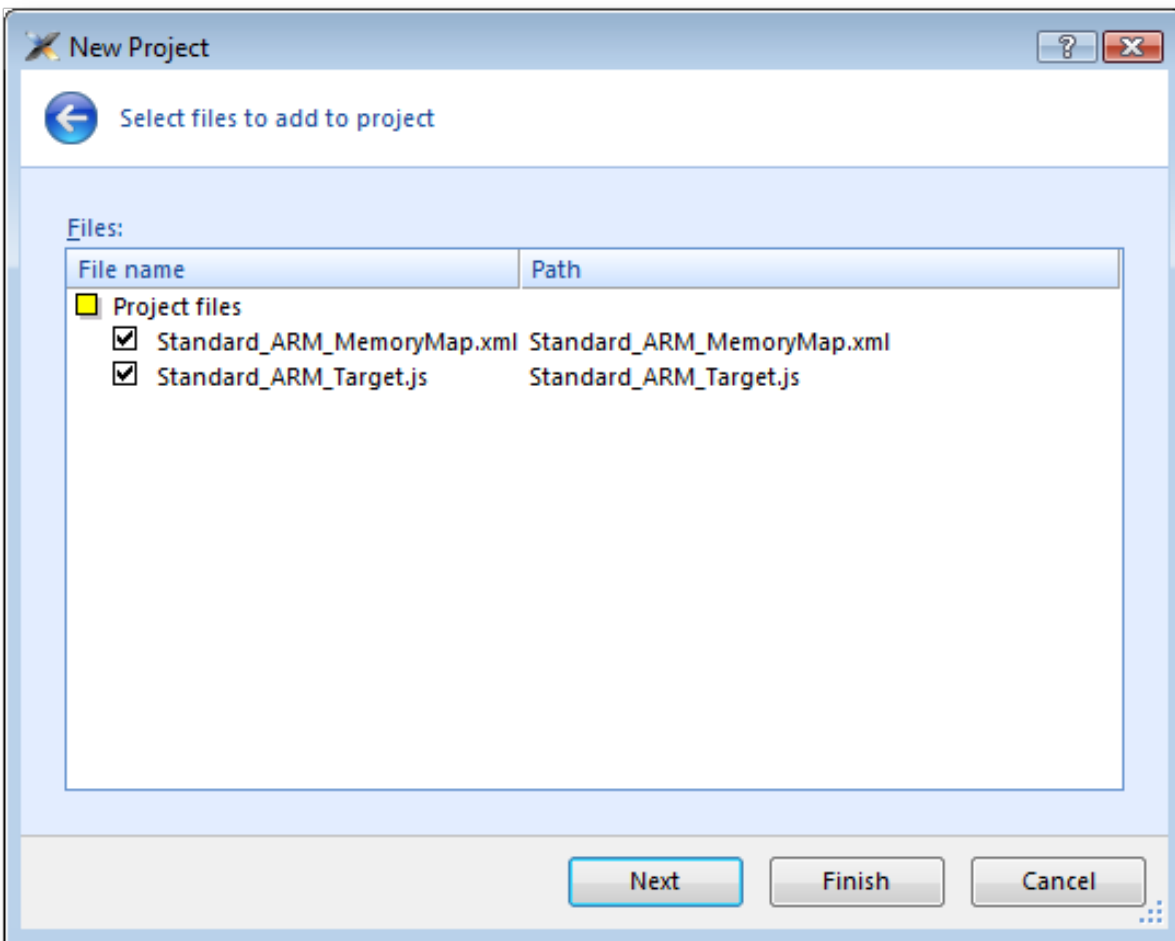
Click **OK**.

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



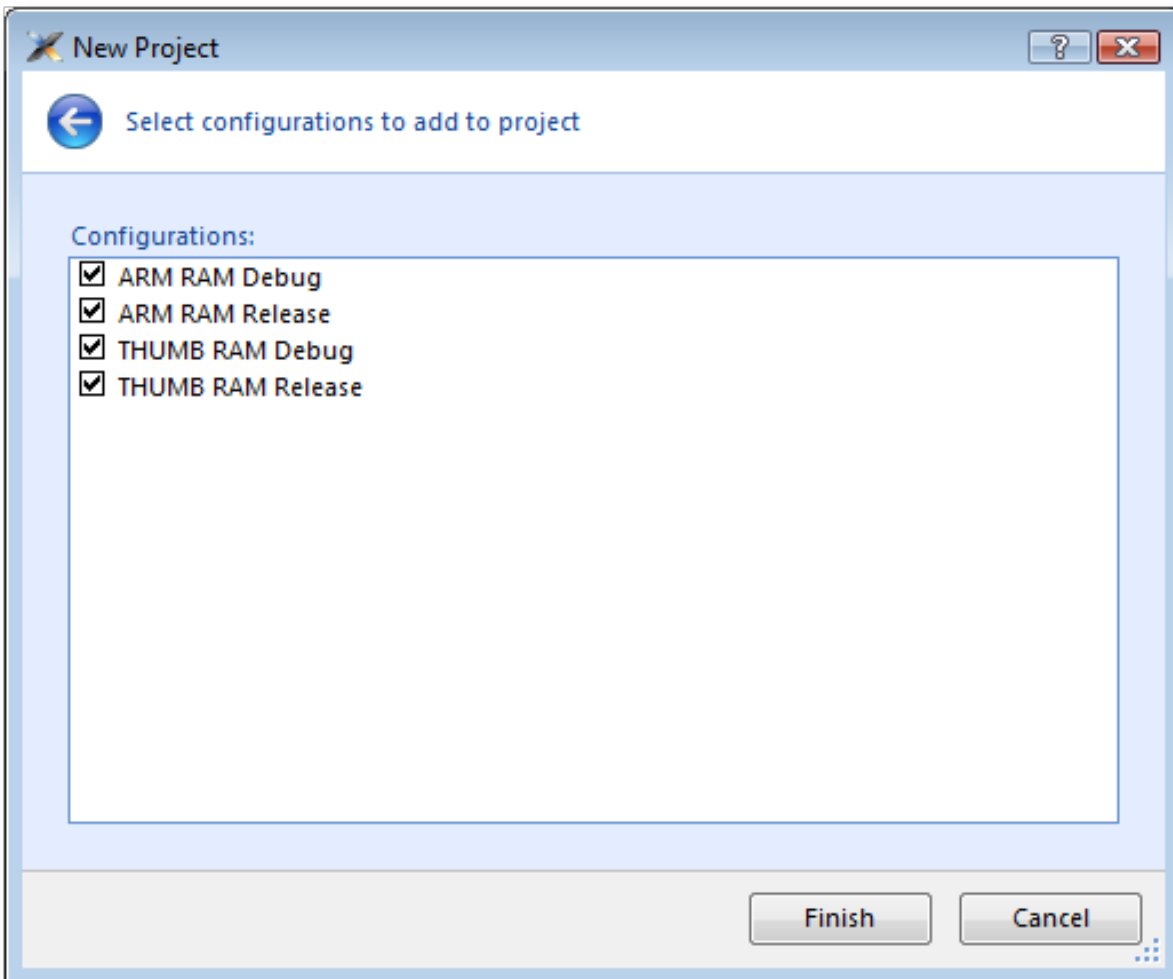
In the **Executable File** field, type the path to the `Tutorial.elf` executable file we generated earlier. For example, if the project was created in the `C:/CrossWorks Projects/Tutorial` directory and was built using the **ARM RAM Debug** configuration, the path to the executable file will be `C:/CrossWorks 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. Tap **Return** to display the menu.
3. Use the **Left** and **Right** keys to select the required menu.
4. Use the **Up** or **Down** key to select the required command or submenu.
5. Press **Enter** to execute the selected command.
6. 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 insertion position position in the editor window. For text files, the caret position pane displays the line number and column number of the insertion point 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.

- As you start dragging, all window groups, including hidden window groups, become visible.

- 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 for editing; when you return to **Debug** perspective on the next debug session, the windows will be restored to how you laid them out in that for debugging.

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 you have subscribed to. 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 subscribed 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 insertion point within 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 > Contents** or press **Ctrl+Alt+F1**.

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 **hzip**, 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 is not built by the CrossWorks internal build process.

Library: a group of object files collected into a single file (sometimes called an *archive*).

Externally Built Library: a library that is not built by the CrossWorks internal build process.

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.

Project properties and configurations

Project 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 build and a different value for a debug build.

Folders and Dynamic 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**.

Projects can also contain *dynamic folders* which will can show the directories and files contained in the file system in the project explorer. You can specify if the dynamic folder is recursive and use wildcards to include and exclude files.

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 **.cpp** or **.cxx** 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 object-file extension **.o** will be linked.

A file with the library-file extension **.a** 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.

Externally Built Executables

You can use an external build process for **Externally Built Executable** project types by setting the **Build Command** project property, for example to **make target**. Alternatively you can set command lines for specific build steps to compile/assemble and link. When you create an **Externally Built Executable** project type configurations will be created that create command lines for a variety of external tool chains.

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** or press **Ctrl+P, A**.

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.

To create a dynamic folder:

1. In the **Project Explore**, right click on the project to which you wish to add a new folder.
2. Choose **New Folder....**
3. Using the **New Folder** dialog name the folder and then show the dynamic folder options.
4. Specify the required **Source Folder** and the **Filter Specification**.

The files that match the filter specification in the source folder will appear in the newly created folder.

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**.

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 Options > Building > Show Build Information** property. To see the build commands themselves, set the **Environment Options > Building > 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.

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 > Previous 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

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

Global macros are store in the environment option [Global Macros](#).

To define a global macro:

1. Use **Tools > Options** to show the environment options dialog.
2. In the **Environment Options** dialog's **Building** group, select the **Global Macros** property.
3. Click the ellipsis button on the right.
4. Set the macro using the syntax *name = replacement text*.

Project macros

To define a project macro:

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)`.

You can also specify a default value for a macro if it is undefined using the `$(macro:default)` syntax. For example, `$(MyMacro:0)` would expand to 0 if the macro `MyMacro` has not been defined.

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.

Precompile Header File support

You can specify a single file in your project to be a precompiled header by setting the project property **Precompiled Header File** on the file node of the project. The file should be project local i.e. in the same directory as the project file and should include the header files that you wish to be compiled.

You must set the project level property **Enable Precompiled Header File** which supplies the output file generated by the precompiled header file to the compilation of each file in the project.

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" />
</Root>
```

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 order memory segments are processed is bottom up; so in this example the sections in the **SRAM** segment will be placed prior to the sections in the **FLASH** segment.

Multiple memory segments can be specified by separating them with a semicolon. In the following example, the **.stack** section will be placed in the SRAM2 memory segment if it exists in the memory map, otherwise it will be placed in the SRAM memory segment. Sections can only be placed in one segment, they will not be placed in a second segment when the first is full.

```
<Root name="Flash Section Placement">
```

```
<MemorySegment name="FLASH" >
  <ProgramSection name=".vectors" load="Yes" />
  <ProgramSection name=".text" load="Yes" />
</MemorySegment>
<MemorySegment name="SRAM2;SRAM" >
  <ProgramSection name=".stack" load="No" />
</MemorySegment>
</Root>
```

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** 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="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 copy the contents of the **.data** section to the **.data_run** section. To enable this, symbols named **__section-name_start**, **__section-name_end**, **__section-name_load_start** and **__section-name_load_end** are generated marking the section start, end, load start and load end addresses of each section. The startup code uses these symbols to copy the sections from their load positions to their run positions.

You can also create your own load and run section, for example the following placement file adds a **.mydata** section:

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

As the startup code doesn't know about this section, the following code will need to be added to the program to initialise the section:

```
/* Section image located in flash */
extern const unsigned char __mydata_load_start__[];
extern const unsigned char __mydata_load_end__[];

/* Where to locate the section image in RAM. */
extern unsigned char __mydata_start__[];
extern unsigned char __mydata_end__[];

...

/* Copy image from flash to RAM. */
memcpy(__mydata_start__,
       __mydata_load_start__,
       __mydata_end__ - __mydata_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 *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.
- Fetching files in the project from source control.
- Optionally locking and unlocking files in the project for editing.
- Comparing a file in the project with the latest version in source control.
- Updating a file in the project by merging changes from the latest version in source control.
- Committing changes made to project files into source control.

Configuring source-control providers

CrossStudio supports Subversion, Git, and Mercurial as source-control systems. To enable CrossStudio to utilize source-control features, you need to install, on your operating system, the appropriate command line client for the source-control systems that you will use.

Once you have installed the command line client, you must configure CrossStudio to use it.

To configure Subversion:

1. Choose **Tools > Options** or press **Alt+,**.
2. Select the **Source Control** category in the options dialog.
3. Set the **Executable** property of the **Subversion Options** group to point to Subversion `svn` command. On Windows operating systems, the Subversion command is `svn.exe`.

To configure Git:

1. Choose **Tools > Options** or press **Alt+,**.
2. Select the **Source Control** category in the options dialog.
3. Set the **Executable** property of the **Git Options** group to point to Git `git` command. On Windows operating systems, the Git command is `git.exe`.

To configure Mercurial:

1. Choose **Tools > Options** or press **Alt+,**.
2. Select the **Source Control** category in the options dialog.
3. Set the **Executable** property of the **Mercurial Options** group to point to Git `hg` command. On Windows operating systems, the Git command is `hg.exe`.

Connecting to the source-control system

When CrossStudio loads a project, it examines the file system folder that contains the project to determine the source-control system the project uses. If CrossStudio cannot determine, from the file system, the source-control system in use, it disables source-control integration.

That is, if you have not set up the paths to the source-control command line clients, even if a working copy exists and the appropriate command line client is installed, CrossStudio cannot establish source-control integration for the project.

User credentials

You can set the credentials that the source-control system uses, for commands that require credentials, using **VCS > Options > Configure**. From here you can set the user name and password. These details are saved to the session file (the password is encrypted) so you won't need to specify this information each time the project is loaded.

Note

CrossStudio has no facility to create repositories from scratch, nor to clone, pull, or checkout repositories to a working copy: it is your responsibility to create a working copy outside of CrossStudio using your selected command-line client or Windows Explorer extension.

The "Tortoise" products are a popular set of tools to provide source-control facilities in the Windows shell. Use Google to find **TortoiseSVN**, **TortoiseGit**, and **TortoiseHG** and see if you like them.

File source-control status

Determining the source-control status of a file can be expensive for large repositories, so CrossWorks updates the source-control status in the background. Priority is given to items that are displayed.

A file will be in one of the following states:

Clean:The file is in source control and matches the tip revision.

Not Controlled:The file is not in source control.

Conflicted:The file is in conflict with changes made to the repository.

Locked:The file is locked.

Update Available:The file is older than the most-recent version in source control.

Added:The file is scheduled to be added to the repository.

Removed:The file is scheduled to be removed from the repository.

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 **VCS > Refresh**.

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 files to the source-control system:

1. In the **Project Explorer**, select the file to add. If you select a folder, project, or solution, any eligible child items will also be added to source control.
2. choose **Source Control > Add** or press **Ctrl+R, A**.
3. The dialog will list the files that can be added.
4. In that dialog, you can deselect any files you don't want to add to source control.
5. Click **Add**.

Note

Files are scheduled to be added to source control and will only be committed to source control (and seen by others) when you commit the file.

Enabling the **VCS > Options > Add Immediately** option will bypass the dialog and immediately add (but not commit) the files.

Updating files

To update files from source control:

1. In the **Project Explorer**, select the file to update. If you select a folder, project, or solution, any eligible child items will also be updated from source control.
2. choose **Source Control > Update** or press **Ctrl+R, U**.
3. The dialog will list the files that can be updated.
4. In that dialog, you can deselect any files you don't want to update from source control.
5. Click **Update**.

Note

Enabling the **VCS > Options > Update Immediately** option will bypass the dialog and immediately update the files.

Committing files

To commit files:

1. In the **Project Explorer**, select the file to commit. If you select a folder, project, or solution, any eligible child items will also be committed.
2. Choose **Source Control > Commit** or press **Ctrl+R, C**.
3. The dialog will list the files that can be committed.
4. In that dialog, you can deselect any files you don't want to commit and enter an optional comment.
5. Click **Commit**.

Note

Enabling the **VCS > Options > Commit Immediately** option will bypass the dialog and immediately commit the files without a comment.

Reverting files

To revert files:

1. In the **Project Explorer**, select the file to revert. If you select a folder, project, or solution, any eligible child items will also be reverted.
2. Choose **Source Control > Revert** or press **Ctrl+R, V**.
3. The dialog will list the files that can be reverted.
4. In that dialog, you can deselect any files you don't want to revert.
5. Click **Revert**.

Note

Enabling the **VCS > Options > Revert Immediately** option will bypass the dialog and immediately revert files.

Locking files

To lock files:

1. In the **Project Explorer**, select the file to lock. If you select a folder, project, or solution, any eligible child items will also be locked.
2. Choose **Source Control > Lock** or press **Ctrl+R, L**.
3. The dialog will list the files that can be locked.
4. In that dialog, you can deselect any files you don't want to lock and enter an optional comment.
5. Click **Lock**.

Note

Enabling the **VCS > Options > Lock Immediately** option will bypass the dialog and immediately lock files without a comment.

Unlocking files

To unlock files:

1. In the **Project Explorer**, select the file to lock. If you select a folder, project, or solution, any eligible child items will also be unlocked.
2. Choose **Source Control > Unlock** or press **Ctrl+R, N**.
3. The dialog will list the files that can be unlocked.
4. In that dialog, you can deselect any files you don't want to unlock.
5. Click **Unlock**.

Note

Enabling the **VCS > Options > Unlock Immediately** option will bypass the dialog and immediately unlock files.

Removing files from source control

To remove files from source control:

1. In the **Project Explorer**, select the file to remove. If you select a folder, project, or solution, any eligible child items will also be removed.
2. choose **Source Control > Remove** or press **Ctrl+R, R**.
3. The dialog will list the files that can be removed.
4. In that dialog, you can deselect any files you don't want to remove.
5. Click **Remove**.

Note

Files are scheduled to be removed from source control and will still be and seen by others, giving you the opportunity to revert the removal. When you commit the file, the file is removed from source control.

Enabling the **VCS > Options > Remove Immediately** option will bypass the dialog and immediately remove (but not commit) files.

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 > Compare**.

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)*.

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
CrossStudio Status	The source-control status of working copy as viewed by CrossStudio.
last Author	The author of the file's head revision.
Path: Relative	The item's path relative to the repository root.
Path: Repository	The pathname of the file in the source-control system, typically a URL.
Path: Working Copy	The pathname of the file in the working copy.
Provider	The name of the source-control system managing this file.
Provider Status	The status of the file as reported by the source-control provider.
Revision: Local	The revision number/name of the local file.
Revision: Remote	The revision number/name of the most-recent version in source control.
Status: In Conflict?	If Yes , updates merged into the file using Update conflict with the changes you made locally; if No , the file is not locked. When conflicted, must resolve the conflicts and mark them Resolved before committing the file.
Status: Locked?	If Yes , the file is lock by you; if No , the file is not locked.
Status: Modified?	If Yes , the checked-out file differs from the version in the source control system; if No , they are identical.
Status: Update Available?	If Yes , the file in the project location is an old version compared to the latest version in the source-control system use Update to merge in the latest changes.

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 <code>svn</code> executable.
Lock Supported	If Yes , check out and undo check out operations are supported. Check out will issue the <code>svn lock</code> command; check in and undo check out will issue the <code>svn unlock</code> command.
Authentication	Selects whether authentication (user name and password) is sent with every command.
Show Updates	Selects whether the update (<code>-u</code> flag) is sent with status requests in order to show that new versions are available in the repository. Note that this requires a live connection to the repository: if you are working without a network connection to your repository, you can disable this switch and continue to enjoy source control status information in the Project Explorer and Pending Changes windows.

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 are implemented using Subversion commands. Mapping CrossWorks source-control operations to Subversion source-control operations is straightforward:

Operation	Command
Commit	<code>svn commit</code> for the file, with optional comment.
Update	<code>svn update</code> for each file.
Revert	<code>svn revert</code> for each file.

Resolved	<code>svn resolved</code> for each file.
Lock	<code>svn lock</code> for each file, with optional comment.
Unlock	<code>svn unlock</code> for each file.
Add	<code>svn add</code> for each file.
Remove	<code>svn remove</code> for each file.
Source Control Explorer	<code>svn list</code> with a remote root. <code>svn mkdir</code> 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 <code>cvs edit</code> command; any check-in or undo-check-out operation will issue the <code>cvs unedit</code> command; the status operation will issue the <code>cvs ss</code> command.
Executable	The path to the <code>cvs</code> executable.
Login/Logout Required	If Yes , Connect will issue the <code>cvs login</code> 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	<code>cvs status</code> and optional <code>cvs editors</code> for local directories in CVS control. <code>cvs rls -e</code> for directories in the repository.
Add To Source Control	<code>cvs add</code> for each directory not in CVS control. <code>cvs add</code> for the file. <code>cvs commit</code> for the file and directories.
Get Latest	<code>cvs update -l -d</code> for each directory not in CVS control. <code>cvs update</code> to merge the local file. <code>cvs update -C</code> to overwrite the local file.

Check Out	Optional <code>cvs update -C</code> to get the latest version. <code>cvs edit</code> to lock the file.
Undo Check Out	<code>cvs unedit</code> to unlock the file. Optional <code>cvs update</code> to get the latest version.
Check In	<code>cvs commit</code> for the file.
Source Control Explorer	<code>cvs rls -e</code> with a remote root starting with <code>..</code> <code>cvs import</code> 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 website, you cannot use the **Package Manager** to install packages and it will be necessary to manually install them.

To manually install a package:

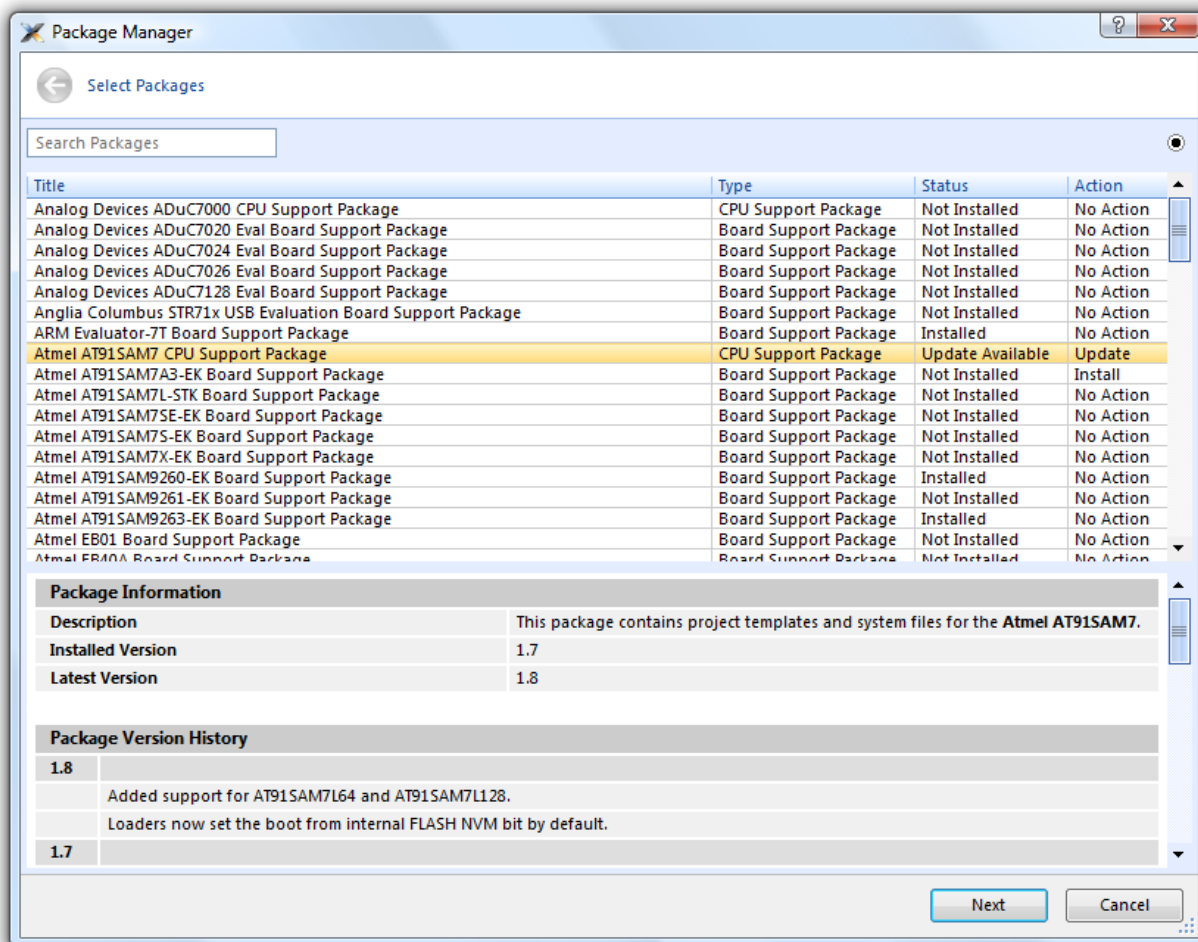
1. Choose **Tools > Packages > Manually Install Packages**.
2. Select one or more package files you want to install.
3. Click **Open** to install the packages.

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 > Package Manager** and set 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 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 > Package Manager** and set 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 > Package Manager** and set 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 > Package Manager** and set 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 reveal or hide the node's children. Double-clicking a project node selects 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
	Add a new file to the active project using the New File dialog.
	Add existing files to the active project.

	Remove files, folders, projects, and links from the project.
	Create a new folder in the active project.
	Menu of build operations.
	Disassemble the active project.
	Menu of Project Explorer options.
	Display 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 filenote 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.

To activate the Source Navigator:

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

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 <code>private</code> or a function declared with <code>static</code> linkage.
	A C++ member function declared <code>protected</code> .
	A C++ member function declared <code>public</code> or a function declared with <code>extern</code> linkage.
	A C++ member variable declared <code>private</code> or a variable declared with <code>static</code> linkage.
	A C++ member variable declared <code>protected</code> .
	A C++ member variable declared <code>public</code> or a variable declared with <code>extern</code> linkage.

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 all files in the active project, and any dependent project, and updates the **Source Navigator** with the changes. Parsing progress is shown as a progress bar in the in the **Source Navigator** window. Errors and warnings detected during parsing are sent to the Source Navigator Log in the **Output** window you can show the log quickly by clicking the **Show Source Navigator Log** tool button on the **Source Navigator** toolbar.

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**

References window

The **References** window shows the results of the last **Find References** operation. The **Find References** facility is closely related to the **Source Navigator** in that it indexes your project and searches for references within the active source code regions.

To activate the References window:

If you have hidden the **References** window and want to see it again:

Choose **Navigate > References** or press **Ctrl+Alt+R**.

To find all references in a project:

1. Open a source file that is part of the active project, or one of its dependent projects.
2. In the editor, move the insertion point within the name of the function, variable, method, or macro to find.
3. Choose **Search > Find References** or press **Alt+R**.
4. CrossStudio shows the **References** window, without moving focus, and searches your project in the background.

You can also find references directly from the text editor's context menu: right-click the item to find and choose **Find References**. As a convenience, CrossStudio is configured to also run **Find References** when you Alt+Right-click in the text editorsee [Mouse-click accelerators](#).

To search within the results:

Type the text to search for in the Reference window's search box. As you type, the search results are narrowed.

Click the close button to clear the search text and show all references.

To replace within the results:

Type the replacement text in the Reference window's replace box.






Use the buttons to navigate and replace the text.

The documents that have had replaced text will appear unsaved in the text editor.








Symbol browser window

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
	Group symbols by source filename.
	Group symbols by symbol type (equates, functions, labels, sections, and variables).
	Group symbols by the section where they are defined.
	Move the insertion point to the statement that defined the symbol.
	Select 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
	<i>Private Equate</i> A private symbol not defined relative to a section.
	<i>Public Equate</i> A public symbol that is not defined relative to a section.
	<i>Private Function</i> A private function symbol.
	<i>Public Function</i> A public function symbol.
	<i>Private Label</i> A private data symbol, defined relative to a section.
	<i>Public Label</i> A public data symbol, defined relative to a section.
	<i>Section</i> A program section.

Choosing what to show

To activate the Symbol Browser window:

Choose **Navigate > 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.

Frame Size:The amount of stack space used by a call to the function symbol. If the source file that defines the symbol is compiled with debugging information turned off, frame size 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 *combobox* 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 **Navigate > 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 **Navigate > 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.

Stack usage window

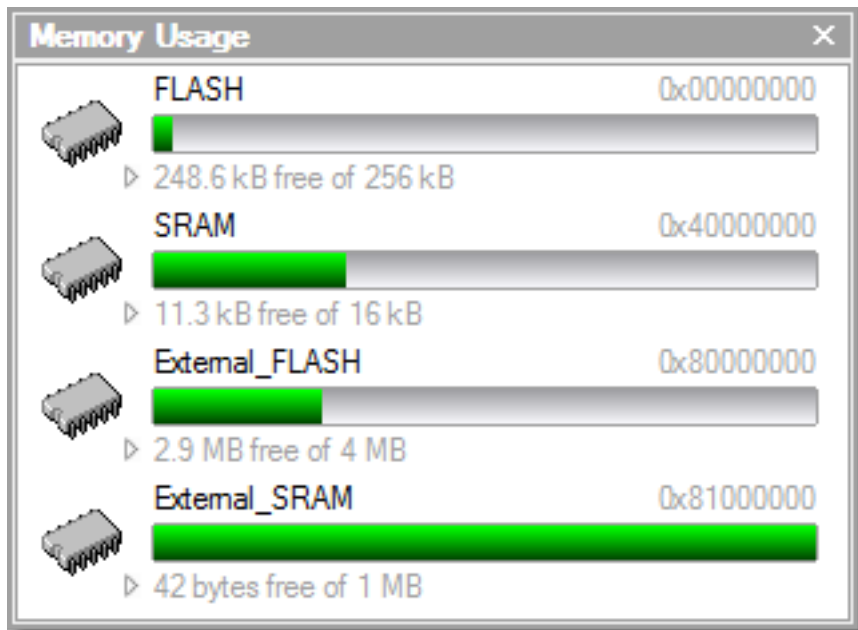
The **Stack Usage Window** finds the call paths of your linked application and displays them as a call tree together with their minimal stack requirements. A call path of your application is any function that has been linked in but has no direct call made to it but will make calls to other functions. The main function is the most obvious example of a call path, an interrupt handler or a function that is called only as a function pointer are other examples. To use the stack usage window your linked application must be compiled with debugging information enabled.

User interface

Button	Description
	Move the insertion point to the statement that defined the symbol.
	Collapse the selected open call tree.
	Open the selected open call tree.
	Show only the deepest call path through the selected call tree.

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.

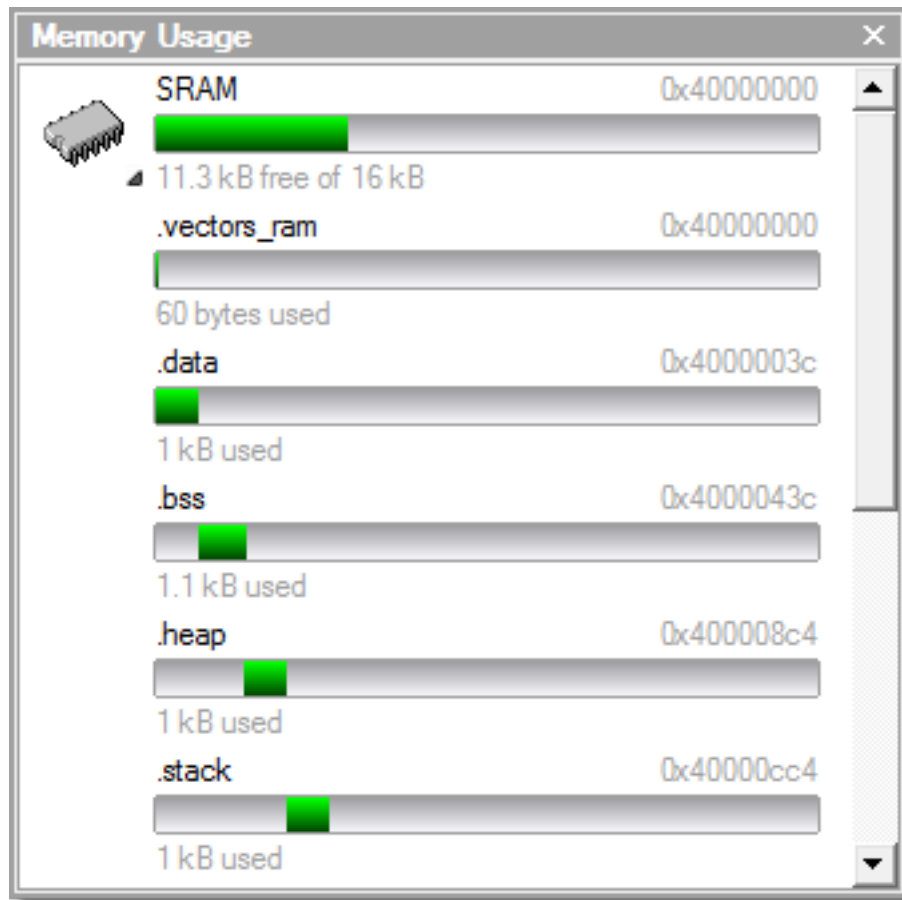
To activate the Memory Usage window:

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

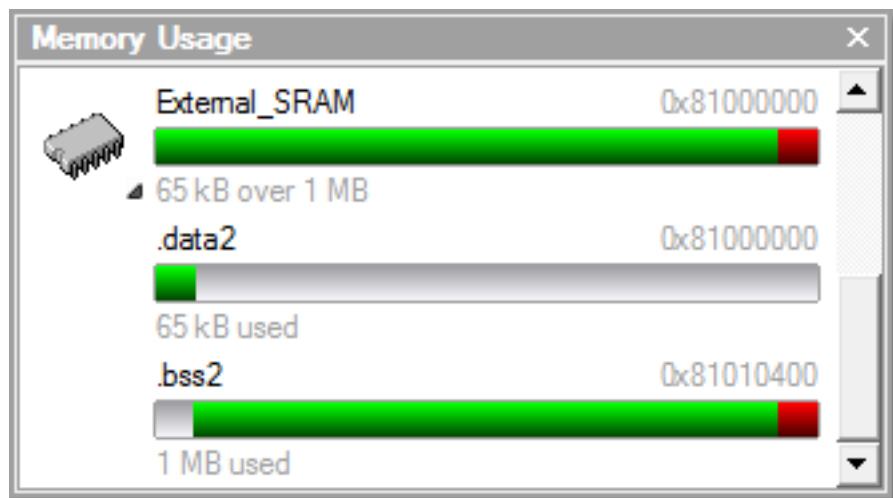
The **Memory Usage** window can also be used to visualize how program sections have been placed in memory. To display the program sections, simply click the memory segment to expand it; or, alternatively, right-click and choose **Show Memory Sections** from the shortcut menu.



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 represent 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 window](#).

Bookmarks window

The **Bookmarks** window contains a list of bookmarks that are set in the project. The bookmarks are stored in the session file associated with the project and persist across runs of CrossStudio. If you remove the session file, the bookmarks associated with the project are lost.

User interface

Button	Description
	Toggle a bookmark at the insertion point in the active editor. Equivalent to choosing Edit > Bookmarks > Toggle Bookmark or pressing Ctrl+F2 .
	Go to the previous bookmark in the bookmark list. Equivalent to choosing Edit > Bookmarks > Previous Bookmark or pressing Alt+Shift+F2 .
	Go to the next next bookmark in the bookmark list. Equivalent to choosing Edit > Bookmarks > Next Bookmark or pressing Alt+F2 .
	Clear all bookmarks. You confirm the action using a dialog. Equivalent to choosing Edit > Bookmarks > Clear All Bookmarks or pressing Ctrl+K, Alt+F2 .
	Selects the fill color for newly created bookmarks.

Double-clicking a bookmark in the bookmark list moves focus to the bookmark.

You can set bookmarks with the mouse or using keystrokes. See [Using bookmarks](#).

Code Outline Window

The **Code Outline** window shows the structure of the text of the focused code editor. For C and C++ documents the top level symbols and types are displayed, for XML documents the nodes are displayed. For C and C++ documents the **Preview** tab can display documentation on the top level symbols and types. The default standard doxygen commands are supported for example:

```
/**
 * \brief Convert a given full parsed comment to an XML document.
 *
 * A Relax NG schema for the XML can be found in comment-xml-schema.rng file
 * inside clang source tree.
 *
 * \param Comment a \c CXComment_FullComment AST node.
 *
 * \returns string containing an XML document.
 */
CINDEX_LINKAGE CXString clang_FullComment_getAsXML(CXComment Comment);
```

Analyzing Source Code

The **Analyze** action is available on the context menu of the project explorer at project, folder and file level. The analyze action will run the <https://clang.llvm.org/extra/clang-tidy> linter tool on the C/C++ files selected by the project explorer node and display warnings in the output window. The default checks will be the same as the clang analyzer. You can enable additional checks by setting the **Clang Tidy Checks** project property. For example you can enable the bugprone code constructs check and disable a specific clang analyzer diagnostic check as follows

```
bugprone-*  
-clang-diagnostic-parentheses-equality
```

You can also set the project property **Analyze After Compile** which will run the analyzer each time the compiler is run.

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 browse 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 the mouse or the keyboard's cursor keys.

Using the mouse

You can move the insertion point within a document by clicking the mouse inside the editor window.

Using the keyboard

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

Keystroke	Description
Up	Move the insertion point up one line
Down	Move the insertion point down one line
Left	Move the insertion point left one character
Right	Move the insertion point right one character
Home	Move the insertion point to the first non-whitespace character on the line pressing Home a second time moves the insertion point to the leftmost column
End	Move the insertion point to the end of the line
PageUp	Move the insertion point up one page
PageDown	Move the insertion point down one page
Ctrl+Home	Move the insertion point to the start of the document
Ctrl+End	Move the insertion point to the end of the document
Ctrl+Left	Move the insertion point left one word
Ctrl+Right	Move the insertion point right one word

CrossStudio offers additional movement keystrokes, though most users are more comfortable using repeated simple keystrokes to accomplish the same thing:

Keystroke	Description
Alt+Up	Move the insertion point up five lines
Alt+Down	Move the insertion point down five lines
Alt+Home	Move the insertion point to the top of the window
Alt+End	Move the insertion point to the bottom of the window
Ctrl+Up	Scroll the document up one line in the window without moving the insertion point

Ctrl+Down	Scroll the document down one line in the window without moving the insertion point
------------------	--

If you are editing source code, the are source-related keystrokes too:

Keystroke	Description
Ctrl+PgUp	Move the insertion point backwards to the previous function or method.
Ctrl+PgDn	Move the insertion point forwards to the next function or method.

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 insertion point 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**.

When overstrike mode is enabled, the mode indicator changes from **INS** to **OVR** 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 place the editor into overstrike mode.

To add or insert text on multiple lines:

1. Hold down the **Alt** key and use block selection to mark the place text is to be inserted.
2. Enter the text using the keyboard.

Deleting text

The text editor supports the following common editing keystrokes:

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

To delete a text block:

1. Hold down the **Alt** key and use block selection to mark the text you want to delete.
2. Press **Delete** or **Backspace** to delete it.

To delete characters on multiple lines:

1. Hold down the **Alt** key and use block selection to mark the lines.
2. Press **Delete** or **Backspace** as many times as needed to delete the characters.

Using the clipboard

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 mouse to end selecting.

To select a block of text with the keyboard:

Hold down the **Shift+Alt** keys while using the cursor keys.

To select a block of text with the mouse:

1. Hold down the **Alt** key.
2. Click the start of the selection.
3. Drag the mouse to mark the selection.
4. Release the mouse to end selecting.

To copy selected text to the clipboard:

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

The standard Windows 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 standard Windows 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 standard Windows 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 standard Windows 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 > Others > Undo All** or press **Ctrl+K, Ctrl+Z**.

To redo one editing action:

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

The standard Windows 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 > Others > 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 mouse pointer 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.

If you press **Ctrl+F** a second time, CrossStudio pops up the standard **Find** dialog to search the file. If you wish to bring up the **Find** dialog without pressing **Ctrl+F** twice, choose **Search > Find**.

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 uses the **Tab** key to increase or decrease the indentation level of the selected text.

To increase indentation:

Select the text to indent.

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

To decrease indentation:

Select the text to indent.

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

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 indentation size:

Choose **Tools > Options** or press **Alt+,**.

Select the **Languages** page.

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

You can choose to use spaces or tab characters to fill whitespace when indenting.

To set tab or space fill when indenting:

Choose **Tools > Options** or press **Alt+,**.

Select the **Languages** page.

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 insertion point 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:

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

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

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

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

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

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

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

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.

Adjusting 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+K, 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.

Using bookmarks

To edit a document elsewhere and then return to your current location, add a bookmark. The **Bookmarks** window maintains a list of the bookmarks set in source files see [Bookmarks window](#).

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

To place a bookmark using the mouse:

1. Right-click the margin gutter where the bookmark should be set.
2. Choose **Toggle Bookmark**.

The default color to use for new bookmarks is configured in the **Bookmarks** window. You can choose a specific color for the bookmark as follows:

1. Press and hold the **Alt** key.
2. Click the margin gutter where the bookmark should be set.
3. From the palette, click the bookmark color to use for the bookmark.

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 moves 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 moves to the last bookmark in the document.

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 is no longer set.

To remove all bookmarks in a document:

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

Quick reference for bookmark operations

Keystroke	Menu	Description
Ctrl+F2	Edit > Bookmarks > Toggle Bookmark	Toggle a bookmark at the insertion point.
Ctrl+K, 0		Clear the bookmark at the insertion point.
F2	Edit > Bookmarks > Next Bookmark In Document	Move the insertion point to next bookmark in the document.
Shift+F2	Edit > Bookmarks > Previous Bookmark In Document	Move the insertion point to previous bookmark in the document.
Ctrl+Q, F2	Edit > Bookmarks > First Bookmark In Document	Move the insertion point to the first bookmark in the document.
Ctrl+Q, Shift+F2	Edit > Bookmarks > Last Bookmark In Document	Move the insertion point to the last bookmark in the document.
Ctrl+K, F2	Edit > Bookmarks > Clear Bookmarks In Document	Clear all bookmarks in the document.
Alt+F2	Edit > Bookmarks > Next Bookmark	Move the insertion point to the next bookmark in the Bookmarks list.
Alt+Shift+F2	Edit > Bookmarks > Previous Bookmark	Move the insertion point to the previous bookmark in the Bookmarks list.
Ctrl+Q, Alt+F2	Edit > Bookmarks > First Bookmark	Move the insertion point to the first bookmark in the Bookmarks list.
Ctrl+Q, Alt+Shift+F2	Edit > Bookmarks > Last Bookmark	Move the insertion point to the last bookmark in the Bookmarks list.
Ctrl+K, Alt+F2	Edit > Bookmarks > Clear All Bookmarks	Clear all bookmarks in all documents.

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 the current or in all open documents.

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**.

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 a specific item from the clipboard ring:

1. Move the insertion point to the position to paste the item in the document.
2. Click the arrow at the right of the item to paste.
3. Choose *Paste* from the pop-up menu.

or

1. Click the item to paste to make it the current item.
2. Move the insertion point to the position to paste the item in 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 insertion point 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. Click the arrow at the right of the item to remove.
2. Choose **Delete** from the pop-up menu.

To remove all items from the clipboard ring:

Choose **Edit > Clipboard Ring > Clear Clipboard Ring**.

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** category 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.

Mouse-click accelerators

CrossStudio provides a number of mouse-click accelerators in the editor that speed access to commonly used functions. The mouse-click accelerators are user configurable using **Tools > Options**.

Default mouse-click assignments

Click	Default
Left	Not configurable start selection.
Shift+Left	Not configurable extend selection.
Ctrl+Left	Select word.
Alt+Left	Execute Go To Definition .
Middle	No action.
Shift+Middle	Display Go To Include menu.
Ctrl+Middle	No action.
Alt+Middle	Display Go To Method menu.
Right	Not configurable show context menu.
Shift+Right	No action.
Ctrl+Right	No action.
Alt+Right	Execute Find References .

Each accelerator can be assigned one of the following actions:

*Default:*The system default for that click.

*Go To Definition:*Go to the definition of the item clicked, equivalent to choosing **Navigate > Go To Definition** or pressing **Alt+G**.

*Find References:*Find references to the item clicked, equivalent to choosing **Search > Find References** or pressing **Alt+R**.

*Find in Solution:*Textually find the item clicked in all the files in the solution, equivalent to choosing **Search > Find Extras > Find in Solution** or pressing **Alt+U**.

*Find Help:*Use F1-help on the item clicked, equivalent to choosing **Help > Help** or pressing **F1**.

*Go To Method:*Display the **Go To Method** menu, equivalent to choosing **Navigate > Find Method** or pressing **Ctrl+M**.

*Go To Include:*Display the **Go To Include** menu, equivalent to choosing **Navigate > Find Include** or pressing **Ctrl+Shift+M**.

*Paste:*Paste the clipboard at the position clicked, equivalent to choosing **Edit > Paste** or pressing **Ctrl+V**.

Configuring Mac OS X

On Mac OS X you must configure the mouse to pass middle clicks and right clicks to the application if you wish to use mouse-click accelerators in CrossStudio. Configure the mouse preferences in the **Mouse** control panel in Mac OS X **System Preferences** to the following:

Right mouse button set to **Secondary Button**.
Middle mouse button set to **Button 3**.

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).
\t	Match ASCII horizontal tab character (ASCII code 9).
\v	Match ASCII vertical tab character.
\xhhh	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.

<code>{n}</code>	Match <i>n</i> occurrences of the preceding expression.
<code>{n,}</code>	Match at least <i>n</i> occurrences of the preceding expression.
<code>{,m}</code>	Match at most <i>m</i> occurrences of the preceding expression.
<code>{n,m}</code>	Match at least <i>n</i> and at most <i>m</i> occurrences of the preceding expression.
<code>^</code>	Beginning of line.
<code>\$</code>	End of line.
<code>\b</code>	Word boundary.
<code>\B</code>	Non-word boundary.
<code>(e)</code>	Capture expression <i>e</i> .
<code>\n</code>	Back-reference to <i>n</i> 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
<code>u\w.d</code>		Search for any-length string containing one or more word characters beginning with the character <i>u</i> and ending in the character <i>d</i> .
<code>^.*;\$</code>		Search for any lines ending in a semicolon.
<code>(typedef.+s+)(\S+);</code>	<code>\1TEST_\2;</code>	Find C type definition and insert the string TEST onto the beginning of the type name.

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
	Display the selected item in binary.
	Display the selected item in octal.
	Display the selected item in decimal.
	Display the selected item in hexadecimal.
	Display the selected item as a signed decimal.
	Display the selected item as a character or Unicode character.
	Set the range displayed in the active Memory window to span the memory allocated to the selected item.
	Sort variables alphabetically by name.
	Sort 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
	Display the selected item in binary.
	Display the selected item in octal.
	Display the selected item in decimal.
	Display the selected item in hexadecimal.
	Display the selected item as a signed decimal.
	Display the selected item as a character or Unicode character.
	Set the range displayed in the active Memory window to span the memory allocated to the selected item.
	Sort variables alphabetically by name.
	Sort 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
	Display the selected item in binary.
	Display the selected item in octal.
	Display the selected item in decimal.
	Display the selected item in hexadecimal.
	Display the selected item as a signed decimal.
	Display the selected item as a character or Unicode character.
	Set the range displayed in the active Memory window to span the memory allocated to the selected item.
	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.
	View pointer value.
	Set watch value to zero.
	Set watch value to one.
	Increment watched variable by one.
	Decrement watched variable by one.

	Negated watched variable.
	Invert watched variable.
	View the properties of the watch value.

You can view details of the watched 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
	Display the CPU, special function register, and peripheral register groups.
	Display the CPU registers.
	Hide the CPU registers.
	Force-read a register, ignoring the access property of the register.
	Update the selected register group.
	Set 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** window shows the contents of the connected target's memory areas and allows the memory to be edited. CrossStudio provides four memory windows, you can configure each memory window to display different memory ranges.

The **Memory** window has a toolbar and a data display/edit area

Field/Button	Description
<i>Address</i>	Address to display. This can be a numeric value or a debug expression.
<i>Size</i>	Number of bytes to display. This can be a number or a debug expression. If unspecified, the number of bytes required to fill the window will be automatically calculated.
<i>Columns</i>	Number of columns to display. If unspecified, the number of columns required to fill the window will be automatically calculated.
	Select binary display.
	Select octal display.
	Select unsigned decimal display.
	Select signed decimal display.
	Select hexadecimal display (<i>default</i>).
	Select byte display (<i>default</i>).
	Select 2-byte display.
	Select 4-byte display.
	Display both data and text (<i>default</i>).
	Display data only.
	Display text only.
	Display an incrementing address range that starts from the selected address (<i>default</i>).
	Display a decrementing address range that starts from the selected address.

	Display an incrementing address range that ends at the selected address.
	Display a decrementing address range that ends at the selected address.
	Evaluate the address and size expressions, and update the Memory window.

Using the memory window

The memory window does not show the complete address space of the target, instead you must enter both the address and the number of bytes to display. You can specify the address and size using numeric values or **debug expressions** which enable you to position the memory display at the address of a variable or at the value of 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 single step and whenever you traverse the call stack. If any values that were previously displayed have changed, they are highlighted in red.

To activate the first Memory window:

Choose **Debug > Other Windows > Memory > Memory 1** or press **Ctrl+T, M, 1**.

Other register windows can be similarly activated.

Using the mouse

You can move the memory window's edit cursor by clicking on a data or text entry.

The vertical scroll bar can be used to modify the address being viewed by clicking the up and down buttons, the page up and down areas or using the vertical scroll wheel when the scroll bar is at it's furthest extent. Holding down the **Shift** key while scrolling will prevent the address being modified.

Using the keyboard

Keystroke	Description
Up	Move the cursor up one line, or if the cursor is on the first line, move the address up one line.
Down	Move the cursor down one line, or if the cursor is on the last line, move the address down line line.
Left	Move the cursor left one character.
Right	Move the cursor right one character.
Home	Move the cursor to the first entry.
End	Move the cursor to the last entry.

PageUp	Move the cursor up one page, or if the cursor is on first page, move the address up one page.
PageDown	Move the cursor down one page, or if the cursor is on the last page, move the address down one page.
Ctrl+E	Toggle the cursor between data and text editing.

Editing memory

To edit memory, simply move the cursor to the data or text entry you want to modify and start typing. The memory entry will be written and read back as you type.

Shortcut menu commands

The shortcut menu contains the following commands:

Action	Description
Access Memory By Display Width	Access memory in terms of the display width.
Address Order	Specify whether the address range shown uses Address as the start or end address and whether addresses should increment or decrement.
Auto Evaluate	Re-evaluate Address and Size each time the Memory window is updated.
Auto Refresh	Specify how frequently the memory window should automatically refresh.
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 .

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.

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.

Copying to clipboard

You can copy the contents of the memory window to the clipboard as text. If an address range is selected, the data or text of the selected range will be copied to the clipboard depending on whether the selection has been made in the data or text view. If no address range is selected, the current memory window view will be copied to the clipboard.

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
	Create a new breakpoint using the New Breakpoint dialog.
	Toggle the selected breakpoint between enabled and disabled states.
	Remove the selected breakpoint.
	Move the insertion point to the statement where the selected breakpoint is set.
	Delete all breakpoints.
	Disable all breakpoints.
	Enable all breakpoints.
	Create a new breakpoint group and makes it active.

The main part of the **Breakpoints** window shows what breakpoints are 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 press **Ctrl+B, N**.

or

On the **Breakpoints** window toolbar, click the **Enable All Breakpoints** button.

To disable all breakpoints:

Choose **Breakpoints > Disable All Breakpoints** or press **Ctrl+B, X**.







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
	Move the insertion point to where the call was made to the selected frame.
	Set the debugger context to the selected stack frame.
	Move the debugger context down one stack to the called function.
	Move the debugger context up one stack to the calling function.
	Select the fields to display for each entry in the call stack.
	Set the debugger context to the most recent stack frame and move the insertion point 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 insertion point 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 insertion point 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 insertion point 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 **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**, **Threads.setSortByNumber** and **Threads.setColor** can be called from the **function init()**.

```
function init()
{
    Threads.setColumns("Name", "Priority", "State", "Time");
    Threads.setSortByNumber("Time");
    Threads.setColor("State", "Ready", "Executing", "Waiting");
}
```

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. The states **Ready**, **Executing** and **Waiting** will have yellow, green and red colored pixmaps respectively.

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,
    { 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 },
    { 0,1,2,3 }
};

struct task task1 =
{
    "Task1",
    0,
    "Executing",
    1000,
    &task2,
    { 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 },
};
```

```
{ 0,1,2,3 }
};
```

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("(" + x + ")>registers");
}
```

If you use thread local storage, implementing the **gettls(x)** function enables you to return the base address of the thread local storage, for example:

```
function gettls(x)
{
    return Debug.evaluate("((struct task*)"+x+"->thread_local_storage");
}
```

The **gettls(x)** function can also be called with null as a parameter. In this case you will have to evaluate an expression that returns the current thread local storage, for example:

```
function gettls(x)
{
    if (x==null)
        x = Debug.evaluate("&currentTask");
    return Debug.evaluate("((struct task*)"+x+"->thread_local_storage");
}
```

The debugger may require the name of a thread which you can provide by implementing the **getname(x)** function, for example:

```
function getname(x)
{
    return Debug.evaluate("((struct task*)"+x+"->name");
}
```

Adding extra queues to the threads window

You can add extra information to the threads window to display other RTOS queues. In the **function init()** you can use **Threads.setColumns2** to create an additional display in the threads window, for example:

```
function init()
{
    ...
    Threads.setColumns2("Timers", "Id(Timers)", "Name", "Hook", "Timeout", "Period", "Active");
}
```

The first argument is identifier of the queue which is also supplied to **Threads.add2** in the **function update()** as follows

```
function update()
{
    ...
    Threads.add2("Timers", "0x1FF0A30", "MyTimer", "0x46C8 (Timer50)", "50(550)", "50", "1");
}
```

You can avoid updating queues that aren't displayed using the **Threads.shown** function as follows

```
function update()
{
    ...
    if (Threads.shown("Timers"))
        Threads.add2("Timers", "0x1FF0A30", "MyTimer", "0x46C8
(Timer50)", "50(550)", "50", "1");
}
```

Execution Profile window

The **Execution Profile** 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 Profile window:

Choose **Debug > Other Windows > Execution Profile** or press **Ctrl+T, P**.

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.

Execution 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** or press **Ctrl+T, T**.

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.

Debug Terminal window

The **Debug Terminal** window displays debug output from the target application and can also be used to provide debug input to the target application.

To activate the Debug Terminal window:

Choose **Debug > Debug Terminal** or press **Ctrl+Alt+D**.

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 Environment Options dialog:

Choose **Tools > Options** or press **Alt+,**.

To activate the Debug Immediate window:

Choose **Debug > Other Windows > Debug Immediate**.

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 break 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 0x10000x10FF is accessed.

You can specify an inverse memory range using the `!` operator. For example `! (char[256]) (0x1000)` will breakpoint when memory outside the range 0x10000x10FF 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 `offsetof (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.

Arrays can be sliced using `[a:b]` where `a` is the first element and `b` is the last element to display.

Operators have the precedence and associativity one expects of a C-like programming language.

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 keyvalue 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
	Connect the target interface selected in the Targets window.
	Disconnect the connected target interface.
	Reconnect the connected target interface.
	Reset the connected target interface.
	Display 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

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 is sent to the communications port and any data received from the communications port is 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 for use in other applications.

Supported control codes

The terminal supports a limited set of control codes:

Control code	Description
<BS>	Backspace
<CR>	Carriage return
<LF>	Linefeed
<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.

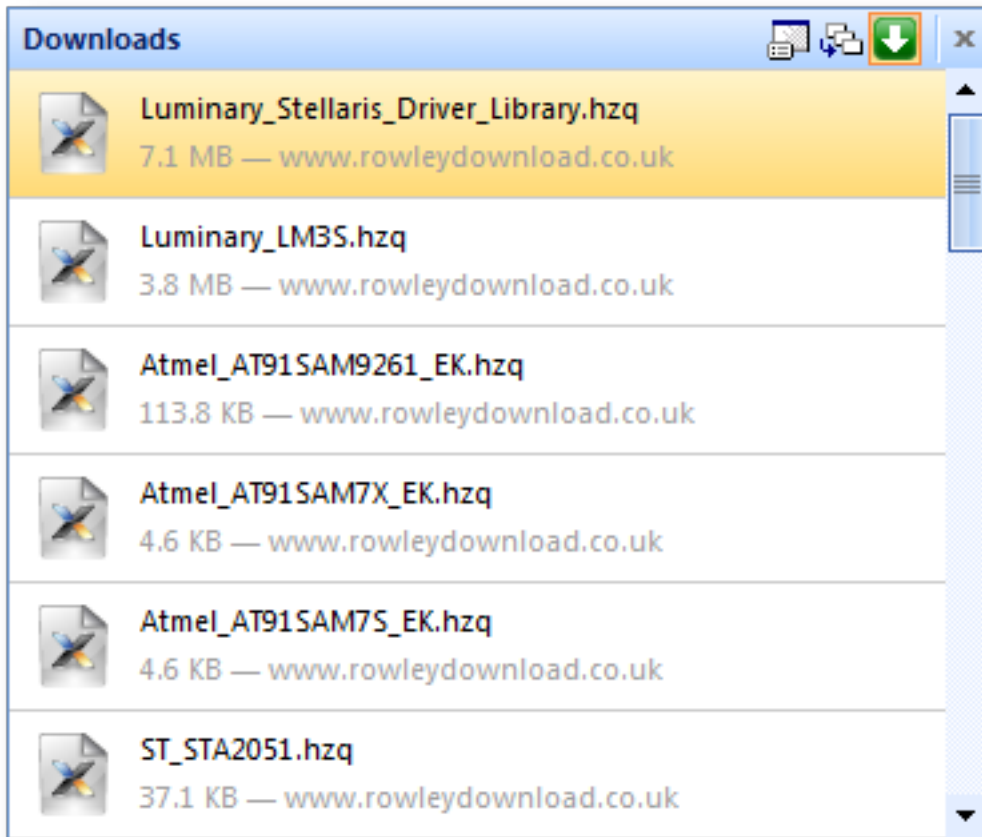
The JavaScript method **load**(*filepath*) loads and executes the JavaScript contained in *filepath* returns a Boolean indicating success.

To activate the Script Console window:

Choose **View > Script Console** or press **Ctrl+Alt+J**.

Downloads window

The **Downloads Window** displays a historical list of files downloaded over the Internet by CrossStudio.

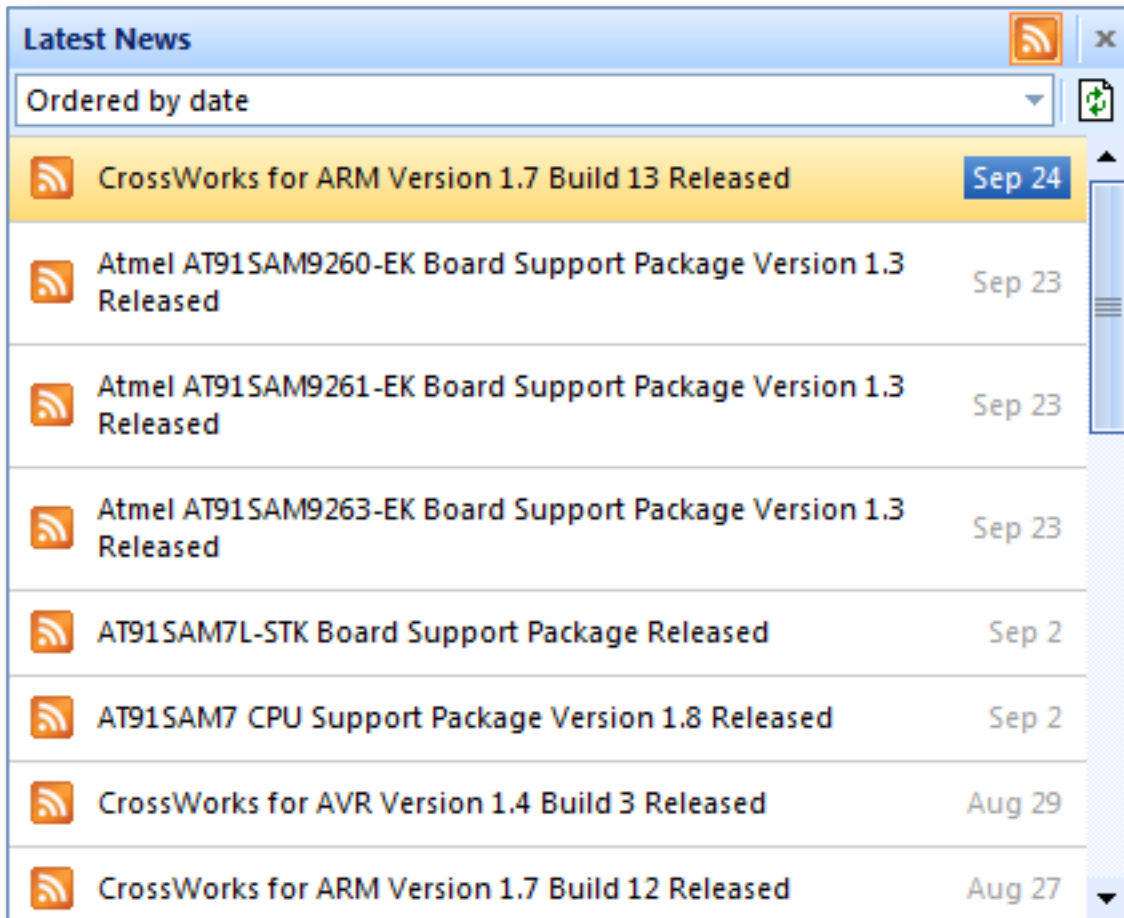


To activate the Downloads window:

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**.



Command-line options

This section describes the command-line options accepted by CrossStudio.

Usage

crossstudio [*options*] [*files*]

-D (Define macro)

Syntax

-D *macro=value*

Description

Define a CrossWorks macro value.

-noclang (Disable Clang support)

Syntax

-noclang

Description

Disable Clang support.

-noload (Disable loading of last project)

Syntax

-noload

Description

Disable loading of last project on startup.

-packagesdir (Specify packages directory)

Syntax

-packagesdir *dir*

Description

Override the default value of the **\$(PackagesDir)** macro.

-permit-multiple-studio-instances (Permit multiple studio instances)

Syntax

-permit-multiple-studio-instances

Description

Allow multiple instances of CrossStudio to run at the same time. This behaviour can also be enabled using the **Environment > Startup Options > Allow Multiple CrossStudios** environment option.

-rootuserdir (Set the root user data directory)

Syntax

-rootuserdir *dir*

Description

Set the CrossWorks root user data directory.

-save-settings-off (Disable saving of environment settings)

Syntax

-save-settings-off

Description

Disable the saving of modified environment settings.

-set-setting (Set environment setting)

Syntax

-set-setting *environment_setting=value*

Description

Sets an environment setting to a specified value. For example:

```
-set-setting "Environment/Build/Show Command Lines=Yes"
```

-templatesfile (Set project templates path)

Syntax

-templatesfile *path*

Description

Sets the search path for finding project template files.



Uninstalling CrossWorks for ARM

This section describes how to completely uninstall CrossWorks for ARM for each supported operating system:

[Uninstalling CrossWorks for ARM from Windows](#)

[Uninstalling CrossWorks for ARM from macOS](#)

[Uninstalling CrossWorks for ARM from Linux](#)

Uninstalling CrossWorks for ARM from Windows

Removing user data and settings

The uninstaller does not remove any user data such as settings or installed packages. To completely remove the user data you will need to carry out the following operations for each user that has used CrossWorks for ARM on your system.

To remove user data using CrossStudio:

1. Start CrossStudio.
2. Click **Tools > Admin > Remove All User Data...**

Alternatively, if CrossWorks for ARM has already been uninstalled you can manually remove the user data as follows:

1. Click the Windows Start button.

2. Type `%LOCALAPPDATA%` in the search field and press enter to open the local application data folder.
3. Open the *Rowley Associates Limited* folder.
4. Open the *CrossWorks for ARM* folder.
5. Delete the *v5* folder.
6. If you want to delete user data for all versions of the software, delete the *CrossWorks for ARM* folder as well.

Uninstalling CrossWorks for ARM

To uninstall CrossWorks for ARM:

1. If CrossStudio is running, click **File > Exit** to shut it down.
2. Click the Start Menu and select Control Panel. The Control Panel window will open.
3. In the Control Panel window, click the **Uninstall a program** link under the Programs section.
4. From the list of currently installed programs, select **CrossWorks for ARM 5.0**.
5. To begin the uninstall, click the **Uninstall** button at the top of the list.

Uninstalling CrossWorks for ARM from macOS

Removing user data and settings

Uninstalling does not remove any user data such as settings or installed packages. To completely remove the user data you will need to carry out the following operations for each user that has used CrossWorks for ARM on your system.

To remove user data using CrossStudio:

1. Start CrossStudio.
2. Click **Tools > Admin > Remove All User Data...**

Alternatively, if CrossWorks for ARM has already been uninstalled you can manually remove the user data as follows:

1. Open Finder.
2. Go to the `$HOME/Library/Rowley Associates Limited/CrossWorks for ARM` directory.
3. Drag the *v5* folder to the Trash.
4. If you want to delete user data for all versions of the software, drag the *CrossWorks for ARM* folder to the Trash as well.

Uninstalling CrossWorks for ARM

To uninstall CrossWorks for ARM:

1. If CrossStudio is running, shut it down.
2. Open the */Applications* folder in Finder.
3. Drag the *CrossWorks for ARM 5.0* folder to the Trash.

Uninstalling CrossWorks for ARM from Linux

Removing user data and settings

The uninstaller does not remove any user data such as settings or installed packages. To completely remove the user data you will need to carry out the following operations for each user that has used CrossWorks for ARM on your system.

To remove user data using CrossStudio:

1. Start CrossStudio.
2. Click **Tools > Admin > Remove All User Data...**

Alternatively, if CrossWorks for ARM has already been uninstalled you can manually remove the user data as follows:

1. Open a terminal window or file browser.
2. Go to the `$HOME/.rowley_associates_limited/CrossWorks for ARM` directory.
3. Delete the `v5` directory.
4. If you want to delete user data for all versions of the software, delete the *CrossWorks for ARM* directory as well.

Uninstalling CrossWorks for ARM

To uninstall CrossWorks for ARM:

1. If CrossStudio is running, click **File > Exit** to shut it down.
2. Open a terminal window.
3. Go to the CrossWorks for ARM bin directory (this is `/usr/share/crossworks_for_arm_5.0/bin` by default).
4. Run `sudo ./uninstall` to start the uninstaller.



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/thumb_crt0.s The CrossWorks standard C runtime. See [Startup code](#).

Target_MemoryMap.xml The target-specific memory map file for the board. See [Section Placement](#). 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 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 and Cortex-A/Cortex-R 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 can be placed at a specific address which is usually 0x00000000 or the start of Flash memory. The vector table contains jump instructions to the particular exception handlers. It is recommended that absolute jump instructions are used `ldr pc, =handler_address` 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 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 the reset handler with an endless loop which will stop the device running potentially dangerous in-development code directly out of reset. In development the debugger will start the device from the specified debug entry point.

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 and Cortex-A/Cortex-R 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 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 can be placed at a specific address which is usually 0x00000000 or the start of Flash memory.

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 `thumb_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. In development the debugger will start the device from the specified debug entry point.

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-M/Thumb equivalent **thumb_crt0.s**).

When you create a new project to produce an executable file using a target-specific project template, the **crt0.s**/**thumb_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.

<code>.fast</code>	Code to copy from flash to RAM for fast execution.
<code>.data</code>	The initialized static data.
<code>.bss</code>	The zeroed static data.
<code>.rodata</code>	The read-only constants and literals of the program.
<code>.ARM.exidx</code>	The C++ exception table.
<code>.tbss</code>	Thread local storage zero'd data followed by
<code>.tdata</code>	Thread local storage initialised data.

Stacks

ARM and Cortex-A/Cortex-R devices have 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
<code>.stack</code>	<code>__STACKSIZE__</code>	System and User mode stack.
<code>.stack_svc</code>	<code>__STACKSIZE_SVC__</code>	Supervisor mode stack
<code>.stack_irq</code>	<code>__STACKSIZE_IRQ__</code>	IRQ mode stack
<code>.stack_fiq</code>	<code>__STACKSIZE_FIQ__</code>	FIQ mode stack
<code>.stack_abt</code>	<code>__STACKSIZE_ABT__</code>	Abort mode stack
<code>.stack_und</code>	<code>__STACKSIZE_UND__</code>	Undefined mode stack

Cortex-M devices have the following stacks and linker symbol stack sizes are defined:

Section name	Linker size symbol	Description
<code>.stack</code>	<code>__STACKSIZE__</code>	Main stack.
<code>.stack_process</code>	<code>__STACKSIZE_PROCESS__</code>	Process stack.

The `crt0.s/thumb 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/thumb 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/thumb 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/thumb_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/thumb_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/thumb_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/thumb_crt0.s` file.

Section Placement

Section placement files map program sections used in your program into the memory spaces defined in the memory map or in the **Memory Segments** project property. 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.

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.

ARM section placement

The following placement files are supplied for ARM targets:

File	Description
<code>flash_placement.xml</code>	Single FLASH segment with internal RAM segment and optional external RAM segment.
<code>flash_run_text_from_ram_placement.xml</code>	Single FLASH segment with internal RAM segment and optional external RAM segments. Text section is copied from FLASH to RAM.
<code>internal_sram_placement.xml</code>	Single internal RAM segment.
<code>multi_flash_placement.xml</code>	Two FLASH segments with internal RAM segment and optional external RAM segment.
<code>sram_placement.xml</code>	Internal RAM segment and optional external RAM segment.
<code>tcm_placement.xml</code>	Data and Instruction tightly coupled memory segments.

Cortex-M section placement

The following placement files are supplied for Cortex-M targets:

File	Description
<code>flash_placement.xml</code>	Two FLASH segments and two RAM segments.
<code>flash_placement_tcm.xml</code>	One FLASH segments, two RAM segments, Data and Instruction tightly coupled memory segments.

<code>flash_placement2.xml</code>	One FLASH segment and two RAM segments.
<code>flash_to_ram_placement.xml</code>	One FLASH segment and one RAM segment. Text section is copied from FLASH to RAM.
<code>flash_to_ram_placement_tcm.xml</code>	One FLASH segment, two RAM segments, Data and Instruction tightly coupled memory segments. Text section is copied from FLASH to RAM.
<code>flash_to_ram_placement2.xml</code>	One FLASH segment and two RAM segments. Text section is copied from FLASH to RAM.
<code>flash_to_tcm_placement.xml</code>	Two FLASH segments, two RAM segments, Data and Instruction tightly coupled memory segments.
<code>ram_placement.xml</code>	Two RAM segments.
<code>tcm_placement.xml</code>	Data and Instruction tightly coupled memory segments.

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 project will have 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, 0x07FFFA0); // 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 programthe *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.

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 debug input/output will be enabled. This enables you to debug an application that copies 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** and **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.

Semihosting

The debugger supports the ARM semihosting interface. The operations `SYS_READC` and `SYS_READ` from standard input will return immediately i.e. they do not block.

Trace Capabilities

The following tracing capabilities are supported in CrossStudio

- Instruction tracing using the simulator target interface.
- Instruction and data tracing using ETMv1 on ARM7/ARM9 to ETB or external trace port.
- Instruction tracing using ETMv3 on Cortex-M to ETB or external trace port.
- Instruction tracing using MTB on Cortex-M0.
- Instruction and data tracing using ETMv3 on Cortex-A to ETB.
- Instrumentation, data tracing, exception tracing and program counter sampling using ITM/DWT on Cortex-M to ETB, external trace port or single wire output.
- Program counter sampling using the debug port on Cortex-M.

Tracing is controlled by the CrossStudio debugger i.e. tracing starts when a program runs or restarts from a breakpoint and stops when the program stops on a breakpoint. With ETM tracing it is also possible to start/stop tracing and to include/exclude functions using trace breakpoints.

Trace output from the last run is displayed in the [Execution Trace window](#) and instruction counts are accumulated in the [Execution Profile window](#) for each run of a debug session.

Simulator Tracing

The simulator maintains a list of the last *N* instructions that were executed or not executed if the condition failed. The size of the list is specified using the simulator project property **Num Trace Entries**.

ETM Tracing

The target trace project property **ETM TraceID** should be non-zero to enable the ETM when the target interface is connected.

For ARM7/ARM9 the ETB is assumed to follow the debug TAP on the JTAG scan chain. For Cortex-M/Cortex-A the ETB will be identified by the CoreSight ROM table. ETB tracing is selected by setting the target trace project property **Trace Interface Type** to be **ETB** when the target interface is connected.

The external trace port is assumed to be a four-bit half-rate clocked port and is selected by setting the target trace project property **Trace Interface Type** to be *TracePort* when the target interface is connected.

You can start and stop tracing with breakpoints by setting hardware breakpoints and specifying the breakpoint action to be **Trace Start** and **Trace Stop**.

You can choose to include/exclude functions by setting hardware breakpoints on the functions and specifying the breakpoint action to be **Trace Include** or **Trace Exclude**. Note that you cannot mix include and exclude ranges.

ITM/DWT Tracing

The target trace project property **ITM TraceID** should be non-zero to enable the ITM when the target interface is connected.

The target trace project properties **ITM Stimulus Ports Enable** and **ITM Stimulus Ports Privilege** are used to specify which ITM channels can be accessed. The library `<itm.h>` can be used to write to the ITM channels. The following ITM channels are treated specially by CrossStudio:

Channel 0: printable characters written to this channel will be buffered to implement **printf**-style output.

Channel 28: words written to this channel will be considered to be program counter values.

Channel 29 and 30: words written to these channels will be considered to be the start addresses of a function. Channel 30 indicates function entry and 29 indicates function exit. This functionality is used to implement the **Instrument Functions** compilation project property.

Channel 31: words written to this channel are considered to be thread scheduling information and as such are interpreted by the threads script.

You can enable local and/or global timestamping on the ITM packets using the **ITM Timestamping** and **ITM Global Timestamping Frequency** target trace project properties.

You can specify DWT program counter sampling and exception tracing using the **DWT PC Sampling** and **DWT Trace Exceptions** target trace project properties.

Like ETM tracing the ITM/DWT tracing can be directed to an *ETB* or a *TracePort* but it can also be directed to a single wire output (*SWO*) pin using the **Trace Interface Type** target trace project property. When the *SWO* pin is used the **Trace Clock Speed** target trace project property should be set to speed of the *TRACECLKIN* signal which is typically the processor clock speed.

Data Tracing

You can trace specific data items by setting a data breakpoint and specifying the action to be **Trace Data**.

Configuring Hardware for Tracing

The script contained in the target trace project property **Trace Initialize Script** will be executed when debug start or debug attach are selected. This script has the macro `$(TraceInterfaceType)` expanded with the value of the **Trace Interface Type** target trace project property. This script, for example, can be used to set up the pins for the external trace port. The Board/CPU support package should provide an implementation of this in the target script.

Supported Trace Capture Devices

The Segger J-Trace ARM and J-Trace Cortex-M supports trace capture from 4-bit half-rate clocked external Trace Ports.

The Segger J-Link - JTAG/SWD supports SWO trace capture.

The STLink/V2 supports SWO trace capture.

Some FTDI-2232 based devices have the second UART channel connected to the SWO. Since this is a target interface independent capability CrossStudio supports this for all target interfaces.



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	Yes	No	Yes	Yes	Yes
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	Yes

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
PandE 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
CMSIS-DAP	Yes	Yes	Yes	Yes	Yes	Yes	Yes

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 to 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, ARM V7-M, ARM V7-EM, ARM V7A and ARMV7R instruction sets, as defined in the 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 ISS supports a limited subset of VFP instructions (CP10 and CP11) that enables C programs that use the VFP to execute. NEON instructions are not simulated.

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 ISS supports MCR and MRC access to the Debug Communication Channel (CP14), as defined in the ARM7TDMI Technical Reference Manual.

The instruction set simulator (ISS) simulates the PPB, bit banding and systick capabilities of the ARM V6-M, ARM V7-M and ARM V7-EM architectures.

The memory system simulated by the ISS is implemented by the dynamic link library specified by the Memory Simulation Filename and Memory Simulation Parameter defined in the project's simulator properties. Any access to memory not defined by the memory system is reported as an error.

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.

Amontec JTAGkey Target Interface

Interface

Property	Description
Serial Number <code>connectedSerialNumberString</code>	The serial number of the currently connected FT2232.
Use Serial Number <code>connectToSerialNumberString</code>	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 <code>interfaceVersionString</code>	The target interface version number.

JTAG

Property	Description
Adaptive Clocking <code>adaptiveClockingEnumeration</code>	Specifies whether JTAG adaptive clocking should be used.
nSRST Open Drain <code>srstOpenDrainBoolean</code>	Specifies whether the nSRST signal is open-drain or push-pull.
nTRST Open Drain <code>trstOpenDrainBoolean</code>	Specifies whether the nTRST signal is open-drain or push-pull.

JTAG/SWD

Property	Description
Speed <code>speedIntegerRange</code>	The maximum JTAG/SWD clock frequency in Hz (0 for best possible).

Target

Property	Description
Device Type <code>String</code>	The detected type of the currently connected target device.

Fast Memory Accesses <code>fastMemoryAccessesEnabled</code> Boolean	<p>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.</p>
Memory Access Timeout <code>memoryAccessTimeout</code> IntegerRange	<p>The timeout period for memory accesses in milliseconds.</p>

Trace

Property	Description
UART-SWO COM Port <code>UARTSWOPortCOMPort</code>	<p>Name of COM port that SWO is connected to.</p>

CMSIS-DAP Target Interface

Interface

Property	Description
CMSIS-DAP Capabilities cmsisDapCapabilitiesString	The capabilities of the currently connected CMSIS-DAP interface.
CMSIS-DAP Protocol Version cmsisDapProtocolVersionString	The CMSIS-DAP Protocol version of the currently connected CMSIS-DAP interface.
Serial Number connectedSerialNumberString	The serial number of the currently connected CMSIS-DAP.
Use Serial Number connectToSerialNumberString	The serial number of the CMSIS-DAP device you want to connect to. If multiple CMSIS-DAP devices are connected to your system, this property allows you to specify which one to use. If no serial number is specified, the first matching available CMSIS-DAP device will be used.

JTAG/SWD

Property	Description
Speed speedIntegerRange	The maximum JTAG/SWD clock frequency in Hz (0 for best possible).

Target

Property	Description
Device Type String	The detected type of the currently connected target device.

USB

Property	Description
Connected Interface Mode connectedUsbInterfaceModeString	The CMSIS-DAP USB interface mode currently being used.
HID Report Length hidReportLengthIntegerRange	Specifies the HID report length in bytes (0 for the interface's default value).
Interface Mode usbInterfaceModeEnumeration	The CMSIS-DAP USB interface mode to use.

Maximum Packet Count usbPacketCountIntegerRange	The maximum number of USB packets that can be buffered for a single operation (0 for the interface's default value).
PID usbPidString	Specifies the USB product ID of the CMSIS-DAP device. If USB vendor and product IDs are both unspecified, the first matching available CMSIS-DAP device will be used.
VID usbVidString	Specifies the USB vendor ID of the CMSIS-DAP device. If USB vendor and product IDs are both unspecified, the first matching available CMSIS-DAP device will be used.

CrossConnect Target Interface

Interface

Property	Description
Information <code>interfaceInformationString</code>	Interface connection information.
Model <code>modelInformationString</code>	CrossConnect Model.
Serial Number <code>connectedSerialNumberString</code>	The serial number of the currently connected CrossConnect.
Target Voltage <code>target_voltageString</code>	The target's JTAG reference voltage.
Version <code>interfaceVersionString</code>	The target interface version number.

JTAG

Property	Description
Adaptive Clocking <code>adaptiveClockingEnumeration</code>	Specifies whether JTAG adaptive clocking should be used.

JTAG/SWD

Property	Description
Speed <code>speedIntegerRange</code>	The maximum JTAG/SWD clock frequency in Hz (0 for best possible).

Target

Property	Description
Device Type <code>String</code>	The detected type of the currently connected target device.

Fast Memory Accesses <code>fastMemoryAccessesEnabledBoolean</code>	<p>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.</p>
Host Connection <code>ConnectionEnumeration</code>	<p>The USB serial number of the CrossConnect to use.</p>
Memory Access Timeout <code>memoryAccessTimeoutIntegerRange</code>	<p>The timeout period for memory accesses in milliseconds.</p>

Trace

Property	Description
Current SWO Speed <code>currentSwoSpeedIntegerRange</code>	<p>The current SWO speed.</p>
Current Trace Buffer Size <code>currentTraceBufferSizeIntegerRange</code>	<p>The current size of the trace buffer.</p>
SWO Speed <code>swoSpeedIntegerRange</code>	<p>The required SWO speed (0 for maximum supported).</p>
Trace Buffer Size <code>traceBufferSizeIntegerRange</code>	<p>The size of the trace buffer.</p>

Generic FT2232 Target Interface

FT2232 Pin Configuration

Property	Description
Connected LED Inversion Mask connectedLedXORMaskIntegerHex	Specifies the FT2232 output pin(s) to invert when setting 'connected' LED.
Connected LED Mask connectedLedMaskIntegerHex	Specifies the FT2232 output pin(s) to use for the 'connected' LED.
Disconnected Output Pins disconnectedOutputDirectionIntegerHex	Specifies the FT2232 pins that are to be configured for output when disconnected.
Disconnected Output Value disconnectedOutputValueIntegerHex	Specifies the value of the FT2232 output pins when disconnected.
Output Pins outputDirectionIntegerHex	Specifies the FT2232 pins that are to be configured for output.
Output Value outputValueIntegerHex	Specifies the initial value of the FT2232 output pins on connection.
Output Value 2 outputValue2IntegerHex	If non-zero the 2nd initial value of the FT2232 output pins on connection.
Running LED Inversion Mask runningLedXORMaskIntegerHex	Specifies the FT2232 output pin(s) to invert when setting the 'running' LED.
Running LED Mask runningLedMaskIntegerHex	Specifies the FT2232 output pin(s) to use for the 'running' LED
SWD Direction Inversion Mask swdDirectionXORMaskIntegerHex	Specifies the FT2232 output pin(s) to invert to set serial wire debug to output.
SWD Direction Mask swdDirectionMaskIntegerHex	Specifies the FT2232 output pin(s) to use to set serial wire debug to output.
SWD Enable Inversion Mask swdEnableXORMaskIntegerHex	Specifies the FT2232 output pin(s) to invert when enabling serial wire .
SWD Enable Mask swdEnableMaskIntegerHex	Specifies the FT2232 output pin(s) to use when enabling serial wire debug.
nSRST Inversion Mask srstXORMaskIntegerHex	Specifies the FT2232 output pin(s) to invert when setting the nSRST signal.
nSRST Mask srstMaskIntegerHex	Specifies the FT2232 output pin(s) to use for the nSRST signal.
nTRST Inversion Mask trstXORMaskIntegerHex	Specifies the FT2232 output pin(s) to invert when setting the nTRST signal.
nTRST Mask trstMaskIntegerHex	Specifies the FT2232 output pin(s) to use for the nTRST signal.

FT2232 USB

Property	Description
Channel channelEnumeration	Specifies the FT2232 channel to use
PID usbPidStringList	Specifies the USB product ID of the FT2232 device.
VID usbVidString	Specifies the USB vendor ID of the FT2232 device.

Interface

Property	Description
Serial Number connectedSerialNumberString	The serial number of the currently connected FT2232.
Use Serial Number connectToSerialNumberString	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 interfaceVersionString	The target interface version number.

JTAG

Property	Description
Adaptive Clocking adaptiveClockingEnumeration	Specifies whether JTAG adaptive clocking should be used.

JTAG/SWD

Property	Description
Speed speedIntegerRange	The maximum JTAG/SWD clock frequency in Hz (0 for best possible).

Target

Property	Description
----------	-------------

Device Type String	The detected type of the currently connected target device.
Fast Memory Accesses fastMemoryAccessesEnabledBoolean	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 memoryAccessTimeoutIntegerRange	The timeout period for memory accesses in milliseconds.

Trace

Property	Description
UART-SWO COM Port UARTSWOPortCOMPort	Name of COM port that SWO is connected to.

Generic Target Interface

Generic

Property	Description
Applicable Host OS hostStringList	The names of host OS that are supported.
Generic DLL File DLLFileNameFileName	The file path of the .dll to use.

Olimex ARM-USB-OCD Target Interface

Interface

Property	Description
Serial Number <code>connectedSerialNumberString</code>	The serial number of the currently connected FT2232.
Use Serial Number <code>connectToSerialNumberString</code>	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 <code>interfaceVersionString</code>	The target interface version number.

JTAG

Property	Description
Adaptive Clocking <code>adaptiveClockingEnumeration</code>	Specifies whether JTAG adaptive clocking should be used.
nTRST Open Drain <code>trstOpenDrainBoolean</code>	Specifies whether the nTRST signal is open-drain or push-pull.

JTAG/SWD

Property	Description
Speed <code>speedIntegerRange</code>	The maximum JTAG/SWD clock frequency in Hz (0 for best possible).

Target

Property	Description
Device Type <code>String</code>	The detected type of the currently connected target device.

Fast Memory Accesses <code>fastMemoryAccessesEnabled</code> Boolean	<p>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.</p>
Memory Access Timeout <code>memoryAccessTimeout</code> IntegerRange	<p>The timeout period for memory accesses in milliseconds.</p>

Trace

Property	Description
UART-SWO COM Port <code>UARTSWOPortCOMPort</code>	<p>Name of COM port that SWO is connected to.</p>

Kinetis OSJTAG Target Interface

Kinetis OSJTAG

Property	Description
Firmware Version String	The Firmware version of the Kinetis OSJTAG.

Target

Property	Description
Device Type String	The detected type of the currently connected target device.

P&E UNIT Interface DLL Target Interface

Generic

Property	Description
Applicable Host OS hostStringList	The names of host OS that are supported.
Generic DLL File DLLFileNameFileName	The file path of the .dll to use.

Segger J-Link Target Interface

J-Link

Property	Description
Additional J-Link Options JLinkExecuteCommandStringList	Specify additional J-Link options to allow enabling or disabling advanced features and fine tuning. For more information see J-Link Command Strings
Current Speed IntegerRange	The JTAG/SWD clock frequency the J-Link is currently using.
DLL Version String	The J-Link DLL version.
Enable Adaptive Clocking adaptiveEnumeration	Adaptive clocking is enabled.
Exclude Flash Cache Range JLinkExcludeFlashCacheRangeString	Define a memory range that should not be cached by J-Link. Per default, all areas that J-Link knows to be Flash memory, are cached. This means that it is assumed that the contents of this areas do not change during program execution. If this assumption does not hold true, typically because the target program modifies the flash content for data storage, then the affected area should be excluded from the cache. This may slightly reduce the debugging speed. Syntax: either 'start_address-end_address' or 'address,size'. For example: 0x08000000,0x1000.
Firmware Version String	The J-Link firmware version.
Hardware Version String	The J-Link hardware version.
J-Link DLL File JLinkARMDLLFileNameFileName	The file path of the libjlinkarm.so to use.
Log File JLinkLogFileNameFileName	The file to output the J-Link log to.
Max SWO Speed IntegerRange	The maximum supported SWO speed.
Reset Type resetTypeIntegerRange	The reset strategy to use.
Script File JLinkScriptFileNameFileName	The file path of the optional J-Link script file to use.

Serial Number String	The serial number of the connected J-Link
Settings File JLinkProjectFileNameFileName	The file path of the automatically generated J-Link settings file to use.
Show Log Messages In Output Window showLogEnumeration	Display the J-Link log messages to the output window.
Speed speedIntegerRange	The required JTAG/SWD clock frequency in kHz (0 to auto-detect best possible).
Supply Power supplyPowerEnumeration	The J-Link supplies power to the target.
Target Voltage String	The target reference voltage.
Trace Buffer Size traceBufferSizeIntegerRange	The size of the trace buffer
Use Built-in Flash Loader JLinkUseFlashLoaderEnumeration	The built-in debug component identify, flash loader and breakpoint support is used.
Use Built-in RTT support JLinkUseRTTEnumeration	The built-in RTT handling is used
Use Built-in TRACE support JLinkUseSTRACEEnumeration	The built-in trace handling is used
Verify Read Operations checkModeAfterReadEnumeration	The CPU mode is checked after each read operation.

Target

Property	Description
Device Type String	The detected type of the currently connected target device.

Stellaris ICDI Target Interface

Target

Property	Description
Device Type String	The detected type of the currently connected target device.

ST-LINK Target Interface

Generic

Property	Description
Applicable Host OS hostStringList	The names of host OS that are supported.
Generic DLL File DLLFileNameFileName	The file path of the .dll to use.

ST-LINK/V2 Target Interface

ST-LINK

Property	Description
Firmware Version String	The Main, JTAG and SWIM firmware versions.

Target

Property	Description
Device Type String	The detected type of the currently connected target device.
Host Connection <code>ConnectionEnumeration</code>	A number specifying the device to connect to.
Speed String	The target JTAG/SWD clock frequency in kHz.
Voltage String	The target reference voltage.

Macraigor Wiggler (20 and 14 pin) Target Interface

Connection

Property	Description
Parallel Port <code>portNameString</code>	The parallel port connection to use to connect to target.
Parallel Port Address <code>portAddressString</code>	The base address of the currently connected parallel port.
Parallel Port Sharing <code>portSharingBoolean</code>	Specifies whether sharing of the parallel port with other device drivers or programs is permitted.

Interface

Property	Description
Version <code>interfaceVersionString</code>	The target interface version number.

JTAG

Property	Description
Invert nSRST <code>invertNSRSTBoolean</code>	Specify whether the nSRST signal should be inverted.

JTAG/SWD

Property	Description
Speed <code>speedIntegerRange</code>	The maximum JTAG/SWD clock frequency in Hz (0 for best possible).

Target

Property	Description
Device Type <code>String</code>	The detected type of the currently connected target device.

Fast Memory Accesses <code>fastMemoryAccessesEnabled</code> 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 <code>memoryAccessTimeout</code> IntegerRange	The timeout period for memory accesses in milliseconds.



Using an external ARM GCC toolchain

You can use CrossStudio for ARM with a third party supplied ARM GCC toolchain. To do this you must set the project properties **Build > Use External GCC** to **Yes**, the **Build > GCC Prefix** to **arm-none-eabi-** and **Build > Tool Chain Directory** to the directory containing the gcc executable for example **C:/Program Files (x86)/GNU Tools ARM Embedded/4.7 2012q4/bin**.

To be able to use the code completion and source navigation features you can set the project property **Source Code > Additional Code Completion Compiler Options** to specify the directories that the ARM GCC toolchain will access for example **-isystemC:/Program Files (x86)/GNU Tools ARM Embedded/4.7 2012q4/arm-none-eabi/include**.



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.



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, __printf_t *ctx);
```

Sending all output to the CrossStudio virtual terminal

The default implementation of the **__putchar** function uses **debug_putchar** if the debugIO library is used in the project. You can remove usage of the debugIO library from your project by setting the **Library > Debug I/O Implementation** property to **None**.

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, __printf_t *ctx);  
int uart1_putc(int ch, __printf_t *ctx);
```

These functions return a positive value if there is no error outputting the character and EOF if there was an error. The second parameter, *ctx*, is the *context* that the high-level formatting routines use to implement the C standard library functions.

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, 0); // null context
    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 which captures the current state and parameters of the format conversion:

```
typedef struct __printf_tag
{
    size_t charcount;
    size_t maxchars;
    char *string;
    int (*output_fn)(int, struct __printf_tag *ctx);
} __printf_t;
```

This type is used by the library functions to direct what the formatting routines do with each character they need to output. If **string** is non-zero, the character is appended is appended to the string pointed to by **string**; if **output_fn** is non-zero, the character is output through the function **output_fn** with the context passed as the second parameter.

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.string = 0;
```

```

    iod.maxchars = INT_MAX;
    iod.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.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\n..", 0);
uart_printf(1, "..and this is uart %d\n", 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 = __vfprintf((__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 6.2 definition as a base. Hence, whenever you see the term Unicode in this document, it is equivalent to Unicode 6.2 and ISO/IEC 10646:2011.

The CrossWorks C library supports both 16-bit and 32-bit wide characters, depending upon the setting of wide character width in the project.

When compiling with 16-bit wide characters, all characters in the Basic Multilingual Plane are representable in a single `wchar_t` (values 0 through 0xFFFF). When compiling with 32-bit wide characters, all characters in the Basic Multilingual Plane and planes 1 through 16 are representable in a single `wchar_t` (values 0 through 0x10FFFF).

The wide character type will hold Unicode code points in a locale that is defined to use Unicode and character type functions such as `iswalpha` 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 in the `$(StudioDir)/source` folder 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.

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; the locale name is terminated either by a zero character or by a semicolon. The locale name, up to the semicolon or zero, 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 source directory as described in the previous section. Add this file to your project.

Although this adds the data needed for the locale, it does not make the locale 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 = {
    "hu_HU.utf8",
    &__RAL_hu_HU_locale,
    &__RAL_codeset_utf8
};

static const __RAL_locale_t hu_HU_iso_8859_2 = {
    "hu_HU.iso_8859_2",
    &__RAL_hu_HU_locale,
    &codeset_iso_8859_2
};

const __RAL_locale_t *
__user_find_locale(const char *locale)
{
    if (__RAL_compare_locale_name(locale, hu_HU_utf8.name) == 0)
        return &hu_HU_utf8;
    else if (__RAL_compare_locale_name(locale, hu_HU_iso_8859_2.name) == 0)
        return &hu_HU_iso_8859_2;
    else
        return 0;
}
```

The function `__RAL_compare_locale_name` matches locale names up to a terminating null character, or a semicolon (which is required by the implementation of `setlocale` in the C library when setting multiple locales using `LC_ALL`).

In addition to this, you must provide a buffer, `__user_locale_name_buffer`, for locale names encoded by `setlocale`. The buffer must be large enough to contain five locale names, one for each category. In the above example, the longest locale name is `hu_HU.iso_8859_2` which is 16 characters in length. Using this information, buffer must be at least $(16+1)5 = 85$ characters in size:

```
const char __user_locale_name_buffer[85];
```

Setting a locale directly

Although we support **setlocale** in its full generality, most likely you'll want to set a locale once and forget about it. You can do that by including the locale in your application and writing to the instance variables that hold the underlying locale data for the CrossWorks C library.

For instance, you might wish to use Czech locale with a UTF codeset:

```
static __RAL_locale_t cz_locale =
{
    "cz_CZ.utf8",
    &__RAL_cs_CZ_locale,
    &__RAL_codeset_utf8
};
```

You can install this directly into the locale without using **setlocale**:

```
__RAL_global_locale.__category[LC_COLLATE] = &cz_locale;
__RAL_global_locale.__category[LC_CTYPE]   = &cz_locale;
__RAL_global_locale.__category[LC_MONETARY] = &cz_locale;
__RAL_global_locale.__category[LC_NUMERIC] = &cz_locale;
__RAL_global_locale.__category[LC_TIME]    = &cz_locale;
```



Complete API reference

This section contains a complete reference to the CrossWorks C library API.

File	Description
<assert.h>	Describes the diagnostic facilities which you can build into your application.
<debugio.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.
<itm.h>	Describes ITM access library 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 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.

<complex.h>

API Summary

Trigonometric functions	
cacos	Compute inverse cosine of a complex float
cacosf	Compute inverse cosine of a complex float
casin	Compute inverse sine of a complex float
casinf	Compute inverse sine of a complex float
catan	Compute inverse tangent of a complex float
catanf	Compute inverse tangent of a complex float
ccos	Compute cosine of a complex float
ccosf	Compute cosine of a complex float
csin	Compute sine of a complex float
csinf	Compute sine of a complex float
ctan	Compute tangent of a complex float
ctanf	Compute tangent of a complex float
Hyperbolic trigonometric functions	
cacosh	Compute inverse hyperbolic cosine of a complex float
cacoshf	Compute inverse hyperbolic cosine of a complex float
casinh	Compute inverse hyperbolic sine of a complex float
casinhf	Compute inverse hyperbolic sine of a complex float
catanh	Compute inverse hyperbolic tangent of a complex float
catanhf	Compute inverse hyperbolic tangent of a complex float
ccosh	Compute hyperbolic cosine of a complex float
ccoshf	Compute hyperbolic cosine of a complex float
csinh	Compute hyperbolic sine of a complex float
csinhf	Compute hyperbolic sine of a complex float
ctanh	Compute hyperbolic tangent of a complex float
ctanhf	Compute hyperbolic tangent of a complex float
Exponential and logarithmic functions	
cexp	Computes the base-e exponential of a complex float
cexpf	Computes the base-e exponential of a complex float
clog	Computes the base-e logarithm of a complex float

clogf	Computes the base-e logarithm of a complex float
Power and absolute value functions	
cabs	Computes the absolute value of a complex float
cabsf	Computes the absolute value of a complex float
cpow	Compute a complex float raised to a power
cpowf	Compute a complex float raised to a power
csqrt	Compute square root of a complex float
csqrtf	Compute square root of a complex float
Manipulation functions	
carg	Compute argument of a complex float
cargf	Compute argument of a complex float
cimag	Compute imaginary part of a complex float
cimagf	Compute imaginary part of a complex float
conj	Compute conjugate of a complex float
conjf	Compute conjugate of a complex float
cproj	Compute projection on the Riemann sphere
cprojf	Compute projection on the Riemann sphere
creal	Compute real part of a complex float
crealf	Compute real part of a complex float

cabs

Synopsis

```
double cabs(double complex z);
```

Description

cabs returns the absolute value of **z**.

cabsf

Synopsis

```
float cabsf(float complex z);
```

Description

cabsf returns the absolute value of **z**.

cacos

Synopsis

```
double complex cacos(double complex z);
```

Description

cacos returns the principal value the inverse cosine of **z** with branch cuts outside the interval $[-1,+1]$ on the real axis. The principal value lies in the interval $[0, \pi]$ on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

cacosf

Synopsis

```
float complex cacosf(float complex z);
```

Description

cacosf returns the principal value the inverse cosine of **z** with branch cuts outside the interval $[-1,+1]$ on the real axis. The principal value lies in the interval $[0, \pi]$ on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

cacosh

Synopsis

```
double complex cacosh(double complex z);
```

Description

cacosh returns the principal value the inverse hyperbolic cosine of **z** with branch cuts of values less than 1 on the real axis. The principal value lies in the range of a half-strip of non-negative values on the real axis and in the interval $[-i, +i]$ on the imaginary axis.

cacoshf

Synopsis

```
float complex cacoshf(float complex _z);
```

Description

cacoshf returns the principal value the inverse hyperbolic cosine of **z** with branch cuts of values less than 1 on the real axis. The principal value lies in the range of a half-strip of non-negative values on the real axis and in the interval $[-i, +i]$ on the imaginary axis.

carg

Synopsis

```
double carg(double complex z);
```

Description

carg computes the argument of **z** with a branch cut along the negative real axis.

cargf

Synopsis

```
float cargf(float complex z);
```

Description

cargf computes the argument of **z** with a branch cut along the negative real axis.

casin

Synopsis

```
double complex casin(double complex z);
```

Description

casin returns the principal value the inverse sine of **z** with branch cuts outside the interval $[-1,+1]$ on the real axis. The principal value lies in the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$ on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

casinf

Synopsis

```
float complex casinf(float complex z);
```

Description

casinf returns the principal value the inverse sine of **z** with branch cuts outside the interval $[-1,+1]$ on the real axis. The principal value lies in the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$ on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

casinh

Synopsis

```
double complex casinh(double complex z);
```

Description

casinh returns the principal value the inverse hyperbolic sine of **z** with branch cuts outside the interval $[-i,+i]$ on the imaginary axis. The principal value lies in the range of a strip mathematically unbounded on the real axis and in the interval $[-i,+i]$ on the imaginary axis.

casinhf

Synopsis

```
float complex casinhf(float complex z);
```

Description

casinhf returns the principal value the inverse hyperbolic sine of **z** with branch cuts outside the interval $[-i, +i]$ on the imaginary axis. The principal value lies in the range of a strip mathematically unbounded on the real axis and in the interval $[-i, +i]$ on the imaginary axis.

catan

Synopsis

```
double complex catan(double complex z);
```

Description

catan returns the principal value the inverse sine of **z** with branch cuts outside the interval $[-1,+1]$ on the real axis. The principal value lies in the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$ on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

catanf

Synopsis

```
float complex catanf(float complex z);
```

Description

catanf returns the principal value the inverse sine of **z** with branch cuts outside the interval $[-1,+1]$ on the real axis. The principal value lies in the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$ on the real axis and in the range of a strip mathematically unbounded on the imaginary axis.

catanh

Synopsis

```
double complex catanh(double complex z);
```

Description

catanh returns the principal value the inverse hyperbolic sine of **z** with branch cuts outside the interval $[-1,+1]$ on the real axis. The principal value lies in the range of a strip mathematically unbounded on the real axis and in the interval $[-i,+i]$ on the imaginary axis.

catanhf

Synopsis

```
float complex catanhf(float complex z);
```

Description

catanhf returns the principal value the inverse hyperbolic sine of **z** with branch cuts outside the interval $[-1,+1]$ on the real axis. The principal value lies in the range of a strip mathematically unbounded on the real axis and in the interval $[-i,+i]$ on the imaginary axis.

CCOS

Synopsis

```
double complex ccos(double complex z);
```

Description

ccos returns the complex cosine of **z**.

ccosf

Synopsis

```
float complex ccosf(float complex z);
```

Description

ccosf returns the complex cosine of **z**.

ccosh

Synopsis

```
double complex ccosh(double complex z);
```

Description

ccosh returns the complex hyperbolic cosine of **z**.

ccoshf

Synopsis

```
float complex ccoshf(float complex z);
```

Description

ccoshf returns the complex hyperbolic cosine of **z**.

cexp

Synopsis

```
double complex cexp(double complex z);
```

Description

cexp returns the complex base-e exponential value of **z**.

cexpf

Synopsis

```
float complex cexpf(float complex z);
```

Description

cexpf returns the complex base-e exponential value of **z**.

cimag

Synopsis

```
double cimag(double complex);
```

Description

cimag computes the imaginary part of **z**.

cimagf

Synopsis

```
float cimagf(float complex);
```

Description

cimagf computes the imaginary part of **z**.

clog

Synopsis

```
double complex clog(double complex z);
```

Description

clog returns the complex base-e logarithm value of **z**.

clogf

Synopsis

```
float complex clogf(float complex z);
```

Description

clogf returns the complex base-e logarithm value of **z**.

conj

Synopsis

```
double complex conj(double complex);
```

Description

conj computes the conjugate of **z** by reversing the sign of the imaginary part.

conjf

Synopsis

```
float complex conjf(float complex);
```

Description

conjf computes the conjugate of **z** by reversing the sign of the imaginary part.

cpow

Synopsis

```
double complex cpow(double complex x,  
                    double complex y);
```

Description

cpow computes x raised to the power y with a branch cut for the x along the negative real axis.

cpowf

Synopsis

```
float complex cpowf(float complex x,  
                   float complex y);
```

Description

cpowf computes x raised to the power y with a branch cut for the x along the negative real axis.

cproj

Synopsis

```
double complex cproj(double complex);
```

Description

cproj computes the projection of **z** on the Riemann sphere.

cprojf

Synopsis

```
float complex cprojf(float complex);
```

Description

cprojf computes the projection of **z** on the Riemann sphere.

creal

Synopsis

```
double creal(double complex);
```

Description

creal computes the real part of **z**.

crealf

Synopsis

```
float crealf(float complex);
```

Description

crealf computes the real part of **z**.

csin

Synopsis

```
double complex csin(double complex z);
```

Description

csin returns the complex sine of **z**.

csinf

Synopsis

```
float complex csinf(float complex z);
```

Description

csinf returns the complex sine of **z**.

csinh

Synopsis

```
double complex csinh(double complex z);
```

Description

csinh returns the complex hyperbolic sine of **z**.

csinhf

Synopsis

```
float complex csinhf(float complex z);
```

Description

csinhf returns the complex hyperbolic sine of **z**.

csqrt

Synopsis

```
double complex csqrt(double complex z);
```

Description

csqrt computes the complex square root of **z** with a branch cut along the negative real axis.

csqrtf

Synopsis

```
float complex csqrtf(float complex z);
```

Description

csqrtf computes the complex square root of **z** with a branch cut along the negative real axis.

ctan

Synopsis

```
double complex ctan(double complex z);
```

Description

ctan returns the complex tangent of **z**.

ctanf

Synopsis

```
float complex ctanf(float complex z);
```

Description

ctanf returns the complex tangent of **z**.

ctanh

Synopsis

```
double complex ctanh(double complex z);
```

Description

ctanh returns the complex hyperbolic tangent of **z**.

ctanhf

Synopsis

```
float complex ctanhf(float complex z);
```

Description

ctanhf returns the complex hyperbolic tangent of **z**.

<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?
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
Classification functions (extended)	
isalnum_l	Is character alphanumeric?
isalpha_l	Is character alphabetic?
isblank_l	Is character a space or horizontal tab?
iscntrl_l	Is character a control character?
isdigit_l	Is character a decimal digit?
isgraph_l	Is character any printing character except space?
islower_l	Is character a lowercase letter?
isprint_l	Is character printable?
ispunct_l	Is character a punctuation mark?
isspace_l	Is character a whitespace character?
isupper_l	Is character an uppercase letter?
isxdigit_l	Is character a hexadecimal digit?
Conversion functions (extended)	
tolower_l	Convert uppercase character to lowercase

[toupper_l](#)

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.

isalnum_l

Synopsis

```
int isalnum_l(int c,  
             locale_t loc);
```

Description

isalnum_l returns nonzero (true) if and only if the value of the argument **c** is a alphabetic or numeric character in locale **loc**.

isalpha

Synopsis

```
int isalpha(int c);
```

Description

isalpha returns true if the character **c** is alphabetic. That is, any character for which **isupper** or **islower** returns true is considered alphabetic in addition to any of the locale-specific set of alphabetic characters for which none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true.

In the C locale, **isalpha** returns nonzero (true) if and only if **isupper** or **islower** return true for value of the argument **c**.

isalpha_l

Synopsis

```
int isalpha_l(int c,  
              locale_t loc);
```

Description

isalpha_l returns nonzero (true) if and only if **isupper** or **islower** return true for value of the argument **c** in locale **loc**.

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 ').

isblank_l

Synopsis

```
int isblank_l(int c,  
              locale_t loc);
```

Description

isblank_l returns nonzero (true) if and only if the value of the argument **c** is either a space character (' ') or the horizontal tab character ('\\t ') in locale **loc**.

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.

isctrl_l

Synopsis

```
int isctrl_l(int c,  
             locale_t loc);
```

Description

isctrl_l returns nonzero (true) if and only if the value of the argument **c** is a control character in locale **loc**.

isdigit

Synopsis

```
int isdigit(int c);
```

Description

isdigit returns nonzero (true) if and only if the value of the argument **c** is a digit.

isdigit_l

Synopsis

```
int isdigit_l(int c,  
             locale_t loc);
```

Description

isdigit_l returns nonzero (true) if and only if the value of the argument **c** is a decimal digit in locale **loc**.

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 (' ').

isgraph_l

Synopsis

```
int isgraph_l(int c,  
              locale_t loc);
```

Description

isgraph_l returns nonzero (true) if and only if the value of the argument **c** is any printing character except space (' ') in locale **loc**.

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.

islower_l

Synopsis

```
int islower_l(int c,  
              locale_t loc);
```

Description

islower_l returns nonzero (true) if and only if the value of the argument **c** is an lowercase letter in locale **loc**.

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 (' ').

isprint_l

Synopsis

```
int isprint_l(int c,  
             locale_t loc);
```

Description

isprint_l returns nonzero (true) if and only if the value of the argument **c** is any printing character including space (' ') in locale **loc**.

ispunct

Synopsis

```
int ispunct(int c);
```

Description

ispunct returns nonzero (true) for every printing character for which neither **isspace** nor **isalnum** is true.

ispunct_l

Synopsis

```
int ispunct_l(int c,  
              locale_t loc);
```

Description

ispunct_l returns nonzero (true) for every printing character for which neither **isspace** nor **isalnum** is true in in locale **loc**.

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 ').

isspace_l

Synopsis

```
int isspace_l(int c,  
              locale_t loc);
```

Description

isspace_l returns nonzero (true) if and only if the value of the argument **c** is a standard white-space character in locale **loc**.

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.

isupper_l

Synopsis

```
int isupper_l(int c,  
              locale_t loc);
```

Description

isupper_l returns nonzero (true) if and only if the value of the argument **c** is an uppercase letter in locale **loc**.

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.

isxdigit_l

Synopsis

```
int isxdigit_l(int c,  
               locale_t loc);
```

Description

isxdigit_l returns nonzero (true) if and only if the value of the argument **c** is a hexadecimal digit in locale **loc**.

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 and there are one or more corresponding characters, as specified by the current locale, for which **islower** is true, the **tolower** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

Note that even though **isupper** can return true for some characters, **tolower** may return that uppercase character unchanged as there are no corresponding lowercase characters in the locale.

tolower_l

Synopsis

```
int tolower_l(int c,  
              locale_t loc);
```

Description

tolower_l converts an uppercase letter to a corresponding lowercase letter in locale **loc**. If the argument **c** is a character for which **isupper** is true in locale **loc**, **tolower_l** 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 and there are one or more corresponding characters, as specified by the current locale, for which **isupper** is true, **toupper** returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged. Note that even though **islower** can return true for some characters, **toupper** may return that lowercase character unchanged as there are no corresponding uppercase characters in the locale.

toupper_l

Synopsis

```
int toupper_l(int c,  
              locale_t loc);
```

Description

toupper_l converts a lowercase letter to a corresponding uppercase letter in locale **loc**. If the argument **c** is a character for which **islower** is true in locale **loc**, **toupper_l** returns the corresponding uppercase letter; otherwise, the argument is returned unchanged.

<debugio.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	Return 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_evaluate	Evaluate debug expression
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_clock	get clock
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_clock

Synopsis

```
long debug_clock(void);
```

Description

debug_clock returns the number of milli-seconds since the start of execution.

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. For this to work correctly, the **Startup Completion Breakpoint** project property needs to be set to a point in the program where the startup code has finished initialising, this is typically **main**.

debug_evaluate

Synopsis

```
void debug_evaluate(const char *expression);
```

Description

debug_evaluate instructs the debugger to evaluate the **expression** and display it in the debug terminal.

debug_exit

Synopsis

```
__noreturn 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(const 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 **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 **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

```
long debug_time(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.

<errno.h>

API Summary

Error numbers	
EDOM	Domain error
EILSEQ	Illegal byte sequence
EINVAL	Invalid argument
ENOMEM	No memory available
ERANGE	Result too large or too small
Macros	
errno	Last-set error condition

EDOM

Synopsis

```
#define EDOM ...
```

Description

EDOM - an input argument is outside the defined domain of a mathematical function.

EILSEQ

Synopsis

```
#define EILSEQ    ...
```

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.

EINVAL

Synopsis

```
#define EINVAL 0x06
```

Description

EINVAL - An argument was invalid, or a combination of arguments was invalid.

ENOMEM

Synopsis

```
#define ENOMEM 0x05
```

Description

ENOMEM - no memory can be allocated by a function in the library. Note that **malloc**, **calloc**, and **realloc** do not set **errno** to **ENOMEM** on failure, but other library routines (such as **duplocale**) may set **errno** to **ENOMEM** when memory allocation fails.

ERANGE

Synopsis

```
#define ERANGE    ...
```

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;
```

Description

errno is treated as a writable l-value, but the implementation of how the l-value is read and written is hidden from the user.

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**.

In this implementation, **errno** expands to a function call to `__errno` (MSP430, AVR, MAXQ) or `__aeabi_errno_addr` (ARM) that returns a pointer to a volatile **int**. This function can be implemented by the application to provide a thread-specific **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 $1.0 + \text{DBL_EPSILON} \neq 1.0$.

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 $1.0 + \text{FLT_EPSILON} \neq 1.0$.

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
Coproprocessor 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

__mcr2	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
__fma	VFMA.F64 instruction
__fmaf	VFMA.F32 instruction
__rintn	VRINTN.F64 instruction
__rintnf	VRINTN.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
__ldrexh	LDREXH instruction
__ldrexw	LDREXD instruction
__ldrexh	LDREXH instruction
__ldrht	LDRHT instruction
__ldrsbt	LDRSBT instruction
__ldrsht	LDRSHT instruction
__ldrt	LDRT instruction
__strbt	STRBT instruction
__strex	STREX instruction
__strexh	STREXH instruction
__strexw	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
__qdsb	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**.

__fma

Synopsis

```
double __fma(double a,  
             double b,  
             double c);
```

Description

`__fma` inserts a VFMA.F64 instruction. Returns the value of $a + b * c$.

__fmaf

Synopsis

```
double __fmaf(double a,  
              double b,  
              double c);
```

Description

__fmaf inserts a VFMA.F32 instruction. Returns the value of $a + b \times c$.

__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**.

__ldrex

Synopsis

```
unsigned long long __ldrex(unsigned long long *ptr);
```

Description

__ldrex 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.

__mcr

Synopsis

```
void __mcr(unsigned coproc,  
           unsigned opc1,  
           unsigned src1,  
           unsigned src2,  
           unsigned CRn);
```

Description

__mcr inserts a MCRR instruction. Where **coproc**, **opc1** and **CRn** are compile time constants and **src1**, **src2** are the values to write.

__mcr2

Synopsis

```
void __mcr2(unsigned coproc,  
            unsigned opc1,  
            unsigned src1,  
            unsigned src2,  
            unsigned CRn);
```

Description

__mcr2 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 $\text{res} = \text{val1} + \text{val2}$. This operation sets the Q flag if saturation occurs.

__qadd16

Synopsis

```
int16x2_t __qadd16(int16x2_t val1,  
                  int16x2_t val2);
```

Description

__qadd16 inserts a QADD16 instruction. **__qadd16** returns the 16-bit signed saturated equivalent of

$\text{res}[0] = \text{val1}[0] + \text{val2}[0],$

$\text{res}[1] = \text{val1}[1] + \text{val2}[1]$

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

$$\text{res}[0] = \text{val1}[1] - \text{val2}[1]$$
$$\text{res}[1] = \text{val1}[1] + \text{val2}[0]$$

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.

__qdsb

Synopsis

```
int __qdsb(int val1,  
           int val2);
```

Description

__qdsb inserts a QDSUB instruction. **__qdsb** 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

$\text{res}[0] = \text{val1}[0] + \text{val2}[1]$

$\text{res}[1] = \text{val1}[1] - \text{val2}[0]$

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 $\text{res}=\text{val1}-\text{val2}$. This operation sets the Q flag if saturation occurs.

__qsub16

Synopsis

```
int16x2_t __qsub16(int16x2_t val1,  
                  int16x2_t val2);
```

Description

__qsub16 inserts a QSUB16 instruction. **__qsub16** returns the 16-bit signed saturated equivalent of

$\text{res}[0] = \text{val1}[0] - \text{val2}[0]$

$\text{res}[1] = \text{val1}[1] - \text{val2}[1]$

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

$\text{res}[0] = \text{val1}[0] - \text{val2}[0]$

$\text{res}[1] = \text{val1}[1] - \text{val2}[1]$

$\text{res}[2] = \text{val1}[2] - \text{val2}[2]$

$\text{res}[3] = \text{val1}[3] - \text{val2}[3]$

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.

__rintn

Synopsis

```
double __rintn(double val);
```

Description

__rintn inserts a VRINTN.F64 instruction. Returns the rounded integer value of **val**.

__rintnf

Synopsis

```
float __rintnf(float val);
```

Description

__rintnf inserts a VRINTN.F32 instruction. Returns the rounded integer value of **val**.

__sadd16

Synopsis

```
int16x2_t __sadd16(int16x2_t val1,  
                  int16x2_t val2);
```

Description

__sadd16 inserts a SADD16 instruction. **__sadd16** returns the 16-bit signed equivalent of

$\text{res}[0] = \text{val1}[0] + \text{val2}[0]$

$\text{res}[1] = \text{val1}[1] + \text{val2}[1]$

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

$\text{res}[0] = \text{val1}[0] + \text{val2}[0]$

$\text{res}[1] = \text{val1}[1] + \text{val2}[1]$

$\text{res}[2] = \text{val1}[2] + \text{val2}[2]$

$\text{res}[3] = \text{val1}[3] + \text{val2}[3]$

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_t __sasx(int16x2_t val1,  
                 int16x2_t val2);
```

Description

__sasx inserts a SASX instruction. __sasx returns the 16-bit signed equivalent of

$$\text{res}[0] = \text{val1}[0] - \text{val2}[1]$$
$$\text{res}[1] = \text{val1}[1] + \text{val2}[0]$$

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

$\text{res}[0] = \text{GE}[0] ? \text{val1}[0] : \text{val2}[0]$

$\text{res}[1] = \text{GE}[1] ? \text{val1}[1] : \text{val2}[1]$

$\text{res}[2] = \text{GE}[2] ? \text{val1}[2] : \text{val2}[2]$

$\text{res}[3] = \text{GE}[3] ? \text{val1}[3] : \text{val2}[3]$

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_t __shadd16(int16x2_t val1,  
                    int16x2_t val2);
```

Description

__shadd16 inserts a SHADD16 instruction. **__shadd16** returns the 16-bit signed equivalent of

$$\text{res}[0] = (\text{val1}[0] + \text{val2}[0]) / 2$$
$$\text{res}[1] = (\text{val1}[1] + \text{val2}[1]) / 2$$

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

$$\text{res}[0] = (\text{val1}[0] + \text{val2}[0])/2$$
$$\text{res}[1] = (\text{val1}[1] + \text{val2}[1])/2$$
$$\text{res}[2] = (\text{val1}[2] + \text{val2}[2])/2$$
$$\text{res}[3] = (\text{val1}[3] + \text{val2}[3])/2$$

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

$$\text{res}[0] = (\text{val1}[0] - \text{val2}[1])/2$$
$$\text{res}[1] = (\text{val1}[1] + \text{val2}[0])/2$$

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

$$\text{res}[0] = (\text{val1}[0] + \text{val2}[1])/2$$
$$\text{res}[1] = (\text{val1}[1] - \text{val2}[0])/2$$

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

$$\text{res}[0] = (\text{val1}[0] - \text{val2}[0])/2$$
$$\text{res}[1] = (\text{val1}[1] - \text{val2}[1])/2$$

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

$$\text{res}[0] = (\text{val1}[0] - \text{val2}[0])/2$$
$$\text{res}[1] = (\text{val1}[1] - \text{val2}[1])/2$$
$$\text{res}[2] = (\text{val1}[2] - \text{val2}[2])/2$$
$$\text{res}[3] = (\text{val1}[3] - \text{val2}[3])/2$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[0] + \text{val3}$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[1] + \text{val3}$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[0] + \text{val1}[1] * \text{val2}[1] + \text{val3}$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[1] + \text{val1}[1] * \text{val2}[0] + \text{val3}$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[0] + \text{val3}$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[1] + \text{val3}$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[0] + \text{val1}[1] * \text{val2}[1] + \text{val3}$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[1] + \text{val1}[1] * \text{val2}[0] + \text{val3}$$

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

$$\text{res} = \text{val1}[1] * \text{val2}[0] + \text{val3}$$

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

$$\text{res} = \text{val1}[1] * \text{val2}[1] + \text{val3}$$

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

$$\text{res} = \text{val1}[1] * \text{val2}[0] + \text{val3}$$

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

$$\text{res} = \text{val1}[1] * \text{val2}[1] + \text{val3}$$

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

$$\text{res} = (\text{val1} * \text{val2}[0] + (\text{val3} \ll 16)) \gg 16$$

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

$$\text{res} = (\text{val1} * \text{val2}[1] + (\text{val3} \ll 16)) \gg 16$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[0] - \text{val1}[1] * \text{val2}[1] + \text{val3}$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[1] - \text{val1}[1] * \text{val2}[0] + \text{val3}$$

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 SMLS LD instruction. **__smlsld** returns the equivalent of

$$\text{res} = \text{val1}[0] * \text{val2}[0] - \text{val1}[1] * \text{val2}[1] + \text{val3}$$

where [0] is the lower 16 bits and [1] is the upper 16 bits.

__smlddx

Synopsis

```
long long __smlddx(int16x2 val1,  
                  int16x2 val2,  
                  long long val3);
```

Description

__smlddx inserts a SMLSLDX instruction. **__smlddx** returns the equivalent of

$$\text{res} = \text{val1}[0] * \text{val2}[1] - \text{val1}[1] * \text{val2}[0] + \text{val3}$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[0] + \text{val1}[1] * \text{val2}[1]$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[1] + \text{val1}[1] * \text{val2}[0]$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[0]$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[1]$$

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

$$\text{res} = \text{val1}[1] * \text{val2}[0]$$

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

$$\text{res} = \text{val1}[1] * \text{val2}[1]$$

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

$$\text{res} = (\text{val1}[0] * \text{val2}) \gg 16$$

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

$$\text{res} = (\text{val1}[1] * \text{val2}) \gg 16$$

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

$$\text{res} = \text{val1}[0] * \text{val2}[0] - \text{val1}[1] * \text{val2}[1]$$

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 SMUSD instruction. **__smusdx** returns the equivalent of

$$\text{res} = \text{val1}[0] * \text{val2}[1] - \text{val1}[1] * \text{val2}[0]$$

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

$$\text{res}[0] = \text{val1}[0] + \text{val2}[1]$$
$$\text{res}[1] = \text{val1}[1] - \text{val2}[0]$$

where [0] is the lower 16 bits and [1] is the upper 16 bits. This operation sets the GE bits.

__ssub16

Synopsis

```
int16x2_t __ssub16(int16x2_t val1,  
                  int16x2_t val2);
```

Description

__ssub16 inserts a SSUB16 instruction. **__ssub16** returns the 16-bit signed equivalent of

$$\text{res}[0] = \text{val1}[0] - \text{val2}[0]$$
$$\text{res}[1] = \text{val1}[1] - \text{val2}[1]$$

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.

__strex

Synopsis

```
int __strex(unsigned char val,  
            unsigned *char ptr);
```

Description

__strex inserts a STREXB instruction.

__strex

Synopsis

```
int __strex(unsigned long long val,  
            unsigned *long long ptr);
```

Description

__strex inserts a STREX 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

$\text{res}[0] = \text{val1}[0] + (\text{short})\text{val2}[0]$

$\text{res}[1] = \text{val1}[1] + (\text{short})\text{val2}[2]$

where $\text{res}[0]$ & $\text{val1}[0]$ are the lower 16 bits, $\text{res}[1]$ & $\text{val1}[1]$ are the upper 16 bits, $\text{val2}[0]$ is the lower 8 bits and $\text{val2}[2]$ 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

$$\text{res}[0] = \text{val1}[0] + \text{val2}[0]$$
$$\text{res}[1] = \text{val1}[1] + \text{val2}[1]$$

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

$$\text{res}[0] = \text{val1}[0] - \text{val2}[1]$$
$$\text{res}[1] = \text{val1}[1] + \text{val2}[0]$$

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

$$\text{res}[0] = (\text{val1}[0] + \text{val2}[0])/2$$
$$\text{res}[1] = (\text{val1}[1] + \text{val2}[1])/2$$

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

$$\text{res}[0] = (\text{val1}[0] + \text{val2}[0])/2$$
$$\text{res}[1] = (\text{val1}[1] + \text{val2}[1])/2$$
$$\text{res}[2] = (\text{val1}[2] + \text{val2}[2])/2$$
$$\text{res}[3] = (\text{val1}[3] + \text{val2}[3])/2$$

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

$$\text{res}[0] = (\text{val1}[0] - \text{val2}[1])/2$$
$$\text{res}[1] = (\text{val1}[1] + \text{val2}[0])/2$$

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

$$\text{res}[0] = (\text{val1}[0] + \text{val2}[1])/2$$
$$\text{res}[1] = (\text{val1}[1] - \text{val2}[0])/2$$

where [0] is the lower 16 bits and [1] is the upper 16 bits.

__uhsb16

Synopsis

```
uint16x2 __uhsb16(uint16x2 val1,  
                  uint16x2 val2);
```

Description

__uhsb16 inserts a UHSUB16 instruction. **__uhsb16** returns the 16-bit unsigned equivalent of

$$\text{res}[0] = (\text{val1}[0] - \text{val2}[0])/2$$
$$\text{res}[1] = (\text{val1}[1] - \text{val2}[1])/2$$

where [0] is the lower 16 bits and [1] is the upper 16 bits.

__uhsb8

Synopsis

```
uint8x4 __uhsb8(uint8x4 val1,  
                uint8x4 val2);
```

Description

__uhsb8 inserts a UHSUB8 instruction. **__uhsb8** returns the 8-bit unsigned equivalent of

$$\text{res}[0] = (\text{val1}[0] - \text{val2}[0]) / 2$$
$$\text{res}[1] = (\text{val1}[1] - \text{val2}[1]) / 2$$
$$\text{res}[2] = (\text{val1}[2] - \text{val2}[2]) / 2$$
$$\text{res}[3] = (\text{val1}[3] - \text{val2}[3]) / 2$$

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

$\text{res}[0] = \text{val1}[0] + \text{val2}[0]$

$\text{res}[1] = \text{val1}[1] + \text{val2}[1]$

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

$\text{res}[0] = \text{val1}[0] + \text{val2}[0]$

$\text{res}[1] = \text{val1}[1] + \text{val2}[1]$

$\text{res}[2] = \text{val1}[2] + \text{val2}[2]$

$\text{res}[3] = \text{val1}[3] + \text{val2}[3]$

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

$\text{res}[0] = \text{val1}[0] - \text{val2}[1]$

$\text{res}[1] = \text{val1}[1] + \text{val2}[0]$

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

$$\text{res}[0] = \text{val1}[0] + \text{val2}[1]$$
$$\text{res}[1] = \text{val1}[1] - \text{val2}[0]$$

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

$\text{res}[0] = \text{val1}[0] - \text{val2}[0]$

$\text{res}[1] = \text{val1}[1] - \text{val2}[1]$

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

$\text{res}[0] = \text{val1}[0] - \text{val2}[0]$

$\text{res}[1] = \text{val1}[1] - \text{val2}[1]$

$\text{res}[2] = \text{val1}[2] - \text{val2}[2]$

$\text{res}[3] = \text{val1}[3] - \text{val2}[3]$

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

$$\text{res} = \text{abs}(\text{val1}[0] - \text{val2}[0]) + \text{abs}(\text{val1}[1] - \text{val2}[1]) + (\text{val1}[2] - \text{val2}[2]) + (\text{val1}[3] - \text{val2}[3])$$

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

$$\text{res} = \text{abs}(\text{val1}[0] - \text{val2}[0]) + \text{abs}(\text{val1}[1] - \text{val2}[1]) + (\text{val1}[2] - \text{val2}[2]) + (\text{val1}[3] - \text{val2}[3]) + \text{val3}$$

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

$$\text{res}[0] = \text{val1}[0] + \text{val2}[1]$$
$$\text{res}[1] = \text{val1}[1] - \text{val2}[0]$$

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

$\text{res}[0] = \text{val1}[0] + (\text{unsigned short})\text{val2}[0]$

$\text{res}[1] = \text{val1}[1] + (\text{unsigned short})\text{val2}[2]$

where $\text{res}[0]$ & $\text{val1}[0]$ are the lower 16 bits, $\text{res}[1]$ & $\text{val1}[1]$ are the upper 16 bits, $\text{val2}[0]$ is the lower 8 bits and $\text{val2}[2]$ is the 8 bits starting at bit position 16.

__uxtb16

Synopsis

```
int16x2_t __uxtb16(int8x4_t 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 `^=`.

<itm.h>

API Summary

Variables	
ITM_base	The base address of the ITM peripheral
Functions	
ITM_channel_enabled	Check if an ITM channel is enabled
ITM_send_byte	Send a byte to an ITM channel
ITM_send_half_word	Send a half word to an ITM channel
ITM_send_pc	Send the program counter of the caller to an ITM channel
ITM_send_word	Send a word to an ITM channel

ITM_base

Synopsis

```
unsigned *ITM_base;
```

Description

ITM_base is the base address of the ITM peripheral. It must be assigned for ITM on V7A/V7R architectures. It is not required for V7M architectures.

ITM_channel_enabled

Synopsis

```
int ITM_channel_enabled(int n);
```

Description

ITM_channel_enabled returns 1 if the given ITM channel is enabled otherwise it returns 0.

n is the ITM channel number from 0 to 31.

ITM_send_byte

Synopsis

```
void ITM_send_byte(int n,  
                  unsigned char b);
```

Description

ITM_send_byte sends the byte **b** to the ITM channel **n**.

n is the ITM channel number from 0 to 31.

ITM_send_half_word

Synopsis

```
void ITM_send_half_word(int n,  
                        unsigned short s);
```

Description

ITM_send_half_word sends the half word **s** to the ITM channel **n**.

n is the ITM channel number from 0 to 31.

ITM_send_pc

Synopsis

```
void ITM_send_pc(int n);
```

Description

ITM_send_pc sends the program counter of the caller to the ITM channel **n**.

n is the ITM channel number from 0 to 31.

ITM_send_word

Synopsis

```
void ITM_send_word(int n,  
                  unsigned w);
```

Description

ITM_send_word sends the word **w** to the ITM channel **n**.

n is the ITM channel number from 0 to 31.

<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	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();  
  
// 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 16BBytes 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 created 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();  
  
// 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);

// 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);

// 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
Multi-byte values	
MB_LEN_MAX	maximum number of bytes in a multi-byte 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**.

MB_LEN_MAX

Synopsis

```
#define MB_LEN_MAX 4
```

Description

MB_LEN_MAX is the maximum number of bytes in a multi-byte character for any supported locale. Unicode (ISO 10646) characters between 0 and 10FFFF inclusive are supported which convert to a maximum of four bytes in the UTF-8 encoding.

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 maximum 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 maximum value for an object of type **unsigned short int**.

<locale.h>

API Summary

Structures	
lconv	Formatting info for numeric values
Functions	
localeconv	Get current locale data
setlocale	Set Locale

lconv

Synopsis

```
typedef struct {
    char *decimal_point;
    char *thousands_sep;
    char *grouping;
    char *int_curr_symbol;
    char *currency_symbol;
    char *mon_decimal_point;
    char *mon_thousands_sep;
    char *mon_grouping;
    char *positive_sign;
    char *negative_sign;
    char int_frac_digits;
    char frac_digits;
    char p_cs_precedes;
    char p_sep_by_space;
    char n_cs_precedes;
    char n_sep_by_space;
    char p_sign_posn;
    char n_sign_posn;
    char int_p_cs_precedes;
    char int_n_cs_precedes;
    char int_p_sep_by_space;
    char int_n_sep_by_space;
    char int_p_sign_posn;
    char int_n_sign_posn;
} lconv;
```

Description

lconv structure holds formatting information on how numeric values are to be written. Note that the order of fields in this structure is not consistent between implementations, nor is it consistent between C89 and C99 standards.

The members **decimal_point**, **grouping**, and **thousands_sep** are controlled by **LC_NUMERIC**, the remainder by **LC_MONETARY**.

The members **int_n_cs_precedes**, **int_n_sep_by_space**, **int_n_sign_posn**, **int_p_cs_precedes**, **int_p_sep_by_space**, and **int_p_sign_posn** are added by the C99 standard.

We have standardized on the ordering specified by the ARM EABI for the base of this structure. This ordering is neither that of C89 nor C99.

Member	Description
currency_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.

Return Value

setlocale returns the name of the current locale.

<math.h>

API Summary

Comparison Macros	
isgreater	Is greater
isgreaterequal	Is greater or equal
isless	Is less
islessequal	Is less or equal
islessgreater	Is less or greater
isunordered	Is unordered
Classification 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	

exp	Compute exponential of a double
exp2	Compute binary exponential of a double
exp2f	Compute binary exponential of a float
expf	Compute exponential of a float
expm1	Compute exponential minus one of a double
expm1f	Compute exponential minus one of a float
frexp	Set exponent of a double
frexpf	Set exponent of a float
ilogb	Compute integer binary logarithm of a double
ilogbf	Compute integer binary logarithm 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
log1p	Compute natural logarithm plus one of a double
log1pf	Compute natural logarithm plus one of a float
log2	Compute binary logarithm of a double
log2f	Compute binary logarithm of a float
logb	Compute floating-point base logarithm of a double
logbf	Compute floating-point base logarithm of a float
logf	Compute natural logarithm of a float
scalbln	Scale a double
scalblnf	Scale a float
scalbn	Scale a double
scalbnf	Scale a float
Rounding and remainder 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 double
floorf	Compute largest integer not greater than a float
fmod	Compute remainder after division of two doubles
fmodf	Compute remainder after division of two floats
llrint	Round and cast double to long long
llrintf	Round and cast float to long long

llround	Round and cast double to long long
llroundf	Round and cast float to long long
lrint	Round and cast double to long
lrintf	Round and cast float to long
lround	Round and cast double to long
lroundf	Round and cast float to long
modf	Break a double into integer and fractional parts
modff	Break a float into integer and fractional parts
nearbyint	Round double to nearby integral value
nearbyintf	Round float to nearby integral value
remainder	Compute remainder of a double
remainderf	Compute remainder of a float
remquo	Compute remainder and quotient of a double
remquof	Compute remainder and quotient of a float
rint	Round a double to an integral value
rintf	Round a float to an integral value
round	Round a double to the nearest integral value
roundf	Round a float to the nearest integral value
trunc	Truncate a double value
truncf	Truncate a float value
Power functions	
cbrt	Compute cube root of a double
cbrtf	Compute cube root of a float
hypot	Compute complex magnitude of two doubles
hypotf	Compute complex magnitude of two floats
pow	Raise a double to a power
powf	Raise a float to a power
sqrt	Compute square root of a double
sqrtf	Compute square root of a float
Absolute value functions	
fabs	Compute absolute value of a double
fabsf	Compute absolute value of a float
Maximum, minimum, and positive difference functions	
fdim	Compute positive difference of two doubles
fdimf	Compute positive difference of two floats

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
Fused multiply functions	
fma	Compute fused multiply-add of doubles
fmaf	Compute fused multiply-add of floats
Floating-point manipulation functions	
copysign	Copy magnitude and sign of a double
copysignf	Copy magnitude and sign of a float
nextafter	Next representable double value
nextafterf	Next representable float value
Error and Gamma functions	
erf	Compute error function of a double
erfc	Compute complementary error function of a double
erfcf	Compute complementary error function of a float
erff	Compute error function of a float
lgamma	Compute log-gamma function of a double
lgammaf	Compute log-gamma function of a float
tgamma	Compute gamma function of a double
tgammaf	Compute gamma function 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 $[-\frac{\pi}{2}, \frac{\pi}{2}]$ 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 $[-\pi/2, \pi/2]$ 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 $|x| > \sim 709.782$, **errno** is set to **EDOM** and **asinh** returns **HUGE_VAL**.

If **x** is +, , or NaN, **asinh** returns $|x|$. If $|x| > \sim 709.782$, **asinh** returns + or depending upon the sign of **x**.

asinhf

Synopsis

```
float asinhf(float x);
```

Description

asinhf calculates the hyperbolic sine of **x**.

If $|x| > \sim 88.7228$, **errno** is set to **EDOM** and **asinhf** returns **HUGE_VALF**.

If **x** is +, , or NaN, **asinhf** returns $|x|$. If $|x| > \sim 88.7228$, **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 $[-\frac{\pi}{2}, \frac{\pi}{2}]$ radians.

atan2

Synopsis

```
double atan2(double y,  
             double x);
```

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 $[-\pi, +\pi]$ 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 .

atan2((anything but 0 and NaN), 0) is .

atan2((anything but and NaN), +) is 0.

atan2((anything but and NaN),) is .

atan2(, +) is .

atan2(,) is .

atan2(, (anything but 0, NaN, and)) is .

atan2f

Synopsis

```
float atan2f(float y,  
            float x);
```

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 $[-\pi, +\pi]$ 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 .

atan2f((anything but 0 and NaN), 0) is .

atan2f((anything but and NaN), +) is 0.

atan2f((anything but and NaN),) is .

atan2f(, +) is .

atan2f(,) is .

atan2f(, (anything but 0, NaN, and)) is .

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 $[-\pi/2, \pi/2]$ radians.

atanh

Synopsis

```
double atanh(double x);
```

Description

atanh returns the inverse hyperbolic tangent of **x**.

If $|x| \geq 1$, **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 **x**.

If $|x| > 1$ **atanhf** returns NaN. If **x** is NaN, **atanhf** returns that NaN. If **x** is 1, **atanhf** returns . If **x** 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 .

copysign

Synopsis

```
double copysign(double x,  
                double y);
```

Description

copysign returns a value with the magnitude of **x** and the sign of **y**.

copysignf

Synopsis

```
float copysignf(float x,  
               float y);
```

Description

copysignf returns a value with the magnitude of **x** and the sign of **y**.

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 $|x| > \sim 709.782$, **errno** is set to **EDOM** and **cosh** returns **HUGE_VAL**.

If **x** is +, , or NaN, **cosh** returns $|x|$.> If $|x| > \sim 709.782$, **cosh** returns + or depending upon the sign of **x**.

coshf

Synopsis

```
float coshf(float x);
```

Description

coshf calculates the hyperbolic sine of **x**.

If $|x| > \sim 88.7228$, **errno** is set to **EDOM** and **coshf** returns **HUGE_VALF**.

If **x** is +, , or NaN, **coshf** returns $|x|$.

If $|x| > \sim 88.7228$, **coshf** returns + or depending upon the sign of **x**.

erf

Synopsis

```
double erf(double x);
```

Description

erf returns the error function for **x**.

erfc

Synopsis

```
double erfc(double x);
```

Description

erfc returns the complementary error function for **x**.

erfcf

Synopsis

```
float erfcf(float x);
```

Description

erfcf returns the complementary error function for **x**.

erff

Synopsis

```
float erff(float x);
```

Description

erff returns the error function for **x**.

exp

Synopsis

```
double exp(double x);
```

Description

exp computes the base-e exponential of **x**.

If $|x| > \sim 709.782$, **errno** is set to **EDOM** and **exp** returns **HUGE_VAL**.

If **x** is NaN, **exp** returns NaN.

If **x** is 0, **exp** returns 1.

If **x** is negative infinity, **exp** returns 0.

exp2

Synopsis

```
double exp2(double x);
```

Description

exp2 returns 2 raised to the power of **x**.

exp2f

Synopsis

```
float exp2f(float x);
```

Description

exp2f returns 2 raised to the power of **x**.

expf

Synopsis

```
float expf(float x);
```

Description

expf computes the base-*e* exponential of *x*.

If $|x| > \sim 88.722$, **errno** is set to **EDOM** and **expf** returns **HUGE_VALF**. If *x* is NaN, **expf** returns NaN.

If *x* is **0**, **expf** returns **1**.

If *x* is **0**, **expf** returns **0**.

expm1

Synopsis

```
double expm1(double x);
```

Description

expm1 returns e raised to the power of x minus one.

expm1f

Synopsis

```
float expm1f(float x);
```

Description

expm1f returns e raised to the power of x minus one.

fabs

Synopsis

```
double fabs(double x);
```

fabsf

Synopsis

```
float fabsf(float x);
```

Description

fabsf computes the absolute value of the floating-point number **x**.

fdim

Synopsis

```
double fdim(double x,  
            double y);
```

Description

fdim returns the positive difference between **x** and **y**.

fdimf

Synopsis

```
float fdimf(float x,  
            float y);
```

Description

fdimf returns the positive difference between **x** and **y**.

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 .

fma

Synopsis

```
double fma(double x,  
           double y,  
           double z);
```

Description

fma computes $x \times y + z$ with a single rounding.

fmaf

Synopsis

```
float fmaf(float x,  
           float y,  
           float z);
```

Description

fmaf computes $x \times y + z$ with a single rounding.

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**. **fmod** 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**.

fmod (NaN, **y**) is NaN. **fmod** (**x**, NaN) is NaN. **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**.

fmodf (NaN, **y**) is NaN. **fmodf** (**x**, NaN) is NaN. **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) ( __is_float32(x) ? __float32_classify(x) : __float64_classify(x) )
```

Description

fpclassify classifies *x* as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. **fpclassify** returns 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 $x * 2^{\text{exp}}$.

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 $[1, 1)$ or zero, and value equals $x * 2^{\text{exp}}$.

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**, $\sqrt{x^2 + y^2}$, 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.

ilogb

Synopsis

```
int ilogb(double x);
```

Description

ilogb returns the integral part of the logarithm of **x**, using **FLT_RADIX** as the base for the logarithm.

ilogbf

Synopsis

```
int ilogbf(float x);
```

Description

ilogbf returns the integral part of the logarithm of **x**, using **FLT_RADIX** as the base for the logarithm.

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). **isfinite** returns a non-zero value if and only if **x** has a finite value.

isgreater

Synopsis

```
#define isgreater(x,y) (!isunordered(x, y) && (x > y))
```

Description

isgreater returns whether **x** is greater than **y**.

isgreaterequal

Synopsis

```
#define isgreaterequal(x,y) (!isunordered(x, y) && (x >= y))
```

Description

isgreaterequal returns whether **x** is greater than or equal to **y**.

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.

isless

Synopsis

```
#define isless(x,y) (!isunordered(x, y) && (x < y))
```

Description

isless returns whether **x** is less than **y**.

islessequal

Synopsis

```
#define islessequal(x,y) (!isunordered(x, y) && (x <= y))
```

Description

islessequal returns whether **x** is less than or equal to **y**.

islessgreater

Synopsis

```
#define islessgreater(x,y) (!isunordered(x, y) && (x < y || x > y))
```

Description

islessgreater returns whether **x** is less than or greater than **y**.

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).. **isnormal** returns a non-zero value if and only if **x** has a normal value.

isunordered

Synopsis

```
#define isunordered(a,b) (fpclassify(a) == FP_NAN || fpclassify(b) == FP_NAN)
```

Description

isunordered returns whether **x** or **y** are unordered values.

ldexp

Synopsis

```
double ldexp(double x,  
             int exp);
```

Description

ldexp multiplies a floating-point number by an integral power of 2.

ldexp returns $x * 2^{\text{exp}}$.

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 $x * 2^{\text{exp}}$. If the result overflows, **errno** is set to **ERANGE** and **ldexpf** returns **HUGE_VALF**.

If **x** is NaN, **ldexpf** returns **x**. If the result overflows, **ldexpf** returns .

lgamma

Synopsis

```
double lgamma(double x);
```

Description

lgamma returns the natural logarithm of the gamma function for **x**.

lgammaf

Synopsis

```
float lgammaf(float x);
```

Description

lgammaf returns the natural logarithm of the gamma function for **x**.

llrint

Synopsis

```
long long int llrint(double x);
```

Description

llrint rounds *x* to an integral value and returns it as a long long int.

llrintf

Synopsis

```
long long int llrintf(float x);
```

Description

llrintf rounds **x** to an integral value and returns it as a long long int.

llround

Synopsis

```
long long int llround(double x);
```

Description

llround rounds *x* to an integral value, with halfway cases rounded away from zero, and returns it as a long long int.

llroundf

Synopsis

```
long long int llroundf(float x);
```

Description

llroundf rounds *x* to an integral value, with halfway cases rounded away from zero, and returns it as a long long int.

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**.

log1p

Synopsis

```
double log1p(double x);
```

Description

log1p computes the base-*e* logarithm of *x* plus one.

log1pf

Synopsis

```
float log1pf(float x);
```

Description

log1pf computes the base-e logarithm of **x** plus one.

log2

Synopsis

```
double log2(double x);
```

Description

log2 computes the base-2 logarithm of **x**.

log2f

Synopsis

```
float log2f(float x);
```

Description

log2f computes the base-2 logarithm of **x**.

logb

Synopsis

```
double logb(double x);
```

Description

logb computes the base-*FLT_RADIX* logarithm of **x**.

logbf

Synopsis

```
float logbf(float x);
```

Description

logbf computes the base-*FLT_RADIX* logarithm of **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**.

lrint

Synopsis

```
long int lrint(double x);
```

Description

lrint rounds *x* to an integral value and returns it as a long int.

lrintf

Synopsis

```
long int lrintf(float x);
```

Description

lrintf rounds **x** to an integral value and returns it as a long int.

lround

Synopsis

```
long int lround(double x);
```

Description

lround rounds *x* to an integral value, with halfway cases rounded away from zero, and returns it as a long int.

lroundf

Synopsis

```
long int lroundf(float x);
```

Description

lroundf rounds *x* to an integral value, with halfway cases rounded away from zero, and returns it as a long int.

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**.

nearbyint

Synopsis

```
double nearbyint(double);
```

Description

nearbyint Rounds *x* to an integral value.

nearbyintf

Synopsis

```
float nearbyintf(float);
```

Description

nearbyintf Rounds *x* to an integral value.

nextafter

Synopsis

```
double nextafter(double x,  
                 double y);
```

Description

nextafter Returns the next representable value after **x** in the direction of **y**.

nextafterf

Synopsis

```
float nextafterf(float x,  
                float y);
```

Description

nextafterf Returns the next representable value after **x** in the direction of **y**.

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 $y = \text{NaN}$, **pow** returns NaN.

If $x = \text{NaN}$ 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 x,  
          float y);
```

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 $y = \text{NaN}$, **powf** returns NaN.

If $x = \text{NaN}$ 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.

remainder

Synopsis

```
double remainder(double numer,  
                double denom);
```

Description

remainder computes the remainder of **numer** divided by **denom**.

remainderf

Synopsis

```
float remainderf(float numer,  
                float denom);
```

Description

remainderf computes the remainder of **numer** divided by **denom**.

remquo

Synopsis

```
double remquo(double numer,  
              double denom,  
              int *quot);
```

Description

remquo computes the remainder of **numer** divided by **denom** and the quotient pointed by **quot**.

remquoof

Synopsis

```
float remquoof(float numer,  
               float denom,  
               int *quot);
```

Description

remquoof computes the remainder of **numer** divided by **denom** and the quotient pointed by **quot**.

rint

Synopsis

```
double rint(double x);
```

Description

rint rounds **x** to an integral value.

rintf

Synopsis

```
float rintf(float x);
```

Description

rintf rounds **x** to an integral value.

round

Synopsis

```
double round(double x);
```

Description

round rounds *x* to an integral value, with halfway cases rounded away from zero.

roundf

Synopsis

```
float roundf(float x);
```

Description

roundf rounds *x* to an integral value, with halfway cases rounded away from zero.

scalbln

Synopsis

```
double scalbln(double x,  
               long int exp);
```

Description

scalbln multiplies **x** by **FLT_RADIX** raised to the power **exp**.

scalblnf

Synopsis

```
float scalblnf(float x,  
              long int exp);
```

Description

scalblnf multiplies **x** by FLT_RADIX raised to the power **exp**.

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 $x * \text{DBL_RADIX}^{\text{exp}}$.

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 $x * \text{FLT_RADIX}^{\text{exp}}$.

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 determines whether the sign of **x** is negative. **signbit** returns a non-zero value if and only if **x** 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 $|x| > 709.782$, **errno** is set to **EDOM** and **sinh** returns **HUGE_VAL**.

If **x** is +, -, or NaN, **sinh** returns $|x|$. If $|x| > \sim 709.782$, **sinh** returns + or - depending upon the sign of **x**.

sinhf

Synopsis

```
float sinhf(float x);
```

Description

sinhf calculates the hyperbolic sine of **x**.

If $|x| > \sim 88.7228$, **errno** is set to **EDOM** and **sinhf** returns **HUGE_VALF**.

If **x** is +, , or NaN, **sinhf** returns $|x|$. If $|x| > \sim 88.7228$, **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 **sqrt** 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 **sqrtf** 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.

tgamma

Synopsis

```
double tgamma(double x);
```

Description

tgamma returns the gamma function for **x**.

tgammaf

Synopsis

```
float tgammaf(float x);
```

Description

tgammaf returns the gamma function for **x**.

trunc

Synopsis

```
double trunc(double x);
```

Description

trunc rounds **x** to an integral value that is not larger in magnitude than **x**.

truncf

Synopsis

```
float truncf(float x);
```

Description

truncf rounds *x* to an integral value that is not larger in magnitude than *x*.

<setjmp.h>

API Summary

Functions	
longjmp	Restores the saved environment
setjmp	Save calling environment for non-local jump

longjmp

Synopsis

```
__noreturn 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	
max_align_t	max_align_t type
ptrdiff_t	ptrdiff_t type
size_t	size_t type

NULL

Synopsis

```
#define NULL 0
```

Description

NULL is the null pointer constant.

max_align_t

Synopsis

```
typedef long double max_align_t;
```

Description

max_align_t is a type whose alignment requirement is at least as strict (as large) as that of every scalar type.

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 __RAL_PTRDIFF_T ptrdiff_t;
```

Description

ptrdiff_t is the signed integral type of the result of subtracting two pointers.

size_t

Synopsis

```
typedef __RAL_SIZE_T size_t;
```

Description

size_t is the unsigned integral type returned by the sizeof operator.

<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 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 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 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 [-]ddd. 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 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.dddedd`, 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 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 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 `l`, `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 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 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 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 **Scnf 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 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**st 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
MB_CUR_MAX	Maximum number of bytes in a multi-byte character in the current locale
RAND_MAX	RAND_MAX
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
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
Multi-byte/wide character conversion functions	
mblen	Determine number of bytes in a multi-byte character
mblen_l	Determine number of bytes in a multi-byte character
Multi-byte/wide string conversion functions	
mbstowcs	Convert multi-byte string to wide string
mbstowcs_l	Convert multi-byte string to wide string using specified locale
mbtowc	Convert multi-byte character to wide character
mbtowc_l	Convert multi-byte character to wide character

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.

MB_CUR_MAX

Synopsis

```
#define MB_CUR_MAX  __RAL_mb_max(&__RAL_global_locale)
```

Description

MB_CUR_MAX expands to a positive integer expression with type **size_t** that is the maximum number of bytes in a multi-byte character for the extended character set specified by the current locale (category LC_CTYPE).

MB_CUR_MAX is never greater than **MB_LEN_MAX**.

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

```
__noreturn 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.

mblen

Synopsis

```
int mblen(const char *s,  
          size_t n);
```

Description

mblen determines the number of bytes contained in the multi-byte character pointed to by **s** in the current locale.

If **s** is a null pointer, **mblen** returns a nonzero or zero value, if multi-byte character encodings, respectively, do or do not have state-dependent encodings

If **s** is not a null pointer, **mblen** either returns 0 (if **s** points to the null character), or returns the number of bytes that are contained in the multi-byte character (if the next **n** or fewer bytes form a valid multi-byte character), or returns 1 (if they do not form a valid multi-byte character).

Note

Except that the conversion state of the **mbtowc** function is not affected, it is equivalent to

```
mbtowc((wchar_t *)0, s, n);
```

Note

It is guaranteed that no library function in the Standard C library calls **mblen**.

See Also

[mblen_l](#), [mbtowc](#)

mblen_l

Synopsis

```
int mblen_l(const char *s,
            size_t n,
            __locale_t *loc);
```

Description

mblen_l determines the number of bytes contained in the multi-byte character pointed to by **s** in the locale **loc**.

If **s** is a null pointer, **mblen_l** returns a nonzero or zero value, if multi-byte character encodings, respectively, do or do not have state-dependent encodings.

If **s** is not a null pointer, **mblen_l** either returns 0 (if **s** points to the null character), or returns the number of bytes that are contained in the multi-byte character (if the next **n** or fewer bytes form a valid multi-byte character), or returns 1 (if they do not form a valid multi-byte character).

Note

Except that the conversion state of the **mbtowc_l** function is not affected, it is equivalent to

```
mbtowc((wchar_t *)0, s, n, loc);
```

Note

It is guaranteed that no library function in the Standard C library calls **mblen_l**.

See Also

[mblen_l](#), [mbtowc_l](#)

mbstowcs

Synopsis

```
size_t mbstowcs(wchar_t *pwcs,  
                const char *s,  
                size_t n);
```

Description

mbstowcs converts a sequence of multi-byte characters that begins in the initial shift state from the array pointed to by **s** into a sequence of corresponding wide characters and stores not more than **n** wide characters into the array pointed to by **pwcs**.

No multi-byte characters that follow a null character (which is converted into a null wide character) will be examined or converted. Each multi-byte character is converted as if by a call to the **mbtowc** function, except that the conversion state of the **mbtowc** function is not affected.

No more than **n** elements will be modified in the array pointed to by **pwcs**. If copying takes place between objects that overlap, the behavior is undefined.

mbstowcs returns 1 if an invalid multi-byte character is encountered, otherwise **mbstowcs** returns the number of array elements modified (if any), not including a terminating null wide character.

mbstowcs_l

Synopsis

```
size_t mbstowcs_l(wchar_t *pwcs,  
                  const char *s,  
                  size_t n,  
                  __locale_s *loc);
```

Description

mbstowcs_l is as **mbstowcs** except that the local **loc** is used for the conversion as opposed to the current locale.

See Also

[mbstowcs](#).

mbtowc

Synopsis

```
int mbtowc(wchar_t *pwc,  
           const char *s,  
           size_t n);
```

Description

mbtowc converts a single multi-byte character to a wide character in the current locale.

If **s** is a null pointer, **mbtowc** returns a nonzero value if multi-byte character encodings are state-dependent in the current locale, and zero otherwise.

If **s** is not null and the object that **s** points to is a wide-character null character, **mbtowc** returns 0.

If **s** is not null and the object that points to forms a valid multi-byte character, **mbtowc** returns the length in bytes of the multi-byte character.

If the object that points to does not form a valid multi-byte character within the first **n** characters, it returns 1.

See Also

[mbtowc_l](#)

mbtowc_l

Synopsis

```
int mbtowc_l(wchar_t *pwc,  
             const char *s,  
             size_t n,  
             __locale_t *loc);
```

Description

mbtowc_l converts a single multi-byte character to a wide character in locale **loc**.

If **s** is a null pointer, **mbtowc_l** returns a nonzero value if multi-byte character encodings are state-dependent in the locale **loc**, and zero otherwise.

If **s** is not null and the object that **s** points to is a wide-character null character, **mbtowc_l** returns 0.

If **s** is not null and the object that points to forms a valid multi-byte character, **mbtowc_l** returns the length in bytes of the multi-byte character.

If the object that **s** points to does not form a valid multi-byte character within the first **n** characters, it returns 1.

See Also

[mbtowc](#)

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** function. 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 **malloc** 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](#)

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>

Overview

The header file <string.h> defines functions that operate on arrays that are interpreted as null-terminated strings.

Various methods are used for determining the lengths of the arrays, but in all cases a **char *** or **void *** argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

Where an argument declared as **size_t n** specifies the length of an array for a function, *n* can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function, pointer arguments must have valid values on a call with a zero size. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

API Summary

Copying functions	
memccpy	Copy memory with specified terminator (POSIX extension)
memcpy	Copy memory
memcpy_fast	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**.

memcpy_fast

Synopsis

```
void *memcpy_fast(void *s1,  
                  const void *s2,  
                  size_t n);
```

Description

memcpy_fast copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The behavior of **memcpy_fast** is undefined if copying takes place between objects that overlap. The implementation of **memcpy_fast** is optimized for speed for all cases of **memcpy** and as such has a large code memory requirement. This function is implemented for little-endian ARM and 32-bit Thumb-2 instruction sets only.

memcpy_fast 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**.

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 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 **nstrlen(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.

strncpy

Synopsis

```
size_t strncpy(char *s1,  
               const char *s2,  
               size_t n);
```

Description

strncpy copies up to **n** 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 **strncpy** is undefined if copying takes place between objects that overlap.

strncpy returns the number of characters it tried to copy, which is the length of the string **s2** or **n**, whichever is smaller.

Note

strncpy 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

Types	
clock_t	Clock type
time_t	Time type
tm	Time structure
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

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. **asctime** 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_mday**, **tm_mon** and **tm_year** fields are consistent. The validated ***tp** struct is converted to the number of seconds since UTC 1 January 1970 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
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for locale
%#c	Date and time formatted as "%A, %B %#d, %Y, %H:%M:%S" (Microsoft extension)
%C	Century number
%d	Day of month as a decimal number [01,31]
%#d	Day of month without leading zero [1,31]
%D	Date in the form %m/%d/%y (POSIX.1-2008 extension)
%e	Day of month [1,31], single digit preceded by space
%F	Date in the format %Y-%m-%d
%h	Abbreviated month name as %b
%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]
%k	Hour in 24-hour clock format [0,23] (POSIX.1-2008 extension)

%l	Hour in 12-hour clock format [0,12] (POSIX.1-2008 extension)
%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]
%n	Insert newline character (POSIX.1-2008 extension)
%p	Locale's a.m or p.m indicator for 12-hour clock
%r	Time as %l:%M:%s %p (POSIX.1-2008 extension)
%R	Time as %H:%M (POSIX.1-2008 extension)
%S	Second as a decimal number [00,59]
%t	Insert tab character (POSIX.1-2008 extension)
%T	Time as %H:%M:%S
%#S	Second as a decimal number without leading zeros [0,59]
%U	Week of year as a decimal number [00,53], Sunday is first day of the week
%#U	Week of year 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]
%Y	Year with century, as decimal number
%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 January 1970, negative values indicate time before UTC 1 January 1970.

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
Constants	
WEOF	End of file indication
Types	
wchar_t	Wide character type
wint_t	Wide integer type
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
wcsnspn	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
wcsbrk	Find first occurrence of characters within string
wcsrchr	Find last occurrence of character within string

wcssp	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
Multi-byte/wide string conversion functions	
mbrtowc	Convert multi-byte character to wide character
mbrtowc_l	Convert multi-byte character to wide character
msbinit	Query conversion state
wctomb	Convert wide character to multi-byte character (restartable)
wctomb_l	Convert wide character to multi-byte character (restartable)
wctob	Convert wide character to single-byte character
wctob_l	Convert wide character to single-byte character
Multi-byte to wide character conversions	
mbrlen	Determine number of bytes in a multi-byte character
mbrlen_l	Determine number of bytes in a multi-byte character
mbsrtowcs	Convert multi-byte string to wide character string
mbsrtowcs_l	Convert multi-byte string to wide character string
Single-byte to wide character conversions	
btowc	Convert single-byte character to wide character
btowc_l	Convert single-byte character to wide character

WCHAR_MAX

Synopsis

```
#define WCHAR_MAX ...
```

Description

WCHAR_MAX is the maximum value for an object of type **wchar_t**. Although capable of storing larger values, the maximum value implemented by the conversion functions in the library is the value 0x10FFFF defined by ISO 10646.

WCHAR_MIN

Synopsis

```
#define WCHAR_MIN    ...
```

Description

WCHAR_MIN is the minimum value for an object of type `wchar_t`.

WEOF

Synopsis

```
#define WEOF ((wint_t)~0U)
```

Description

WEOF expands to a constant value that does not correspond to any character in the wide character set. It is typically used to indicate an end of file condition.

btowc

Synopsis

```
wint_t btowc(int c);
```

Description

btowc function determines whether **c** constitutes a valid single-byte character. If **c** is a valid single-byte character, **btowc** returns the wide character representation of that character

btowc returns WEOF if **c** has the value **EOF** or if `(unsigned char)c` does not constitute a valid single-byte character in the initial shift state.

btowc_l

Synopsis

```
wint_t btowc_l(int c,  
               locale_t loc);
```

Description

btowc_l function determines whether **c** constitutes a valid single-byte character in the locale **loc**. If **c** is a valid single-byte character, **btowc_l** returns the wide character representation of that character

btowc_l returns WEOF if **c** has the value **EOF** or if `(unsigned char)c` does not constitute a valid single-byte character in the initial shift state.

mbrlen

Synopsis

```
size_t mbrlen(const char *s,  
              size_t n,  
              mbstate_t *ps);
```

Note

mbrlen function is equivalent to the call:

```
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);
```

where **internal** is the **mbstate_t** object for the **mbrlen** function, except that the expression designated by **ps** is evaluated only once.

mbrlen_l

Synopsis

```
size_t mbrlen_l(const char *s,  
                size_t n,  
                mbstate_t *ps,  
                locale_t loc);
```

Note

mbrlen_l function is equivalent to the call:

```
mbrtowc_l(NULL, s, n, ps != NULL ? ps : &internal, loc);
```

where **internal** is the **mbstate_t** object for the **mbrlen** function, except that the expression designated by **ps** is evaluated only once.

mbrtowc

Synopsis

```
size_t mbrtowc(wchar_t *pwc,  
               const char *s,  
               size_t n,  
               mbstate_t *ps);
```

Description

mbrtowc converts a single multi-byte character to a wide character in the current locale.

If **s** is a null pointer, **mbrtowc** is equivalent to `mbrtowc(NULL, "", 1, ps)`, ignoring **pwc** and **n**.

If **s** is not null and the object that **s** points to is a wide-character null character, **mbrtowc** returns 0.

If **s** is not null and the object that points to forms a valid multi-byte character with a most **n** bytes, **mbrtowc** returns the length in bytes of the multi-byte character and stores that wide character to the object pointed to by **pwc** (if **pwc** is not null).

If the object that points to forms an incomplete, but possibly valid, multi-byte character, **mbrtowc** returns 2. If the object that points to does not form a partial multi-byte character, **mbrtowc** returns 1.

See Also

[mbtowc](#), [mbrtowc_l](#)

mbrtowc_l

Synopsis

```
size_t mbrtowc_l(wchar_t *pwc,  
                 const char *s,  
                 size_t n,  
                 mbstate_t *ps,  
                 locale_t loc);
```

Description

mbrtowc_l converts a single multi-byte character to a wide character in the locale **loc**.

If **s** is a null pointer, **mbrtowc_l** is equivalent to `mbrtowc(NULL, "", 1, ps)`, ignoring **pwc** and **n**.

If **s** is not null and the object that **s** points to is a wide-character null character, **mbrtowc_l** returns 0.

If **s** is not null and the object that points to forms a valid multi-byte character with a most **n** bytes, **mbrtowc_l** returns the length in bytes of the multi-byte character and stores that wide character to the object pointed to by **pwc** (if **pwc** is not null).

If the object that points to forms an incomplete, but possibly valid, multi-byte character, **mbrtowc_l** returns 2. If the object that points to does not form a partial multi-byte character, **mbrtowc_l** returns 1.

See Also

[mbrtowc](#), [mbtowc_l](#)

mbsrtowcs

Synopsis

```
size_t mbsrtowcs(wchar_t *dst,  
                 const char **src,  
                 size_t len,  
                 mbstate_t *ps);
```

Description

mbsrtowcs converts a sequence of multi-byte characters that begins in the conversion state described by the object pointed to by **ps**, from the array indirectly pointed to by **src** into a sequence of corresponding wide characters. If **dst** is not a null pointer, the converted characters are stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null character, which is also stored.

Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multi-byte character, or (if **dst** is not a null pointer) when **len** wide characters have been stored into the array pointed to by **dst**. Each conversion takes place as if by a call to the **mbtowc** function.

If **dst** is not a null pointer, the pointer object pointed to by **src** is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multi-byte character converted (if any). If conversion stopped due to reaching a terminating null character and if **dst** is not a null pointer, the resulting state described is the initial conversion state.

See Also

[mbsrtowcs_l](#), [mbrtowc](#)

mbsrtowcs_l

Synopsis

```
size_t mbsrtowcs_l(wchar_t *dst,  
                   const char **src,  
                   size_t len,  
                   mbstate_t *ps,  
                   locale_t loc);
```

Description

mbsrtowcs_l converts a sequence of multi-byte characters that begins in the conversion state described by the object pointed to by **ps**, from the array indirectly pointed to by **src** into a sequence of corresponding wide characters. If **dst** is not a null pointer, the converted characters are stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null character, which is also stored.

Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multi-byte character, or (if **dst** is not a null pointer) when **len** wide characters have been stored into the array pointed to by **dst**. Each conversion takes place as if by a call to the **mbrtowc** function.

If **dst** is not a null pointer, the pointer object pointed to by **src** is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multi-byte character converted (if any). If conversion stopped due to reaching a terminating null character and if **dst** is not a null pointer, the resulting state described is the initial conversion state.

See Also

[mbsrtowcs_l](#), [mbrtowc](#)

msbinit

Synopsis

```
int msbinit(const mbstate_t *ps);
```

Description

msbinit function returns nonzero if **ps** is a null pointer or if the pointed-to object describes an initial conversion state; otherwise, **msbinit** returns zero.

wchar_t

Synopsis

```
typedef __RAL_WCHAR_T wchar_t;
```

Description

wchar_t holds a single wide character.

Depending on implementation you can control whether **wchar_t** is represented by a short 16-bit type or the standard 32-bit type.

wcrtomb

Synopsis

```
size_t wcrtomb(char *s,  
               wchar_t wc,  
               mbstate_t *ps);
```

If **s** is a null pointer, **wcrtomb** function is equivalent to the call `wcrtomb(buf, L'\0', ps)` where **buf** is an internal buffer.

If **s** is not a null pointer, **wcrtomb** determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by **wc**, and stores the multibyte character representation in the array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **wc** is a null wide character, a null byte is stored; the resulting state described is the initial conversion state.

wcrtomb returns the number of bytes stored in the array object. When **wc** is not a valid wide character, an encoding error occurs: **wcrtomb** stores the value of the macro **EILSEQ** in **errno** and returns `(size_t)(-1)`; the conversion state is unspecified.

wcrtomb_l

Synopsis

```
size_t wcrtomb_l(char *s,  
                 wchar_t wc,  
                 mbstate_t *ps,  
                 locale_t loc);
```

If **s** is a null pointer, **wcrtomb_l** function is equivalent to the call `wcrtomb_l(buf, L'\0', ps, loc)` where **buf** is an internal buffer.

If **s** is not a null pointer, **wcrtomb_l** determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by **wc**, and stores the multibyte character representation in the array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **wc** is a null wide character, a null byte is stored; the resulting state described is the initial conversion state.

wcrtomb_l returns the number of bytes stored in the array object. When **wc** is not a valid wide character, an encoding error occurs: **wcrtomb_l** stores the value of the macro **EILSEQ** in **errno** and returns `(size_t)(-1)`; the conversion state is unspecified.

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** wide 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,  
                wchar_t **ptr);
```

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 wide 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](#).

wctob

Synopsis

```
int wctob(wint_t c);
```

Description

wctob determines whether **c** corresponds to a member of the extended character set whose multi-byte character representation is a single byte when in the initial shift state in the current locale.

Description

this returns **EOF** if **c** does not correspond to a multi-byte character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character as an **unsigned char** converted to an **int**.

wctob_l

Synopsis

```
int wctob_l(wint_t c,  
            locale_t loc);
```

Description

wctob_l determines whether **c** corresponds to a member of the extended character set whose multi-byte character representation is a single byte when in the initial shift state in locale **loc**.

Description

wctob_l returns **EOF** if **c** does not correspond to a multi-byte character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character as an **unsigned char** converted to an **int**.

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.

<wctype.h>

API Summary

Classification functions	
iswalnum	Is character alphanumeric?
iswalpha	Is character alphabetic?
iswblank	Is character blank?
iswcntrl	Is character a control?
iswctype	Determine character type
iswdigit	Is character a decimal digit?
iswgraph	Is character a control?
iswlower	Is character a lowercase letter?
iswprint	Is character printable?
iswpunct	Is character punctuation?
iswspace	Is character a whitespace character?
iswupper	Is character an uppercase letter?
iswxdigit	Is character a hexadecimal digit?
wctype	Construct character class
Conversion functions	
towctrans	Translate character
tolower	Convert uppercase character to lowercase
toupper	Convert lowercase character to uppercase
wctrans	Construct character mapping
Classification functions (extended)	
iswalnum_l	Is character alphanumeric?
iswalpha_l	Is character alphabetic?
iswblank_l	Is character blank?
iswcntrl_l	Is character a control?
iswctype_l	Determine character type
iswdigit_l	Is character a decimal digit?
iswgraph_l	Is character a control?
iswlower_l	Is character a lowercase letter?
iswprint_l	Is character printable?
iswpunct_l	Is character punctuation?

iswspace_l	Is character a whitespace character?
iswupper_l	Is character an uppercase letter?
iswxdigit_l	Is character a hexadecimal digit?
Conversion functions (extended)	
towctrans_l	Translate character
towlower_l	Convert uppercase character to lowercase
towupper_l	Convert lowercase character to uppercase
wctrans_l	Construct character mapping

iswalnum

Synopsis

```
int iswalnum(wint_t c);
```

Description

iswalnum tests for any wide character for which **iswalpha** or **iswdigit** is true.

iswalnum_l

Synopsis

```
int iswalnum_l(wint_t c,  
               locale_t loc);
```

Description

iswalnum_l tests for any wide character for which **iswalpha_l** or **iswdigit_l** is true in the locale **loc**.

iswalpha

Synopsis

```
int iswalpha(wint_t c);
```

Description

iswalpha returns true if the wide character **c** is alphabetic. Any character for which **iswupper** or **iswlower** returns true is considered alphabetic in addition to any of the locale-specific set of alphabetic characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

In the C locale, **iswalpha** returns nonzero (true) if and only if **iswupper** or **iswlower** return true for the value of the argument **c**.

iswalpha_l

Synopsis

```
int iswalpha_l(wint_t c,  
               locale_t loc);
```

Description

iswalpha_l returns true if the wide character **c** is alphabetic in the locale **loc**. Any character for which **iswupper_l** or **iswlower_l** returns true is considered alphabetic in addition to any of the locale-specific set of alphabetic characters for which none of **iswcntrl_l**, **iswdigit_l**, **iswpunct_l**, or **iswspace_l** is true.

iswblank

Synopsis

```
int iswblank(wint_t c);
```

Description

iswblank tests for any wide character that is a standard blank wide character or is one of a locale-specific set of wide characters for which **iswspace** is true and that is used to separate words within a line of text. The standard blank wide are space and horizontal tab.

In the C locale, **iswblank** returns true only for the standard blank characters.

iswblank_l

Synopsis

```
int iswblank_l(wint_t c,  
               locale_t loc);
```

Description

iswblank_l tests for any wide character that is a standard blank wide character in the locale **loc** or is one of a locale-specific set of wide characters for which **iswspace_l** is true and that is used to separate words within a line of text. The standard blank wide are space and horizontal tab.

iswcntrl

Synopsis

```
int iswcntrl(wint_t c);
```

Description

iswcntrl tests for any wide character that is a control character.

iswcntrl_l

Synopsis

```
int iswcntrl_l(wint_t c,  
               locale_t loc);
```

Description

iswcntrl_l tests for any wide character that is a control character in the locale **loc**.

iswctype

Synopsis

```
int iswctype(wint_t c,  
             wctype_t t);
```

Description

iswctype determines whether the wide character **c** has the property described by **t** in the current locale.

iswctype_l

Synopsis

```
int iswctype_l(wint_t c,  
               wctype_t t,  
               locale_t loc);
```

Description

iswctype_l determines whether the wide character **c** has the property described by **t** in the locale **loc**.

iswdigit

Synopsis

```
int iswdigit(wint_t c);
```

Description

iswdigit tests for any wide character that corresponds to a decimal-digit character.

iswdigit_l

Synopsis

```
int iswdigit_l(wint_t c,  
               locale_t loc);
```

Description

iswdigit_l tests for any wide character that corresponds to a decimal-digit character in the locale **loc**.

iswgraph

Synopsis

```
int iswgraph(wint_t c);
```

Description

iswgraph tests for any wide character for which **iswprint** is true and **iswspace** is false.

iswgraph_l

Synopsis

```
int iswgraph_l(wint_t c,  
               locale_t loc);
```

Description

iswgraph_l tests for any wide character for which **iswprint** is true and **iswspace** is false in the locale **loc**.

iswlower

Synopsis

```
int iswlower(wint_t c);
```

Description

iswlower tests for any wide character that corresponds to a lowercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

iswlower_l

Synopsis

```
int iswlower_l(wint_t c,  
               locale_t loc);
```

Description

iswlower_l tests for any wide character that corresponds to a lowercase letter in the locale **loc** or is one of a locale-specific set of wide characters for which none of **iswcntrl_l**, **iswdigit_l**, **iswpunct_l**, or **iswspace_l** is true.

iswprint

Synopsis

```
int iswprint(wint_t c);
```

Description

iswprint returns nonzero (true) if and only if the value of the argument **c** is any printing character.

iswprint_l

Synopsis

```
int iswprint_l(wint_t c,  
               locale_t loc);
```

Description

iswprint_l returns nonzero (true) if and only if the value of the argument **c** is any printing character in the locale **loc**.

iswpunct

Synopsis

```
int iswpunct(wint_t c);
```

Description

iswpunct tests for any printing wide character that is one of a locale-specific set of punctuation wide characters for which neither **iswspace** nor **iswalnum** is true.

iswpunct_l

Synopsis

```
int iswpunct_l(wint_t c,  
               locale_t loc);
```

Description

iswpunct_l tests for any printing wide character that is one of a locale-specific set of punctuation wide characters in locale **loc** for which neither **iswspace_l** nor **iswalnum_l** is true.

iswspace

Synopsis

```
int iswspace(wint_t c);
```

Description

iswspace tests for any wide character that corresponds to a locale-specific set of white-space wide characters for which none of **iswalnum**, **iswgraph**, or **iswpunct** is true.

iswspace_l

Synopsis

```
int iswspace_l(wint_t c,  
               locale_t loc);
```

Description

iswspace_l tests for any wide character that corresponds to a locale-specific set of white-space wide characters in the locale **loc** for which none of **iswalnum**, **iswgraph_l**, or **iswpunct_l** is true.

iswupper

Synopsis

```
int iswupper(wint_t c);
```

Description

iswupper tests for any wide character that corresponds to an uppercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

iswupper_l

Synopsis

```
int iswupper_l(wint_t c,  
               locale_t loc);
```

Description

iswupper_l tests for any wide character that corresponds to an uppercase letter or is one of a locale-specific set of wide characters in the locale **loc** for which none of **iswcntrl_l**, **iswdigit_l**, **iswpunct_l**, or **iswspace_l** is true.

iswxdigit

Synopsis

```
int iswxdigit(wint_t c);
```

Description

iswxdigit tests for any wide character that corresponds to a hexadecimal digit.

iswxdigit_l

Synopsis

```
int iswxdigit_l(wint_t c,  
                locale_t loc);
```

Description

iswxdigit_l tests for any wide character that corresponds to a hexadecimal digit in the locale **loc**.

towctrans

Synopsis

```
wint_t towctrans(wint_t c,  
                 wctrans_t t);
```

Description

towctrans maps the wide character **c** using the mapping described by **t** in the current locale.

towctrans_l

Synopsis

```
wint_t towctrans_l(wint_t c,  
                  wctrans_t t,  
                  locale_t loc);
```

Description

towctrans_l maps the wide character **c** using the mapping described by **t** in the current locale.

towlower

Synopsis

```
wint_t tolower(wint_t c);
```

Description

towlower converts an uppercase letter to a corresponding lowercase letter.

If the argument **c** is a wide character for which **iswupper** is true and there are one or more corresponding wide characters, in the current locale, for which **iswlower** is true, **towlower** returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, **c** is returned unchanged.

towlower_l

Synopsis

```
wint_t towlower_l(wint_t c,  
                  locale_t loc);
```

Description

towlower_l converts an uppercase letter to a corresponding lowercase letter in locale **loc**.

If the argument **c** is a wide character for which **iswupper_l** is true and there are one or more corresponding wide characters, in the locale **loc**, for which **iswlower_l** is true, **towlower_l** returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, **c** is returned unchanged.

towupper

Synopsis

```
wint_t towupper(wint_t c);
```

Description

towupper converts a lowercase letter to a corresponding uppercase letter.

If the argument **c** is a wide character for which **iswlower** is true and there are one or more corresponding wide characters, in the current current locale, for which **iswupper** is true, **towupper** returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, **c** is returned unchanged.

towupper_l

Synopsis

```
wint_t towupper_l(wint_t c,  
                  locale_t loc);
```

Description

towupper_l converts a lowercase letter to a corresponding uppercase letter in locale **loc**.

If the argument **c** is a wide character for which **iswlower_l** is true and there are one or more corresponding wide characters, in the locale **loc**, for which **iswupper_l** is true, **towupper_l** returns one (and always the same one for any given locale) of the corresponding wide characters; otherwise, **c** is returned unchanged.

wctrans

Synopsis

```
wctrans_t wctrans(const char *property);
```

Description

wctrans constructs a value of type **wctrans_t** that describes a mapping between wide characters identified by the string argument **property**.

If **property** identifies a valid mapping of wide characters in the current locale, **wctrans** returns a nonzero value that is valid as the second argument to **towctrans**; otherwise, it returns zero.

Note

The only mappings supported are "tolower" and "toupper".

wctrans_l

Synopsis

```
wctrans_t wctrans_l(const char *property,  
                   locale_t loc);
```

Description

wctrans_l constructs a value of type **wctrans_t** that describes a mapping between wide characters identified by the string argument **property** in locale **loc**.

If **property** identifies a valid mapping of wide characters in the locale **loc**, **wctrans_l** returns a nonzero value that is valid as the second argument to **towctrans_l**; otherwise, it returns zero.

Note

The only mappings supported are "tolower" and "toupper".

wctype

Synopsis

```
wctype_t wctype(const char *property);
```

Description

wctype constructs a value of type **wctype_t** that describes a class of wide characters identified by the string argument **property**.

If **property** identifies a valid class of wide characters in the current locale, **wctype** returns a nonzero value that is valid as the second argument to **iswctype**; otherwise, it returns zero.

Note

The only mappings supported are "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper", and "xdigit".

<xlocale.h>

API Summary

Functions	
duplocale	Duplicate current locale data
freelocale	Free a locale
localeconv_l	Get locale data
newlocale	Create a new locale

duplocale

Synopsis

```
locale_t duplocale(locale_t loc);
```

Description

duplocale duplicates the locale object referenced by **loc**.

If there is insufficient memory to duplicate **loc**, **duplocale** returns **NULL** and sets **errno** to **ENOMEM** as required by POSIX.1-2008.

Duplicated locales must be freed with **freelocale**.

This is different behavior from the GNU glibc implementation which makes no mention of setting **errno** on failure.

Note

This extension is derived from BSD, POSIX.1, and glibc.

freelocale

Synopsis

```
int freelocale(locale_t loc);
```

Description

freelocale frees the storage associated with **loc**.

freelocale zero on success, 1 on error.

localeconv_l

Synopsis

```
localeconv_l(locale_t loc);
```

Description

localeconv_l returns a pointer to a structure of type **lconv** with the corresponding values for the locale **loc** filled in.

newlocale

Synopsis

```
locale_t newlocale(int category_mask,  
                  const char *locale,  
                  locale_t base);
```

Description

newlocale creates a new locale object or modifies an existing one. If the **base** argument is **NULL**, a new locale object is created.

category_mask specifies the locale categories to be set or modified. Values for **category_mask** are constructed by a bitwise-inclusive OR of the symbolic constants **LC_CTYPE_MASK**, **LC_NUMERIC_MASK**, **LC_TIME_MASK**, **LC_COLLATE_MASK**, **LC_MONETARY_MASK**, and **LC_MESSAGES_MASK**.

For each category with the corresponding bit set in **category_mask**, the data from the locale named by **locale** is used. In the case of modifying an existing locale object, the data from the locale named by **locale** replaces the existing data within the locale object. If a completely new locale object is created, the data for all sections not requested by **category_mask** are taken from the default locale.

The locales C and POSIX are equivalent and defined for all settings of **category_mask**:

If **locale** is **NULL**, then the C locale is used. If **locale** is an empty string, **newlocale** will use the default locale.

If **base** is **NULL**, the current locale is used. If **base** is **LC_GLOBAL_LOCALE**, the global locale is used.

If **mask** is **LC_ALL_MASK**, **base** is ignored.

Note

POSIX.1-2008 does not specify whether the locale object pointed to by **base** is modified or whether it is freed and a new locale object created.

Implementation

The category mask **LC_MESSAGES_MASK** is not implemented as POSIX messages are not implemented.



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 .
<locale>	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 .
<cstdint>	C++ wrapper on stdint.h .
<cstdio>	C++ wrapper on stdio.h .
<cstdlib>	C++ wrapper on stdlib.h .

<code><cstring></code>	C++ wrapper on string.h .
<code><ctime></code>	C++ wrapper on time.h .
<code><cwchar></code>	C++ wrapper on wchar.h .
<code><cwctype></code>	C++ wrapper on wctype.h .
<code><exception></code>	Definitions for exceptions.
<new>	Types and definitions for placement new and delete.
<code><typeinfo></code>	Definitions for RTTI.

Standard template library

The C++ STL functionality of STLPort is provided as a separate library package use **Tools > Package Manager** to install this package.

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 flashl_max_geometry_regions = 4;
    libmem_driver_handle_t flashl_handle;
    libmem_geometry_t flashl_geometry[flashl_max_geometry_regions];
    libmem_flash_info_t flashl_info;
    uint8_t *flashl_start = (uint8_t *)0x10000000;
    uint8_t *write_dest = flashl_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(&flashl_handle,
                                    flashl_start,
                                    flashl_geometry,
                                    flashl_max_geometry_regions,
                                    &flashl_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 **flashl_start**.

```
// Register the FLASH LIBMEM driver
res = libmem_register_cfi_driver(&flashl_handle,
                                flashl_start,
                                flashl_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 size)
{
    // 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, erase_size);
    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 crc)
{
    // TODO: Implement CRC-32 operation.
    return crc;
}

static const libmem_ext_driver_functions_t ext_driver_functions =
{
    libmem_inrange_impl,
    libmem_read_impl,
    libmem_crc32_impl
};

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_functions);
    return LIBMEM_STATUS_SUCCESS;
}

```

Some types of memory require you to carry out paged writes. The paged write driver helper functions have been provided to simplify the writing of drivers of this type.

To use these functions, you need to call **libmem_driver_paged_write_init** supplying a paged write control block, a page buffer, the page size, a pointer to a function that will carry out the actual page write operation and the byte alignment of the source data required by the page write function. You can then use the **libmem_driver_paged_write**, **libmem_driver_paged_write_fill** and **libmem_driver_paged_write_flush** functions to implement your driver's write, fill and flush functions.

For example, the following code demonstrates how you might implement a driver for a device with a page size of 256 bytes:

```
static uint8_t page_buffer[256];
static libmem_driver_paged_write_ctrlblk_t paged_write_ctrlblk;

static int
flash_write_page(libmem_driver_handle_t *h, uint8_t *dest, const uint8_t *src)
{
    // TODO: Implement function that writes a page of data from src to page
    //        starting at dest.
    return LIBMEM_STATUS_SUCCESS;
}

static int
libmem_write_impl(libmem_driver_handle_t *h, uint8_t *dest, const uint8_t *src, size_t size)
{
    return libmem_driver_paged_write(h, dest, src, size, &paged_write_ctrlblk);
}

static int
libmem_fill_impl(libmem_driver_handle_t *h, uint8_t *dest, uint8_t c, size_t size)
{
    return libmem_driver_paged_write_fill(h, dest, c, size, &paged_write_ctrlblk);
}

static int
libmem_flush_impl(libmem_driver_handle_t *h)
{
    return libmem_driver_paged_write_flush(h, &paged_write_ctrlblk);
}

int
libmem_register_example_driver_4(libmem_driver_handle_t *h, uint8_t *start, size_t size)
{
    libmem_register_driver(h, start, size, 0, 0, &driver_functions, 0);
    libmem_driver_paged_write_init(&paged_write_ctrlblk,
                                   page_buffer, sizeof(page_buffer),
                                   flash_write_page, 4,
                                   0);
    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 *flashl_start = (uint8_t *)0x10000000;
    const int flashl_max_geometry_regions = 4;
    libmem_driver_handle_t flashl_handle;
    libmem_geometry_t flashl_geometry[flashl_max_geometry_regions];
    libmem_flash_info_t flashl_info;
    int res;

    // Register FLASH driver.
    res = libmem_register_cfi_driver(&flashl_handle,
                                    flashl_start,
                                    flashl_geometry,
                                    flashl_max_geometry_regions,
                                    &flashl_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.

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
Driver helper macros	
LIBMEM_DRIVER_PAGED_WRITE_OPTION_DISABLE_D	Option to disable direct writes bypassing page buffer
LIBMEM_DRIVER_PAGED_WRITE_OPTION_DISABLE_P	Option to disable paged write data pre-loading
Data types	
_libmem_driver_functions_t	Structure containing pointers to a LIBMEM driver's functions
_libmem_driver_handle_t	LIBMEM driver handle structure
_libmem_driver_paged_write_ctrlblk_t	Paged write control block
_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_page_write_fn_t	A function pointer to a function implementing a paged write operation
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
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
Driver helper functions	
libmem_driver_paged_write	A driver helper function that implements a paged write operation
libmem_driver_paged_write_fill	A driver helper function that implements a paged write fill operation
libmem_driver_paged_write_flush	A driver helper function that implements a paged write flush operation
libmem_driver_paged_write_init	A driver helper function that initializes the paged write control block
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

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 (0x006)
```

Description

A definition representing the CFI command set number for the Winbond standard command set.

LIBMEM_DRIVER_PAGED_WRITE_OPTION_DISABLE_DIRECT_W

Synopsis

```
#define LIBMEM_DRIVER_PAGED_WRITE_OPTION_DISABLE_DIRECT_WRITES (1 << 1)
```

Description

LIBMEM_DRIVER_PAGED_WRITE_OPTION_DISABLE_DIRECT_WRITES disables direct writes bypassing page buffer.

This option can be passed to **libmem_driver_paged_write_init** to stop the **libmem_driver_paged_write** function carrying out direct writes bypassing the page buffer when it is possible to do so. This should be used if the source data for the write page function must be in RAM.

LIBMEM_DRIVER_PAGED_WRITE_OPTION_DISABLE_PAGE_PRELOAD

Synopsis

```
#define LIBMEM_DRIVER_PAGED_WRITE_OPTION_DISABLE_PAGE_PRELOAD (1 << 0)
```

Description

LIBMEM_DRIVER_PAGED_WRITE_OPTION_DISABLE_PAGE_PRELOAD disables paged write data pre-loading.

This option can be passed to **libmem_driver_paged_write_init** to disable pre-loads to the page buffer when switching to a new page. The pre-load is required if you want the driver to support arbitrary writes without corrupting existing data, however it may not be supported by all hardware.

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_RANG
```

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_ADDR
```

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_ADDRES
```

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 4
```

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_driver_paged_write_ctrlblk_t

Synopsis

```
typedef struct {
    uint8_t *page_buffer;
    size_t page_size;
    uint32_t page_mask;
    libmem_driver_page_write_fn_t page_write_fn;
    uint32_t page_write_src_alignment;
    uint32_t options;
    uint8_t *current_page;
} _libmem_driver_paged_write_ctrlblk_t;
```

Description

_libmem_driver_paged_write_ctrlblk_t is a structure describing the paged write helper functions control block.

`_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
<code>inrange</code>	A pointer to a LIBMEM driver's inrange function
<code>read</code>	A pointer to a LIBMEM driver's read function
<code>crc32</code>	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 *flashl_start = (uint8_t *)0x10000000;
libmem_flash_info_t flashl_info;
const int flashl_max_geometry_regions = 4;
libmem_geometry_t flashl_geometry[flashl_max_geometry_regions];
size_t flashl_size;
int res;

res = libmem_cfi_get_info(flashl_start,
                          &flashl_size,
                          flashl_geometry,
                          flashl_max_geometry_regions,
                          &flashl_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 *h, const uint8_t *start, size_t size, uint32_t crc);
```

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 *h, uint8_t *start, size_t size, uint8_t ** erase_start, size_t *erase_size);
```

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 *h, uint8_t *dest, uint8_t c, size_t size);
```

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 **c** 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 *h);
```

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 *h, const uint8_t *dest);
```

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 the 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 *h, uint8_t *start, size_t size);
```

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_page_write_fn_t

Synopsis

```
typedef int (*libmem_driver_page_write_fn_t)
(libmem_driver_handle_t *h, uint8_t *dest, const uint8_t *src);
```

Description

libmem_driver_page_write_fn_t is a function pointer to a function implementing a paged write operation.

h A pointer to the handle of the LIBMEM driver.

dest A pointer to the start address of the page to write to.

src A pointer to the address to copy the page data from.

libmem_driver_page_write_fn_t returns The LIBMEM status result. If any value other than LIBMEM_STATUS_SUCCESS is returned from this function the **libmem_driver_paged_write** or **libmem_driver_paged_write_fill** functions will terminate and return the response.

libmem_driver_paged_write

Synopsis

```
int libmem_driver_paged_write(libmem_driver_handle_t *h,
                             uint8_t *dest,
                             const uint8_t *src,
                             size_t size,
                             libmem_driver_paged_write_ctrlblk_t *paged_write_ctrlblk);
```

Description

libmem_driver_paged_write is a driver helper function that implements a paged write operation.

h A pointer to the handle of the 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.

paged_write_ctrlblk A pointer to the paged write control block.

libmem_driver_paged_write returns The LIBMEM status result.

libmem_driver_paged_write_fill

Synopsis

```
int libmem_driver_paged_write_fill(libmem_driver_handle_t *h,  
                                   uint8_t *dest,  
                                   uint8_t c,  
                                   size_t size,  
  
                                   libmem_driver_paged_write_ctrlblk_t *paged_write_ctrlblk);
```

Description

libmem_driver_paged_write_fill is a driver helper function that implements a paged write fill operation.

h A pointer to the handle of the LIBMEM driver.

dest A pointer to the address to write the block of data.

c The data value to fill the memory with.

size The number of bytes to write.

paged_write_ctrlblk A pointer to the paged write control block.

libmem_driver_paged_write_fill returns The LIBMEM status result.

libmem_driver_paged_write_flush

Synopsis

```
int libmem_driver_paged_write_flush(libmem_driver_handle_t *h,  
    libmem_driver_paged_write_ctrlblk_t *paged_write_ctrlblk);
```

Description

libmem_driver_paged_write_flush is a driver helper function that implements a paged write flush operation.

h A pointer to the handle of the LIBMEM driver.

paged_write_ctrlblk A pointer to the paged write control block.

libmem_driver_paged_write_flush returns The LIBMEM status result.

libmem_driver_paged_write_init

Synopsis

```
int libmem_driver_paged_write_init(libmem_driver_paged_write_ctrlblk_t *paged_write_ctrlblk,
                                   uint8_t *page_buffer,
                                   size_t page_size,
                                   libmem_driver_page_write_fn_t page_write_fn,
                                   uint32_t page_write_src_alignment,
                                   uint32_t options);
```

Description

libmem_driver_paged_write_init is a driver helper function that initializes the paged write control block.

paged_write_ctrlblk A pointer to the paged write control block.

page_buffer A pointer to the page buffer to use for paged write operations.

page_size The page size, this value must be equal to the size of the buffer pointed to by **page_buffer**.

page_write_fn A pointer to a function that carries out the page write operation.

page_write_src_alignment The byte alignment of source data required by the page write function when bypassing the page buffer. Set this to zero if the write function only supports writes directly from the page buffer.

options Paged write configuration options.

libmem_driver_paged_write_init returns The LIBMEM status result.

libmem_driver_read_fn_t

Synopsis

```
typedef int (*libmem_driver_read_fn_t)
(libmem_driver_handle_t *h, uint8_t *dest, const uint8_t *src, size_t size);
```

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 *h, uint8_t *start, size_t size);
```

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 *h, uint8_t *dest, const uint8_t *src, size_t size);
```

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 *h);
```

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 *h, libmem_sector_info_t *sector_info);
```

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 described 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 described 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 described 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 flashl_max_geometry_regions = 4;
libmem_driver_handle_t flashl_handle;
uint8_t *flashl_start = (uint8_t *)0x10000000;
libmem_geometry_t flashl_geometry[flashl_max_geometry_regions];
libmem_flash_info_t flashl_info;
size_t flashl_size;
int res;

// Detect the type, size and geometry of the Intel FLASH.
res = libmem_cfi_get_info(flashl_start,
                        &flashl_size,
                        flashl_geometry,
                        flashl_max_geometry_regions,
                        &flashl_info);

if (res == LIBMEM_STATUS_SUCCESS)
{
    // Register the driver
    res = libmem_register_cfi_amd_driver(&flashl_handle,
                                        flashl_start,
                                        flashl_size,
                                        flashl_geometry,
                                        &flashl_info);

    if (res == LIBMEM_STATUS_SUCCESS)
```

```
    printf("libmem_register_cfi_aml_driver : success\n");  
else  
    printf("libmem_register_cfi_aml_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 flashl_max_geometry_regions = 4;
libmem_driver_handle_t flashl_handle;
uint8_t *flashl_start = (uint8_t *)0x10000000;
libmem_geometry_t flashl_geometry[flashl_max_geometry_regions];
libmem_flash_info_t flashl_info;
size_t flashl_size;
int res;

// Detect the type, size and geometry of the Intel FLASH.
res = libmem_cfi_get_info(flashl_start,
                        &flashl_size,
                        flashl_geometry,
                        flashl_max_geometry_regions,
                        &flashl_info);

if (res == LIBMEM_STATUS_SUCCESS)
{
    // Register the driver
    res = libmem_register_cfi_intel_driver(&flashl_handle,
                                         flashl_start,
                                         flashl_size,
                                         flashl_geometry,
                                         &flashl_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 raml_handle;
uint8_t *raml_start = (uint8_t *)0x10000000;
const size_t raml_size = 1024;
int res;

res = libmem_register_ram_driver(&ram_handle, raml_start, raml_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_regions, &flash1_info);

if (res == LIBMEM_STATUS_SUCCESS)
{
    res = libmem_register_sst39xFx00A_16_driver(&flash1_handle, flash1_start, size, flash1_geometry, &flash1_info);

    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_LOADER_VERSION_NUMBER	LIBMEM loader interface version number.
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.
LIBMEM_RPC_LOADER_OPTION_HOST_ERASE	Enables host erase loader mode.
LIBMEM_RPC_LOADER_OPTION_HOST_WRITE	Enables host write loader mode.
Functions	
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).
libmem_rpc_loader_start_ex	Start up a LIBMEM loader that uses direct RPC (remote procedure calls) with additional options flags.

LIBMEM_LOADER_VERSION_NUMBER

Synopsis

```
#define LIBMEM_LOADER_VERSION_NUMBER 3
```

Description

The LIBMEM loader interface version number.

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 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_RPC_LOADER_OPTION_HOST_ERASE

Synopsis

```
#define LIBMEM_RPC_LOADER_OPTION_HOST_ERASE (1 << 30)
```

Description

LIBMEM loader option that enables host erase mode when used in the *options* parameter of `libmem_rpc_loader_start_ex`.

LIBMEM_RPC_LOADER_OPTION_HOST_WRITE

Synopsis

```
#define LIBMEM_RPC_LOADER_OPTION_HOST_WRITE (1 << 31)
```

Description

LIBMEM loader option that enables host write mode when used in the *options* parameter of `libmem_rpc_loader_start_ex`.

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 *flashl_start = (uint8_t *)0x10000000;  
    const int flashl_max_geometry_regions = 4;  
    libmem_driver_handle_t flashl_handle;  
    libmem_geometry_t flashl_geometry[flashl_max_geometry_regions];  
    libmem_flash_info_t flashl_info;  
    int res;  
    const char *error = 0;  
  
    if (initialise())  
    {  
        // Register FLASH driver.  
        res = libmem_register_cfi_driver(&flashl_handle,  
                                       flashl_start,  
                                       flashl_geometry,  
                                       flashl_max_geometry_regions,  
                                       &flashl_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.

This function starts up a LIBMEM loader that uses direct remote procedure calls of the LIBMEM library.

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 fewer RPC calls will be required.

Example:

```
static unsigned char buffer[256];

int main(void)
{
    uint8_t *flashl_start = (uint8_t *)0x10000000;
    const int flashl_max_geometry_regions = 4;
    libmem_driver_handle_t flashl_handle;
    libmem_geometry_t flashl_geometry[flashl_max_geometry_regions];
    libmem_flash_info_t flashl_info;
    int res;

    // Register FLASH driver.
    res = libmem_register_cfi_driver(&flashl_handle,
                                    flashl_start,
                                    flashl_geometry,
                                    flashl_max_geometry_regions,
                                    &flashl_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;
}
```

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 functionality is not required then this flag can be ignored.

Note that older versions of LIBMEM loader required that you always link certain LIBMEM functions such as **libmem_write()** and **libmem_erase()** into the loader using the "Linker | Keep Symbols" project property. This is now only required if you want the loader to be compatible with versions of CrossWorks prior to version 3.

libmem_rpc_loader_start_ex

Synopsis

```
int libmem_rpc_loader_start_ex(void *comm_buffer_start,
                              void *comm_buffer_end,
                              uint32_t options);
```

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.

options This parameter is usually 0, but can be used to enable different loader behaviour by specifying a combination of the following loader options:

LIBMEM_RPC_LOADER_OPTION_HOST_WRITE - This option flag enables host write loader mode. This mode is intended for devices with small amounts RAM. When enabled it guarantees that all write operations will be the size of, and aligned to, the communication buffer. This allows a lot of the complexity involved in carrying out a memory write operation to be avoided, thereby saving code size at the cost of download performance.

LIBMEM_RPC_LOADER_OPTION_HOST_ERASE - This option flag enables host erase loader mode. This mode is intended for devices with small amounts RAM. When enabled it guarantees that all erase operations will be the size of, and aligned to, the device's geometry. This allows a lot of the complexity involved in carrying out a memory erase operation to be avoided, thereby saving code size at the cost of download performance.

libmem_rpc_loader_start_ex returns The last error result returned from a LIBMEM function or **LIBMEM_STATUS_SUCCESS** if there has been no error.

This function is the same as **libmem_rpc_loader_start** except that it allows a loader option parameter to be specified.

Example:

```
static unsigned char buffer[256];

int main(void)
{
    uint8_t *flashl_start = (uint8_t *)0x10000000;
    const int flashl_max_geometry_regions = 4;
    libmem_driver_handle_t flashl_handle;
    libmem_geometry_t flashl_geometry[flashl_max_geometry_regions];
    libmem_flash_info_t flashl_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_ex(buffer,
                               buffer + sizeof(buffer) - 1,
                               LIBMEM_LOADER_OPTION_HOST_WRITE |
                               LIBMEM_LOADER_OPTION_HOST_ERASE);
}

libmem_rpc_loader_exit(res, NULL);

return 0;
}
```



Utilities Reference

Command-Line Compiler

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.

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. Alternatively you can specify that it is a C file using

```
cc -x c cfile.notc ...
```

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. Alternatively you can specify that it is a C++ file using

```
cc -x c++ cppfile.notcpp ...
```

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. Alternatively you can specify that it is an assembly language file using

```
cc -x asm asmfile.ots ...
```

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.

Compilation

To compile or assemble a file you should supply the **-c** option together with the source file and provide a name for the output file using the **-o** option

```
cc -c file.c -o file.o
```

if you don't supply an output name then the output file will be use the basename of the source file.

You can supply the file to be compiled from the standard input using the

```
cc -c - -o main.o
int main()
{
    return 3;
}
<EOF>
```

You can preprocess the source file rather than compile it using the **-E** option

```
cc -E main.c
```

This will send the output to the standard output or you can use **-o** to send it to a named file.

You can show the preprocessor defines that are defined for the compilation using the **-dM** option

```
cc -E -dM main.c
```

You can supply preprocessor defines and include directories using **-D** and **-I** options

```
cc -c file.c -Dmydefine -Imyincludedir
```

You can include a file before compilation using **-include**

```
cc -c file.c -include file.h
```

There is also a variant that will just use the **#defines** that are declared in the included file

```
cc -c file.c -imacros file.h
```

If you wish to not use the default C/C++ library you can use **-l-** and then supply your own system library directory using **-isystem**

```
cc -c file.c -l- -isystemmysystemincludedir
```

You can using the **-g** option to include debugging information in the output file

```
cc -c file.c -g
```

You can use the **-O** option to set the desired optimization level

```
cc -c file.c -O0
```

Linking

You can compile/link a number of files with the standard libraries

```
cc Cortex_M_Startup.s thumb_crt0.s main.o -o main.elf
```

You'll also need to supply linker control details. There are a number of ways of doing this

```
cc .. -placement ram_placement.xml -placementsegments "SRAM RW 0x0 0x1000" -erreset_handler
```

```
cc .. -memorymap map.xml -placementsegments "SRAM RW 0x0 0x1000" -erreset_handler
```

```
cc .. -Tlinker.icf -erreset_handler
```

Target Selection

You can specify the target cpu using **-cpu=** or **-mcpu=**

```
cc -c file.c -mcpu=cortex-m7
```

or the equivalent architecture using **-arch=** or **-march=**

```
cc -c file.c -march=armv7e-m
```

You can select the fpu using **-fpu=** or **-mfpu=**

```
cc -c file.c -march=armv7e-m -mfpu=fpv4-sp-d16
```

and the floating point abi to use

```
cc -c file.c -march=armv7e-m -mfpu=fpv4-sp-d16 -mfloat-abi=hard
```

Advanced

You can create a precompile header using the **-pch** option

```
cc -c -xc -pch main.h -o main.h.pch
```

Note that the output file must be in the same directory as the input file. You can use this precompiled header file

```
cc -c main.c -include-pch main.h.pch
```

You can create a C++ 20 module using

```
cc++ -c -xc++ main.cxx -std=c++20 -fmodules-ts -fmodule-file=main=main.o -o main.o
```

The module file will be named either **.gcm** or **.pcm** and can be used by another file using

```
cc++ -c -xc++ another.cxx -std=c++20 -fmodules-ts -fmodule-file=main=main.o -o another.o
```

Options:

Option	Description
-	input is taken from standard input
-###	show commands but don't execute them
-allow-multiple-definition	allow multiple symbol definition when linking
-ansi	enforce ANSI checking

-ar	create library from input files
-arch=val	set cpu architecture to 'val', use list to display supported
-arm	generate ARM code
-arm64	generate ARM64 code
-be	big endian target
-be8	big endian target
-builtins	use builtin compiler functions
-c	compile the files, no link/library
-clang	use clang compiler/assembler/ld
-cmselib=l	create cmse output library in 'l'
-codec=c	set file codec to 'c', use list to display supported
-common	allocate global variables in the common section
-cpu=val	set cpu core to 'val', use list to display supported
-debugio=bkpt	use breakpoint implementation for debugio
-debugio=dcc	use dcc implementation for debugio
-debugio=mempoll	use memory polling implementation for debugio
-depend file	generate dependency file in 'file'
-dependu file	generate dependency file in 'file' with user header files only
-dM	show #defines
-Dname	define the preprocessor macro 'name'
-Dname=val	define the preprocessor macro 'name' as 'val'
-dname=val	define the linker symbol 'name' as 'val'
-E	preprocess file and write to standard output
-emit-relocs	emit relocations into executable
-ename	set program entry symbol to 'name'
-exceptions	enable C++ exceptions
-Fbin	create an additional binary output file
-fbuiltin	enable compiler builtin functions
-fcommon	place global variables in COMMON section
-fcoroutines	enable C++ coroutine support
-fdebug-types-section	generate .debug_types section
-fdiagnostics-color=always	color diagnostic output of the compiler
-fdiagnostics-color=never	do not color diagnostic output of the compiler

-fdiagnostics-show-caret	show caret in diagnostic output of the compiler
-fexceptions	enable C++ exception support
-Fhex	create an additional hex output file
-fill=b	fill gaps in the additional output file with byte 'b'
-flto	generate code suitable for link time optimization
-fmath-errno	set errno after calling math functions
-fmodule-file='name'	get module dependencies from the file 'name'
-fmodules-ts	enable c++20 modules
-fno-builtin	disable compiler builtin functions
-fno-common	place global variables in bss section
-fno-diagnostics-show-caret	do not show caret in diagnostic output of the compiler
-fno-exceptions	disable C++ exception support
-fno-math-errno	set errno after calling math functions
-fno-omit-frame-pointer	disable framepointer generation
-fno-rtti	disable C++ RTTI support
-fno-short-enums	enumerations are int sized
-fno-short-wchar	wide characters are 32-bit
-fno-signed-char	char is considered to be unsigned char
-fomit-frame-pointer	disable framepointer generation
-fpabi=hard	generate FPU instructions passing fp arguments in FPU registers
-fpabi=soft	do not generate FPU instructions
-fpabi=softfp	generate FPU instructions passing fp arguments in CPU registers
-fpu=val	set fpu to 'val', use list to display supported
-framepointer	generate code to maintain a frame pointer register
-frtti	enable C++ RTTI support
-fshort-enums	enumerations are minimal container sized
-fshort-wchar	wide characters are 16-bit
-fsigned-char	char is considered to be signed char
-Fsrec	create an additional srec output file
-ftree-vectorize	perform vectorization on trees
-funwind-tables	generate unwind tables
-g1	generate only backtrace and line number debugging information

-g2	generate level 1 and variable display debugging information
-g3	generate level 2 and macro display debugging information
-gcc	use gcc assembler/compiler/lto
-gcc-target=name	select gcc 'name' tools to use
-gdwarf-2	generate dwarf-2 debugging information
-gdwarf-3	generate dwarf-3 debugging information
-gdwarf-4	generate dwarf-4 debugging information
-gdwarf-5	generate dwarf-5 debugging information
-gpubnames	generate .debug_pubnames and .debug_pubtypes sections
-hascmse	v8m architecture has cmse instructions
-hascrc	v8a architecture has crc instructions
-hascrypto	v8a architecture has crypto instructions
-hasdsp	v8m architecture has dsp instructions
-hasdiv	v7ar architecture has integer divide instructions
-hassmallmultiplier	cortex-m0/m0+/m1 architecture has small multiplier
-help	show this text
-I-	do not search any standard directories for include files
-Idir	add 'dir' to the end of the user include search list
-imacros file	same as -include but only keep #defines
-include file	#include 'file' before the source file
-include-pch file	#include precompiled header 'file' before the source file
-inputfiles file	list of files in 'file' to link or archive
-instrument	instrument functions
-isystem dir	add 'dir' to the end of the system include search list
-Jdir	add 'dir' to the end of the system include search list
-kasm	keep assembly code output
-kind	keep indirect files
-kldscript	keep generated linker script
-klto	keep lto generated files
-Kname	keep symbol 'name' in the linked output
-kpp	keep preprocessor output
-l-	disable linking of standard libraries

-Ldir	search directory 'dir' to find libraries
-le	little endian target
-libdir dir	specify system library directory 'dir'
-lname	search library 'name' to resolve symbols
-longcalls	generate long calling instruction sequences
-lunwind	generate stack unwind tables
-M	generate linkage map file
-march=val	set cpu architecture to 'val', use list to display supported
-marm	generate arm code
-mbe8	big endian target
-mbig-endian	big endian target
-mcmse	v8m architecture has cmse instructions
-mcpu=val	set cpu core to 'val', use list to display supported
-memorymap file	supply memory map file in 'file'
-memorymapmacros macros	define macros for memory map file in 'macros'
-mfloat-abi=val	specify the floating-point abi to use, val can be 'soft', 'softfp', 'hard'
-mfp16-format=ieee	specify the format of the __fp16 half-precision floating-point type
-mfpu=val	set fpu to 'val', use list to display supported
-mlittle-endian	little endian target
-mno-thumb-interwork	do not generate interworking code for v4t architecture
-mno-unaligned-access	disable unaligned word and half-word load/store instructions
-mthumb	generate thumb code, default is to generate ARM code for processors that support it
-mthumb-interwork	generate thumb interworking code for v4t architecture
-mtp=soft	specify the thread local storage model
-munaligned-access	enable unaligned word and half-word load/store instructions
-n	show commands but don't execute them
-nodefaultlibs	disable linking of standard libraries
-nointerwork	do not generate interworking code for v4t architecture
-noshortenums	enumerations are int sized
-noshortwchar	wide characters are 32-bit
-nostderr	redirect output from stderr to stdout

<code>-nostdinc</code>	do not search any standard directories for include files
<code>-nostdlib</code>	disable linking of standard libraries
<code>-nowarn-enumsize</code>	no linker warning on mismatched enum sized input files
<code>-nowarn-mismatch</code>	no linker warning on mismatched architecture input files
<code>-nowarn-rwx-segments</code>	no linker warning on load segments with RWX permissions
<code>-nowarn-wcharsize</code>	no linker warning on mismatched wchar sized input files
<code>-o file</code>	leave output in 'file'
<code>-O0</code>	set optimization level to level 0
<code>-O1</code>	set optimization level to level 1
<code>-O2</code>	set optimization level to level 2
<code>-O3</code>	set optimization level to level 3
<code>-Og</code>	set optimization level to debug
<code>-Os</code>	set optimization level to optimize for size
<code>-patch cmd</code>	run 'cmd' after link but before the creation of the additional output file
<code>-pch</code>	generate a precompiled header file
<code>-pedantic</code>	warning on non-standard language usage
<code>-pedantic-errors</code>	error on non-standard language usage
<code>-placement file</code>	supply placement file in 'file'
<code>-placementmacros macros</code>	define macros for placement file in 'macros'
<code>-placementsegments segments</code>	memory segments for placement in 'segments'
<code>-printf=d[ll][w]</code>	double, optional long long, optional wchar
<code>-printf=f[ll][w]</code>	float, optional long long, optional wchar
<code>-printf=i[p][w]</code>	integer, optional width and precision, optional wchar
<code>-printf=ll[p][w]</code>	long long integer, optional width and precision, optional wchar
<code>-Rc, name</code>	name the default code section to 'name'
<code>-Rd, name</code>	name the default data section to 'name'
<code>-Rk, name</code>	name the default const section to 'name'
<code>-rtti</code>	enable C++ rtti
<code>-Rz, name</code>	name the default bss section to 'name'
<code>-scanf=d[ll][c]</code>	double, optional long long, optional %[...] and %[^...] character class

<code>-scanf=ll[c]</code>	long long integer, optional %[...] and %[^...] character class
<code>-shortenums</code>	enumerations are minimal container sized
<code>-shortwchar</code>	wide characters are 16-bit
<code>-simd=neon</code>	generate simd vector processing code
<code>-stack-sizes</code>	generate stack-sizes section
<code>-std=s</code>	set language standard to 's', use list to display supported
<code>-stripdebug</code>	strip debug information from linked executable
<code>-stripsymbols</code>	strip symbols from linked executable
<code>-symbols=s</code>	link symbols file 's' into executable
<code>-Tfile</code>	use 'file' as linker script
<code>-thumb</code>	generate thumb code, default is to generate ARM code for processors that support it
<code>-unwindtables</code>	generate stack unwind tables
<code>-v</code>	show command lines as they are executed
<code>-vectorize</code>	enable auto vectorization code generation
<code>-W</code>	supply option to the compiler
<code>-w</code>	suppress warnings
<code>-Wa,x</code>	pass 'x' to the assembler
<code>-Wc,x</code>	pass 'x' to the compiler
<code>-we</code>	treat warnings as errors
<code>-Werror</code>	treat warnings as errors
<code>-Wl,x</code>	pass 'x' to the linker
<code>-xt</code>	subsequent files are considered to be of file type 't'
<code>-xa</code>	subsequent files are considered to be library files
<code>-xasm</code>	subsequent files are considered to be assembly code
<code>-xassembler-with-cpp</code>	subsequent files are considered to be assembly code
<code>-xc</code>	subsequent files are considered to be C code
<code>-xc++</code>	subsequent files are considered to be C++ code
<code>-Xlinker x</code>	pass 'x' to the linker
<code>-xo</code>	subsequent files are considered to be object code

Command-Line Project Builder

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
```

Options:

Option	Description
-batch	batch build
-clean	remove all output and intermediate files
-config 'name'	(batch) build in 'name' configuration
-D 'name'='value'	set global macro \$('name') to 'value'
-echo	show command lines as they are executed
-export-build	show the command lines and dependencies
-file 'name'	add file 'name' to template-created project
-help	show help text
-keepgoing	keep building when errors occur
-nostderr	redirect output from stderr to stdout
-nounity	disable unity building
-packagesdir 'name'	set \$(PackagesDir) to 'name'
-project 'name'	(batch) build 'name' project
-property 'name'='value'	set project property 'name' to 'value'
-rebuild	always do the build steps
-sconfig 'name'	solution properties are set in 'name' configuration
-show	show the build steps but don't execute them

-solution 'name'	(batch) build 'name' solution
-sproperty 'name'='value'	set solution property 'name' to 'value'
-studiodir 'name'	set \$(StudioDir) to 'name'
-template 'name'	create the project from the template 'name'
-templatesfile 'path'	get project template files from 'path'
-threadnum 'n'	use 'n' threads for build
-time	show time taken
-type 'type'	create the project from the project 'type'
-verbose	show build information

Command-Line Simulator

CrossSim is a program that allows you to run CrossStudio's instruction set simulator from the command line.

The primary purpose of **CrossSim** is to enable command line tests to be run. The debug I/O provided by CrossStudio is supported, as are command line arguments and exit.

CrossSim will accept a single elf file, it will allocate and load memory regions based on the program sections in the elf file. CrossSim will start execution from the entry point symbol contained in the elf file. CrossSim will terminate when exit is called or execution reaches a specified symbol.

Example

Assuming that app.c contains the following

```
#include <stdio.h>
#include <stdlib.h>
int
main(int argc, const char *argv[])
{
    int i;
    for (i = 1; i < argc; i++)
        printf("argv[%d]=%s\n", i, argv[i]);
    exit(EXIT_SUCCESS);
}
```

and app.elf has been built with the preprocessor definition FULL_LIBRARY then

```
crosssim app.elf hello world
```

will produce the output

```
argv[1]=hello
argv[2]=world
```

if the debug I/O implementation has to set breakpoints or poll memory locations then you can supply the name of the symbol to breakpoint on that will enable the debug I/O

```
crosssim app.elf -startup __startup_complete hello world
```

if the application uses memory that isn't allocated in the elf file then you can supply the memory segments that the simulator should create

```
crosssim app.elf -segments 0x08000000;0x10000;0x20000000;0x10000
```

and there is an alternative form of this

```
crosssim app.elf -memory-segments "FLASH RX 0x08000000 0x10000;SRAM RWX 0x20000000 0x10000"
```

The simulator attempts to determine the machine architecture from data in the elf file this can be override using

```
crosssim app.elf -arch v7m
```


If the simulator doesn't support the architecture then the the list of supported architectures will be displayed.

The simulator will start executing at the entry point symbol in the elf file. If the application doesn't set the stack pointer then you can supply this

```
crosssim app.elf -stackpointer __stack_end__
```

If the application doesn't call exit then you can supply a symbol to breakpoint on that will terminate the simulation

```
crosssim app.elf -end _Exit
```

You can show an instruction trace

```
crosssim app.elf -trace
```

Which will show the addresses of instructions, the instruction opcode and the disassembly

```
0x000002d8 E59F003C    ldr r0, =0xE01FC000
0x000002dc E3A01000    mov r1, #0
0x000002e0 E5801000    str r1, [r0]
0x000002e4 E3A01003    mov r1, #3
0x000002e8 E5801004    str r1, [r0, #4]
0x000002ec E3A01002    mov r1, #2
```

You can display the instruction execution counts at the end of the simulation

```
crosssim app.elf -counts
```

Which will show the addresses of instructions, with the number of times executed

```
0x08000298=1
0x0800029a=1
...
0x080008ac=278
0x080008ae=2
...
```

Usage:

`crosssim file [options] args`

Option	Description
<code>-arch 'a'</code>	Specify architecture to simulate
<code>-arch list</code>	List supported architectures
<code>-counts</code>	Show execution counts when the simulation ends
<code>-end 's'</code>	Specify the symbol to end simulation
<code>-max 'c'</code>	Specify the maximum number of instructions to simulate
<code>-memory-segments 'name [access] start size;'</code>	Specify the list of memory segments

-segments 'start;size;'	Specify the list of memory segments
-stackpointer 's'	Specify the starting stackpoint symbol
-startup 's'	Specify the startup completion point symbol
-trace	Show instruction execution

Command-Line Project Download and Debug

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
programs is permitted.
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"="Lpt2"
```

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)")
[0xeafffffe, 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 *framenumbers* which can be supplied to the **Debug.locate(framenumbers)** method. Framenumbers start at zero and are incremented for each function call. So framenumbers zero is the current location, framenumbers 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.

Support packages

The preceding examples assume that the support packages required to carry out the download or debugging have already been installed using CrossStudio's package manager. On some systems however, it is not possible or desirable to use CrossStudio to do this. This section describes how to manually install packages from the command line and specify where CrossLoad should look for them.

The first thing to do is decide on the directory path to store the installed packages, we're going to use an environment variable **PACKAGES_DIR** to specify this. For example:

```
export PACKAGES_DIR=/my_crossload_packages
echo $PACKAGES_DIR
```

Please note, Windows command prompt users should use **set** instead of **export** and **%PACKAGES_DIR%** instead of **\$PACKAGES_DIR**.

Next, we need to get hold of the package .hzb or .hbr files to be installed. These can be downloaded from our [package website](#) or [package archive](#).

Once we have got the package files, the **mkpkg** tool can be used to install the packages. For example:

```
mkpkg -x CMSIS_3.hzb $PACKAGES_DIR
mkpkg -x LPC1000.hzb $PACKAGES_DIR
```

By default, CrossLoad will look for packages in CrossStudio's packages directory. We can override this so that our local package installation is used instead with CrossLoad's **-packagesdir** option. For example:

```
crossload -packagesdir $PACKAGES_DIR -target usb -solution MySolution.hzb -project
MyProject -config Debug
```

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.

-connection (Specify connection)

Syntax

-connection *name*

Description

Specify the connection 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.

-eraseall (Erase all flash memory)

Syntax

-eraseall

Description

Erase all flash memory rather than just the flash memory to be programmed.

-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.

-listprojectprops (Display all project properties)

Syntax

-listprojectprops

Description

List all properties of selected project and configuration

-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.

-loader (Specify loader configuration)

Syntax

-loader *config*

Description

Select the loader configuration to use for the download.

-nodifferential (Inhibit differential download)

Syntax

-nodifferential

Description

Do not use differential downloading.

-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.

-reset (Reset only)

Syntax

-reset

Description

Reset the target and don't carry out download.

-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.

Command-Line Scripting

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 CrossScript.

-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:

[BinaryFile](#)

[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);
}

// Executed when script loaded.
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"`.

Embed

Embed is a program that converts a binary file into a C/C++ array definition.

The primary purpose of the **Embed** tool is to provide a simple method of embedding files into an application. This may be useful if you want to include firmware images, bitmaps, etc. in your application without having to read them first from an external source.

Syntax

embed *variable_name* *input_file* *output_file*

variable_name is the name of the C/C++ array to be initialised with the binary data.

input_file is the path to the binary input file.

output_file is the path to the C/C++ source file to generate.

Example

To convert a binary file *image.bin* to a C/C++ file called *image.h*:

```
embed img image.bin image.h
```

This will generate the following output in *image.h*:

```
static const unsigned char img[] = {  
    0x5B, 0x95, 0xA4, 0x56, 0x16, 0x5F, 0x2D, 0x47,  
    0xC5, 0x04, 0xD4, 0x8D, 0x73, 0x40, 0x31, 0x66,  
    0x3E, 0x81, 0x90, 0x39, 0xA3, 0x8E, 0x22, 0x37,  
    0x3C, 0x63, 0xC8, 0x30, 0x90, 0x0C, 0x54, 0xA4,  
    0xA2, 0x74, 0xC2, 0x8C, 0x1D, 0x56, 0x57, 0x05,  
    0x45, 0xCE, 0x3B, 0x92, 0xAD, 0x0B, 0x2C, 0x39,  
    0x92, 0x59, 0xB9, 0x9D, 0x01, 0x30, 0x59, 0x9F,  
    0xC5, 0xEA, 0xCE, 0x35, 0xF6, 0x4B, 0x05, 0xBF  
};
```

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 generate 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 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>
  ...
</RegisterGroup>
```

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_SMR0_SRCTYPE_BIT);
```

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.

Linker script file generator

The command line program **mkld** generates a GNU ld linker script from a CrossWorks memory map or section placement file.

Syntax

```
mkld -memory-map-file inputfile outputfile [options]
```

```
mkld -memory-map-segments segments outputfile [options]
```

Description

inputfile is the name of the CrossWorks memory map file to generate the ld script from.

segments is a list of memory segments of the form *SegmentName* *RWX* *Address* *Size*

outputfile is the the name of the ld script file to write.

Command-line options

This section describes the command-line options accepted by *mkld*.

-check-section-overflow

Syntax

-check-section-overflow

Description

Add checks for memory section overflow to the linker script.

-check-segment-overflow

Syntax

-check-segment-overflow

Description

Add checks for memory segment overflow to the linker script.

-disable-missing-runin-error

Syntax

-disable-missing-runin-error

Description

Discard any sections with a missing run in section.

-memory-map-macros

Syntax

-memory-map-macros *macro=value* [*macro=value*]

Description

Define CrossWorks macros to use when reading a memory map file.

-no-check-unplaced-sections

Syntax

-no-check-unplaced-sections

Description

Removes checks for unplaced memory sections from the linker script.

-no-ctors

Syntax

-no-ctors

Description

Ignore the .ctors section.

-no-dtors

Syntax

-no-ctors

Description

Ignore the .dtors section.

-section-placement-file

Syntax

-section-placement-file *filename*

Description

Generate a GNU ld linker script from the CrossWorks section placement file *filename*. If this option is used, a memory map file should also be specified with the *-memory-map-file* option.

-section-placement-macros

Syntax

-section-placement-macros *macro=value*;*macro=value*

Description

Define CrossWorks macros to use when reading a section placement file.

-symbols

Syntax

-symbols *symbol=value*;*symbol=value*]

Description

Add extra symbol definitions to the ld linker script.

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 exclude specific files or directories from being added to a package using the **-exclude** option:

```
mkpkg -c packages/FX150.hzq targets/FX150 -exclude targets/FX150/excluded_file.txt -exclude targets/FX150/excluded_directory 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
<code>-c</code>	Create a new package.
<code>-compress level</code>	Change compression level (0 for none, 9 for maximum).
<code>-d</code>	Remove files from a package.
<code>-exclude path</code>	Exclude path when adding files to a package.
<code>-f</code>	Output files to stdout.
<code>-overwrite</code>	Overwrite existing files.
<code>-no-date</code>	Do not add date attribute to package.
<code>-r</code>	Replace files in a package.
<code>-readonly</code>	Force all files to have read only attribute.
<code>-set-attr attribute=value</code>	Set package attribute to value.
<code>-sub-arch-endian</code>	Create architecture and endian specific sub packages.
<code>-sub-arch-endian-compatiblity</code>	Create architecture and endian specific sub packages including compatibility packages for versions of the IDE that don't have <code>\$(LibEndian)</code> macro.
<code>-sub-base-type</code>	Specify the type description of the base package.
<code>-sub-type</code>	Specify the type description of the sub packages.
<code>-t</code>	List the contents of a package.
<code>-v</code>	Be chatty.
<code>-V</code>	Show version information.
<code>-x</code>	Extract files from a package.

Package manager

The **pkg** program can be used to download, install, remove and search for packages from the command line.

Usage	Description
<code>pkg history <i>package_names...</i></code>	List version history of packages
<code>pkg install <i>package_names...</i></code>	Download and install packages
<code>pkg install -manual <i>package_files...</i></code>	Manually install package files
<code>pkg list</code>	List all available packages
<code>pkg list -installed</code>	List installed packages
<code>pkg list -installed-names</code>	List installed package names
<code>pkg list -dependencies <i>package_names...</i></code>	List package dependencies
<code>pkg list -dependents <i>package_names...</i></code>	List dependent packages
<code>pkg remove <i>package_names...</i></code>	Remove packages
<code>pkg remove -all</code>	Remove all packages
<code>pkg search <i>keywords...</i></code>	Search for packages
<code>pkg update</code>	Update list of available packages
<code>pkg upgrade</code>	Upgrade all installed packages
<code>pkg upgrade <i>package_names...</i></code>	Upgrade selected packages
Option	Description
<code>-D <i>macro</i>=value</code>	Set a global macro
<code>-keepgoing</code>	Continue when errors occur
<code>-legacy</code>	Include legacy packages
<code>-nodelete</code>	Don't delete downloaded packages after installation
<code>-noverify</code>	Don't verify downloaded packages
<code>-outputformat <i>string</i></code>	Specify list/search output format string
<code>-packagesdir <i>directory</i></code>	Set the packages directory to be <i>directory</i>
<code>-packagesurl <i>url</i></code>	Set the URL of the packages website to be <i>url</i>
<code>-quiet</code>	Do not output any progress messages
<code>-rootuserdir <i>directory</i></code>	Set the root user data directory to <i>directory</i>
<code>-verbose</code>	Produce verbose output
<code>-yes</code>	Answer yes to all questions without prompting
Macro	Description
<code>\$(Description)</code>	Package description
<code>\$(Name)</code>	Package name
<code>\$(Title)</code>	Package title

<code>\$(Version)</code>	Package version
--------------------------	-----------------

Before you can download, install or search for packages you must first update the local list of available packages:

```
$ pkg update
```

The **search** command can be used to search for a specific package:

```
$ pkg search libcxx
libcxx_arm - ARM libcxx Library Package (1.1)
```

The **install** command can be used to install a package:

```
$ pkg install libcxx_arm
```

The **list** command can be used to list installed packages:

```
$ pkg list -installed
libcxx_arm - ARM libcxx Library Package (1.1)
```

The **history** command can be used to show package history:

```
$ pkg history libcxx_arm
libcxx_arm - libcxx Library Package [ARM]

  1.1 (Installed)
    - Fixed name of Type Interpretation File.

  1.0
    - Initial release.
```

Specific versions of a package can be installed:

```
$ pkg install libcxx_arm:1.0
```

The **upgrade** command can be used to upgrade to the latest version of a package:

```
$ pkg upgrade libcxx_arm
```

The **remove** command can be used to uninstall a package:

```
$ pkg remove libcxx_arm
```




Appendices

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.

External Tools file format

Describes the format of external tool configuration files you use to extend CrossStudio.

Debugger Type Interpretation file format

Describes the format of the debugger type interpretation file.

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.

end_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, 031.

size:The total number of bits, 132.

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:

address_symbol: A symbolic name for the start address of the section.

alignment: The required alignment of the program section; a decimal number specifying the byte alignment.

end_symbol: A symbolic name for the end address of the section.

fill: The optional value used to fill unspecified regions of memory, a hexadecimal number with a 0x prefix.

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`, `.ctors`, `.ctors`, `.data`, `.rodata`, or `.bss`, then the equivalent input section of `*(.name.name.*)` is supplied to the linker.

keep: If **Yes**, the section will be kept even if none of the symbols are referenced by the rest of the program.

load: If **Yes**, the section is loaded. If **No**, the section isn't loaded.

place_from_segment_end: If **Yes**, this section and following sections will be placed at the end of the segment. Please note that this will only succeed if the section and all following sections have a fixed size specified with the `size` attribute.

runin: This specifies the name of the section to copy this section to. Multiple sections can be specified separated by a semicolon, the first section that exists will be used.

runoffset: This specifies an offset from the load address that the section will be run from.

size: The optional size of the program section in bytes, a hexadecimal number with a 0x prefix. The macro `$(SEGMENT_SIZE_REMAINING)` can be used for size calculations based on the remaining number of bytes in the segment.

size_symbol: A symbolic name for the size of the section.

start: The optional start address of the program section, 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 `.hzip` 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 be 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 `$(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.hzip" />
```

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 Evaluator7T." type="Executable"/>
```

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_ResetWithRamAtZero()" />
```

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 `source` attribute:

```
<file name="main.c" source="$(StudioDir)/samples/Shared/main.c" />
<file name="$(StudioDir)/source/thumb_crt0.s" />
```

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>
  <grouphdots

  <grouphdots
</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="Select a set of target options" name="Target" default="STR912FW44" />
```

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_versions="8:1.6-" author="Rowley Associates Ltd" >
  <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
and 3.</description>
      <description>Modified loader memory map so that .libmem sections will be placed
correctly.</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_manufacturer="Rowley Associates" board_name="CrossFire LPC2138"
  dependencies="Philips_LPC2000" version="1.0">
  <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 <i>(if omitted, CPU manufacturer will be used)</i> .
board_name	The name of the specific board supported by the package <i>(only required for board-support packages)</i> .
company_name	The name of the company to group the package under in the package dialogs. <i>(if omitted, the Board/CPU manufacturer will be used)</i> .
cpu_family	The family name of the CPU supported by the package <i>(optional)</i> .
cpu_manufacturer	The manufacturer of the CPU supported by the package.
cpu_name	The name of the specific CPU supported by the package <i>(may be omitted if the CPU family is specified)</i> .
crossstudio_versions	A string describing which version of CrossStudio supports the package <i>(optional)</i> . The format of the string is <i>target_id_number:version_range_string</i> .
description	A description of the package <i>(optional)</i> .
dependencies	A semicolon-separated list of packages the package requires to be installed in order to work <i>(optional)</i> .
installation_directory	The directory in which the package should be installed <i>(optional - if undefined, defaults to "\$(PackagesDir)")</i> .

<code>replaces</code>	A semicolon-separated list of package names listing the packages that this package replaces. The replaced packages are marked as legacy packages and are only visible in the package manager if the if the <i>Include Legacy Packages</i> option is selected (<i>optional</i>).
<code>deprecated</code>	If set to true, indicates that the package has been deprecated. Deprecated packages are marked as legacy packages and are only visible in the package manager if the if the <i>Include Legacy Packages</i> option is selected (<i>optional</i>).
<code>title</code>	A short description of the package (<i>optional</i>).
<code>uninstalls</code>	A semicolon-separated list of packages names listing the packages to be uninstalled if this package is installed (<i>optional</i>).
<code>version</code>	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
<code>file_name</code>	The file path of the file.
<code>title</code>	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
<code>title</code>	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.

External Tools file format

CrossStudio external-tool configuration files are structured using XML syntax for its simple construction and parsing.

Tool configuration files

The CrossStudio application will read the tool configuration file when it starts up. By default, CrossStudio will read the file `$(StudioUserDir)/tools.xml`.

Structure

All tools are wrapped in a **tools** element:

```
<tools>  
  
</tools>
```

Inside the tools element are **item** elements that define each tool:

```
<tools>  
  <item name="logical name">  
  
  </item>  
</tools>
```

The **item** element requires an **name** attribute, which is an internal name for the tool, and has an optional *wait* element. When CrossStudio invokes the tool on a file or project, it uses the *wait* element to determine whether it should wait for the external tool to complete before continuing. If the *wait* attribute is not provided or is set to *yes*, CrossStudio will wait for external tool to complete.

The way that the tool is presented in CrossStudio is configured by elements inside the **menu** element.

menu

The **menu** element defines the wording used inside menus. You can place a shortcut to the menu using an ampersand, which must be escaped using **&** in XML, before the shortcut letter. For instance:

```
<menu>&amp;PC-lint (Unit Check)</menu>
```

text

The optional **text** element defines the wording used in contexts other than menus, for instance when the tool appears as a tool button with a label. If **text** is not provided, the tool's textual appearance outside the menu is taken from the **menu** element (and is presented without an shortcut underline). For instance:

```
<text>PC-lint (Unit Check)</text>
```

tip

The optional **tip** element defines the status tip, shown on the status line, when moving over the tool inside CrossStudio:

```
<tip>Run a PC-lint unit checkout on the selected file or folder</tip>
```

key

The optional **key** element defines the accelerator key, or key chord, to use to invoke the tool using the keyboard. You can construct the key sequence using modifiers **Ctrl**, **Shift**, and **Alt**, and can specify more than one key in a sequence (note: Windows and Linux only; OS X does not provide key chords). For instance:

```
<key>Ctrl+L, Ctrl+I</key>
```

message

The optional **message** element defines the text shown in the tool log in CrossStudio when running the tool. For example:

```
<message>Linting</message>
```

match

The optional **match** element defines which documents the tool will operator on. The match is performed using the file extension of the document. If the file extension of the document matches one of the wildcards provided, the tool will run on that document. If there is no **match** element, the tool will run on all documents. For instance:

```
<match>*.c;*.cpp</match>
```

output

The optional **output** element defines the name of the output file created by the tool. If this element is specified the the output file will be opened in the editor when the tool has finished execution. The macros **\$(InputPath)** and **\$(InputBaseName)** can be used to name the output file. For Instance:

```
<output>$(InputBaseName).txt</output>
```

commands

The **commands** element defines the command line to run to invoke the tool. The command line is expanded using macros applicable to the file derived from the current build configuration and the project settings. Most importantly, the standard **\$(InputPath)** macro expands to a full pathname for the target file.

Additional macros constructed by CrossStudio are:

\$(DEFINES) is the set of **-D** options applicable to the current file, derived from the current configuration and project settings.

\$(INCLUDES) is the set of **-I** options applicable to the current file, derived from the current configuration and project settings.

For instance:

```
<commands>
    &quot;$(LINTDIR)/lint-nt&quot; -i$(LINTDIR)/lnt &quot;$(LINTDIR)/lnt/co-gcc.lnt&quot;
    $(DEFINES) $(INCLUDES) -D__GNUC__ -u -b +macros -w2 -e537 +fie +ffn -width(0,4) -hF1
    &quot;-format=%f:%l:%C:s%t:s%m&quot; &quot;$(InputPath)&quot;
</commands>
```

In this example we intend **\$(LINTDIR)** to point to the directory where PC-lint is installed and for **\$(LINTDIR)** to be defined as a CrossStudio global macro. You can set global macros using **Project > Macros...** or **Tools > Options > Building > Global Macros**.

Note that additional **"** entities are placed around pathnames in the **commands** section this is to ensure that paths that contain spaces are correctly interpreted when the command is executed by CrossStudio.

Debugger Type Interpretation file format

CrossStudio debugger type interpretation files are used by the debugger to provide list and string displays of C++ template container types. The files are structured using XML syntax for its simple construction and parsing.

Consider the following C++ template type

```
template <class _Type> class VeryBasicArray
{
private:
    size_t m_Count;
    _Type *m_pData;
public:
    VeryBasicArray(size_t count)
        : m_Count(count)
        , m_pData(new _Type[count])
    {
    }
}

VeryBasicArray<int> basicArray(5);
```

To display a variable of this type as a list the type interpretation file contains the following entry

```
<List Name="VeryBasicArray<int>"
      Head="((($ (T)*)HEAD).m_pData"
      Data="*((($ (T0)*)CURRENT)"
      Length="((($ (T)*)HEAD).m_Count"
      Next="CURRENT+sizeof($ (T0))"/>
```

The **Name** attribute is used to match the template type name note that the **<** and **>** xml entities are used to match the template argument.

When an entry has been matched the head of the list is located by evaluating the debugger expression in the **Head** attribute. The debugger expressions can contain macros that refer to the matched template type and will use the symbols **HEAD** and **CURRENT**.

The macro **\$(T)** refers to the instantiated template type, for the above example **\$(T)=VeryBasicArray<int>**.

The template arguments are referred to using macros **\$(T0)**, for the above example **\$(T0)=int**.

The symbol **HEAD** is the address of the variable being displayed, for the above example if the variable **basicArray** is allocated at address **0x20004000** then the **Head** expression

```
((VeryBasicArray<int>*)0x20004000).m_pData
```

will be evaluated by the debugger, note that the **.** operator and the **->** operator are equivalent in debugger expressions.

To display an element the debugger will evaluate the **Data** expression. This expression contains the symbol **CURRENT** which is the address of the element to display, for the above example the first element is at the address **basicArray.m_pData** which is allocated at address **0x20008000** then the **Data** expression

```
(*(int*)0x20008000)
```

will be evaluated by the debugger.

To increment the **CURRENT** symbol the **Next** expression

```
0x20008000+sizeof(int)
```

will be evaluated by the debugger.

Before the **CURRENT** symbol is incremented the debugger needs to check if it is at the end of list. This can be done either as a **Condition** expression or as a **Length** expression

```
((VeryBasicArray<int>*)0x20004000).m_Count
```

The **String** display is simpler than the **List** display since the characters are contiguous and optionally null terminated. The **Data** and **Length** expressions are supported, for example

```
<String Name="string"
  Data="*(($(T) *)HEAD)._M_start_of_storage._M_data"
  Length="(($(T) *)HEAD)._M_finish-(($(T) *)HEAD)._M_start_of_storage._M_data"/>
```

is used to display STLPort std::string types.

Building Environment Options

Build

Property	Description
Automatically Build Before Debug Environment/Build/Build Before DebugBoolean	Enables auto-building of a project before downloading if it is out of date.
Confirm Automatically Build Before Debug Environment/Build/Show Build Before DebugBoolean	Enables the display of the auto-building popup.
Confirm Debugger Stop Environment/Build/Confirm Debugger StopBoolean	Present a warning when you start to build that requires the debugger to stop.
Display ETA Environment/Build/Display ETABoolean	Selects whether to attempt to compute and display the ETA on building.
Display Progress Bar Environment/Build/Display Progress BarBoolean	Selects whether to display progress bar on building.
Echo Build Command Lines Environment/Build/Show Command LinesBoolean	Selects whether build command lines are written to the build log.
Echo Raw Error/Warning Output Environment/Build/Show Unparsed Error OutputBoolean	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 BuildBoolean	Moves the cursor to the first diagnostic after a build completes with errors.
Global Macros Environment/Macros/Global MacrosStringList	Build macros that are shared across all solutions and projects e.g. paths to library files.
Keep Going On Error Environment/Build/Keep Going On ErrorBoolean	Build doesn't stop on error.
Save Project File Before Building Environment/Build/Save Project File On BuildBoolean	Selects whether to save the project file prior to build.
Show Build Information Environment/Build/Show Build InformationBoolean	Show build information.

Build Acceleration

Property	Description
Disable Unity Build Environment/Build/Disable Unity BuildBoolean	Ignore Unity Build project properties and always build individual project components.
Parallel Building Threads Environment/Build/Building ThreadsIntegerRange	The number of threads to launch when building.

Compatibility

Property	Description
Installation Directory ARM/Build/StudioDir DirectoryDirPath	The installation directory to be used for building - the value \$(StudioDir) is set to.

Window

Property	Description
Show Build Log On Build Environment/Show Transcript On BuildBoolean	Show the build log when a build starts.

Debugging Environment Options

Breakpoint

Property	Description
Disassembly Breakpoints Environment/Debugger/Disassembly BreakpointsBoolean	What to do with disassembly breakpoints on debug stop/start.
Focus On Breakpoint Environment/Debugger/Focus On BreakpointBoolean	Focus IDE when breakpoint is hit.

Display

Property	Description
Close Disassembly On Mode Switch Environment/Debugger/Close Disassembly On Mode SwitchBoolean	Close Disassembly On Mode Switch.
Data Tips Display a Maximum Of Environment/Debugger/Maximum Array Elements DisplayedIntegerRange	Selects the maximum number of array elements displayed in a data tip.
Default Display Mode Environment/Debugger/Default Variable Display ModeEnumeration	Selects the format that data values are shown in.
Display Floating Point Number In Environment/Debugger/Floating Point Format DisplayCustom	The printf format directive used to display floating point numbers.
Maximum Backtrace Calls Environment/Debugger/Maximum Backtrace CallsIntegerRange	Selects the maximum number of calls when backtracing.
Prompt To Display If More Than Environment/Debugger/Array Elements Prompt SizeIntegerRange	The array size to display with prompt.
Show Data Tips In Text Editor Environment/Debugger/Show Data TipsBoolean	Show Data Tips In Text Editor.
Show ELF Header ElfDwarf/Environment/Show ELF HeaderBoolean	Display ELF Headers when executable and object files are displayed in text editor.
Show Folds In Disassembly Environment/Debugger/Disassembly Show FoldsBoolean	Show Folds In Disassembly.

Show Labels In Disassembly Environment/Debugger/Disassembly Show LabelsBoolean	Show Labels In Disassembly.
Show Source In Disassembly Environment/Debugger/Disassembly Show SourceBoolean	Show Source In Disassembly.
Show char * as null terminated string Environment/Debugger/Display Char Ptr As StringBoolean	Show char * as null terminated string.
Source Path Environment/Debugger/Source PathStringList	Global search path to find source files.
Use objdump For File Disassembly ElfDwarf/Environment/Use Objdump For DisassemblyBoolean	Specifies whether to use objdump to disassemble files rather than the built-in disassembler.

Extended Data Tips

Property	Description
ASCII Environment/Debugger/Extended Tooltip Display Mode/ASCIIBoolean	Display ASCII extended data tips.
Binary Environment/Debugger/Extended Tooltip Display Mode/BinaryBoolean	Display Binary extended data tips.
Decimal Environment/Debugger/Extended Tooltip Display Mode/DecimalBoolean	Display Decimal extended data tips.
Hexadecimal Environment/Debugger/Extended Tooltip Display Mode/HexadecimalBoolean	Display Hexadecimal extended data tips.
Octal Environment/Debugger/Extended Tooltip Display Mode/OctalBoolean	Display Octal extended data tips.
Unsigned Decimal Environment/Debugger/Extended Tooltip Display Mode/Unsigned DecimalBoolean	Display Unsigned Decimal extended data tips.

Target

Property	Description
----------	-------------

Automatically Connect When Starting Debug Target/Auto ConnectBoolean	Enable automatic connection to last connected target when debug start pressed.
Automatically Disconnect When Stopping Debug Target/Auto DisconnectBoolean	Enable automatic disconnection on debug stop.
Background Scan for Debug Pod Presence Environment/Targets Window/Background Target ScanBoolean	Scan USB devices to detect if debug pods are plugged in which may affect CrossStudio response.
Check Project And Target Processor Compatibility Target/Enable Processor CheckBoolean	Verify that the project-defined processor is compatible with the connected target processor.
Enable Differential Download Target/Enable Differential DownloadBoolean	Verify the contents of memory prior to download and only download the code and data that is different.
Identify Target On Connect Target/IdentifyBoolean	Note that turning this off may make a malfunctioning target connection appear as if it is working.
Step Using Hardware Step Environment/Debugger/Step Using Hardware StepBoolean	Step using hardware single stepping rather than setting breakpoints.
Switch Project To Text Editor Environment/Debugger/Switch ProjectBoolean	Switch Debugger Project To Text Editor.
Verify Program After Download Target/Enable Load VerificationBoolean	Verify that a program has been successfully downloaded after download.

Window

Property	Description
Clear Debug Terminal On Run Environment/Clear Debug Terminal On RunBoolean	Clear the debug terminal automatically when a program is run.
Hide Output Window On Successful Load Debugging/Hide Transcript On Successful LoadBoolean	Hide the Output window when a load completes without error.
Show Target Log On Load Debugging/Show Transcript On LoadBoolean	Show the target log when a load starts.

IDE Environment Options

Browser

Property	Description
Text Size Environment/Browser/Text SizeEnumeration	Sets the text size of the integrated HTML and help browser.
Underline Hyperlinks In Browser Environment/Browser/Underline Web LinksBoolean	Enables underlining of hypertext links in the integrated HTML and help browser.

File Extension

Property	Description
ELF Archive File Extensions ElfDwarf/Environment/Archive File ExtensionsStringList	The file extensions used for ELF archive files.
ELF Executable File Extensions ElfDwarf/Environment/Executable File ExtensionsStringList	The file extensions used for ELF executable files.
ELF Object File Extensions ElfDwarf/Environment/Object File ExtensionsStringList	The file extensions used for ELF object files.

File Search

Property	Description
Collapse Search Results Find In Files/Collapse ResultsBoolean	Whether to initially collapse search results.
Files To Exclude Find In Files/Exclude File TypesStringList	The wildcard used to exclude files in Find In Files searches.
Files To Search Find In Files/File TypeStringList	The wildcard used to match files in Find In Files searches.
Find History Find In Files/Find HistoryStringList	The list of strings recently used in searches.
Flat Search Result Output Find In Files/Flat OutputBoolean	Whether to display file search results as a flat list.

Folder History Find In Files/Folder HistoryStringList	The set of folders recently used in file searches.
Match Case Find In Files/Match CaseBoolean	Whether the case of letters must match exactly when searching.
Match Whole Word Find In Files/Match Whole WordBoolean	Whether the whole word must match when searching.
Replace History Find In Files/Replace HistoryStringList	The list of strings recently used in searches.
Search Dependencies Find In Files/Search DependenciesBoolean	Controls searching of dependent files."
Search In Find In Files/ContextEnumeration	Where to look to find files.
Use Regular Expressions Find In Files/Use RegExpBoolean	Whether to use a regular expression or plain text search.

Find And Replace

Property	Description
Greedy Regular Expressions Find/Greedy RegExpBoolean	Enables greedy matching when using regular expressions.

Internet

Property	Description
Automatically Check For Packages Environment/Internet/Check PackagesBoolean	Specifies whether to enable downloading of the list of available packages.
Automatically Check For Updates Environment/Internet/Check UpdatesBoolean	Specifies whether to check for software updates.
Check For Latest News Environment/Internet/RSS UpdateBoolean	Specifies whether to update the latest news window.
Enable Connection Debugging Environment/Internet/Enable DebuggingBoolean	Controls debugging traces of internet connections and downloads.
External Web Browser Environment/External Web BrowserFileName	The path to the external web browser to use when accessing non-local files.
HTTP Caching Environment/Internet/HTTP CachingBoolean	Specifies if caching should be permitted when carrying out HTTP requests.
HTTP Proxy Host Environment/Internet/HTTP Proxy ServerString	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 PortIntegerRange	Specifies the HTTP proxy server's port number.
Maximum Download History Items Environment/Internet/Max Download History ItemsIntegerRange	The maximum amount of download history kept in the downloads window.

Language Server

Property	Description
Additional clangd Options Environment/Language Server/Additional clangd OptionsStringList	Specifies additional command line options to use when starting clangd.
Enable Language Server Debugging Environment/Language Server/Enable DebuggingBoolean	Controls debugging traces of language servers.

Launcher

Property	Description
Confirm Check Solution Target Environment/Launcher/Confirm Check Solution TargetBoolean	Specifies whether the CrossStudio launcher should present a warning if the solution being launched targets a device it does not support.
Launch Latest Installations Only Environment/Launcher Use Latest Installations OnlyBoolean	Specifies whether the CrossStudio launcher should only consider the latest installations when deciding which one to use.
Launcher Enabled Environment/Launcher EnabledBoolean	Specifies whether the CrossStudio launcher should be used when the operating system or an external application requests a file to be opened.

Package Manager

Property	Description
Check Solution Package Dependencies Environment/Package/Check Solution Package DependenciesBoolean	Specifies whether to check package dependencies when a solution is loaded.
Delete Package Downloads Environment/Package/Delete DownloadsBoolean	Specifies whether to delete downloaded package files after they have been installed.

Install Default Packages Environment/Package/Install Default PackagesBoolean	Specifies whether default packages should be installed on startup if they are not installed already.
Package Directory Environment/Package/Destination DirectoryString	Specifies the directory packages are installed to.
Parallel Download And Install Environment/Package/Parallel Download And InstallBoolean	Specifies whether the package manager should download and install packages in parallel.
Show Logos Environment/Package/Show LogosEnumeration	Specifies whether the package manager should display company logos.
Verify Package Downloads Environment/Package/Verify DownloadsBoolean	Specifies whether to carry out an MD5 sum check on downloaded package files.

Print

Property	Description
Bottom Margin Environment/Printing/Bottom MarginIntegerRange	The page's bottom margin in millimetres.
Left Margin Environment/Printing/Left MarginIntegerRange	The page's left margin in millimetres.
Page Orientation Environment/Printing/OrientationEnumeration	The page's orientation.
Page Size Environment/Printing/Page SizeEnumeration	The page's size.
Right Margin Environment/Printing/Right MarginIntegerRange	The page's right margin in millimetres.
Top Margin Environment/Printing/Top MarginIntegerRange	The page's top margin in millimetres.

Startup

Property	Description
Allow Multiple CrossStudios Environment/Permit Multiple Studio InstancesBoolean	Allow more than one CrossStudio to run at the same time.

Load Last Project On Startup Environment/Load Last Project On StartupBoolean	Specifies whether to load the last project the next time CrossStudio runs.
New Project Directory Environment/General/Solution DirectoryString	The directory where projects are created.
Sort Project File On Save Environment/Sort Project FileBoolean	The project file is sorted when it is saved.
Splash Screen Environment/Splash ScreenEnumeration	How to display the splash screen on startup.

Status Bar

Property	Description
(Visible) Environment/Status BarBoolean	Show or hide the status bar.
Show Build Status Pane Environment/General/Status Bar/Show Build StatusBoolean	Show or hide the Build pane in the status bar.
Show Caret Position Pane Environment/General/Status Bar/Show Caret PosBoolean	Show or hide the Caret Position pane in the status bar.
Show Insert/Overwrite Status Pane Environment/General/Status Bar/Show Insert ModeBoolean	Show or hide the Insert/Overwrite pane in the status bar.
Show Read-Only Status Pane Environment/General/Status Bar/Show Read OnlyBoolean	Show or hide the Read Only pane in the status bar.
Show Size Grip Environment/General/Status Bar/Show Size GripBoolean	Show or hide the status bar size grip.
Show Target Pane Environment/General/Status Bar/Show TargetBoolean	Show or hide the Target pane in the status bar.
Show Time Pane Environment/General/Status Bar/Show TimeBoolean	Show or hide the Time pane in the status bar.

Title Bar

Property	Description
----------	-------------

Show Full Solution Path Environment/General/Title Bar/Show Full Solution PathBoolean	Show the full solution path in title bar.
---	---

User Interface

Property	Description
Application Main Font Environment/Application Main FontFont	The font to use for the user interface as a whole.
Application Monospace Font Environment/Application Monospace FontFixedPitchFont	The fixed-size font to use for the user interface as a whole.
Error Display Timeout Environment/Error Display TimeoutIntegerRange	The minimum time, in seconds, that errors are shown for in the status bar.
Errors Are Displayed Environment/Error Display ModeEnumeration	How errors are reported in CrossStudio.
File Size Display Units Environment/Size Display UnitEnumeration	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 MenusBoolean	Number the first nine file names in menus for quick keyboard access.
Qt Style Sheet Environment/Qt Style SheetFileName	The Qt style sheet to use in order to customize the user interface (experimental).
Show Large Icons In Toolbars Environment/General/Large IconsEnumeration	Show large or small icons on toolbars.
Show Ribbon Environment/General/Ribbon/ShowBoolean	Show or hide the ribbon.
Show Window Selector On Ctrl+Tab Environment/Show SelectorBoolean	Present the Window Selector on Next Window and Previous Window commands activated from the keyboard.
Theme Environment/Studio ThemeEnumeration	The user interface style and color theme to use.
Window Menu Contains At Most Environment/Max Window Menu ItemsIntegerRange	The maximum number of windows appearing in the Windows menu.

Programming Language Environment Options

Assembly Language

Property	Description
Column Guide Columns Text Editor/Indent/Assembly Language/ Column GuidesString	The columns that guides are drawn for.
Indent Closing Brace Text Editor/Indent/Assembly Language/ Close BraceBoolean	Indent the closing brace of compound statements.
Indent Context Text Editor/Indent/Assembly Language/ Context LinesIntegerRange	The number of lines to use for context when indenting.
Indent Mode Text Editor/Indent/Assembly Language/ Indent ModeEnumeration	How to indent when a new line is inserted.
Indent Opening Brace Text Editor/Indent/Assembly Language/Open BraceBoolean	Indent the opening brace of compound statements.
Indent Size Text Editor/Indent/Assembly Language/ SizeIntegerRange	The number of columns to indent a code block.
Tab Size Text Editor/Indent/Assembly Language/Tab SizeIntegerRange	The number of columns between tabstops.
Use Tabs Text Editor/Indent/Assembly Language/Use TabsBoolean	Insert tabs when indenting.
User-Defined Keywords Text Editor/Indent/Assembly Language/ KeywordsStringList	Additional identifiers to highlight as keywords.

C and C++

Property	Description
Column Guide Columns Text Editor/Indent/C and C++/Column GuidesString	The columns that guides are drawn for.

Indent Closing Brace Text Editor/Indent/C and C++/Close BraceBoolean	Indent the closing brace of compound statements.
Indent Context Text Editor/Indent/C and C++/Context LinesIntegerRange	The number of lines to use for context when indenting.
Indent Mode Text Editor/Indent/C and C++/Indent ModeEnumeration	How to indent when a new line is inserted.
Indent Opening Brace Text Editor/Indent/C and C++/Open BraceBoolean	Indent the opening brace of compound statements.
Indent Size Text Editor/Indent/C and C++/ SizeIntegerRange	The number of columns to indent a code block.
Tab Size Text Editor/Indent/C and C++/Tab SizeIntegerRange	The number of columns between tabstops.
Use Tabs Text Editor/Indent/C and C++/Use TabsBoolean	Insert tabs when indenting.
User-Defined Keywords Text Editor/Indent/C and C++/ KeywordsStringList	Additional identifiers to highlight as keywords.

Default

Property	Description
Column Guide Columns Text Editor/Indent/Default/Column GuidesString	The columns that guides are drawn for.
Indent Closing Brace Text Editor/Indent/Default/Close BraceBoolean	Indent the closing brace of compound statements.
Indent Context Text Editor/Indent/Default/Context LinesIntegerRange	The number of lines to use for context when indenting.
Indent Mode Text Editor/Indent/Default/Indent ModeEnumeration	How to indent when a new line is inserted.

Indent Opening Brace Text Editor/Indent/Default/Open BraceBoolean	Indent the opening brace of compound statements.
Indent Size Text Editor/Indent/Default/SizeIntegerRange	The number of columns to indent a code block.
Tab Size Text Editor/Indent/Default/Tab SizeIntegerRange	The number of columns between tabstops.
Use Tabs Text Editor/Indent/Default/Use TabsBoolean	Insert tabs when indenting.
User-Defined Keywords Text Editor/Indent/Default/KeywordsStringList	Additional identifiers to highlight as keywords.

Java

Property	Description
Column Guide Columns Text Editor/Indent/Java/Column GuidesString	The columns that guides are drawn for.
Indent Closing Brace Text Editor/Indent/Java/Close BraceBoolean	Indent the closing brace of compound statements.
Indent Context Text Editor/Indent/Java/Context LinesIntegerRange	The number of lines to use for context when indenting.
Indent Mode Text Editor/Indent/Java/Indent ModeEnumeration	How to indent when a new line is inserted.
Indent Opening Brace Text Editor/Indent/Java/Open BraceBoolean	Indent the opening brace of compound statements.
Indent Size Text Editor/Indent/Java/SizeIntegerRange	The number of columns to indent a code block.
Tab Size Text Editor/Indent/Java/Tab SizeIntegerRange	The number of columns between tabstops.
Use Tabs Text Editor/Indent/Java/Use TabsBoolean	Insert tabs when indenting.
User-Defined Keywords Text Editor/Indent/Java/KeywordsStringList	Additional identifiers to highlight as keywords.

XML

Property	Description
----------	-------------

Column Guide Columns Text Editor/Indent/XML/Column GuidesString	The columns that guides are drawn for.
Indent Context Text Editor/Indent/XML/Context LinesIntegerRange	The number of lines to use for context when indenting.
Indent Mode Text Editor/Indent/XML/Indent ModeEnumeration	How to indent when a new line is inserted.
Indent Size Text Editor/Indent/XML/SizeIntegerRange	The number of columns to indent a code block.
Tab Size Text Editor/Indent/XML/Tab SizeIntegerRange	The number of columns between tabstops.
Use Tabs Text Editor/Indent/XML/Use TabsBoolean	Insert tabs when indenting.
User-Defined Keywords Text Editor/Indent/XML/KeywordsStringList	Additional identifiers to highlight as keywords.

Source Control Environment Options

External Tools

Property	Description
Diff Command Line Environment/Source Code Control/ DiffCommandStringList	The diff command line.
Merge Command Line Environment/Source Code Control/ MergeCommandStringList	The merge command line.

Preference

Property	Description
Add Immediately Environment/Source Code Control/Immediate AddBoolean	Bypasses the confirmation dialog and immediately adds items to source control.
Commit Immediately Environment/Source Code Control/Immediate CommitBoolean	Bypasses the confirmation dialog and immediately commits items.
Get Immediately Environment/Source Code Control/Immediate GetBoolean	Bypasses the confirmation dialog and immediately gets items from source control.
Lock Immediately Environment/Source Code Control/Immediate LockBoolean	Bypasses the confirmation dialog and immediately locks items.
Remove Immediately Environment/Source Code Control/Immediate RemoveBoolean	Bypasses the confirmation dialog and immediately removes items from source control.
Resolved Immediately Environment/Source Code Control/Immediate ResolvedBoolean	Bypasses the confirmation dialog and immediately mark items resolved.
Revert Immediately Environment/Source Code Control/Immediate RevertBoolean	Bypasses the confirmation dialog and immediately revert items.
Unlock Immediately Environment/Source Code Control/Immediate UnlockBoolean	Bypasses the confirmation dialog and immediately unlocks items.

Update Immediately Environment/Source Code Control/Immediate UpdateBoolean	Bypasses the confirmation dialog and immediately updates items.
---	---

Text Editor Environment Options

Auto Recovery

Property	Description
Auto Recovery Backup Time Text Editor/Auto Recovery Backup TimeIntegerRange	The time in minutes between saving of auto recovery backups files or 0 to disable generation of backup files.
Auto Recovery Keep Time Text Editor/Auto Recovery Keep TimeIntegerRange	The time in days to keep unrecovered backup files or 0 to disable deletion of unrecovered backup files.

Cursor Fence

Property	Description
Bottom Margin Text Editor/Margins/BottomIntegerRange	The number of lines in the bottom margin.
Keep Cursor Within Fence Text Editor/Margins/EnabledBoolean	Enable margins to fence and scroll around the cursor.
Left Margin Text Editor/Margins/LeftIntegerRange	The number of characters in the left margin.
Right Margin Text Editor/Margins/RightIntegerRange	The number of characters in the right margin.
Top Margin Text Editor/Margins/TopIntegerRange	The number of lines in the right margin.

Editing

Property	Description
Allow Drag and Drop Editing Text Editor/Drag Drop EditingBoolean	Enables dragging and dropping of selections in the text editor.
Bold Popup Diagnostic Messages Text Editor/Bold Popup DiagnosticsBoolean	Displays popup diagnostic messages in bold for easier reading.
Column-mode Tab Text Editor/Column Mode TabBoolean	Tab key moves to the next textual column using the line above.
Confirm Modified File Reload Text Editor/Confirm Modified File ReloadBoolean	Display a confirmation prompt before reloading a file that has been modified on disk.

Copy Action When Nothing Selected Text Editor/Copy ActionEnumeration	What Copy copies when nothing is selected.
Cut Action When Nothing Selected Text Editor/Cut ActionEnumeration	What Cut cuts when nothing is selected.
Cut Single Blank Line Text Editor/Cut Blank LinesBoolean	Selects whether to place text on the clipboard when a single blank line is cut. When set to Yes, cutting a single blank line will put the blank line on the clipboard. When set to No, cutting a single blank line deletes the line and does not place it on the clipboard.
Diagnostic Cycle Mode Text Editor/Diagnostic Cycle ModeEnumeration	Iterates through diagnostics either from most severe to least severe or in reported order.
Edit Read-Only Files Text Editor/Edit Read OnlyBoolean	Allow editing of read-only files.
Enable Virtual Space Text Editor/Enable Virtual SpaceBoolean	Permit the cursor to move into locations that do not currently contain text.
Numeric Keypad Editing Text Editor/Numeric Keypad EnabledBoolean	Selects whether the numeric keypad plus and minus buttons copy and cut text.
Tab Key Indents Preprocessor Directives Text Editor/Tab Key Indents Preprocessor DirectivesBoolean	Enables or disables the indentation of C preprocessor directives when using tab key indentation on a selection.
Undo And Redo Behavior Text Editor/Undo ModeEnumeration	How Undo and Redo group your typing when it is undone and redone.

Find And Replace

Property	Description
Case Sensitive Matching Text Editor/Find/Match CaseBoolean	Enables or disables the case sensitivity of letters when searching.
Find History Text Editor/Find/HistoryStringList	The list of strings recently used in searches.
Regular Expression Matching Text Editor/Find/Use RegExpBoolean	Enables regular expression matching rather than plain text matching.
Replace History Text Editor/Replace/HistoryStringList	The list of strings recently used in replaces.
Whole Word Matching Text Editor/Find/Match Whole WordBoolean	Enables or disables whole word matching when searching.

Formatting

Property	Description
----------	-------------

Access Modifier Offset Text Editor/Formatting/ AccessModifierOffsetInteger	The extra indent or outdent of access modifiers, e.g. public:..
Additional Formatting Styles Text Editor/Additional Formatting StylesStringList	Additional styles to pass to clang-format.
Align After Open Bracket Text Editor/Formatting/ AlignAfterOpenBracketBoolean	If enabled, horizontally aligns arguments after an open bracket.
Align Consecutive Assignments Text Editor/Formatting/ AlignConsecutiveAssignmentsBoolean	If enabled, aligns consecutive assignments.
Align Consecutive Declarations Text Editor/Formatting/ AlignConsecutiveDeclarationsBoolean	If enabled, aligns consecutive declarations.
Align Escaped Newlines Left Text Editor/Formatting/ AlignEscapedNewlinesLeftBoolean	If enabled, aligns escaped newlines as far left as possible otherwise puts them into the right-most column.
Align Operands Text Editor/Formatting/ AlignOperandsBoolean	If enabled, horizontally align operands of binary and ternary expressions.
Align Trailing Comments Text Editor/Formatting/ AlignTrailingCommentsBoolean	If enabled, aligns trailing comments.
Allow All Parameters Of Declaration On Next Line Text Editor/Formatting/ AllowAllParametersOfDeclarationOnNextLineBoolean	Allow putting all parameters of a function declaration onto the next line even if Bin-pack Parameters is disabled.
Allow Short 'if' Statements On A Single Line Text Editor/Formatting/ AllowShortIfStatementsOnASingleLineBoolean	If enabled, short 'if' statements are put on a single line.
Allow Short Blocks On A Single Line Text Editor/Formatting/ AllowShortBlocksOnASingleLineBoolean	If enabled, allows contracting simple braced statements to a single line.
Allow Short Case Labels On A Single Line Text Editor/Formatting/ AllowShortCaseLabelsOnASingleLineBoolean	If enabled, short case labels will be contracted to a single line.
Allow Short Functions On A Single Line Text Editor/Formatting/ AllowShortFunctionsOnASingleLineEnumeration	Optionally compress small functions to a single line.
Allow Short Loop Statements On A Single Line Text Editor/Formatting/ AllowShortLoopsOnASingleLineBoolean	If enabled, short loop statements are put on a single line.

Always Break After Return Type Text Editor/Formatting/ AlwaysBreakAfterReturnTypeEnumeration	The function declaration return type breaking style to use.
Always Break Before Multiline Strings Text Editor/Formatting/ AlwaysBreakBeforeMultilineStringsBoolean	If enabled, always break before multiline strings.
Always Break Template Declarations Text Editor/Formatting/ AlwaysBreakTemplateDeclarationsBoolean	If enabled, always break after the 'template<...>' of a template declaration.
Bin-Pack Arguments Text Editor/Formatting/ BinPackArgumentsBoolean	If disabled, a function call's arguments will either be all on the same line or will have one line each.
Bin-Pack Parameters Text Editor/Formatting/ BinPackParametersBoolean	If disabled, a function call's or function definition's parameters will either all be on the same line or will have one line each.
Break Before Binary Operators Text Editor/Formatting/ BreakBeforeBinaryOperatorsBoolean	The way to wrap binary operators.
Break Before Braces Text Editor/Formatting/ BreakBeforeBracesEnumeration	The brace breaking style to use.
Break Before Inheritance Comma Text Editor/Formatting/ BreakBeforeInheritanceCommaBoolean	If enabled, the class inheritance expression will break before : and , if there is multiple inheritance.
Break Before Ternary Operators Text Editor/Formatting/ BreakBeforeTernaryOperatorsBoolean	If enabled, ternary operators will be placed after line breaks.
Break Constructor Initializers Before Comma Text Editor/Formatting/ BreakConstructorInitializersBeforeCommaBool	If enabled, always break constructor initializers before commas and align the commas with the colon.
Break String Literals Text Editor/Formatting/ BreakStringLiteralsBoolean	Allow breaking string literals when formatting.
C++11 Braced List Style Text Editor/Formatting/ Cpp11BracedListStyleBoolean	If enabled, format braced lists as best suited for C++11 braced lists.
Column Limit Text Editor/Formatting/ColumnLimitInteger	The column limit which limits the width of formatted lines.
Comment Pragmas Text Editor/Formatting/CommentPragmasString	A regular expression that describes comments with special meaning, which should not be split into lines or otherwise changed.

Compact Namespaces Text Editor/Formatting/ CompactNamespacesBoolean	If enabled, consecutive namespace declarations will be on the same line. If disabled, each namespace is declared on a new line.
Constructor Initializer All On One Line Or One Per Line Text Editor/Formatting/ ConstructorInitializerAllOnOneLineOrOnePer	If enabled and the constructor initializers don't fit on a line, put each initializer on its own line.
Constructor Initializer Indent Width Text Editor/Formatting/ ConstructorInitializerIndentWidthInteger	The number of characters to use for indentation of constructor initializer lists.
Continuation Indent Width Text Editor/Formatting/ ContinuationIndentWidthInteger	Indent width for line continuations.
Derive Pointer Alignment Text Editor/Formatting/ DerivePointerAlignmentBoolean	If enabled, analyze the formatted file for the most common alignment of address of and dereference. PointerAlignment is then used only as fallback.
Empty Lines At End Of File Text Editor/Extra Formatting/ LinesAtEOFIntegerRange	The number of lines to add at the end of the file.
Fix Namespace Comments Text Editor/Formatting/ FixNamespaceCommentsBoolean	If enabled, add missing namespace end comments and fix invalid existing ones.
For-Each Macros Text Editor/Formatting/ ForEachMacrosStringList	A list of macros that should be interpreted as foreach loops rather than function calls.
Formatting Indent Width Text Editor/Formatting/IndentWidthInteger	The number of columns the code formatter uses for indentation. Note that this is not the indent width used by the text editor, that value is specified in the 'Languages' environment option group.
Formatting Style Text Editor/FormattingStyleEnumeration	Select a set of formatting options based on a named standard.
Formatting Tab Width Text Editor/Formatting/TabWidthIntegerRange	The number of columns the code formatter uses for tab stops. Note that this is not the tab width used by the text editor, that value is specified in the 'Languages' environment option group.
Include Is Main Regex Text Editor/Formatting/ IncludeIsMainRegexString	Specify a regular expression of suffixes that are allowed in the file-to-main-include mapping.
Indent Case Labels Text Editor/Formatting/ IndentCaseLabelsBoolean	If enabled, indent case labels one level from the switch statement.
Indent Wrapped Function Names Text Editor/Formatting/ IndentWrappedFunctionNamesBoolean	If enabled, Indent if a function definition or declaration is wrapped after the type.

Keep Empty Lines At The Start Of Blocks Text Editor/Formatting/ KeepEmptyLinesAtTheStartOfBlocksBoolean	If enabled, empty lines at the start of blocks are kept.
Macro Block Begin Text Editor/Formatting/ MacroBlockBeginString	A regular expression matching macros that start a block.
Macro Block End Text Editor/Formatting/MacroBlockEndString	A regular expression matching macros that end a block.
Maximum Empty Lines To Keep Text Editor/Formatting/ MaxEmptyLinesToKeepInteger	The maximum number of consecutive empty lines to keep.
Namespace Indentation Text Editor/Formatting/ NamespaceIndentationEnumeration	The indentation used for namespaces.
Penalty Break Assignment Text Editor/Formatting/ PenaltyBreakAssignmentIntegerRange	The penalty for breaking around an assignment operator.
Penalty Break Before First Call Parameter Text Editor/Formatting/ PenaltyBreakBeforeFirstCallParameterIntegerRange	The penalty for breaking a function call after 'call('.
Penalty Break Before First Less-Less Text Editor/Formatting/ PenaltyBreakFirstLessLessIntegerRange	The penalty for breaking before the first less-less.
Penalty Break Comment Text Editor/Formatting/ PenaltyBreakCommentIntegerRange	The penalty for each line break introduced inside a comment.
Penalty Break String Text Editor/Formatting/ PenaltyBreakStringIntegerRange	The penalty for each line break introduced inside a string literal.
Penalty Excess Character Text Editor/Formatting/ PenaltyExcessCharacterIntegerRange	The penalty for each character outside of the column limit.
Penalty Return Type On Its Own Line Text Editor/Formatting/ PenaltyReturnTypeOnItsOwnLineIntegerRange	Penalty for putting the return type of a function onto its own line.
Pointer Alignment Text Editor/Formatting/ PointerAlignmentEnumeration	Pointer and reference alignment style.
Reflow Comments Text Editor/Formatting/ ReflowCommentsBoolean	If enabled, clang-format will attempt to re-flow comments.
Sort Includes Text Editor/Formatting/SortIncludesBoolean	If enabled, sort #includes.

Sort Using Declarations Text Editor/Formatting/ SortUsingDeclarationsBoolean	If enabled, sort using declarations.
Space After C Style Cast Text Editor/Formatting/ SpaceAfterCStyleCastBoolean	If enabled, a space may be inserted after C style casts.
Space After Template Keyword Text Editor/Formatting/ SpaceAfterTemplateKeywordBoolean	If enabled, a space will be inserted after the ?template? keyword.
Space Before Assignment Operators Text Editor/Formatting/ SpaceBeforeAssignmentOperatorsBoolean	If disabled spaces will be removed before assignment operators.
Space Before Parentheses Text Editor/Formatting/ SpaceBeforeParensEnumeration	Defines in which cases to put a space before opening parentheses.
Space In Empty Parentheses Text Editor/Formatting/ SpaceInEmptyParenthesesBoolean	If enabled, spaces may be inserted into '()'.
Spaces Before Trailing Comments Text Editor/Formatting/ SpacesBeforeTrailingCommentsIntegerRange	The number of spaces before trailing line comments.
Spaces In Angles Text Editor/Formatting/ SpacesInAnglesBoolean	If enabled, spaces will be inserted around the angle brackets in template argument lists.
Spaces In C-style Cast Parentheses Text Editor/Formatting/ SpacesInCStyleCastParenthesesBoolean	If enabled, spaces may be inserted into C style casts.
Spaces In Container Literals Text Editor/Formatting/ SpacesInContainerLiteralsBoolean	If enabled, spaces are inserted inside container literals.
Spaces In Parentheses Text Editor/Formatting/ SpacesInParenthesesBoolean	If true, spaces will be inserted after '(' and before ')'.
Spaces In Square Brackets Text Editor/Formatting/ SpacesInSquareBracketsBoolean	If true, spaces will be inserted after '[' and before ']'.
Standard Text Editor/Formatting/StandardEnumeration	Format compatible with this standard
Tab Style Text Editor/Formatting/UseTabEnumeration	The way to use hard tab characters in the resulting file.

Use .clang-format File Text Editor/Use .clang-format FileBoolean	Load code formatting style configuration from a .clang-format file located in one of the parent directories of the source file rather than use the formatting options.
--	--

International

Property	Description
Auto-Detect UTF-8 Text Editor/Auto-Detect UTF-8Boolean	Auto-detect UTF-8 encoding without signature.
Default Text File Encoding Text Editor/Default CodecEnumeration	The encoding to use if not overridden by a project property or file is not in a known format.
Verify Text File Decoding Text Editor/Verify DecodeBoolean	Specifies whether the decoding of a text file should be verified when file is loaded.

Mouse

Property	Description
Alt+Left Click Action Environment/Project Explorer/Alt+Left Click ActionEnumeration	The action the editor performs on Alt+Left Click.
Alt+Middle Click Action Environment/Project Explorer/Alt+Middle Click ActionEnumeration	The action the editor performs on Alt+Middle Click.
Alt+Right Click Action Environment/Project Explorer/Alt+Right Click ActionEnumeration	The action the editor performs on Alt+Right Click.
Copy On Mouse Select Text Editor/Copy On Mouse SelectBoolean	Automatically copy text to clipboard when marking a selection with the mouse.
Ctrl+Left Click Action Environment/Project Explorer/Ctrl+Left Click ActionEnumeration	The action the editor performs on Ctrl+Left Click.
Ctrl+Middle Click Action Environment/Project Explorer/Ctrl+Middle Click ActionEnumeration	The action the editor performs on Ctrl+Middle Click.
Ctrl+Right Click Action Environment/Project Explorer/Ctrl+Right Click ActionEnumeration	The action the editor performs on Ctrl+Right Click.
Middle Click Action Environment/Project Explorer/Middle Click ActionEnumeration	The action the editor performs on Middle Click.

Mouse Wheel Adjusts Font Size Text Editor/Mouse Wheel Adjusts Font SizeBoolean	Enable or disable resizing of font by mouse wheel when CTRL key pressed.
Shift+Middle Click Action Environment/Project Explorer/Shift+Middle Click ActionEnumeration	The action the editor performs on Shift+Middle Click.
Shift+Right Click Action Environment/Project Explorer/Shift+Right Click ActionEnumeration	The action the editor performs on Shift+Right Click.

Programmer Assistance

Property	Description
ATTENTION Tag List Text Editor/ATTENTION TagsStringList	Set the tags to display as ATTENTION comments.
Ask For Index Text Editor/Ask For IndexBoolean	Ask to index the project if goto symbol fails in current editor context.
Auto-Comment Text Text Editor/Auto CommentBoolean	Enable or disable automatically swapping commenting on source lines by typing '/' with an active selection.
Auto-Surround Text Text Editor/Auto SurroundBoolean	Enable or disable automatically surrounding selected text when typing triangular brackets, quotation marks, parentheses, brackets, or braces.
Check Spelling Text Editor/Spell CheckingBoolean	Enable spell checking in comments.
Code Completion Characters Text Editor/Code Completion CharactersIntegerRange	The minimum number of word characters required before showing the code completion suggestions while typing.
Code Completion Replaces Existing Word Text Editor/Completion Replaces Existing WordBoolean	Replace existing word with completion suggestion if cursor is located on one.
Code Completion Suggestion Selection Key Text Editor/Suggestion Selection KeyEnumeration	The key used to select a code completion suggestion.
Display Code Completion Suggestions While Typing Text Editor/Suggest Completion While TypingBoolean	Enable code completion as you type without needing to use the show suggestions key (Ctrl+J).
Enable Popup Diagnostics Text Editor/Enable Popup DiagnosticsBoolean	Enables on-screen popup diagnostics messages.
FIXME Tag List Text Editor/FIXME TagsStringList	Set the tags to display as FIXME comments.

Inactive Code Opacity Text Editor/Inactive Code OpacityIntegerRange	Specifies the opacity of code that has been conditionally excluded by the preprocessor.
Include Preprocessor Definitions in Suggestions Text Editor/Preprocessor Definition SuggestionsBoolean	Include or exclude preprocessor definitions in code completion suggestions.
Include Templates in Suggestions Text Editor/Template SuggestionsBoolean	Include or exclude templates in code completion suggestions.
Lint Tag List Text Editor/LINT TagsStringList	Set the tags to display as Lint directives.
Select First Code Completion Selection Text Editor/Select First SuggestionBoolean	Automatically select first suggestion when showing suggestions
Show Diagnostics Text Editor/Show DiagnosticsEnumeration	Enables on-screen diagnostics in the text editor.
Show Inactive Code Text Editor/Show Inactive CodeBoolean	Show code that has been conditionally excluded by the preprocessor.
Show Inline Diagnostics Text Editor/Show Inline DiagnosticsEnumeration	Enables inline diagnostics in the text editor.
Show Symbol Declaration Tooltips Text Editor/Show TooltipBoolean	Show tooltips when hovering over symbols.
Template Characters To Match Text Editor/Template Suggestions CharactersIntegerRange	The number of characters to match before suggesting a template.

Save

Property	Description
Backup File History Depth Text Editor/Backup File DepthIntegerRange	The number of backup files to keep when saving an existing file.
Default Line Endings Text Editor/Default Line EndingsEnumeration	The line ending format to use for a new file or a file where the existing line ending format cannot be determined.
Delete Trailing Space On Save Text Editor/Delete Trailing Space On SaveBoolean	Deletes trailing whitespace from each line when a file is saved.
Format On Save Text Editor/Format On SaveEnumeration	Formats text when a file is saved.
Tab Cleanup On Save Text Editor/Cleanup Tabs On SaveEnumeration	Cleans up tabs when a file is saved.

Visual Appearance

Property	Description
Fold Comments Text Editor/Fold CommentsBoolean	Allow multiline comments to be collapsed.
Fold Preprocessor Directives Text Editor/Fold Preprocessor DirectivesBoolean	Allow preprocessor directives to be collapsed.
Font Text Editor/FontFixedPitchFont	The font to use for text editors.
Font Rendering Text Editor/Font RenderingEnumeration	The font rendering scheme to use in text editors.
Font Smoothing Threshold Text Editor/Antialias ThresholdIntegerRange	The minimum size for font smoothing; font sizes smaller than this will have antialiasing turned off.
Hide Cursor When Typing Text Editor/Hide Cursor When TypingBoolean	Hide or show the I-beam cursor when you start to type.
Highlight All Selected Text Text Editor/Highlight All Selected TextBoolean	Enable or disable visually highlighting all text that matches the current selection.
Highlight Cursor Line Text Editor/Highlight Cursor LineBoolean	Enable or disable visually highlighting the cursor line.
Highlight References Text Editor/Highlight ReferencesBoolean	Enable or disable visually highlighting of references.
Horizontal Scroll Bar Text Editor/HScroll BarEnumeration	Show or hide the horizontal scroll bar.
Insert Caret Style Text Editor/Insert Caret StyleEnumeration	How the caret is displayed with the editor in insert mode.
Line Numbers Text Editor/Line Number ModeEnumeration	How often line numbers are displayed in the margin.
Mate Match Off Screen Text Editor/Mate Match Off ScreenBoolean	Specifies whether braces, brackets, and parentheses are matched when off screen.
Mate Matching Mode Text Editor/Mate Matching ModeEnumeration	Controls when braces, brackets, and parentheses are matched.
Maximum Collapsed Fold Preview Lines Text Editor/Maximum Collapsed Fold Preview LinesIntegerRange	The maximum number of lines to show in a collapsed fold preview tooltip.
Minimum Scroll Width Text Editor/Minimum Scroll WidthIntegerRange	Specifies the minimum width of the scrolling region in characters.
Overwrite Caret Style Text Editor/Overwrite Caret StyleEnumeration	How the caret is displayed with the editor in overwrite mode.

Selection Opacity Text Editor/Selection OpacityIntegerRange	Specifies the opacity of text selection.
Show Bookmarks In Vertical Scroll Bar Text Editor/Show Bookmarks In Vertical Scroll BarBoolean	Annotate the vertical scroll bar with bookmark positions.
Show Breakpoints In Vertical Scroll Bar Text Editor/Show Breakpoints In Vertical Scroll BarBoolean	Annotate the vertical scroll bar with breakpoint positions.
Show Caret Position In Vertical Scroll Bar Text Editor/Show Caret In Vertical Scroll BarBoolean	Annotate the vertical scroll bar with the caret's position within the document.
Show Diagnostic Icons In Gutter Text Editor/Diagnostic IconsBoolean	Enables display of diagnostic icons in the icon gutter.
Show Errors In Vertical Scroll Bar Text Editor/Show Errors In Vertical Scroll BarBoolean	Annotate the vertical scroll bar with error positions.
Show Fold Gutter Text Editor/Fold GutterBoolean	Show or hide the left-hand gutter containing folding controls.
Show Icon Gutter Text Editor/Icon GutterBoolean	Show or hide the left-hand gutter containing breakpoint, bookmark, and optional diagnostic icons.
Show Mini Toolbar Text Editor/Mini ToolbarBoolean	Show the mini toolbar when selecting text with the mouse.
Show Toolbar Text Editor/ShowWidgetStripBoolean	Show or hide the Editor toolbar in the dock window.
Show Warnings In Vertical Scroll Bar Text Editor/Show Warnings In Vertical Scroll BarBoolean	Annotate the vertical scroll bar with warning positions.
Use I-beam Cursor Text Editor/Ibeam cursorBoolean	Show an I-beam or arrow cursor in the text editor.
Vertical Scroll Bar Text Editor/VScroll BarEnumeration	Show or hide the vertical scroll bar.
View Whitespace Text Editor/View WhitespaceBoolean	Make whitespace characters visible in the text editor.

Windows Environment Options

Autos

Property	Description
Show Digit Separator Environment/AutosWindow/Show Digit SeparatorBoolean	Show digit separator in variable value display.
Show Member Functions Environment/AutosWindow/Show Member FunctionsBoolean	Controls whether C++ class member functions are displayed.
Show Variable Address Column Environment/AutosWindow/Show Address ColumnBoolean	Controls whether the variable address column is displayed.
Show Variable Size Column Environment/AutosWindow/Show Size ColumnBoolean	Controls whether the variable size column is displayed.
Show Variable Type Column Environment/AutosWindow/Show Type ColumnBoolean	Controls whether the variable type column is displayed.

Call Stack

Property	Description
Execution Frame at Top Environment/Call Stack/Most Recent At TopBoolean	Controls whether the most recent call is at the top or the bottom of the list.
Show Call Address Environment/Call Stack/Show Call AddressBoolean	Enables the display of the call address in the call stack.
Show Call Source Location Environment/Call Stack/Show Call LocationBoolean	Enables the display of the call source location in the call stack.
Show Frame Size Environment/Call Stack/Show Stack UsageBoolean	Enables the display of the amount of stack used by the call.
Show Frame Size In Bytes Environment/Call Stack/Show Stack Usage In BytesBoolean	Display the stack usage in bytes rather than words.

Show Parameter Names Environment/Call Stack/Show Parameter NamesBoolean	Enables the display of parameter names in the call stack.
Show Parameter Types Environment/Call Stack/Show Parameter TypesBoolean	Enables the display of parameter types in the call stack.
Show Parameter Values Environment/Call Stack/Show Parameter ValuesBoolean	Enables the display of parameter values in the call stack.
Show Stack Pointer Environment/Call Stack/Show Stack PointerBoolean	Enables the display of the stack pointer in the call stack.
Show Stack Usage Environment/Call Stack/Show Cumulative Stack UsageBoolean	Enables the display of the amount of stack used.
Show Stack Usage In Bytes Environment/Call Stack/Show Cumulative Stack Usage In BytesBoolean	Display the stack usage in bytes rather than words.

Clipboard Ring

Property	Description
Maximum Items Held In Ring Environment/Clipboard Ring/Max EntriesIntegerRange	The maximum number of items held on the clipboard ring before they are recycled.
Preserve Contents Between Runs Environment/Clipboard Ring/SaveBoolean	Save the clipboard ring across CrossStudio runs.

Debug Terminal

Property	Description
Backscroll Buffer Lines Debug Terminal/Backscroll Buffer LinesIntegerRange	The number of lines you can see when you scroll backward in the debug terminal window.
Use Window System Colors Debug Terminal/Use Window System ColorsBoolean	Substitute window system colors for ANSI black background and white foreground in debug terminal.

Find Symbol Dialog

Property	Description
----------	-------------

Group Symbols Windows/Find Symbol Dialog/Group SymbolsBoolean	Group symbols by type.
Scope Windows/Find Symbol Dialog/ScopeEnumeration	Specifies whether to search for symbols in the entire workspace or only the current document.

Frame Buffer

Property	Description
Maximum Frame Buffer Height Environment/Frame Buffer Window/Maximum HeightIntegerRange	Specifies the maximum frame buffer height.
Maximum Frame Buffer Width Environment/Frame Buffer Window/Maximum WidthIntegerRange	Specifies the maximum frame buffer width.
Show Frame Buffer Tooltips Environment/Frame Buffer Window/Display TooltipsBoolean	Specifies whether tooltips are displayed in the frame buffer window.

Globals

Property	Description
Show Digit Separator Environment/GlobalsWindow/Show Digit SeparatorBoolean	Show digit separator in variable value display.
Show Member Functions Environment/GlobalsWindow/Show Member FunctionsBoolean	Controls whether C++ class member functions are displayed.
Show Variable Address Column Environment/GlobalsWindow/Show Address ColumnBoolean	Controls whether the variable address column is displayed.
Show Variable Size Column Environment/GlobalsWindow/Show Size ColumnBoolean	Controls whether the variable size column is displayed.
Show Variable Type Column Environment/GlobalsWindow/Show Type ColumnBoolean	Controls whether the variable type column is displayed.

Latest News

Property	Description
----------	-------------

Article Grouping Environment/Latest News/GroupingEnumeration	How to display the RSS feed articles.
---	---------------------------------------

Locals

Property	Description
Show Digit Separator Environment/LocalsWindow/Show Digit SeparatorBoolean	Show digit separator in variable value display.
Show Member Functions Environment/LocalsWindow/Show Member FunctionsBoolean	Controls whether C++ class member functions are displayed.
Show Struct Offsets Environment/Watch4Window/Show Struct OffsetsBoolean	Show offsets of structure fields in the address column.
Show Struct Offsets Environment/Watch3Window/Show Struct OffsetsBoolean	Show offsets of structure fields in the address column.
Show Struct Offsets Environment/Watch2Window/Show Struct OffsetsBoolean	Show offsets of structure fields in the address column.
Show Struct Offsets Environment/Watch1Window/Show Struct OffsetsBoolean	Show offsets of structure fields in the address column.
Show Struct Offsets Environment/AutosWindow/Show Struct OffsetsBoolean	Show offsets of structure fields in the address column.
Show Struct Offsets Environment/GlobalsWindow/Show Struct OffsetsBoolean	Show offsets of structure fields in the address column.
Show Struct Offsets Environment/LocalsWindow/Show Struct OffsetsBoolean	Show offsets of structure fields in the address column.
Show Variable Address Column Environment/LocalsWindow/Show Address ColumnBoolean	Controls whether the variable address column is displayed.
Show Variable Size Column Environment/LocalsWindow/Show Size ColumnBoolean	Controls whether the variable size column is displayed.
Show Variable Type Column Environment/LocalsWindow/Show Type ColumnBoolean	Controls whether the variable type column is displayed.

Memory

Property	Description
Confirm Large Download Environment/Memory Window/Confirm SizeBoolean	Present a warning if you attempt to download a large amount of memory in the memory window.
Group Auto Columns Environment/Memory Window/Group Auto ColumnsBoolean	Selects whether columns are grouped in automatic column mode.
Locate Sets Entry Width Environment/Memory Window/Locate Sets Entry WidthBoolean	Set the memory window entry width if possible when locating.
Locate Sets Size Environment/Memory Window/Locate Sets SizeBoolean	Set the memory window size when locating.
Scroll Wheel Modifies Start Address Environment/Memory Window/Scroll Wheel Modifies Start AddressBoolean	Selects whether the mouse scroll wheel can change the memory window start address.

Outline

Property	Description
Group Top-Level Declarations Windows/Outline/Group Top Level ItemsBoolean	Group consecutive top-level variable and type declarations.
Show Function Arguments Windows/Outline/Show Function ArgsBoolean	Show function arguments.

Project Explorer

Property	Description
Add Filename Replace Macros Environment/Project Explorer/Filename Replace MacrosStringList	Macros (system and global) used to replace the start of a filename on project file addition.
Check Solution Target Environment/Project Explorer/Check Solution TargetBoolean	Specifies whether to check target is correct when loading a solution.
Color Project Nodes Environment/Project Explorer/Color NodesBoolean	Show the project nodes colored for identification in the Project Explorer.

Confirm Configuration Folder Delete Project Explorer/Confirm Configuration Folder DeleteBoolean	Display a confirmation prompt before deleting a configuration folder containing properties.
Confirm File Replacement Warning Project Explorer/Confirm File Replacement WarningBoolean	Display a confirmation prompt before replacing project files for import and creation
Confirm Forget Modified Properties Project Explorer/Confirm Reject Property ChangesBoolean	Display a confirmation prompt before forgetting property modifications.
Context Menu Uses Common Folder Environment/Project Explorer/Context Menu Common FolderBoolean	Controls how common options are displayed by the Project Explorer's context menu.
Edit Properties At Top Environment/Project Explorer/Context Menu Properties PositionBoolean	Controls where edit properties is displayed by the Project Explorer's context menu.
External Editor Environment/Project Explorer/External EditorFileName	The file name of the application to use as the external text editor. The external editor is started by holding down the Shift key when opening files from the project explorer.
Favorite Properties Environment/Project Explorer/Favorite PropertiesStringList	The favorite list of properties that are displayed starred and before other properties in the Project Explorer.
Highlight Dynamic Items Environment/Project Explorer/Show Dynamic OverlayBoolean	Show an overlay on an item if it is populated from a dynamic folder.
Highlight External Items Environment/Project Explorer/Show Non-Local OverlayBoolean	Show an overlay on an item if it is not held within the project directory.
Output Files Folder Environment/Project Explorer/Show Output FilesBoolean	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 HandlingBoolean	Configures whether read-only data contributes to the Code or Data statistic.
Show Dependencies Environment/Project Explorer/Dependencies DisplayEnumeration	Controls how the dependencies are displayed.
Show Favorite Properties Environment/Project Explorer/Context Menu Show FavoritesBoolean	Controls if favorite properties are displayed by the Project Explorer's context menu.
Show File Count on Folder Environment/Project Explorer/Count FilesBoolean	Show the number of files contained in a folder as a badge in the Project Explorer.

Show Modified Properties on Folder/File Environment/Project Explorer/Show Modified PropertiesBoolean	Show if a folder or file has modified properties as a badge in the Project Explorer.
Show Project Count on Solution Environment/Project Explorer/Count ProjectsBoolean	Show the number of projects contained in a solution as a badge in the Project Explorer.
Show Properties Environment/Project Explorer/Properties DisplayEnumeration	Controls how the properties are displayed.
Show Source Control Annotation Environment/Project Explorer/Show Source Control AnnotationBoolean	Annotate items in the project explorer with their source control status.
Show Statistics Rounded Environment/Project Explorer/Statistics FormatBoolean	Show exact or rounded sizes in the project explorer.
Source Control Status Column Environment/Project Explorer/Show Source Control ColumnBoolean	Show the source control status column in the project explorer.
Starred Files Names Environment/Project Explorer/Starred File NamesStringList	The list of wildcard-matched file names that are highlighted with stars, to bring attention to themselves, in the Project Explorer.
Statistics Column Environment/Project Explorer/Statistics DisplayBoolean	Show the code and data size columns in the Project Explorer.
Synchronize Explorer With Editor Environment/Project Explorer/Sync EditorBoolean	Synchronizes the Project Explorer with the document being edited.
Use Common Properties Folder Environment/Project Explorer/Common Properties DisplayBoolean	Controls how common properties are displayed.

Properties

Property	Description
Enable Favorites Group Environment/Properties Windows/Favorites GroupedEnumeration	Assign favorites to their own group.
Properties Displayed Environment/Properties Windows/Property Display FormatEnumeration	Set how the properties are displayed.

Public Setting Check Environment/Properties Windows/Public Setting CheckEnumeration	Warn when setting property in public configuration.
Show Property Details Environment/Properties Windows/Show DetailsBoolean	Show or hide the property description.

Registers 1

Property	Description
Show Digit Separator Environment/Registers1Window/Show Digit SeparatorBoolean	Show digit separator in register value display.
Show Register Address Column Environment/Registers1Window/Show Address ColumnBoolean	Controls whether the register address column is displayed.

Registers 2

Property	Description
Show Digit Separator Environment/Registers2Window/Show Digit SeparatorBoolean	Show digit separator in register value display.
Show Register Address Column Environment/Registers2Window/Show Address ColumnBoolean	Controls whether the register address column is displayed.

Registers 3

Property	Description
Show Digit Separator Environment/Registers3Window/Show Digit SeparatorBoolean	Show digit separator in register value display.
Show Register Address Column Environment/Registers3Window/Show Address ColumnBoolean	Controls whether the register address column is displayed.

Registers 4

Property	Description
----------	-------------

Show Digit Separator Environment/Registers4Window/Show Digit SeparatorBoolean	Show digit separator in register value display.
Show Register Address Column Environment/Registers4Window/Show Address ColumnBoolean	Controls whether the register address column is displayed.

Source Navigator

Property	Description
Group Symbols Windows/Source Navigator/Group SymbolsBoolean	Group symbols by type.
Scope Windows/Source Navigator/ScopeEnumeration	Specifies whether to search for symbols in the entire workspace or only the current document.

Symbol Browser

Property	Description
Code Field Environment/Symbol Browser/Display CodeBoolean	Selects whether the Code field is displayed.
Const Field Environment/Symbol Browser/Display ConstBoolean	Selects whether the Const field is displayed.
Data Field Environment/Symbol Browser/Display DataBoolean	Selects whether the Data field is displayed.
Frame Size Field Environment/Symbol Browser/Display Frame SizeBoolean	Selects whether the Frame Size field is displayed.
Range Field Environment/Symbol Browser/Display RangeBoolean	Selects whether the Range field is displayed.
Section Field Environment/Symbol Browser/Display SectionBoolean	Selects whether the Section field is displayed.
Size Field Environment/Symbol Browser/Display SizeBoolean	Selects whether the Size field is displayed.

Sort Criteria Environment/Symbol Browser/ GroupingEnumeration	Selects how to sort or group the symbols displayed.
Type Field Environment/Symbol Browser/Display TypeBoolean	Selects whether the Type field is displayed.
Value Field Environment/Symbol Browser/Display ValueBoolean	Selects whether the Value field is displayed.

Terminal Emulator

Property	Description
Backscroll Buffer Lines Terminal Emulator/Backscroll Buffer LinesIntegerRange	The number of lines you can see when you scroll backward in the terminal emulator window.
Baud Rate Terminal Emulator/Communications/Baud RateEnumeration	Baud rate used when transmitting and receiving data.
Data Bits Terminal Emulator/Communications/Data BitsEnumeration	Number of data bits to use when transmitting and receiving data.
Flow Control Terminal Emulator/Communications/Flow ControlEnumeration	The flow control method to use.
Line Feed On Carriage Return Terminal Emulator/Line Feed On Carriage ReturnBoolean	Append a line feed character when a carriage return character is received.
Local Echo Terminal Emulator/Local EchoBoolean	Displays every character typed before sending to the remote computer.
Maximum Input Block Size Terminal Emulator/Maximum Input Block SizeIntegerRange	The maximum number of bytes to read at a time.
Parity Terminal Emulator/Communications/ ParityEnumeration	Parity used when transmitting and receiving data.
Port Terminal Emulator/Communications/ PortCOMPort	The communications port to use, e.g. /dev/ttyS0, /dev/ttyS1, etc.
Port Used By Target Interface Terminal Emulator/Communications/Port Used By Target InterfaceBoolean	The COM port will be disconnected when the target interface is connected and reconnected when the target interface is disconnected.

Set DTR Terminal Emulator/Communications/ DTRBoolean	Set the DTR signal.
Stop Bits Terminal Emulator/Communications/Stop BitsEnumeration	Number of stop bits to use when transmitting data.

Watch 1

Property	Description
Show Digit Separator Environment/Watch1Window/Show Digit SeparatorBoolean	Show digit separator in variable value display.
Show Member Functions Environment/Watch1Window/Show Member FunctionsBoolean	Controls whether C++ class member functions are displayed.
Show Variable Address Column Environment/Watch1Window/Show Address ColumnBoolean	Controls whether the variable address column is displayed.
Show Variable Size Column Environment/Watch1Window/Show Size ColumnBoolean	Controls whether the variable size column is displayed.
Show Variable Type Column Environment/Watch1Window/Show Type ColumnBoolean	Controls whether the variable type column is displayed.

Watch 2

Property	Description
Show Digit Separator Environment/Watch2Window/Show Digit SeparatorBoolean	Show digit separator in variable value display.
Show Member Functions Environment/Watch2Window/Show Member FunctionsBoolean	Controls whether C++ class member functions are displayed.
Show Variable Address Column Environment/Watch2Window/Show Address ColumnBoolean	Controls whether the variable address column is displayed.
Show Variable Size Column Environment/Watch2Window/Show Size ColumnBoolean	Controls whether the variable size column is displayed.

Show Variable Type Column Environment/Watch2Window/Show Type ColumnBoolean	Controls whether the variable type column is displayed.
---	---

Watch 3

Property	Description
Show Digit Separator Environment/Watch3Window/Show Digit SeparatorBoolean	Show digit separator in variable value display.
Show Member Functions Environment/Watch3Window/Show Member FunctionsBoolean	Controls whether C++ class member functions are displayed.
Show Variable Address Column Environment/Watch3Window/Show Address ColumnBoolean	Controls whether the variable address column is displayed.
Show Variable Size Column Environment/Watch3Window/Show Size ColumnBoolean	Controls whether the variable size column is displayed.
Show Variable Type Column Environment/Watch3Window/Show Type ColumnBoolean	Controls whether the variable type column is displayed.

Watch 4

Property	Description
Show Digit Separator Environment/Watch4Window/Show Digit SeparatorBoolean	Show digit separator in variable value display.
Show Member Functions Environment/Watch4Window/Show Member FunctionsBoolean	Controls whether C++ class member functions are displayed.
Show Variable Address Column Environment/Watch4Window/Show Address ColumnBoolean	Controls whether the variable address column is displayed.
Show Variable Size Column Environment/Watch4Window/Show Size ColumnBoolean	Controls whether the variable size column is displayed.
Show Variable Type Column Environment/Watch4Window/Show Type ColumnBoolean	Controls whether the variable type column is displayed.

Windows

Property	Description
Buffer Grouping Environment/Windows/GroupingEnumeration	How the files are grouped or listed in the Windows window.
Show File Path as Tooltip Environment/Windows/Show Filename TooltipsBoolean	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 SizesBoolean	Show the number of lines and size of each file in the windows list.

Code Options

Assembler

Property	Description
Additional Assembler Options asm_additional_optionsStringList	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_fileProjFileName	Enables additional options to be supplied to the assembler from a file. This property will have macro expansion applied to it.
Assembler arm_assembler_variantEnumeration	Specifies which assembler to use.
Backup Additional Assembler Options asm_additional_options_backupString	Value of additional assembler options prior to generic options processing.
Run Preprocessor arm_preprocess_assembly_codeBoolean	The assembly code file is preprocessed before assembly

Build

Property	Description
Always Rebuild build_always_rebuildBoolean	Specifies whether or not to always rebuild the project/folder/file.
Batch Build Configurations batch_build_configurationsStringList	The set of configurations to batch build.
Build Options Generic File Name build_generic_options_file_nameProjFileName	The file name containing the generic options.
Build Quietly build_quietlyBoolean	Suppress the display of startup banners and information messages.
Compilation Database File compilation_database_fileFileName	The name of the compilation database file
Dependency File Name build_dependency_file_nameFileName	The file name to contain the dependencies.
Enable Unused Symbol Removal build_remove_unused_symbolsBoolean	Enable the removal of unused symbols from the executable.
Exclude From Build build_exclude_from_buildBoolean	Specifies whether or not to exclude the project/folder/file from the build.
GCC Prefix gcc_prefixString	The string that is prepended to the gcc toolname e.g arm-none-eabi-. The macro \$(GCCPrefix) is set to this value for external build command lines.

GCC Target gcc_targetString	The macro \$(GCCTarget) is set to this value for build command lines.
GCC Version gcc_versionString	The macro \$(GCCVersion) is set to this value for build command lines.
Generate Compilation Database generate_compilation_database_fileBoolean	Generate a JSON formatted file named by the Compilation Database property.
Generate Dependency File build_generate_dependency_fileEnumeration	Generate a dependency file
Include Debug Information build_debug_informationBoolean	Specifies whether symbolic debug information is generated.
Inputs File inputs_fileFileName	Specifies the inputs file to be used for Linking/ Archiving. The files listed in this file will be used rather than the outputs of the project.
Intermediate Directory build_intermediate_directoryDirPath	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.
Is C++ Project is_cpp_projectEnumeration	Supply C++ include directories and libraries to the project build.
Object File Name build_object_file_nameFileName	Specifies a name to override the default object file name.
Output Directory build_output_directoryDirPath	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.
Preprocess Output File Name build_preprocess_output_file_nameFileName	Specifies a name to override the default preprocess output file name.
Project Dependencies project_dependenciesStringList	Specifies the projects the current project depends upon.
Project Directory project_directoryString	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 macrosStringList	Specifies macro values which are expanded in project properties and for file names in Common configuration only. Each macro is defined as name=value and are separated by ;.
Project Type project_typeEnumeration	Specifies the type of project to build. The options are Executable, Library, Object file, Staging, Combining, Externally Built Executable, Externally Built Library, Externally Built Object file, Preprocess.

Property Groups File property_groups_file_pathProjFileName	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_directoryDirPath	Allows a common root output directory to be specified that can be referenced using the \$(RootOutDir) macro.
Suppress Warnings build_suppress_warningsBoolean	Don't report warnings.
Toolchain Directory build_toolchain_directoryDirPath	Specify the root of the toolchain directory. This property will have macro expansion applied to it. The macro \$(ToolChainDir) is set to this value.
Treat Warnings as Errors build_treat_warnings_as_errorsBoolean	Treat all warnings as errors.
Use External GCC use_external_gccBoolean	The build will issue gcc commands.

Code Analyzer

Property	Description
Analyze After Compile analyze_after_compileBoolean	Run the static code analyzer after compile
Analyze Command analyze_commandCommandLine	The command to execute for the Analyze action. This property will have macro expansion applied to it with the additional macros: \$(DEFINES) contains a space separated list of preprocessor definitions as set in the Preprocessor Definitions property. \$(INCLUDES) contains a space separated list of user include directories as set in the User Include Directories property.
Analyze Command Options C analyze_command_c_optionsStringList	Options to supply to the analyze command for C source files.
Analyze Command Options C++ analyze_command_cpp_optionsStringList	Options to supply to the analyze command for C++ source files.
Clang Tidy Checks C clang_tidy_checks_cStringList	Checks to supply to clang-tidy for C source files.
Clang Tidy Checks C++ clang_tidy_checks_cppStringList	Checks to supply to clang-tidy for C++ source files.

Code Generation

Property	Description
----------	-------------

ARM Advanced SIMD Auto Vectorize arm_advanced_SIMD_auto_vectorizeBoolean	Enable automatic code generation for Advanced SIMD.
ARM Advanced SIMD Type arm_advanced_SIMD_typeEnumeration	Specifies the Advanced SIMD type to generate code for. The options are: NEON - Cortex-A based processors
ARM Architecture arm_architectureEnumeration	<p>Specifies the version of the instruction set to generate code for. The options are:</p> <ul style="list-style-type: none"> v4T - ARM7TDMI and ARM920T processors v5TE - ARM9E, Feroceon and XScale processors v6 - ARM11 processors v6M - Cortex-M0/M1 processors v7M - Cortex-M3 processors v7EM - Cortex-M4/M7 processors v7R - Cortex-R4/R5/R8 processors v7A - Cortex-A5/A7/A8/A9/A17 processors v8R - Cortex-R52 processors v8A - Cortex-A32/A35/A53/A55/A57/A72/A73/A75 processors v8M_Baseline - Cortex M23 processor v8M_Mainline - Cortex M33 processor v8.1M_Mainline - Cortex-M55/M85 processors None <p>The corresponding preprocessor definitions:</p> <ul style="list-style-type: none"> __ARM_ARCH_4T__ __ARM_ARCH_5TE__ __ARM_ARCH_6__ __ARM_ARCH_6M__ __ARM_ARCH_7M__ __ARM_ARCH_7EM__ __ARM_ARCH_7R__ __ARM_ARCH_7A__ __ARM_ARCH_8R__ __ARM_ARCH_8A__ __ARM_ARCH_8M_BASELINE__ __ARM_ARCH_8M_MAINLINE__ __ARM_ARCH_81M_MAINLINE__ <p>are defined.</p>

<p>ARM Core Type arm_core_typeEnumeration</p>	<p>Specifies the core to generate code for. The options are:</p> <p>ARM7TDMI, ARM7TDMI-S, ARM720T ARM920T, ARM946E-S, ARM966E-S, ARM968E-S, ARM926EJ-S ARM1136J-S, ARM1136JF-S, ARM1176JZ-S, ARM1176JZF-S Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M23, Cortex-M3, Cortex-M33, Cortex-M4, Cortex-M55, Cortex-M7 Cortex-R4, Cortex-R4F, Cortex-R5, Cortex-R7, Cortex-R8 Cortex-R52 Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A15, Cortex-A17 Cortex-A32, Cortex-A35, Cortex-A53, Cortex-A55, Cortex-A57, Cortex-A72, Cortex-A73, Cortex-A75 XScale None</p> <p>If this property is set to None then the architecture property is used</p>
<p>ARM FP ABI Type arm_fp_abiEnumeration</p>	<p>Specifies the FP ABI type to generate code for. The options are:</p> <p>Soft generate calls to the 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.</p>

<p>ARM FPU Type</p> <p>arm_fpu_typeEnumeration</p>	<p>Specifies the FPU type to generate code for. The options are:</p> <p>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 VFPv4-D32 - Cortex-A/Cortex-R based processors VFPv4-D16 - Cortex-A/Cortex-R based processors FPv4-SP-D16 - Cortex-M4 processors FPv5-SP-D16 - Cortex-M7/M33/R52 processors FPv5-D16 - Cortex-M7/M55 processors FP-ARMv8 - Cortex-A/Cortex-R processors</p> <p>The corresponding preprocessor definitions:</p> <p>__ARM_ARCH_VFP__ __ARM_ARCH_VFP3_D32__ __ARM_ARCH_VFP3_D16__ __ARM_ARCH_VFP4_D32__ __ARM_ARCH_VFP4_D16__ __ARM_ARCH_FPV4_SP_D16__ __ARM_ARCH_FPV5_SP_D16__ __ARM_ARCH_FPV5_D16__ __ARM_ARCH_FP_ARMv8__</p> <p>are defined.</p>
<p>ARM/Thumb Interworking</p> <p>arm_interworkEnumeration</p>	<p>Specifies whether ARM/Thumb interworking code should be generated. Setting this property to No may result in smaller code sizes when compiling for architecture v4T.</p>
<p>Additional C++ Modules</p> <p>gcc_additional_modulesStringList</p>	<p>Add additional C++ Modules to the module mapper file of the form name=filename.</p>
<p>Byte Order</p> <p>arm_endianEnumeration</p>	<p>Specify the byte order of the target processor. The options are:</p> <p>Little little endian code and data. Big big endian code and data. BE-8 little endian code and big endian data. None do not specify the endian.</p>
<p>CM0/CM0+/CM1 Has Small Multiplier</p> <p>arm_cm0_has_small_multiplierBoolean</p>	<p>The CM0/CM0+/CM1 core has the small multiplier.</p>
<p>Code Model.</p> <p>arm64_code_modelEnumeration</p>	<p>Specify the code model to generate code for.</p>
<p>Data Model.</p> <p>arm64_abiEnumeration</p>	<p>Specify the data model to generate code for.</p>

Debugging Level <code>gcc_debugging_level</code> Enumeration	Specifies the level of debugging information to generate. The options are: None - no debugging information Level 1 - backtrace and line number debugging information Level 2 - Level 1 and variable display debugging information Level 3 - Level 2 and macro display debugging information
Disable Function Inlining <code>gcc_disable_function_inlining</code> Boolean	Disable auto inlining of functions when optimization enables this.
Dwarf Version <code>gcc_dwarf_version</code> Enumeration	Specifies the version of Dwarf debugging information to generate.
Enable Coroutine Support <code>gcc_enable_coroutines</code> Boolean	Specifies whether coroutine support is enabled for C++ programs.
Enable Exception Support <code>cpp_enable_exceptions</code> Enumeration	Specifies whether exception support is enabled for C++ programs.
Enable Modules Support <code>gcc_enable_modules</code> Boolean	Specifies whether modules support is enabled for C++ programs.
Enable Precompiled Header File <code>gcc_enable_precompiled_header</code> Boolean	Enable use of a precompiled header file for the project.
Enable RTTI Support <code>cpp_enable_rtti</code> Enumeration	Specifies whether RTTI support is enabled for C++ programs.
Enable Stack Overflow Prevention <code>stack_overflow_prevention</code> Boolean	Enable Stack Overflow Prevention. For more information read: https://wiki.segger.com/Stack_Overflow_Prevention
Enable Use Of __cxa_atexit <code>gcc_use_cxa_at_exit</code> Boolean	Enable compiler usage of __cxa_atexit.
Enumeration Size <code>gcc_short_enum</code> Enumeration	Select between minimal container sized enumerations and int sized enumerations.
FP16 Format. <code>arm_fp16_format</code> Enumeration	The format of 16-bit floating point numbers.
Generate Dwarf Debug Types <code>gcc_dwarf_generate_debug_types</code> Boolean	Generate Dwarf .debug_types section.
Generate Dwarf Pubnames <code>gcc_dwarf_generate_pubnames</code> Boolean	Generate Dwarf .debug_pubnames and .debug_pubtypes sections.
Generate Listing File <code>asm_generate_listing_file</code> Boolean	An source/assembler listing file is generated which can be found in the output files folder
Instruction Set <code>arm_instruction_set</code> Enumeration	Specifies the instruction set to generate code for.

Instrument Functions arm_instrument_functionsBoolean	Specifies whether instrumentation calls are generated for function entry and exit.
Is C++ Module is_cpp_moduleEnumeration	The file contains an importable C++ module unit.
Keep Link Time Optimization Intermediate Files link_keep_lto_filesBoolean	Specifies whether to keep the link time optimization resolution and object files.
Link Time Optimization link_time_optimizationBoolean	Specifies whether the project should be built for optimization at link time.
Link Time Optimization Additional Options lto_additional_optionsStringList	Enables additional options to be supplied to the link time optimization process
Long Calls arm_long_callsBoolean	Specifies whether function calls are made using absolute addresses.
Machine Outliner [clang] clang_machine_outlinerEnumeration	Select machine outliner mode. An optimization that reduces code size by identifying identical code sequences across functions and replaces them with a call to a function which contains the identical code sequence.
Math Errno arm_math_errnoEnumeration	Set errno after calling math functions that are executed with a single instruction, e.g., sqrt.
Merge Globals [clang] clang_merge_globalsBoolean	Select whether global declarations are merged. This may reduce code size and increase execution speed for some applications. However, if functions are not used in an application and are eliminated by the linker, merged globals may increase the data size requirement of an application.
No COMMON gcc_no_commonEnumeration	Don't put globals in the common section
Omit Frame Pointer gcc_omit_frame_pointerEnumeration	Specifies whether a frame pointer register is omitted if not required.
Optimization Level gcc_optimization_levelEnumeration	Specifies the optimization level to use. The options are: None - don't specify an optimization level Debug - optimize debug experience. Level 0 - no optimization, fastest compilation and best debug experience. Level 1 - optimize minimally. Level 2 - optimize more. Level 3 - optimize even more, will take longer to compile and may produce much larger code. Optimize For Size Optimize For More Size
Precompiled Header File gcc_precompiled_headerBoolean	The precompiled header file for the project.

Relocation Model [clang] clang_relocation_modelEnumeration	Select relocation model.
Signed Char gcc_signed_charEnumeration	The char type is considered to be signed char.
Stack Sizes generate_stack_sizesBoolean	Generate stack sizes section
TLS Model. arm_tls_modelEnumeration	Thread local storage model.
Unaligned Access Support. arm_unaligned_accessEnumeration	Unaligned word and half-words can be accessed. The options are: Yes enable unaligned word and half-words. No disable unaligned word and half-words. Auto disable unaligned word and half-word access for v4T/v5TE/v6M/v8M_Baseline architectures, enable for others.
Unwind Tables arm_unwind_tablesBoolean	Generate unwind tables for C code.
Use Builtins arm_use_builtinsEnumeration	Use built-in library functions e.g. scanf.
Vector Extension arm_v81M_mve_typeEnumeration	Specifies the vector extension type to generate code for. The options are: MVE - integer instructions MVE.FP - integer and single precision floating-point instructions
Wide Character Size gcc_wchar_sizeEnumeration	Select between standard 32-bit or shorter 16-bit size for wide characters and wchar_t.
v7A/v7R Has Integer Divide Instructions arm_v7_has_divide_instructionsBoolean	The v7A architecture has integer divide instructions in both ARM and Thumb instruction sets. The v7R architecture has integer divide instructions in the ARM instruction set. The v7R architecture always has integer divide instructions in the Thumb instruction set.
v8.1M Has PACBTI Instructions arm_v81M_has_pacbtiBoolean	The v8.1M architecture has PACBTI instructions.
v8A Has CRC Instructions arm_v8A_has_crcBoolean	The v8A architecture has CRC instructions.
v8A Has Crypto Instructions arm_v8A_has_cryptoBoolean	The v8A architecture has crypto instructions.
v8M Has CMSE Instructions arm_v8M_has_cmseBoolean	The v8M architecture has CMSE instructions.
v8M Has DSP Instructions arm_v8M_has_dspBoolean	The v8M architecture has DSP instructions.

Combining

Property	Description
Combine Command <code>combine_commandCommandLine</code>	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 <code>combine_command_wdString</code>	The working directory in which the combine command is run. This property will have macro expansion applied to it.
Output File Path <code>combine_output_filepathString</code>	The output file path the stage command will create. This property will have macro expansion applied to it.
Set To Read-only <code>combine_set_readonlyEnumeration</code>	Set the output file to read only or read/write.

Compiler

Property	Description
Additional C Compiler Only Options <code>c_only_additional_optionsStringList</code>	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 <code>c_only_additional_options_from_fileProjFileName</code>	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 <code>cpp_only_additional_optionsStringList</code>	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 <code>cpp_only_additional_options_from_fileProjFileName</code>	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++ Assembler Options <code>c_asm_additional_optionsStringList</code>	Enables additional options to be supplied to the assembler when used by the C/C++ compiler. This property will have macro expansion applied to it.
Additional C/C++ Compiler Options <code>c_additional_optionsStringList</code>	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 <code>c_additional_options_from_fileProjFileName</code>	Enables additional options to be supplied to the C/C++ compiler from a file. This property will have macro expansion applied to it.

Backup Additional C Compiler Only Options c_only_additional_options_backupString	Value of additional C compiler options prior to generic options processing
Backup Additional C++ Compiler Only Options cpp_only_additional_options_backupString	Value of additional C++ compiler options prior to generic options processing
Backup Additional Compiler Options c_additional_options_backupString	Value of additional compiler options prior to generic options processing
C Language Standard gcc_c_language_standardEnumeration	Specifies the language standard to use when compiling C files. The options are: None - don't specify a language standard c89/gnu89 c90/gnu90 c99/gnu99 c11/gnu11 c17/gnu17
C++ Language Standard gcc_cplusplus_language_standardEnumeration	Specifies the language standard to use when compiling C files. The options are: None - don't specify a language standard c++98/gnu++98 c++11/gnu++11 c++14/gnu++14 c++20/gnu++20 c++17/gnu++17
Color Diagnostics compiler_color_diagnosticsEnumeration	Specifies whether to enable color diagnostic output.
Compile C Files As C++ c_files_are_cppBoolean	Compile files that have the .c extension with the C++ compiler.
Compiler arm_compiler_variantEnumeration	Specifies which compiler to use.
Compiler Has -Oz gcc_has_Oz_optimization_levelBoolean	The compiler support the -Oz optimization level.
Enable All Warnings gcc_enable_all_warningsBoolean	Enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
Enable All Warnings C Compiler Only Command Line Options gcc_c_only_all_warnings_command_line_optio	The command line options supplied to the C compiler when Enable All Warnings is enabled.
Enable All Warnings C++ Compiler Only Command Line Options gcc_cpp_only_all_warnings_command_line_opt	The command line options supplied to the C++ compiler when Enable All Warnings is enabled.
Enable All Warnings Command Line Options gcc_all_warnings_command_line_optionsStringL	The command line options supplied to the compiler when Enable All Warnings is enabled.

Enforce ANSI Checking c_enforce_ansi_checkingBoolean	Perform additional checks for ensure strict conformance to the selected ISO (ANSI) C or C++ standard.
Enforce ANSI Checking C Command Line Options gcc_c_only_enforce_ansi_checking_command_l	The command line options supplied to the C compiler when Enforce ANSI Checking is enabled.
Enforce ANSI Checking C++ Command Line Options gcc_cpp_only_enforce_ansi_checking_command	The command line options supplied to the C++ compiler when Enforce ANSI Checking is enabled.
Enforce ANSI Checking Command Line Options gcc_enforce_ansi_checking_command_line_opt	The command line options supplied to the compiler when Enforce ANSI Checking is enabled.
GNU Version [clang] clang_gnu_versionEnumeration	Specifies value of __GNU__ and related macros
Keep Assembly Source arm_keep_assemblyBoolean	Specifies whether assembly code generated by the compiler is kept.
Keep Preprocessor Output arm_keep_preprocessor_outputBoolean	Specifies whether preprocessor output generated by the compiler is kept.
Show Caret compiler_diagnostics_show_caretEnumeration	Specifies whether caret is displayed in compiler diagnostics.
Supply Absolute File Path arm_supply_absolute_file_pathBoolean	Specifies whether absolute file paths are supplied to the compiler.
Supply Execution Character Set compiler_supply_editor_execute_charsetBoole	Specifies whether to supply the editor file encoding as the execution character set.
Supply Input Character Set compiler_supply_editor_input_charsetBoolean	Specifies whether to supply the editor file encoding as the input character set.
Use Compiler Driver use_compiler_driverBoolean	The build will issue cc commands.

Compiler Warning

Property	Description
Main (-Wmain) gcc_main_warningEnumeration	Warn if the type of main is suspicious.
Main Return Type (-Wmain-return-type) [clang] gcc_main_return_typeEnumeration	Warn main return type is not int.
Uninitialized Variables (-Wuninitialized) gcc_uninitialized_variables_warningEnumerat	Warn uninitialized variables
Unused Variable (-Wunused-variable) gcc_unused_variable_warningEnumeration	Warn unused variable
Warn Missing Prototypes (-Wstrict-prototypes) gcc_strict_prototypes_warningEnumeration	Warn if a function is declared or defined without specifying the argument types.

Warn On Narrowing Conversion (-Wnarrowing) C++ Only <code>gcc_narrowing_warningEnumeration</code>	Warn when an implicit narrowing conversion occurs.
Warn Sign Compare (-Wsign-compare) <code>gcc_sign_compare_warningEnumeration</code>	Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.
Warning Level <code>WARNING_LEVELEnumeration</code>	Select a set of warnings based on level

External Build

Property	Description
Archive Command <code>external_archive_commandCommandLine</code>	<p>The command line to archive object files. This property will have macro expansion applied to it with the additional macros:</p> <p><code>\$(TargetPath)</code> contains the full file name of the Library File Name property <code>\$(RelTargePath)</code> contains the project directory relative file name of the Object File Name property. <code>\$(Objects)</code> a space seperated list of files to archive, generated from the source files of the project OR. <code>\$(ObjectsFilePath)</code> contains the full file name of the file containing the list of files to archive <code>\$(RelObjectsFilePath)</code> contains the project directory relative file name of the file containing the list of files to link</p>

<p>Assemble Command</p> <p>external_assemble_commandCommandLine</p>	<p>The command line to assemble an assembly source file. This property will have macro expansion applied to it with the additional macros:</p> <p>\$(TargetPath) contains the full file name of the Object File Name property.</p> <p>\$(RelTargetPath) contains the project directory relative file name of the Object File Name property.</p> <p>\$(AsmOptions) contains a space separated list of options as set in the Additional Assembler Options property.</p> <p>\$(DependencyPath) contains the filename of the .d file that is required to be output by the compilation for dependency support.</p> <p>\$(RelDependencyPath) contains the relative filename of the .d file that is required to be output by the compilation for dependency support.</p> <p>\$(Defines) contains a space separated list of preprocessor definitions as set in the Preprocessor Definitions property.</p> <p>\$(Undefines) contains a space separated list of preprocessor undefinitions as set in the Preprocessor Definitions property.</p> <p>\$(Includes) contains a space separated list of user include directories as set in the User Include Directories property.</p> <p>\$(IncludeFiles) contains a space separated list of include files as set in the Include Files property.</p>
<p>Build Command</p> <p>external_build_commandCommandLine</p>	<p>The command line to build the executable e.g. make. This property will have macro expansion applied to it.</p>

C Compile Command

external_c_compile_commandCommandLine

The command line to compile a C source file. This property will have macro expansion applied to it with the additional macros:

\$(TargetPath) contains the full file name of the **Object File Name** property.

\$(RelTargetPath) contains the project directory relative file name of the **Object File Name** property.

\$(COptions) contains a space separated list of options as set in the **C Additional C/C++ Compiler Options** property.

\$(COnlyOptions) contains a space separated list of options as set in the **C Additional C Compiler Only Options** property.

\$(DependencyPath) contains the filename of the .d file that is required to be output by the compilation for dependency support.

\$(RelDependencyPath) contains the relative filename of the .d file that is required to be output by the compilation for dependency support.

\$(Defines) contains a space separated list of preprocessor definitions as set in the **Preprocessor Definitions** property.

\$(Undefines) contains a space separated list of preprocessor undefinitions as set in the **Preprocessor Definitions** property.

\$(Includes) contains a space separated list of user include directories as set in the **User Include Directories** property.

\$(IncludeFiles) contains a space separated list of include files as set in the **Include Files** property.

C++ Compile Command

`external_cpp_compile_commandCommandLine`

The command line to compile a C++ source file. This property will have macro expansion applied to it with the additional macros:

\$(TargetPath) contains the full file name of the **Object File Name** property.

\$(RelTargetPath) contains the project directory relative file name of the **Object File Name** property.

\$(COptions) contains a space separated list of options as set in the **C Additional C/C++ Compiler Options** property.

\$(CppOnlyOptions) contains a space separated list of options as set in the **C Additional C++ Compiler Only Options** property.

\$(DependencyPath) contains the filename of the .d file that is required to be output by the compilation for dependency support.

\$(RelDependencyPath) contains the relative filename of the .d file that is required to be output by the compilation for dependency support.

\$(Defines) contains a space separated list of preprocessor definitions as set in the **Preprocessor Definitions** property.

\$(Undefines) contains a space separated list of preprocessor undefinitions as set in the **Preprocessor Definitions** property.

\$(Includes) contains a space separated list of user include directories as set in the **User Include Directories** property.

\$(IncludeFiles) contains a space separated list of include files as set in the **Include Files** property.

<p>C++ Link Command</p> <p>external_cpp_link_commandCommandLine</p>	<p>The command line to link an executable. This property will have macro expansion applied to it with the additional macros:</p> <p>\$(TargetPath) contains the full file name of the Executable File Name property.</p> <p>\$(RelTargePath) contains the project directory relative file name of the Executable File Name property.</p> <p>\$(LinkOptions) contains a space seperated list of options as set in the Additional Linker Options property.</p> <p>\$(Objects) a space seperated list of files to link, generated from the source files of the project and the outputs of any dependent projects OR.</p> <p>\$(ObjectsFilePath) contains the full file name of the file containing the list of files to link</p> <p>\$(RelObjectsFilePath) contains the project directory relative file name of the file containing the list of files to link</p> <p>\$(LinkerScriptPath) contains the full file name of the Linker Script File property.</p> <p>\$(RelLinkerScriptPath) contains the project directory relative file name of the Linker Script File property.</p> <p>\$(MapPath) contains the full file name of the required map file.</p> <p>\$(RelMapPath) contains the project directory relative file name of the required map file.</p>
<p>Clean Command</p> <p>external_clean_commandCommandLine</p>	<p>The command line to clean the executable e.g. make clean. This property will have macro expansion applied to it.</p>

<p>Link Command</p> <p>external_link_commandCommandLine</p>	<p>The command line to link an executable. This property will have macro expansion applied to it with the additional macros:</p> <p>\$(TargetPath) contains the full file name of the Executable File Name property.</p> <p>\$(RelTargePath) contains the project directory relative file name of the Executable File Name property.</p> <p>\$(LinkOptions) contains a space seperated list of options as set in the Additional Linker Options property.</p> <p>\$(Objects) a space seperated list of files to link, generated from the source files of the project and the outputs of any dependent projects OR.</p> <p>\$(ObjectsFilePath) contains the full file name of the file containing the list of files to link</p> <p>\$(RelObjectsFilePath) contains the project directory relative file name of the file containing the list of files to link</p> <p>\$(LinkerScriptPath) contains the full file name of the Linker Script File property.</p> <p>\$(RelLinkerScriptPath) contains the project directory relative file name of the Linker Script File property.</p> <p>\$(MapPath) contains the full file name of the required map file.</p> <p>\$(RelMapPath) contains the project directory relative file name of the required map file.</p>
<p>Objects File</p> <p>external_objects_file_nameCommandLine</p>	<p>The name of the file containing the list of files to archive or link, generated from the source files of the project. This property will have macro expansion applied to it. The macro \$(ObjectsFilePath) is set to this value.</p>

File

Property	Description
<p>File Encoding</p> <p>file_codecEnumeration</p>	<p>Specifies the encoding to use when reading and writing the file.</p>

File Name <code>file_nameString</code>	The name of the file. This property will have global macro expansion applied to it. The following macros are set based on the value: <code>\$(InputDir)</code> relative directory of file, <code>\$(InputName)</code> file name without directory or extension, <code>\$(InputFileName)</code> file name, <code>\$(InputExt)</code> file name extension, <code>\$(InputPath)</code> absolute path to the file name, <code>\$(RelInputPath)</code> relative path from project directory to the file name.
File Open Action <code>file_open_withEnumeration</code>	Specifies how to open the file when it is double clicked.
File Type <code>file_typeEnumeration</code>	The type of file. Default setting uses the file extension to determine file type.
Flag <code>file_flagEnumeration</code>	Flag which you can use to draw attention to important files in your project.

Folder

Property	Description
Dynamic Folder Directory <code>pathDirPath</code>	Dynamic folder directory specification - ; seperated directory names that will have global macro expansion applied to them.
Dynamic Folder Exclude <code>excludeStringList</code>	Dynamic folder exclude specification - ; seperated wildcards.
Dynamic Folder Filter <code>filterString</code>	Dynamic folder filter specification - ; seperated wildcards.
Dynamic Folder Recurse <code>recurseBoolean</code>	Dynamic folder recurse into subdirectories.
Unity Build Exclude Filter <code>unity_build_exclude_filterString</code>	The filter specification to exclude from the unity build - ; seperated wildcards.
Unity Build File Name <code>unity_build_file_nameFileName</code>	The file name created that #includes all files in the folder for the unity build.

General

Property	Description
Environment Variables <code>environment_variablesStringList</code>	Environment variables to set on solution load.
Inherited Configurations <code>inherited_configurationsStringList</code>	The list of configurations that are inherited by this configuration.

Library

Property	Description
Debug I/O Implementation <code>arm_link_debugio_type</code> Enumeration	Specifies which Debug I/O mechanism to use for I/O operations. Options are: Breakpoint: Hardware breakpoint instruction and memory locations are used DCC: ARM debug communication channel is used Memory Poll: Memory locations are polled
Exclude Default Library Helper Functions <code>link_use_multi_threaded_libraries</code> Boolean	Specifies whether to exclude default library helper functions.
Include Standard Libraries <code>link_include_standard_libraries</code> Boolean	Specifies whether the standard libraries should be linked into your application.
Library ARM Architecture <code>arm_library_architecture</code> Enumeration	Specifies the architecture variant of the library to link with. The default uses the ARM Architecture value
Library Extension Suffix <code>link_libext_suffix</code> String	Specifies a suffix to add to the \$(LibExt) macro
Library File Name <code>build_output_file_name</code> FileName	Specifies a name to override the default library file name.
Library Instruction Set <code>arm_library_instruction_set</code> Enumeration	Specifies the instruction set variant of the libraries to link with, Default will use the Instruction Set value.
Library Optimization <code>arm_library_optimization</code> Enumeration	Specifies whether to link with libraries optimized for speed or size.
Standard Libraries Configuration Prefix <code>link_standard_libraries_configuration_pref</code>	Specifies the prefix to prepend to the library build configuration.
Standard Libraries Directory <code>link_standard_libraries_directory</code> String	Specifies where to find the standard libraries

Linker

Property	Description
Additional Input Files <code>linker_additional_files</code> StringList	Enables additional object and library files to be supplied to the linker.
Additional Linker Options <code>linker_additional_options</code> StringList	Enables additional options to be supplied to the linker.
Additional Linker Options From File <code>linker_additional_options_from_file</code> ProjFileNa	Enables additional options to be supplied to the linker from a file.
Additional Linker Script Generator Options <code>arm_additional_mkld_options</code> StringList	Enables additional options to be supplied to the linker script generator.

Additional Output File Gap Fill Value arm_linker_additional_output_file_gap_fill	The value to fill gaps between sections in additional output file.
Additional Output Format linker_output_formatEnumeration	The format used when creating an additional linked output file. The options are: None do not create an additional output file. bin create a binary file. srec create a Motorola S-Record file. hex create an Intel Hex file.
Additional System Libraries linker_additional_system_librariesStringList	Enables additional system libraries to be supplied to the linker.
Allow Multiple Symbol Definition arm_linker_allow_multiple_definitionBoolean	Do not report error if the same symbol is defined more than once in object files/libraries.
Backup Additional Linker Options link_additional_options_backupString	Value of additional linker options prior to generic options processing
Breakpad Symbols Directory linker_breakpad_symbols_directoryString	Specifies location of the breakpad symbols directory.
CMSE Import Library File arm_linker_cmse_import_library_file_nameFile	Specifies the name of the CMSE import library to generate.
Check CMSE Import Library File arm_linker_check_cmse_import_library_file	Specifies the name of the file to check the generated CMSE import library with.
Check For Memory Section Overflow arm_library_check_memory_section_overflowBoolean	Specifies whether the linker should check whether program sections exceed their specified size.
Check For Memory Segment Overflow arm_library_check_memory_segment_overflowBoolean	Specifies whether the linker should check whether program sections fit in their memory segments.
Default Fill Pattern arm_linker_script_generator_default_fill_pattern	Specifies the default pattern used to fill unspecified regions of memory in a generated linker script. This pattern maybe overridden by the <i>fill</i> attribute of a program section in the section placement file.
Emit Relocations arm_linker_emit_relocationsBoolean	Output relocation information into the executable.
Entry Point gcc_entry_pointString	Specifies the entry point of the program. None will not supply an entry point to the linker.
Gap Fill Value arm_linker_gap_fillIntegerHex	The value to fill gaps between sections in ELF file. <i>This property has been deprecated, use Linker Options > Additional Output File Gap Fill Value instead.</i>
Generate Breakpad Symbols linker_generate_breakpad_symbolsBoolean	Specifies whether to generate breakpad symbols from the linked image.
Generate Linker Map File linker_map_fileBoolean	Specifies whether to generate a linkage map file.
Indirect File Supported linker_use_indirect_filesBoolean	Linker can use @indirect file for input files.

Keep Indirect Files linker_keep_indirect_filesBoolean	Keep generated linker indirect files.
Keep Linker Script File keep_linker_script_fileBoolean	Keep the generated linker script file.
Keep Symbols linker_keep_symbolsStringList	Specifies the symbols that should be kept by the linker even if they are not reachable.
Link Dependent Projects link_dependent_projectsBoolean	Specifies whether to link the output of dependent library projects.
Link Whole Archive arm_linker_whole_archiveStringList	List the archives that require to be linked in whole.
Linker Map File Name linker_map_file_nameFileName	The file name to contain the linkage map file.
Linker Script File link_linker_script_fileProjFileName	The name of the manual linker script file.
Linker Search Path arm_linker_search_pathStringList	Specify the linker script search path.
Linker Symbol Definitions link_symbol_definitionsStringList	Specifies one or more linker symbol definitions.
Memory Map File linker_memory_map_fileProjFileName	The name of the file containing the memory map description.
Memory Map Macros linker_memory_map_macrosStringList	Macro values to substitute in memory map nodes. Each macro is defined as name=value and are separated by ; .
Memory Segments linker_section_placements_segmentsString	The start, access and size of named segments in the target, these are used when no memory map file is available. Each segment is specified by NAME RWX HEXSTART HEXSIZE for example FLASH RX 0x08000000 0x00010000
No Enum Size Warning arm_linker_no_enum_size_warningBoolean	Do not generate warnings when object files have different ARM EABI enum size attributes.
No Start File arm_linker_no_start_filesBoolean	Do not use startup files when linking.
No Wide Char Size Warning arm_linker_no_wchar_size_warningBoolean	Do not generate warnings when object files have different ARM EABI wide character size attributes.
Section Placement File linker_section_placement_fileProjFileName	The name of the file containing section placement description.
Section Placement Macros linker_section_placement_macrosStringList	Macro values to substitute in section placement nodes - MACRO1=value1;MACRO2=value2.
Start/End Group Required linker_requires_start_groupBoolean	Linker requires --start-group and --end-group for input files.

Strip Debug Information <code>linker_strip_debug_information</code> Boolean	Specifies whether debug information should be stripped from the linked image.
Strip Symbols <code>gcc_strip_symbols</code> Boolean	Specifies whether symbols should be stripped.
Suppress Warning on Executable Stack <code>arm_linker_no_warn_on_executable_stack</code> Boolean	No warning on executable stack.
Suppress Warning on Mismatch <code>arm_linker_no_warn_on_mismatch</code> Boolean	No warning on mismatched object files/libraries.
Suppress Warning on RWX Segments <code>arm_linker_no_warn_on_rwx_segments</code> Boolean	No warning on RWX segments.
Symbols File <code>arm_linker_symbols_files</code> FileName	Specify the name of a symbols file to link.
Treat Libraries As Object Files <code>linker_treat_libraries_as_object_files</code> Boolean	Specifies whether the linker treats libraries as a set of object files.
Treat Linker Warnings as Errors <code>arm_linker_treat_warnings_as_errors</code> Boolean	Treat linker warnings as errors.
Use Manual Linker Script <code>link_use_linker_script_file</code> Boolean	Specifies whether to use a manual linker script.

Package

Property	Description
Package Dependencies <code>package_dependencies</code> StringList	Specifies the packages the current project depends upon.
Package Directory <code>package_directory</code> DirPath	Specifies the directory packages are installed to. If no directory is specified, the default package directory is used.

Preprocessor

Property	Description
Add Property Group Options <code>add_property_group_includes_defines</code> Boolean	Supply the defines and includes that are selected by the property group.
Ignore Includes <code>c_ignore_includes</code> Boolean	Ignore the include directories properties.
Include Files <code>c_include_files</code> StringList	Specifies the list of files to include before preprocessing. This property will have macro expansion applied to it.

Include Files Assembler Only <code>c_include_files_asm_onlyStringList</code>	Specifies the list of files to include before preprocessing. This property will have macro expansion applied to it.
Include Files C Compiler Only <code>c_include_files_c_onlyStringList</code>	Specifies the list of files to include before preprocessing. This property will have macro expansion applied to it.
Include Files C++ Compiler Only <code>c_include_files_cpp_onlyStringList</code>	Specifies the list of files to include before preprocessing. This property will have macro expansion applied to it.
Macro Files <code>c_macros_filesStringList</code>	Specifies the list of macro files to include before preprocessing. This property will have macro expansion applied to it.
Macro Files Assembler Only <code>c_macros_files_asm_onlyStringList</code>	Specifies the list of macro files to include before preprocessing. This property will have macro expansion applied to it.
Macro Files C Compiler Only <code>c_macros_files_c_onlyStringList</code>	Specifies the list of macro files to include before preprocessing. This property will have macro expansion applied to it.
Macro Files C++ Compiler Only <code>c_macros_files_cpp_onlyStringList</code>	Specifies the list of macro files to include before preprocessing. This property will have macro expansion applied to it.
Preprocessor Definitions <code>c_preprocessor_definitionsStringList</code>	Specifies one or more preprocessor definitions. This property will have macro expansion applied to it.
Preprocessor Definitions Assembler Only <code>c_preprocessor_definitions_asm_onlyStringList</code>	Specifies one or more preprocessor definitions. This property will have macro expansion applied to it.
Preprocessor Definitions C Compiler Only <code>c_preprocessor_definitions_c_onlyStringList</code>	Specifies one or more preprocessor definitions. This property will have macro expansion applied to it.
Preprocessor Definitions C++ Compiler Only <code>c_preprocessor_definitions_cpp_onlyStringList</code>	Specifies one or more preprocessor definitions. This property will have macro expansion applied to it.
Preprocessor Undefinitions <code>c_preprocessor_undefinitionsStringList</code>	Specifies one or more preprocessor undefinitions. This property will have macro expansion applied to it.
Preprocessor Undefinitions Assembler Only <code>c_preprocessor_undefinitions_asm_onlyStringList</code>	Specifies one or more preprocessor undefinitions. This property will have macro expansion applied to it.
Preprocessor Undefinitions C Compiler Only <code>c_preprocessor_undefinitions_c_onlyStringList</code>	Specifies one or more preprocessor undefinitions. This property will have macro expansion applied to it.
Preprocessor Undefinitions C++ Compiler Only <code>c_preprocessor_undefinitions_cpp_onlyStringList</code>	Specifies one or more preprocessor undefinitions. This property will have macro expansion applied to it.
System Include Directories <code>c_system_include_directoriesStringList</code>	Specifies the system include path. This property will have macro expansion applied to it.
Undefine All Preprocessor Definitions <code>c_undefine_all_preprocessor_definitionsBool</code>	Does not define any standard preprocessor definitions.

User Include Directories <code>c_user_include_directoriesStringList</code>	Specifies the user include path. This property will have macro expansion applied to it.
User Include Directories Assembler Only <code>c_user_include_directories_asm_onlyStringList</code>	Specifies the user include path. This property will have macro expansion applied to it.
User Include Directories C Compiler Only <code>c_user_include_directories_c_onlyStringList</code>	Specifies the user include path. This property will have macro expansion applied to it.
User Include Directories C++ Compiler Only <code>c_user_include_directories_cpp_onlyStringList</code>	Specifies the user include path. This property will have macro expansion applied to it.

Printf/Scanf

Property	Description
Printf Floating Point Supported <code>linker_printf_fp_enabledEnumeration</code>	Are floating point numbers supported by the printf function group.
Printf Integer Support <code>linker_printf_fmt_levelEnumeration</code>	The largest integer type supported by the printf function group.
Printf Width/Precision Supported <code>linker_printf_width_precision_supportedBoolean</code>	Enables support for width and precision specification in the printf function group.
Scanf Classes Supported <code>linker_scanf_character_group_matching_enabledBoolean</code>	Enables support for %[...] and %^[...] character class matching in the scanf functions.
Scanf Floating Point Supported <code>linker_scanf_fp_enabledBoolean</code>	Are floating point numbers supported by the scanf function group.
Scanf Integer Support <code>linker_scanf_fmt_levelEnumeration</code>	The largest integer type supported by the scanf function group.
Wide Characters Supported <code>linker_printf_wchar_enabledBoolean</code>	Are wide characters supported by the printf function group.

Project

Property	Description
Flag <code>project_flagEnumeration</code>	Flag which you can use to draw attention to important projects in your solution.

Runtime Memory Area

Property	Description
Heap Size <code>arm_linker_heap_sizeIntegerRange</code>	The size of the heap in bytes. The size must be a multiple of 8. The preprocessor define <code>__HEAP_SIZE__</code> is set to this value.

Main Stack Size arm_linker_stack_sizeIntegerRange	The size of the main stack in bytes. The size must be a multiple of 8.
Process Stack Size arm_linker_process_stack_sizeIntegerRange	The size of the process stack in bytes. The size must be a multiple of 8.
Stack Size (Abort Mode) arm_linker_abt_stack_sizeIntegerRange	The size of the Abort mode stack in bytes. The size must be a multiple of 8.
Stack Size (FIQ Mode) arm_linker_fiq_stack_sizeIntegerRange	The size of the FIQ mode stack in bytes. The size must be a multiple of 8.
Stack Size (IRQ Mode) arm_linker_irq_stack_sizeIntegerRange	The size of the IRQ mode stack in bytes. The size must be a multiple of 8.
Stack Size (Supervisor Mode) arm_linker_svc_stack_sizeIntegerRange	The size of the Supervisor mode stack in bytes. The size must be a multiple of 8.
Stack Size (Undefined Mode) arm_linker_und_stack_sizeIntegerRange	The size of the Undefined mode stack in bytes. The size must be a multiple of 8.

Section

Property	Description
Code Section Name default_code_sectionString	Specifies the default name to use for the program code section.
Constant Section Name default_const_sectionString	Specifies the default name to use for the read-only constant section.
Data Section Name default_data_sectionString	Specifies the default name to use for the initialized, writable data section.
ISR Section Name default_isr_sectionString	Specifies the default name to use for the ISR code.
Vector Section Name default_vector_sectionString	Specifies the default name to use for the interrupt vector section.
Zeroed Section Name default_zeroed_sectionString	Specifies the default name to use for the zero-initialized, writable data section.

Solution

Property	Description
Flag solution_flagEnumeration	Flag which you can use to draw attention to important projects in your solution.
Properties Filter properties_filterStringList	The names of project properties that can be displayed at the solution

Source Code

Property	Description
Additional Code Completion Compiler Options <code>code_completion_optionsStringList</code>	Additional source indexing and code completion compiler options.
Inhibit Source Indexing <code>project_inhibit_indexingBoolean</code>	Disable source indexing and code completion for files/folders/projects that would normally be indexed (C/C++ files in executable and library projects).
Source Code Control Directory <code>source_code_control_directoryDirPath</code>	Source code control directory root.

Staging

Property	Description
Output File Path <code>stage_output_filepathString</code>	The output file path the stage command will create. This property will have macro expansion applied to it.
Set To Read-only <code>stage_set_readonlyEnumeration</code>	Set the output file permissions to read only or read/write.
Stage Command <code>stage_commandCommandLine</code>	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 <code>stage_command_wdString</code>	The working directory in which the stage command is run. This property will have macro expansion applied to it.
Stage Project Command <code>stage_post_build_commandCommandLine</code>	The command to execute after staging commands have executed. This property will have macro expansion applied to it.
Stage Project Command Working Directory <code>stage_post_build_command_wdString</code>	The working directory where the post build command runs. This property will have macro expansion applied to it.

User Build Step

Property	Description
Link Patch Command <code>linker_patch_build_commandCommandLine</code>	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_wdDirPath	The working directory where the link patch command is run. This property will have macro expansion applied to it.
Post-Archive Command archive_post_build_commandCommandLine	A command to run after the archive command has completed. This property will have macro expansion applied to it with the additional \$(TargetPath) macro set to the output filepath of the archive command.
Post-Archive Working Directory archive_post_build_command_wdDirPath	The working directory where the post-archive command is run. This property will have macro expansion applied to it.
Post-Build Command post_build_commandCommandLine	The command to execute after a project build. This property will have macro expansion applied to it.
Post-Build Command Control post_build_command_controlEnumeration	Controls when the post-build command is run, either Always Run or when Run When Build Has Occurred .
Post-Build Command Working Directory post_build_command_wdString	The working directory in which the post-build command is run. This property will have macro expansion applied to it.
Post-Compile Command compile_post_build_commandCommandLine	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_wdDirPath	The working directory where the post-compile command is run. This property will have macro expansion applied to it.
Post-Link Command linker_post_build_commandCommandLine	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 and \$(PostLinkOutputFilePath) set to the value of the output filepath of the post link command.
Post-Link Output File linker_post_build_command_output_fileString	The name of the file created by the post-link command. This property will have macro expansion applied to it.
Post-Link Working Directory linker_post_build_command_wdDirPath	The working directory where the post-link command is run. This property will have macro expansion applied to it.
Pre-Build Command pre_build_commandCommandLine	The command to execute before a project build. This property will have macro expansion applied to it.
Pre-Build Command Control pre_build_command_controlEnumeration	Controls when the pre-build command is run, either Always Run or when Run When Build Required .
Pre-Build Command Working Directory pre_build_command_wdString	The working directory in which the pre-build command is run. This property will have macro expansion applied to it.

Pre-Compile Command <code>compile_pre_build_command</code> <code>CommandLine</code>	A command to run before the compile command. This property will have macro expansion applied to it.
Pre-Compile Command Output File Path <code>compile_pre_build_command_output_file_name</code>	The pre-compile generated file name. This property will have macro expansion applied to it.
Pre-Compile Working Directory <code>compile_pre_build_command_wdDirPath</code>	The working directory where the pre-compile command is run. This property will have macro expansion applied to it.
Pre-Link Command <code>linker_pre_build_command</code> <code>CommandLine</code>	A command to run before the link command. This property will have macro expansion applied to it.
Pre-Link Working Directory <code>linker_pre_build_command_wdDirPath</code>	The working directory where the pre-link command is run. This property will have macro expansion applied to it.

Debug Options

Debugger

Property	Description
Alternative LDR Disassembly debug_alternative_ldr_disBoolean	Show alternative disassembly of ldr*/vldr instructions
CPU Register File debug_cpu_registers_fileProjFileName	The name of the file containing CPU register definitions.
Command Arguments debug_command_argumentsString	The command arguments passed to the executable. This property will have macro expansion applied to it.
Debug Additional Configurations debug_additional_configurationsStringList	The debugger will load and debug the specified additional configurations.
Debug Additional Projects debug_dependent_projectsStringList	The debugger will load (if not already loaded by Load Additional Projects) and debug the specified additional projects.
Debug Project Name debug_project_nameString	The name of the project used by the debugger when debugging multiple projects
Debug Symbols File[0] external_debug_symbols_file_nameProjFileName	The name of the debug symbols file. This property will have macro expansion applied to it. If it is not defined then the main load file is used.
Debug Symbols File[1] external_debug_symbols_file_name1ProjFileName	The name of the debug symbols file. This property will have macro expansion applied to it. If it is not defined then the main load file is used.
Debug Symbols File[2] external_debug_symbols_file_name2ProjFileName	The name of the debug symbols file. This property will have macro expansion applied to it. If it is not defined then the main load file is used.
Debug Symbols File[3] external_debug_symbols_file_name3ProjFileName	The name of the debug symbols file. This property will have macro expansion applied to it. If it is not defined then the main load file is used.
Debug Symbols Load Address[0] external_debug_symbols_load_addressString	The (code) address to be added to the debug symbol (code) addresses.
Debug Symbols Load Address[1] external_debug_symbols_load_address1String	The (code) address to be added to the debug symbol (code) addresses.
Debug Symbols Load Address[2] external_debug_symbols_load_address2String	The (code) address to be added to the debug symbol (code) addresses.
Debug Symbols Load Address[3] external_debug_symbols_load_address3String	The (code) address to be added to the debug symbol (code) addresses.
Debug Terminal Log File debug_terminal_log_fileUnknown	A file to write the output from the debug terminal to.

Default debugIO implementation arm_debugIO_ImplementationEnumeration	The default debugIO implementation used by the debugger if symbols are unavailable.
Display DCC data arm_display_DCCBoolean	The debugger will display data that is written to the DCC when debugIO is not used.
Entry Point Symbol debug_entry_point_symbolString	Debugger will start execution at symbol if defined.
Flash Software Breakpoints arm_target_read_only_software_breakpointsEnumeration	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), and Dynamic (software breakpoints are set and cleared as required).
HTML Watch File debug_html_watchProjFileName	The file used by the debugger's HTML Watch window.
Has Hypervisor Mode arm_has_hypervisor_modeBoolean	Show hypervisor mode registers
Has Monitor Mode arm_has_monitor_modeBoolean	Show monitor mode registers
Has Vector Catch arm_has_vector_catchBoolean	Vector catching is supported
Ignore .debug_aranges Section debug_ignore_debug_arangesBoolean	The debugger will not use the .debug_aranges section.
Ignore .debug_frame Section debug_ignore_debug_frameBoolean	The debugger will not use the .debug_frame section.
Initial Breakpoint debug_initial_breakpointString	The initial breakpoint to set
Initial Breakpoint Is Set debug_initial_breakpoint_set_optionEnumeration	Specify when the initial breakpoint should be set
Leave Target Running debug_leave_target_runningBoolean	Debugger will leave the target running on debug stop.
Load Additional Projects debug_load_additional_projectsStringList	The debugger will load the outputs of the specified additional projects.
Memory Upload Page Size debug_memory_upload_page_sizeInteger	The aligned page size the debugger uses when uploading address ranges.
RAM Software Breakpoints arm_target_read_write_software_breakpointsEnumeration	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), and Dynamic (software breakpoints are set and cleared as required).
RTT Control Block Address debug_RTTControlBlockString	The symbol or 0x prefixed address of the RTT control block.

RTT Enable debug_enable_RTTBoolean	If enabled the debugger will service RTT input/output in the debug terminal.
Register Definition File debug_register_definition_fileProjFileName	The name of the file containing register definitions.
Register Definition File Type debug_register_definition_file_typeEnumerati	The type of the file containing register definitions.
Reserved Member Name reservedMember_nameString	The struct reserved member name. Struct members that contain the (case insensitive) string will not be displayed.
Restrict Memory Access debug_restrict_memory_accessBoolean	If enabled the debugger will only display variables and backtrace in the address ranges of the memory map or the sections in the elf file.
Start Address external_start_addressString	The address to start the externally built executable running from.
Start From Entry Point Symbol debug_start_from_entry_point_symbolEnumerati	If Yes the debugger will start execution from the entry point symbol. If No the debugger will start execution from the target specific location. If Don't the debugger will not start execution.
Starting Stack Pointer Value debug_stack_pointer_startString	The symbol or 0x prefixed value to set the stack pointer on start debugging.
Startup Completion Point debug_startup_completion_pointStringList	Specifies the point in the program where startup is complete. Software breakpoints and debugIO will be enabled after this point has been reached.
Target Device arm_target_device_nameString	The name of the device to connect to. The macro \$(Target) is substituted with the Target Processor project property value.
Thread Maximum debug_threads_maxIntegerRange	The maximum number of threads to display.
Threads Script File debug_threads_scriptProjFileName	The threads script used by the debugger.
Type Interpretation File debug_type_fileFileName	Specifies the type interpretation file to use.
Working Directory debug_working_directoryDirPath	The working directory for a debug session. This property will have macro expansion applied to it.

JTAG Chain

Property	Description
JTAG Data Bits After arm_linker_jtag_pad_post_drIntegerRange	Specifies the number of bits to pad the JTAG data register after the target.

JTAG Data Bits Before <code>arm_linker_jtag_pad_pre_drIntegerRange</code>	Specifies the number of bits to pad the JTAG data register before the target.
JTAG Instruction Bits After <code>arm_linker_jtag_pad_post_irIntegerRange</code>	Specifies the number of bits to pad the JTAG instruction register with the BYPASS instruction after the target.
JTAG Instruction Bits Before <code>arm_linker_jtag_pad_pre_irIntegerRange</code>	Specifies the number of bits to pad the JTAG instruction register with the BYPASS instruction before the target.

Loader

Property	Description
Additional Load File Address[0] <code>debug_additional_load_file_addressString</code>	The address to load the additional load file.
Additional Load File Address[1] <code>debug_additional_load_file_address1String</code>	The address to load the additional load file.
Additional Load File Address[2] <code>debug_additional_load_file_address2String</code>	The address to load the additional load file.
Additional Load File Address[3] <code>debug_additional_load_file_address3String</code>	The address to load the additional load file.
Additional Load File Type[0] <code>debug_additional_load_file_typeEnumeration</code>	The file type of the additional load file. The options are Detect, elf, bin, ihex, hex, tihex, srec .
Additional Load File Type[1] <code>debug_additional_load_file_type1Enumeration</code>	The file type of the additional load file. The options are Detect, elf, bin, ihex, hex, tihex, srec .
Additional Load File Type[2] <code>debug_additional_load_file_type2Enumeration</code>	The file type of the additional load file. The options are Detect, elf, bin, ihex, hex, tihex, srec .
Additional Load File Type[3] <code>debug_additional_load_file_type3Enumeration</code>	The file type of the additional load file. The options are Detect, elf, bin, ihex, hex, tihex, srec .
Additional Load File[0] <code>debug_additional_load_fileProjFileName</code>	Additional file to load on debug load. This property will have macro expansion applied to it.
Additional Load File[1] <code>debug_additional_load_file1ProjFileName</code>	Additional file to load on debug load. This property will have macro expansion applied to it.
Additional Load File[2] <code>debug_additional_load_file2ProjFileName</code>	Additional file to load on debug load. This property will have macro expansion applied to it.
Additional Load File[3] <code>debug_additional_load_file3ProjFileName</code>	Additional file to load on debug load. This property will have macro expansion applied to it.
Load ELF Address Limit <code>debug_load_file_offset_limitString</code>	Restrict the Load ELF Offset. The Load ELF Offset will not be added to addresses greater than or equal to this address.

Load ELF Offset debug_load_file_offsetString	The offset to add to the load addresses of the ELF programs. This offset is added to any absolute relocations of symbols (whose address is less than Load ELF Offset Limit) if the load file contains relocation sections.
Load ELF Sections debug_load_sectionsEnumeration	The debugger will load ELF sections rather than ELF programs.
Load File external_build_file_nameProjFileName	The name of the main load file. This property will have macro expansion applied to it. If it is not defined then the output filepath of the linker command is used.
Load File Address external_load_addressString	The address to download the main load file to.
Load File Type external_load_file_typeEnumeration	The file type of the main load file. The options are Detect, elf, bin, ihex, hex, tihex, srec .
No Load Sections target_loader_no_load_sectionsStringList	Names of (loadable) program sections or names of memory segments not to load.

Simulator

Property	Description
Max Instructions arm_simulator_max_instructionsString	Maximum number of instructions to execute before simulator is stopped.
Memory Simulation File arm_simulator_memory_simulation_filenameProj	Specifies the dll that simulates the memory system. This property will have macro expansion applied to it. If not specified then the default memory simulation will be used.
Memory Simulation Parameter arm_simulator_memory_simulation_parametersString	Parameter passed to the memory simulation. This property will have macro expansion applied to it. The format of this is specific to the memory simulation. The default memory simulation takes a list of ROM RAM;START;SIZE for example ROM;0x0;0x10000;RAM;0x20000000;0x1000 or a list of [name] RX RWX 'hex start address', 'hex size in bytes', 'default hex word value' for example RX 00000000, 10000000, FFFFFFFF;RWX 10000000, 10000000, CDCDCDCD .
Memory Simulation Parameter Macros arm_simulator_memory_simulation_parametersString	Macros to apply to the parameter passed to the memory simulation on creation. If null then the macro MemorySegments is set to the value of the address ranges specified by the project.
Stop On Branch . arm_simulator_stop_on_branch_dotBoolean	Stop when the simulator executes a b . instruction.

Stop On Memory Error arm_simulator_stop_on_read_writeEnumeration	Specifies the simulator behaviour when a memory error occurs.
Trace Buffer Size arm_simulator_num_trace_entriesInteger	The number of trace entries to store.

Target Control

Property	Description
ARM Debug Interface arm_target_debug_interface_typeEnumeration	Specifies the type of debug interface the target has. The options are: Default - Select debug interface based on CPU core type ARM7TDI - ARM7TDMI/ARM7TDMI-S/ARM720T ARM9TDI - ARM920T/ARM946E-S/ARM966E-S/ ARM968E-S/ARM926EJ-S ARM11 - ARM1136J-S/ARM1136JF-S/ARM1176JZ-S/ARM1176JZF-S XScale - PXA25x XScale7BitIR - PXA27x ADiv5 - Cortex-A/Cortex-M/Cortex-R Feroceon - Marvell ARM9E ADiv6 - SoC-600
Check Load Sections Fit Target Description target_check_load_sections_fitBoolean	Specifies whether load sections in the program match the memory segments described in the memory map.
Connect With Reset arm_target_connect_with_resetBoolean	Hold the target in hardware reset on connect and stops the target. This requires the nSRST signal to be connected and the target debug hardware to work when in reset.
Connect With Stop arm_target_connect_with_stopBoolean	Stops the target after connect.
Coprocessor Instruction Execution Address arm_target_coprocessor_execute_addressString	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_pathProjFileName	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_addressString	The address to load the debug handler.
Do Not Use bkpt Instruction arm_target_do_not_use_bkptBoolean	Specifies that the bkpt instructions should not be used when setting software breakpoints on ARM architectures that support the instruction.
Identify Target arm_target_identifyBoolean	Identify the target on connect.

Monitor Mode Debug <code>arm_target_monitor_mode_debug</code> Boolean	Specifies whether the debug handler is a monitor mode debug handler.
Monitor Mode Memory <code>arm_target_monitor_mode_memory</code> Boolean	Specifies whether to use monitor mode memory accesses.
Processor Stop Timeout <code>arm_target_processor_stop_timeout</code> IntegerRange	The timeout period for stopping the processor in milliseconds.
Restrict Memory Accesses <code>arm_target_restrict_memory_accesses</code> Boolean	Specifies whether memory accesses should be restricted to known memory segments and their associated access attributes.
Stop CPU Using DBGRQ <code>arm_target_stop_cpu_using_dbgrq</code> Boolean	Specifies whether the CPU should be stopped by asserting DBGRQ rather than by using breakpoints.
Target Interface Clock Speed <code>arm_target_interface_speed</code> IntegerRange	The maximum JTAG/SWD clock frequency in Hz.
Target Interface Type <code>arm_target_interface_type</code> Enumeration	Specifies the type of interface the target has. The options are: Default - Select target interface type based on CPU core type and SWO usage JTAG - Use JTAG interface SWD - Use SWD interface
Use Debug Handler <code>arm_target_use_debug_handler</code> Enumeration	Specifies whether to use a debug handler.

Target Loader

Property	Description
Applicable Loader Configurations <code>arm_target_loader_applicable_loaders</code> StringList	The set of target loader configurations that are applicable
Can Erase All <code>arm_target_loader_can_erase_all</code> Boolean	Loader can erase all of memory
Can Erase Range <code>arm_target_loader_can_erase_range</code> Boolean	Loader can erase a range of memory
Can Lock All <code>arm_target_loader_can_lock_all</code> Boolean	Loader can lock all of memory
Can Lock Range <code>arm_target_loader_can_lock_range</code> Boolean	Loader can lock a range of memory
Can Only Download After Erase <code>arm_target_loader_can_only_download_after_</code>	Loader can only download after erase
Can Only Verify With Download <code>arm_target_loader_can_only_verify_with_dow</code>	Loader can only verify with download

Can Peek arm_target_loader_can_peekBoolean	Loader can peek memory
Can Unlock All arm_target_loader_can_unlock_allBoolean	Loader can unlock all of memory
Can Unlock Range arm_target_loader_can_unlock_rangeBoolean	Loader can unlock a range of memory
Erase All target_loader_erase_allEnumeration	If set to Yes , all of the FLASH memory on the target will be erased prior to downloading the application. If set to No , only the areas of FLASH containing the program being downloaded will be erased. If set to Default the behaviour is target specific.
Erase All Timeout arm_target_loader_erase_all_timeoutIntegerRange	The timeout period for an erase all operation in milliseconds.
First Loader Program Section arm_target_loader_first_program_sectionString	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_sectionString	The loader's last program section. This parameter is only required if the program being downloaded overwrites the loader.
Loader Configurations arm_target_loader_default_loaderStringList	The target loader configuration(s) to use
Loader File Path arm_target_flash_loader_file_pathProjFileName	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 Parameter arm_target_loader_parameterString	The parameter to pass to the loader on startup.
Loader RAM Size arm_target_flash_loader_load_sizeString	The size of the RAM region used by the loader. This is required for FLM and STLDR loader types.
Loader RAM Start arm_target_flash_loader_load_offsetString	The start of the RAM region used by the loader. This is required for FLM and STLDR loader types.
Loader Timeout arm_target_loader_operation_timeoutIntegerRange	The timeout period for loader operations in milliseconds.
Loader Type arm_target_loader_typeEnumeration	The type of loader to use.
Reset After Download arm_target_loader_reset_after_downloadBoolean	Specifies whether the target should be reset after a program has been downloaded by a loader.

Target Script

Property	Description
----------	-------------

Attach Script target_attach_scriptJavaScript	The script that is executed when the target is attached to.
Connect Script target_connect_scriptJavaScript	The script that is executed when the target is connected to.
Debug Begin Script target_debug_begin_scriptJavaScript	The script that is executed when the debugger begins a debug session.
Debug End Script target_debug_end_scriptJavaScript	The script that is executed when the debugger ends a debug session.
Debug Interface Reset Script target_debug_interface_reset_scriptJavaScript	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_scriptJavaScript	The script that is executed when the target is disconnected from.
Get Part Name Script target_get_partname_scriptJavaScript	The script that returns the part name of the connected target.
Load Begin Script target_load_begin_scriptJavaScript	The script that is executed when the debugger begins a load.
Load End Script target_load_end_scriptJavaScript	The script that is executed when the debugger ends a load.
Loader Reset Script target_loader_reset_scriptJavaScript	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_scriptJavaScript	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_scriptJavaScript	The script that is executed when the target is reset.
Run Script target_go_scriptJavaScript	The script that is executed when the target is run.
Stop Script target_stop_scriptJavaScript	The script that is executed when the target is stopped.
TAP Reset Script target_TAP_reset_scriptString	The script that is executed when the TAP is reset.
Target Extras Script target_extras_scriptJavaScript	The script that is executed to supply extra menu entries in the targets window context menu.
Target Script File target_script_fileFileName	The target script file, the contents of this file are prepended to script project properties before they are executed.

Target Trace

Property	Description
ETM Global Timestamping Enable arm_target_etm_global_timestamping_enableBoolean	Enable the ETM global timestamping if supported.
ETM TraceID arm_target_etm_trace_idIntegerRange	Specifies the traceID of the ETM - zero disables usage.
ITM Stimulus Port To Display arm_target_itm_stimulus_port_displayIntegerRange	Specifies the ITM Stimulus port to display in the debug terminal -1 disables this
ITM Stimulus Ports Enable arm_target_itm_stimulus_port_enableIntegerHex	Specifies the ITM Stimulus ports to enable.
ITM Stimulus Ports Privilege arm_target_itm_stimulus_port_privilegeIntegerHex	Specifies the ITM Stimulus ports to enable.
ITM Timestamping arm_target_itm_timestamping_enableEnumeration	Specifies ITM timestamping. The options are: Disable - disable timestamping Local - use the local timestamp clock Global - use the global timestamp clock
ITM TraceID arm_target_itm_trace_idIntegerRange	Specifies the traceID of the ITM - zero disables usage.
ITM/DWT Data Trace PC arm_target_dwt_data_trace_PCBoolean	Specifies whether to trace the PC on data trace.
ITM/DWT PC Sampling arm_target_dwt_pc_sampling_enableEnumeration	Specifies the DWT PC sampling rate.
ITM/DWT Trace Exceptions arm_target_dwt_trace_exceptionsBoolean	Specifies whether to trace exception entry and return.
MTB RAM Address arm_target_mtb_ram_addressIntegerHex	Specifies the MTB RAM Address - note that this must be aligned to the MTB RAM size.
MTB RAM Size arm_target_mtb_ram_sizeEnumeration	Specifies the MTB RAM size in bytes.
SWO Baud Rate arm_target_trace_SWO_speedIntegerRange	Specifies the baud rate of the SWO - zero selects auto detection.
Trace Clock Speed arm_target_trace_clock_speedIntegerRange	The speed of the trace clock. This is usually the same as the CPU clock and is used to program the prescaler for the SWO
Trace Initialize Script target_trace_initialize_scriptJavaScript	The script that is executed to initialize the target trace hardware. When executed this script has the macro \$(TraceInterfaceType) expanded with value of the Trace Interface Type property, typically it is EnableTrace("\$(TraceInterfaceType)").

Trace Interface Type <code>arm_target_trace_interface_typeEnumeration</code>	<p>Specifies the type of trace interface the target has. The options are:</p> <ul style="list-style-type: none">SWO - Use asynchronous SWO trace interface.TracePort - Use synchronous parallel trace interface.ETB - Use on-chip embedded trace buffer.MTB - Use on-chip MTB - Cortex-M0+ only.PC Sampling - sample the PC.None
Trace Port Size <code>arm_target_trace_port_sizeEnumeration</code>	<p>Specifies the trace port size the target has. The options are:</p> <ul style="list-style-type: none">1-bit2-bit4-bit8-bit16-bit24-bit32-bit

System Macros

System Macro Values

Property	Description
<code>\$(Date)</code> <code>\$(Date)String</code>	Day Month Year e.g. 21 June 2011.
<code>\$(DateDay)</code> <code>\$(DateDay)String</code>	Day e.g. 21.
<code>\$(DateMonth)</code> <code>\$(DateMonth)String</code>	Month e.g. 01 to 12.
<code>\$(DateYear)</code> <code>\$(DateYear)String</code>	Year e.g. 2011.
<code>\$(DesktopDir)</code> <code>\$(DesktopDir)String</code>	Path to users desktop directory.
<code>\$(DocumentsDir)</code> <code>\$(DocumentsDir)String</code>	Path to users documents directory.
<code>\$(HomeDir)</code> <code>\$(HomeDir)String</code>	Path to users home directory.
<code>\$(HostArch)</code> <code>\$(HostArch)String</code>	The CPU architecture that CrossStudio is running on e.g. x86.
<code>\$(HostArchClass)</code> <code>\$(HostArchClass)String</code>	The class of CPU architecture that CrossStudio is running on e.g. intel, arm.
<code>\$(HostDLL)</code> <code>\$(HostDLL)String</code>	The file extension for dynamic link libraries on the CPU that CrossStudio is running on e.g. .dll.
<code>\$(HostDLLExt)</code> <code>\$(HostDLLExt)String</code>	The file extension for dynamic link libraries used by the operating system that CrossStudio is running on e.g. .dll, .so, .dylib.
<code>\$(HostEXE)</code> <code>\$(HostEXE)String</code>	The file extension for executables on the CPU that CrossStudio is running on e.g. .exe.
<code>\$(HostOS)</code> <code>\$(HostOS)String</code>	The name of the operating system that CrossStudio is running on e.g. win.
<code>\$(Micro)</code> <code>\$(Micro)String</code>	The CrossStudio target e.g. ARM.
<code>\$(PackagesDir)</code> <code>\$(PackagesDir)String</code>	Path to the users packages directory.
<code>\$(Platform)</code> <code>\$(Platform)String</code>	The target platform.
<code>\$(ProductNameShort)</code> <code>\$(ProductNameShort)String</code>	The product name.

<code>\$(SamplesDir)</code> <code>\$(SamplesDir)String</code>	Path to the samples subdirectory of the packages directory.
<code>\$(StudioArchiveFileExt)</code> <code>\$(StudioArchiveFileExt)String</code>	The filename extension of a studio archive file.
<code>\$(StudioBuildToolExeName)</code> <code>\$(StudioBuildToolExeName)String</code>	The filename of the build tool executable.
<code>\$(StudioBuildToolName)</code> <code>\$(StudioBuildToolName)String</code>	The name of the build tool executable.
<code>\$(StudioDir)</code> <code>\$(StudioDir)String</code>	The install directory of the product.
<code>\$(StudioExeName)</code> <code>\$(StudioExeName)String</code>	The filename of the studio executable.
<code>\$(StudioMajorVersion)</code> <code>\$(StudioMajorVersion)String</code>	The major release version of software.
<code>\$(StudioMinorVersion)</code> <code>\$(StudioMinorVersion)String</code>	The minor release version of software.
<code>\$(StudioName)</code> <code>\$(StudioName)String</code>	The full name of studio.
<code>\$(StudioNameShort)</code> <code>\$(StudioNameShort)String</code>	The short name of studio.
<code>\$(StudioPackageFileExt)</code> <code>\$(StudioPackageFileExt)String</code>	The filename extension of a studio package file.
<code>\$(StudioProjectFileExt)</code> <code>\$(StudioProjectFileExt)String</code>	The filename extension of a studio project file.
<code>\$(StudioRevision)</code> <code>\$(StudioRevision)String</code>	The release revision of software.
<code>\$(StudioScriptToolExeName)</code> <code>\$(StudioScriptToolExeName)String</code>	The filename of the script tool executable.
<code>\$(StudioScriptToolName)</code> <code>\$(StudioScriptToolName)String</code>	The name of the script tool executable.
<code>\$(StudioSessionFileExt)</code> <code>\$(StudioSessionFileExt)String</code>	The filename extension of a studio session file.
<code>\$(StudioSimulatorExeName)</code> <code>\$(StudioSimulatorExeName)String</code>	The filename of the simulator executable.
<code>\$(StudioSimulatorName)</code> <code>\$(StudioSimulatorName)String</code>	The name of the simulator executable.
<code>\$(StudioUserDir)</code> <code>\$(StudioUserDir)String</code>	The directory containing the user data.
<code>\$(TargetID)</code> <code>\$(TargetID)String</code>	ID number representing the CrossStudio target.

<code>\$(TargetsDir)</code> <code>\$(TargetsDir)String</code>	Path to the targets subdirectory of the packages directory.
<code>\$(Time)</code> <code>\$(Time)String</code>	Hour:Minutes:Seconds e.g. 15:34:03.
<code>\$(TimeHour)</code> <code>\$(TimeHour)String</code>	Hour e.g. 15.
<code>\$(TimeMinute)</code> <code>\$(TimeMinute)String</code>	Minute e.g. 34.
<code>\$(TimeSecond)</code> <code>\$(TimeSecond)String</code>	Seconds e.g. 03.
<code>\$(UnixTime)</code> <code>\$(UnixTime)String</code>	Seconds since 00:00, Jan 1 1970 UTC

Build Macros

(Build Macro Values)

Property	Description
<code>\$(AR)</code> <code>\$(AR) String</code>	The path to the binutils ar command.
<code>\$(AS)</code> <code>\$(AS) String</code>	The path to the binutils as command.
<code>\$(Arch)</code> <code>\$(Arch) String</code>	The lower case value of the ARM Architecture project property.
<code>\$(AsmOptions)</code> <code>\$(AsmOptions) String</code>	A space separated list of assembler options for the external assemble command.
<code>\$(CC)</code> <code>\$(CC) String</code>	The path to the cc command.
<code>\$(CC1)</code> <code>\$(CC1) String</code>	The path to the gcc cc1 command.
<code>\$(CCPP)</code> <code>\$(CCPP) String</code>	The path to the cc command.
<code>\$(CLANG)</code> <code>\$(CLANG) String</code>	The path to the clang command.
<code>\$(CLANGTIDY)</code> <code>\$(CLANGTIDY) String</code>	The path to the clang-tidy command.
<code>\$(COnlyOptions)</code> <code>\$(COnlyOptions) String</code>	A space separated list of compiler options for the external c compile command.
<code>\$(COptions)</code> <code>\$(COptions) String</code>	A space separated list of compiler options for the external c and c++ compile commands.
<code>\$(CombiningOutputFilePath)</code> <code>\$(CombiningOutputFilePath) String</code>	The full path of the output file of the combining command.
<code>\$(CombiningRelInputPaths)</code> <code>\$(CombiningRelInputPaths) String</code>	The relative inputs to the combining command.
<code>\$(Configuration)</code> <code>\$(Configuration) String</code>	The build configuration e.g. ARM Flash Debug.
<code>\$(CoreType)</code> <code>\$(CoreType) String</code>	The lower case value of the ARM Core Type project property.
<code>\$(Defines)</code> <code>\$(Defines) String</code>	The preprocessor defines property value for the external compile command.
<code>\$(DependencyPath)</code> <code>\$(DependencyPath) String</code>	The path of the dependency file for the external compile command.

<code>\$(EXE)</code> <code>\$(EXE) String</code>	The default file extension for an executable file including the dot e.g. .elf.
<code>\$(Endian)</code> <code>\$(Endian) String</code>	The lower case value of the Byte Order project property.
<code>\$(FPABI)</code> <code>\$(FPABI) String</code>	The value of the ARM FP ABI Type project property.
<code>\$(FPU)</code> <code>\$(FPU) String</code>	The lower case value of the ARM FPU Type project property.
<code>\$(FPU2)</code> <code>\$(FPU2) String</code>	Alternative value of the ARM FPU Type project property.
<code>\$(FPU3)</code> <code>\$(FPU3) String</code>	Alternative value of the ARM FPU Type project property.
<code>\$(FolderName)</code> <code>\$(FolderName) String</code>	The folder name of the containing folder.
<code>\$(FolderPath)</code> <code>\$(FolderPath) String</code>	The folder path of the containing folders.
<code>\$(GCC)</code> <code>\$(GCC) String</code>	The path to the gcc command.
<code>\$(GCCPrefix)</code> <code>\$(GCCPrefix) String</code>	The macro-expanded value of the GCC Prefix project property.
<code>\$(GCCTarget)</code> <code>\$(GCCTarget) String</code>	The macro-expanded value of the GCC Target project property.
<code>\$(GCCVersion)</code> <code>\$(GCCVersion) String</code>	The macro-expanded value of the GCC Version project property.
<code>\$(GPLUSPLUS)</code> <code>\$(GPLUSPLUS) String</code>	The path to the g++ command.
<code>\$(IncludeFiles)</code> <code>\$(IncludeFiles) String</code>	The user includes property value for the external compile command.
<code>\$(Includes)</code> <code>\$(Includes) String</code>	The user directories property value for the external compile command.
<code>\$(InputDir)</code> <code>\$(InputDir) String</code>	The absolute directory of the input file.
<code>\$(InputExt)</code> <code>\$(InputExt) String</code>	The extension of an input file not including the dot e.g. cpp.
<code>\$(InputFileName)</code> <code>\$(InputFileName) String</code>	The name of an input file relative to the project directory.
<code>\$(InputName)</code> <code>\$(InputName) String</code>	The name of an input file relative to the project directory without the extension.
<code>\$(InputPath)</code> <code>\$(InputPath) String</code>	The absolute name of an input file including the extension.

<code>\$(IntDir)</code> <code>\$(IntDir)String</code>	The macro-expanded value of the Intermediate Directory project property.
<code>\$(LD)</code> <code>\$(LD)String</code>	The path to the binutils ld command.
<code>\$(LIB)</code> <code>\$(LIB)String</code>	The default file extension for a library file including the dot e.g. .lib.
<code>\$(LIBLTO)</code> <code>\$(LIBLTO)String</code>	The path to the LTO dll.
<code>\$(LTO1)</code> <code>\$(LTO1)String</code>	The path to the gcc lto1 command.
<code>\$(LibArch)</code> <code>\$(LibArch)String</code>	The library architecture.
<code>\$(LibEndianExt)</code> <code>\$(LibEndianExt)String</code>	The endian specific library extension.
<code>\$(LibExt)</code> <code>\$(LibExt)String</code>	The architecture and build specific library extension.
<code>\$(LinkLibraries)</code> <code>\$(LinkLibraries)String</code>	The value of the Standard Libraries Directory project property.
<code>\$(LinkOptions)</code> <code>\$(LinkOptions)String</code>	A space separated list of compiler options for the external link command.
<code>\$(LinkerScriptPath)</code> <code>\$(LinkerScriptPath)String</code>	The full path of the linker script file for the link command.
<code>\$(MacroFiles)</code> <code>\$(MacroFiles)String</code>	The user macros property value for the external compile command.
<code>\$(MapPath)</code> <code>\$(MapPath)String</code>	The full path of the map file of the external link command.
<code>\$(MemorySegments)</code> <code>\$(MemorySegments)String</code>	The value of the Memory Segments property supplied to pre/post link command.
<code>\$(OBJ)</code> <code>\$(OBJ)String</code>	The default file extension for an object file including the dot e.g. .o.
<code>\$(OBJCOPY)</code> <code>\$(OBJCOPY)String</code>	The path to the binutils objcopy command.
<code>\$(OBJDUMP)</code> <code>\$(OBJDUMP)String</code>	The path to the binutils objdump command.
<code>\$(Objects)</code> <code>\$(Objects)String</code>	A space separated list of files for the external archive or link command.
<code>\$(ObjectsFilePath)</code> <code>\$(ObjectsFilePath)String</code>	The full path containing the files for the external archive or link command.
<code>\$(OutDir)</code> <code>\$(OutDir)String</code>	The macro-expanded value of the Output Directory project property.

<code>\$(PackageExt)</code> <code>\$(PackageExt)String</code>	The file extension of a package file e.g. hzq.
<code>\$(PostLinkOutputFilePath)</code> <code>\$(PostLinkOutputFilePath)String</code>	The full path of the output file of the post link command.
<code>\$(ProjectDir)</code> <code>\$(ProjectDir)String</code>	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.
<code>\$(ProjectName)</code> <code>\$(ProjectName)String</code>	The project name of the current project.
<code>\$(ProjectNodeName)</code> <code>\$(ProjectNodeName)String</code>	The name of the selected project node.
<code>\$(RANLIB)</code> <code>\$(RANLIB)String</code>	The path to the binutils ranlib command.
<code>\$(RelDependencyPath)</code> <code>\$(RelDependencyPath)String</code>	The relative path of the dependency file for the external compile command.
<code>\$(RelInputDir)</code> <code>\$(RelInputDir)String</code>	The relative path to the directory containing the input file from the project directory or dot if not relative.
<code>\$(RelInputPath)</code> <code>\$(RelInputPath)String</code>	The relative path to the input file from the project directory or the full path if not relative.
<code>\$(RelLinkerScriptPath)</code> <code>\$(RelLinkerScriptPath)String</code>	The relative path of the linker script file for the link command.
<code>\$(RelMapPath)</code> <code>\$(RelMapPath)String</code>	The relative path of the map file of the external link command.
<code>\$(RelObjectsFilePath)</code> <code>\$(RelObjectsFilePath)String</code>	The relative path containing the files for the external archive or link command.
<code>\$(RelTargetPath)</code> <code>\$(RelTargetPath)String</code>	The project directory relative path of the output file of the link or compile command.
<code>\$(RootOutDir)</code> <code>\$(RootOutDir)String</code>	The macro-expanded value of the Root Output Directory project property.
<code>\$(RootRelativeOutDir)</code> <code>\$(RootRelativeOutDir)String</code>	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.
<code>\$(STRIP)</code> <code>\$(STRIP)String</code>	The path to the binutils strip command.
<code>\$(SolutionDir)</code> <code>\$(SolutionDir)String</code>	The absolute path of the directory containing the solution file.
<code>\$(SolutionExt)</code> <code>\$(SolutionExt)String</code>	The extension of the solution file without the dot.
<code>\$(SolutionFileName)</code> <code>\$(SolutionFileName)String</code>	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.
\$(ToolChainDir) \$(ToolChainDir)String	The macro-expanded value of the Tool Chain Directory project property.
\$(Undefines) \$(Undefines)String	The preprocessor undefines property value for the external compile command.

BinaryFile

The following table lists the BinaryFile object's member functions.

BinaryFile.crc32(offset, length) returns the CRC-32 checksum of an address range <i>length</i> bytes long, starting at <i>offset</i> . 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.
BinaryFile.length() returns the length of the binary file in bytes.
BinaryFile.load(path) loads binary file from <i>path</i> .
BinaryFile.loadAppend(path) loads binary file from <i>path</i> and appends it to the binary image.
BinaryFile.peekBytes(offset, length) returns byte array containing <i>length</i> bytes peeked from <i>offset</i> .
BinaryFile.peekUint32(offset, littleEndian) returns a 32-bit word peeked from <i>offset</i> . The <i>littleEndian</i> argument specifies the endianness of the access, if true or undefined it will be little endian, otherwise it will be big endian.
BinaryFile.pokeBytes(offset, byteArray) poke byte array <i>byteArray</i> to <i>offset</i> .
BinaryFile.pokeUint32(offset, value, littleEndian) poke a <i>value</i> to 32-bit word located at <i>offset</i> . The <i>littleEndian</i> argument specifies the endianness of the access, if true or undefined it will be little endian, otherwise it will be big endian.
BinaryFile.resize(length, fill) resizes the binary image to <i>length</i> bytes. If the operation extends the size, the binary image will be padded with bytes of value <i>fill</i> .
BinaryFile.save(path) saves binary file to <i>path</i> .
BinaryFile.saveRange(path, offset, length) saves part of the binary file to <i>path</i> . The <i>offset</i> argument specifies the byte offset to start from. The <i>length</i> argument specifies the maximum number of bytes that should be saved.

CWSys

The following table lists the CWSys object's member functions.

CWSys.appendStringToFile(path, string)	appends <i>string</i> to the end of the file <i>path</i> .
CWSys.copyFile(srcPath, destPath)	copies file <i>srcPath</i> to <i>destPath</i> .
CWSys.crc32(array)	returns the CRC-32 checksum of the byte array <i>array</i> . 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 <i>path</i> exists.
CWSys.fileSize(path)	return the number of bytes in file <i>path</i> .
CWSys.getRunStderr()	returns the stderr output from the last <i>CWSys.run()</i> call.
CWSys.getRunStdout()	returns the stdout output from the last <i>CWSys.run()</i> call.
CWSys.makeDirectory(path)	create the directory <i>path</i> .
CWSys.packU32(array, offset, number, le)	packs <i>number</i> into the <i>array</i> at <i>offset</i> .
CWSys.popup(text, caption)	prompt the user with text and return true for yes and false for no.
CWSys.readByteArrayFromFile(path)	returns the byte array contained in the file <i>path</i> .
CWSys.readStringFromFile(path)	returns the string contained in the file <i>path</i> .
CWSys.removeDirectory(path)	remove the directory <i>path</i> .
CWSys.removeFile(path)	deletes file <i>path</i> .
CWSys.renameFile(oldPath, newPath)	renames file <i>oldPath</i> to be <i>newPath</i> .
CWSys.run(cmd, wait)	runs command line <i>cmd</i> optionally waits for it to complete if <i>wait</i> is true.
CWSys.unpackU32(array, offset, le)	returns the number unpacked from the <i>array</i> at <i>offset</i> .
CWSys.writeByteArrayToFile(path, array)	creates a file <i>path</i> containing the byte array <i>array</i> .
CWSys.writeStringToFile(path, string)	creates a file <i>path</i> containing <i>string</i> .

Debug

The following table lists the Debug object's member functions.

Debug.breakexpr(expression, count, hardware) set a breakpoint on <i>expression</i> , with optional ignore <i>count</i> and use <i>hardware</i> parameters. Return the, none zero, allocated breakpoint number.
Debug.breakline(filename, linenumber, temporary, count, hardware) set a breakpoint on <i>filename</i> and <i>linenumber</i> , with optional <i>temporary</i> , ignore <i>count</i> and use <i>hardware</i> 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 <i>source</i> and <i>labels</i> to be displayed and the number of bytes to disassemble <i>before</i> and <i>after</i> the located program counter.
Debug.echo(s) display string.
Debug.enableexception(exception, enable) <i>enable</i> break on <i>exception</i> .
Debug.evaluate(expression) evaluates debug <i>expression</i> 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 <i>frame</i> context.
Debug.locatepc(pc) locate the debugger to the specified <i>pc</i> .
Debug.locateregisters(registers) locate the debugger to the specified <i>register</i> context.
Debug.print(expression, fmt) evaluate and display debug <i>expression</i> using optional <i>fmt</i> . Supported formats are <i>b</i> binary, <i>c</i> character, <i>d</i> decimal, <i>e</i> scientific float, <i>f</i> decimal float, <i>g</i> scientific or decimal float, <i>i</i> signed decimal, <i>o</i> octal, <i>p</i> pointer value, <i>s</i> null terminated string, <i>u</i> unsigned decimal, <i>x</i> 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 <i>before</i> and <i>after</i> 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* milliseconds 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, programNotSection) 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. If *programNotSection* is true or undefined, data is read using program headers rather than section headers. 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, programNotSection) 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. If *programNotSection* is true or undefined, data is read using program headers rather than section headers.

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 its 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.crc32(address, length) reads a block of bytes from target memory starting at address for length bytes, generates a crc32 on the block of bytes and returns it.
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.executeMRC(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.expandMacro(string) returns the string with macros expanded.
TargetInterface.fillScanChain(bool, lsb, msb) sets bits from lsb (least significant bit) to msb (most significant bit) in internal buffer to bool value.
TargetInterface.findByte(address, length, byte) returns the index of the byte in the specified target memory range.
TargetInterface.findNotByte(address, length, byte) returns the index of the byte that isn't in the specified target memory range.

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.runFromToAddress(from, to, timeout) start the target executing at address from and waits for the 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.setDBGRRQ(v) sets/clears the DBGRRQ bit of the ARM7/ARM9 debug control register.

TargetInterface.setDebugInterfaceProperty("reset_debug_interface_enabled", bool) turn on/off the reset of the debug interface.

TargetInterface.setDebugInterfaceProperty("has_etm", bool) set the ARM7/ARM9 property to enable use of the ETM.

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", N, [start, size]) set the ARMv7A/ARMv7R debug property to turn on/off usage of the ADIV5 AHB MEM-AP for 1+2+4 data sized accesses on the optional address range specified by start and size.

TargetInterface.setDebugInterfaceProperty("use_adiv5_APB", start, size) set the ARMv7M debug property to turn on usage of the ADIV5 APB MEM-AP for word sized data accesses in the address range specified by start and size.

TargetInterface.setDebugInterfaceProperty("set_adiv5_AHB_ap_num", N, [clearCSWbits, setCSWbits]) specify the ADIV5 AHB AP number to use and optional CSW bits to clear and set.

TargetInterface.setDebugInterfaceProperty("set_adiv5_APB_ap_num", N) specify the ADIV5 APB AP number to use.

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 ADIV5 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 specified 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.