

CrossWorks for MSP430 Reference Manual

Version: 2.1.2.2011080305.11906

Contents

Preface	27
Introduction	27
What is CrossWorks?	28
What we don't tell you... ..	29
Activating your product	30
Text conventions	32
Standard syntactic metalanguage	34
Requesting support and reporting problems	35
CrossStudio Tutorial	37
Activating CrossWorks	38
Managing support packages	40
Creating a project	42
Managing files in a project	48
Setting project options	54
Building projects	56
Exploring projects	59
Using the debugger	68
Low-level debugging	73
Debugging externally built applications	77
CrossStudio Reference	81
Overview	82
The title bar	83
The menu bar	85
The status bar	86
The editing workspace	89
Docking windows	90
Project management	91
Overview	93
Creating a project	96
Adding existing files to a project	97
Adding new files to a project	98
Removing a file, folder, project, or project link	99
Project properties	100
Project configurations	103
Project dependencies and build order	104
Project macros	105
Building projects	107
Source control	110

Breakpoint expressions	116
Debug expressions	117
Basic editing	118
Navigation	119
Bookmarks	122
Changing text	124
Using the clipboard	126
Drag and drop editing	128
Undo and redo	129
Indentation	131
File management	133
Find and replace	135
Regular expressions	137
Advanced editor features	139
Code templates	142
Linking and section placement	143
CrossStudio Windows	145
Breakpoints window	147
Clipboard ring window	151
Call stack window	153
Downloads window	157
Execution counts window	158
Find and replace window	159
Globals window	161
CrossStudio Help and Assistance	163
Immediate window	165
Latest news window	166
Locals window	167
Memory window	169
Memory usage window	172
Output window	175
Package manager window	177
Project explorer	181
Properties window	186
Register windows	187
Script Console	190
Source navigator window	191
Symbol browser	194
Targets window	200
Terminal emulator window	205
Threads window	208

Trace window	211
Watch window	212
MSP430 target interfaces	216
MSP430 Target Debug	217
CrossConnect Target Interface	219
MSP430 Flash Emulation Tool Target Interface	221
MSP430 DLL Target Interface	223
MSP430 Core Simulator Target Interface	225
Dialogs	227
Debug file search editor	227
Environment Options Dialog	229
Building Environment Options	230
Debugging Environment Options	231
IDE Environment Options	234
Programming Language Environment Options	239
Source Control Environment Options	242
Text Editor Environment Options	244
Windows Environment Options	248
CrossStudio menu summary	251
File menu	252
New menu	256
Edit menu	257
Clipboard menu	259
Clipboard Ring menu	261
Macros menu	262
Edit Selection menu	264
Bookmarks menu	267
Advanced menu	268
View menu	270
Other Windows menu	272
Browser menu	274
Toolbars menu	275
Search menu	276
Project menu	278
Build menu	280
Debug menu	282
Debug Control menu	286
Breakpoint menu	288
Debug Windows menu	290
Target menu	292
Tools menu	293

Window menu	294
Help menu	296
C Compiler Reference	299
Command line options	300
-D (Define macro symbol)	301
-g (Generate debugging information)	302
-I (Define user include directories)	303
-J (Define system include directories)	304
-mmpy (Enable hardware multiplier)	305
-mmpyinl (Enable inline hardware multiplier)	306
-msd (Treat double as float)	307
-o (Set output file name)	308
-O (Optimize code generation)	309
-Or- (Disable register allocation)	310
-Org (Register allocation of locals and global addresses)	311
-Orl (Register allocation of locals)	312
-Rc (Set default code section name)	313
-Rd (Set default initialised data section name)	314
-Rk (Set default read-only data section name)	315
-Rv (Set default vector section name)	316
-Rz (Set default zeroed data section name)	317
-V (Version information)	318
-w (Suppress warnings)	319
-we (Treat warnings as errors)	320
Functions	321
Interrupt functions	322
Monitor functions	323
Top-level functions	324
Strings	325
Code-space strings	326
GSM 03.38 strings	327
Pragmas	328
#pragma codeseg	329
#pragma dataseg	330
#pragma constseg	331
#pragma vectorseg	332
#pragma zeroedseg	333
#pragma linklib	334
Data definition	335
Type-based enumerations	335
Section reference	336

Preprocessor	337
Preprocessor predefined symbols	338
Assembly language interface	339
Data representation	339
External naming convention	340
Register usage	341
Parameter passing	342
Returning values	343
Examples	344
Customizing runtime behavior	346
Floating-point implementation	347
Customizing putchar	349
Extending I/O library functions	351
Diagnostics	354
Pre-processor warning messages	355
Pre-processor error messages	357
Compiler warning messages	360
Compiler error messages	361
Extensions summary	370
Tasking Library User Guide	371
Overview	373
Tasks	375
Event sets	378
Semaphores	381
Mutexes	383
Message queues	385
Byte queues	388
Global interrupts control	391
Timer support	392
Interrupt service routines	393
Memory areas	394
Task scheduling example	396
MSP430 implementation	398
CTL revisions	399
CTL sources	402
Library Reference	403
MSP430 Library Reference	403
<cross_studio_io.h>	404
debug_abort	407
debug_break	408
debug_clearerr	409

debug_enabled	410
debug_exit	411
debug_fclose	412
debug_feof	413
debug_ferror	414
debug_fflush	415
debug_fgetc	416
debug_fgetpos	417
debug_fgets	418
debug_filesize	419
debug_fopen	420
debug_fprintf	421
debug_fprintf_c	422
debug_fputc	423
debug_fputs	424
debug_fread	425
debug_freopen	426
debug_fscanf	427
debug_fscanf_c	428
debug_fseek	429
debug_fsetpos	430
debug_ftell	431
debug_fwrite	432
debug_getargs	433
debug_getch	434
debug_getchar	435
debug_getd	436
debug_getenv	437
debug_getf	438
debug_geti	439
debug_getl	440
debug_getll	441
debug_gets	442
debug_getu	443
debug_getul	444
debug_getull	445
debug_kbhit	446
debug_loadsymbols	447
debug_perror	448
debug_printf	449
debug_printf_c	450

debug_putchar	451
debug_puts	452
debug_remove	453
debug_rename	454
debug_rewind	455
debug_runtime_error	456
debug_scanf	457
debug_scanf_c	458
debug_system	459
debug_time	460
debug_tmpfile	461
debug_tmpnam	462
debug_ungetc	463
debug_unloadsymbols	464
debug_vfprintf	465
debug_vfscanf	466
debug_vprintf	467
debug_vscanf	468
<cruntime.h>	469
__float32_add	474
__float32_add_1	475
__float32_add_asgn	476
__float32_div	477
__float32_div_asgn	478
__float32_eq	479
__float32_eq_0	480
__float32_lt	481
__float32_lt_0	482
__float32_mul	483
__float32_mul_asgn	484
__float32_neg	485
__float32_sqr	486
__float32_sub	487
__float32_sub_asgn	488
__float32_to_float64	489
__float32_to_int16	490
__float32_to_int32	491
__float32_to_int64	492
__float32_to_uint16	493
__float32_to_uint32	494
__float32_to_uint64	495

__float64_add	496
__float64_add_1	497
__float64_add_asgn	498
__float64_div	499
__float64_div_asgn	500
__float64_eq	501
__float64_eq_0	502
__float64_lt	503
__float64_lt_0	504
__float64_mul	505
__float64_mul_asgn	506
__float64_neg	507
__float64_sqr	508
__float64_sub	509
__float64_sub_asgn	510
__float64_to_float32	511
__int16_asr	512
__int16_asr_asgn	513
__int16_div	514
__int16_div_asgn	515
__int16_lsl	516
__int16_lsl_asgn	517
__int16_lsr	518
__int16_lsr_asgn	519
__int16_mod	520
__int16_mod_asgn	521
__int16_mul	522
__int16_mul_8x8	523
__int16_mul_asgn	524
__int16_to_float32	525
__int16_to_float64	526
__int32_asr	527
__int32_asr_asgn	528
__int32_div	529
__int32_div_asgn	530
__int32_lsl	531
__int32_lsl_asgn	532
__int32_lsr	533
__int32_lsr_asgn	534
__int32_mod	535
__int32_mod_asgn	536

__int32_mul	537
__int32_mul_16x16	538
__int32_mul_asgn	539
__int32_to_float32	540
__int32_to_float64	541
__int64_asr	542
__int64_asr_asgn	543
__int64_div	544
__int64_div_asgn	545
__int64_lsl	546
__int64_lsl_asgn	547
__int64_lsr	548
__int64_lsr_asgn	549
__int64_mod	550
__int64_mod_asgn	551
__int64_mul	552
__int64_mul_32x32	553
__int64_mul_asgn	554
__int64_to_float32	555
__int64_to_float64	556
__uint16_div	557
__uint16_div_asgn	558
__uint16_mod	559
__uint16_mod_asgn	560
__uint16_mul_8x8	561
__uint16_to_float32	562
__uint16_to_float64	563
__uint32_div	564
__uint32_div_asgn	565
__uint32_mod	566
__uint32_mod_asgn	567
__uint32_mul_16x16	568
__uint32_to_float32	569
__uint32_to_float64	570
__uint64_div	571
__uint64_div_asgn	572
__uint64_mod	573
__uint64_mod_asgn	574
__uint64_mul_32x32	575
__uint64_to_float32	576
__uint64_to_float64	577

<ctl.h>	578
CTL_BYTE_QUEUE_t	581
CTL_EVENT_SET_t	582
CTL_MEMORY_AREA_t	583
CTL_MESSAGE_QUEUE_t	584
CTL_MUTEX_t	585
CTL_SEMAPHORE_t	586
CTL_TASK_t	587
CTL_TIME_t	588
ctl_byte_queue_init	589
ctl_byte_queue_num_free	590
ctl_byte_queue_num_used	591
ctl_byte_queue_post	592
ctl_byte_queue_post_multi	593
ctl_byte_queue_post_multi_nb	594
ctl_byte_queue_post_nb	595
ctl_byte_queue_receive	596
ctl_byte_queue_receive_multi	597
ctl_byte_queue_receive_multi_nb	598
ctl_byte_queue_receive_nb	599
ctl_byte_queue_setup_events	600
ctl_current_time	601
ctl_events_init	602
ctl_events_pulse	603
ctl_events_set_clear	604
ctl_events_wait	605
ctl_get_current_time	606
ctl_global_interrupts_disable	607
ctl_global_interrupts_enable	608
ctl_global_interrupts_set	609
ctl_handle_error	610
ctl_increment_tick_from_isr	611
ctl_interrupt_count	612
ctl_last_schedule_time	613
ctl_memory_area_allocate	614
ctl_memory_area_free	615
ctl_memory_area_init	616
ctl_memory_area_setup_events	617
ctl_message_queue_init	618
ctl_message_queue_num_free	619
ctl_message_queue_num_used	620

ctl_message_queue_post	621
ctl_message_queue_post_multi	622
ctl_message_queue_post_multi_nb	623
ctl_message_queue_post_nb	624
ctl_message_queue_receive	625
ctl_message_queue_receive_multi	626
ctl_message_queue_receive_multi_nb	627
ctl_message_queue_receive_nb	628
ctl_message_queue_setup_events	629
ctl_mutex_init	630
ctl_mutex_lock	631
ctl_mutex_unlock	632
ctl_reschedule_on_last_isr_exit	633
ctl_semaphore_init	634
ctl_semaphore_signal	635
ctl_semaphore_wait	636
ctl_task_die	637
ctl_task_executing	638
ctl_task_init	639
ctl_task_list	640
ctl_task_remove	641
ctl_task_reschedule	642
ctl_task_restore	643
ctl_task_run	644
ctl_task_set_priority	645
ctl_task_switch_callout	646
ctl_time_increment	647
ctl_timeout_wait	648
ctl_timeslice_period	649
<inmsp.h>	650
__bcd_add_long	652
__bcd_add_long_long	653
__bcd_add_short	654
__bcd_negate_long	655
__bcd_negate_long_long	656
__bcd_negate_short	657
__bcd_subtract_long	658
__bcd_subtract_long_long	659
__bcd_subtract_short	660
__bic_SR_register	661
__bic_SR_register_on_exit	662

__bis_SR_register	663
__bis_SR_register_on_exit	664
__bit_count_leading_zeros_char	665
__bit_count_leading_zeros_long	666
__bit_count_leading_zeros_long_long	667
__bit_count_leading_zeros_short	668
__bit_reverse_char	669
__bit_reverse_long	670
__bit_reverse_long_long	671
__bit_reverse_short	672
__delay_cycles	673
__disable_interrupt	674
__disable_interrupt	674
__enable_interrupt	675
__enable_interrupt	675
__even_in_range	676
__even_in_range	676
__get_register	677
__insert_opcode	678
__low_power_mode_0	679
__low_power_mode_0	679
__low_power_mode_1	680
__low_power_mode_1	680
__low_power_mode_2	681
__low_power_mode_2	681
__low_power_mode_3	682
__low_power_mode_3	682
__low_power_mode_4	683
__low_power_mode_4	683
__low_power_mode_off_on_exit	684
__low_power_mode_off_on_exit	684
__no_operation	685
__no_operation	685
__read_extended_byte	686
__read_extended_long	687
__read_extended_word	688
__set_interrupt	689
__set_interrupt	689
__set_register	690
__swap_bytes	691
__swap_long_bytes	692

__swap_words	693
__write_extended_byte	694
__write_extended_long	695
__write_extended_word	696
<In430.h>	697
_BIC_SR	698
_BIC_SR_IRQ	699
_BIS_SR	700
_BIS_SR_IRQ	701
_DADD16	702
_DADD32	703
_DADD64	704
_DINT	705
_DNEG16	706
_DNEG32	707
_DNEG64	708
_DSUB16	709
_DSUB32	710
_DSUB64	711
_EINT	712
_LSWPB	713
_LSWPW	714
_NOP	715
_OPC	716
_SWPB	717
Standard C Library Reference	718
Formatted output control strings	719
Formatted input control strings	724
String handling	728
<assert.h>	729
__assert	730
assert	731
<ctype.h>	732
isalnum	733
isalpha	734
isblank	735
iscntrl	736
isdigit	737
isgraph	738
islower	739
isprint	740

ispunct	741
isspace	742
isupper	743
isxdigit	744
tolower	745
toupper	746
<errno.h>	747
EDOM	748
EILSEQ	749
ERANGE	750
__errno	751
errno	752
<float.h>	753
DBL_DIG	754
DBL_EPSILON	755
DBL_MANT_DIG	756
DBL_MAX	757
DBL_MAX_10_EXP	758
DBL_MAX_EXP	759
DBL_MIN	760
DBL_MIN_10_EXP	761
DBL_MIN_EXP	762
DECIMAL_DIG	763
FLT_DIG	764
FLT_EPSILON	765
FLT_EVAL_METHOD	766
FLT_MANT_DIG	767
FLT_MAX	768
FLT_MAX_10_EXP	769
FLT_MAX_EXP	770
FLT_MIN	771
FLT_MIN_10_EXP	772
FLT_MIN_EXP	773
FLT_RADIX	774
FLT_ROUNDS	775
<limits.h>	776
CHAR_BIT	777
CHAR_MAX	778
CHAR_MIN	779
INT_MAX	780
INT_MIN	781

LLONG_MAX	782
LLONG_MIN	783
LONG_MAX	784
LONG_MIN	785
SCHAR_MAX	786
SCHAR_MIN	787
SHRT_MAX	788
SHRT_MIN	789
UCHAR_MAX	790
UINT_MAX	791
ULLONG_MAX	792
ULONG_MAX	793
USHRT_MAX	794
<locale.h>	795
lconv	796
localeconv	798
setlocale	799
<math.h>	800
acos	803
acosf	804
acosh	805
acoshf	806
asin	807
asinf	808
asinh	809
asinhf	810
atan	811
atan2	812
atan2f	813
atanf	814
atanh	815
atanhf	816
cbrt	817
cbrtf	818
ceil	819
ceilf	820
cos	821
cosf	822
cosh	823
coshf	824
exp	825

expf	826
fabs	827
fabsf	828
floor	829
floorf	830
fmax	831
fmaxf	832
fmin	833
fminf	834
fmod	835
fmodf	836
fpclassify	837
frexp	838
frexpf	839
hypot	840
hypotf	841
isfinite	842
isinf	843
isnan	844
isnormal	845
ldexp	846
ldexpf	847
log	848
log10	849
log10f	850
logf	851
modf	852
modff	853
pow	854
powf	855
scalbn	856
scalbnf	857
signbit	858
sin	859
sinf	860
sinh	861
sinhf	862
sqrt	863
sqrtf	864
tan	865
tanf	866

tanh	867
tanhf	868
<setjmp.h>	869
longjmp	870
setjmp	871
<stdarg.h>	872
va_arg	873
va_copy	874
va_end	875
va_start	876
<stddef.h>	877
NULL	878
offsetof	879
ptrdiff_t	880
size_t	881
wchar_t	882
<stdio.h>	883
getchar	884
gets	885
printf	886
putchar	887
puts	888
scanf	889
snprintf	890
sprintf	891
sscanf	892
vprintf	893
vscanf	894
vsnprintf	895
vsprintf	896
vsscanf	897
<stdlib.h>	898
EXIT_FAILURE	900
EXIT_SUCCESS	901
RAND_MAX	902
abs	903
atexit	904
atof	905
atoi	906
atol	907
atoll	908

bsearch	909
calloc	910
div	911
div_t	912
exit	913
free	914
itoa	915
labs	916
ldiv	917
ldiv_t	918
llabs	919
lldiv	920
lldiv_t	921
lltoa	922
ltoa	923
malloc	924
qsort	925
rand	926
realloc	927
srand	928
strtod	929
strtof	930
strtol	931
strtoll	933
strtoul	934
strtoull	935
ulltoa	936
ultoa	937
utoa	938
<string.h>	939
memchr	941
memcmp	942
memcpy	943
memmove	944
memset	945
strcasecmp	946
strcat	947
strchr	948
strcmp	949
strcpy	950
strcspn	951

strdup	952
strerror	953
strlen	954
strncasecmp	955
strncat	956
strnchr	957
strncmp	958
strncpy	959
strndup	960
strnlen	961
strnstr	962
strpbrk	963
strrchr	964
strsep	965
strspn	966
strstr	967
strtok	968
strtok_r	969
<time.h>	970
asctime	971
asctime_r	972
clock_t	973
ctime	974
ctime_r	975
difftime	976
gmtime	977
gmtime_r	978
localtime	979
localtime_r	980
mktime	981
strftime	982
time_t	984
tm	985
Command Line Tools	986
Assembler Reference	986
Command line options	987
-D (Define macro symbol)	988
-g (Generate debugging information)	989
-I (Define include directories)	990
-J (Define system include directories)	991
-o (Set output file name)	992

-Rc (Set default code section name)	993
-Rd (Set default initialized data section name)	994
-Rk (Set default read-only data section name)	995
-Rv (Set default vector section name)	996
-Rz (Set default zeroed data section name)	997
-V (Display version)	998
-we (Treat warnings as errors)	999
-w (Suppress warnings)	1000
Source format	1001
Comments	1002
Data definition and allocation directives	1003
ALIGN directive	1004
DC.A directive	1005
DC.B directive	1006
DC.L directive	1007
DC.W directive	1008
EVEN directive	1009
FILL directive	1010
DS.B directive	1011
DS.W directive	1012
DS.L directive	1013
DV directive	1014
INCLUDEBIN directive	1015
Labels, variables, and sections	1016
Defining sections	1017
Symbolic constants and equates	1019
Labels	1021
CODE directive	1022
CONST directive	1023
DATA directive	1024
VECTORS directive	1025
ZDATA directive	1026
KEEP directive	1027
Data types	1028
Built-in types	1029
Structure and union types	1030
Array types	1031
Pointer types	1032
Combining data types	1033
Expressions and operators	1034
Constants	1035

Integer constants	1035
String constants	1036
Arithmetic operators	1037
+ operator	1038
- operator	1039
* operator	1040
/ operator	1041
% operator	1042
SHL and << operators	1043
SHR and >> operators	1044
ASHR operator	1045
Logical operators	1046
LAND and && operators	1047
LNOT and ! operators	1048
LOR and operators	1049
Bitwise operators	1050
AND and & operators	1051
NOT and ~ operators	1052
OR and operators	1053
XOR and ^ operators	1054
Relational operators	1055
Value extraction operators	1056
HIGH and HBYTE operators	1057
HWORD operator	1058
LOW and LBYTE operators	1059
LWORD operator	1060
Miscellaneous operators	1061
THIS operator	1061
DEFINED operator	1062
SIZEOF operator	1063
Indexing operator	1064
Retyping operator	1065
Compilation units and libraries	1066
INCLUDE directive	1067
Exporting symbols	1068
Importing symbols	1069
Macros, conditions, and loops	1070
Conditional assembly	1071
Macros	1072
Utilities Reference	1074
Compiler driver reference	1075

File naming conventions	1076
-ansi (Warn about potential ANSI problems)	1077
-ar (Archive output)	1078
-c (Compile to object code, do not link)	1079
-D (Define macro symbol)	1080
-e (Override entry symbol)	1081
-F (Set output format)	1082
-g (Generate debugging information)	1083
-h (Display help information)	1084
-I (Define user include directories)	1085
-J (Define system include directories)	1086
-K (Keep linker symbol)	1087
-l (Link library)	1088
-L (Set library directory path)	1089
-l- (Exclude standard include directories)	1090
-l- (Do not link standard libraries)	1091
-mmpy (Enable hardware multiplier)	1092
-mmpyinl (Enable inline hardware multiplier)	1093
-msd (Treat double as float)	1094
-M (Print linkage map)	1095
-n (Dry run, no execution)	1096
-o (Set output file name)	1097
-O (Optimize output)	1098
-Rc (Set default code section name)	1099
-Rd (Set default initialised data section name)	1100
-Rk (Set default read-only data section name)	1101
-Rv (Set default vector section name)	1102
-Rz (Set default zeroed data section name)	1103
-s- (Exclude standard startup code)	1104
-s (Set startup code file)	1105
-v (Verbose execution)	1106
-V (Version information)	1107
-w (Suppress warnings)	1108
-we (Treat warnings as errors)	1109
-Wa (Pass option to assembler)	1110
-Wc (Pass option to C compiler)	1111
-Wl (Pass option to linker)	1112
Linker reference	1113
Command line syntax	1114
-D (Define linker symbol)	1115
-F (Set output format)	1116

-g (Propagate debugging information)	1117
-H (checksum sections)	1118
-l- (Do not link standard libraries)	1120
-l (Link library)	1121
-L (Set library directory path)	1122
-M (Display linkage map)	1123
-o (Set output file name)	1124
-O (Optimize output)	1125
-Obl (Enable block localization optimization)	1126
-Ocm (Enable code motion optimization)	1127
-Ocp (Enable copy propagation optimization)	1128
-Ojc (Enable jump chaining optimization)	1129
-Ojt (Enable jump threading optimization)	1130
-Oph (Enable peephole optimizations)	1131
-Osf (Enable flattening optimizations)	1132
-Otm (Enable tail merging optimization)	1133
-Oxc (Enable code factoring optimization)	1134
-Oxcx (Enable extreme code factoring)	1135
-Oxcp (Set maximum code factoring passes)	1136
-Oxj (Enable cross jumping optimization)	1137
-T (Locate sections)	1138
-we (Treat warnings as errors)	1139
-w (Suppress warnings)	1140
-v (Verbose execution)	1141
-V (Version information)	1142
Hex extractor reference	1143
-F (Set output format)	1144
-o (Set output prefix)	1145
-T (Extract named section)	1146
-V (Display version)	1147
-W (Set bus width)	1148
Librarian reference	1149
-c (Create archive)	1150
-r (Add or replace archive member)	1151
-d (Delete archive members)	1152
-t (List archive members)	1153
CrossBuild	1154
CrossLoad	1156
Appendices	1159
Technical	1159
File formats	1159

Memory Map file format	1159
Section Placement file format	1161
Project file format	1162
Project Templates file format	1163
Property Groups file format	1165
Package Description file format	1167
Project Property Reference	1171
General Build Properties	1172
Combining Project Properties	1174
Compilation Properties	1175
Debugging Properties	1179
Externally Built Executable Project Properties	1181
File and Folder Properties	1182
Library Project Properties	1184
Executable Project Properties	1185
Staging Project Properties	1188
Macros	1189
Build Macros	1189
System Macros	1191
JavaScript Classes Reference	1193
CWSys	1194
Debug	1195
WScript	1197
Code editor command summary	1198
Binary editor command summary	1203
Frequently asked questions	1205
Glossary	1206

Introduction

This guide is divided into a number of sections:

Introduction

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

CrossStudio Tutorial

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

CrossStudio Reference

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

C Compiler Reference

Contains documentation for the C compiler, including syntax and usage details and a description of extensions provided by CrossWorks.

Tasking Library Tutorial

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

MSP430 Library Reference

Contains documentation for the functions that are specific to the MSP430.

Standard C Library Reference

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

Assembler Reference

Contains detailed documentation covering how to use the assembler, the assembler notation, an instruction set reference, macros, and other assembler features and extensions.

Command Line Tools Reference

Contains detailed reference material about the CrossWorks command line tools.

What is CrossWorks?

CrossWorks is a programming system which runs on Windows-based computers. Programs which are prepared on these host machines using the tools in this package and are executed, not on the host, but on an MSP430 Ultra Low Power microcontroller.

C compiler

CrossWorks C is a faithful implementation of the ANSI and ISO standards for the programming language C. We have added some extensions that enhance usability in a microcontroller environment. Because the

Assembler

MSP430 assembly language is largely compatible with the IAR assembler used in the KickStart package, which enables existing IAR users to use CrossWorks without losing their existing software base.

And more...

As well as providing cross-compilation technology, CrossWorks provides a PC-based fully functional simulation of the MSP430 core and hardware multiplier which, together with a windowing debugger, allows you to debug your application quickly. A set of tools for generating output files in multiple formats and a facility for flashing your applications onto the MSP430 provide the final stage of the software development lifecycle.

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 CPU 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

Because the CrossWorks C compiler is a compiler for ANSI C, the following books are especially relevant:

- 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., **A C 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. Written by Guy Steele, a noted language expert.

ANSI C reference

- 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 www.iso.ch.

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.
- From the **Tools** menu, click **License Manager**.
- If you have a default mailer, click the **By Mail** button underneath the text "If you wish to evaluate CrossWorks...".
- Using e-mail, send the registration key to the e-mail address **license@rowley.co.uk**.
- If you don't have a default mailer, click the **Manually** button underneath the text "If you wish to evaluate CrossWorks...".
- CrossStudio copies a registration key onto the clipboard. 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.
- From the **Tools** menu, click **License Manager**.
- Click the **Activate Product** button.
- Type in or paste the returned activation key into the dialog and click **OK**.

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, follow these instructions:

- Install CrossWorks on your computer using the CrossWorks installer and accept the license agreement.
- Run the **CrossStudio** application.
- From the **Tools** menu, click **License Manager**.
- If you have a default mailer, click the **By Mail** button underneath the text "If you have a product key...".
- Using e-mail, send the registration key to the e-mail address **license@rowley.co.uk**.
- If you do have a default mailer, click the **Manually** button underneath the text "If you have a product key...".

- CrossStudio copies a registration key onto the clipboard. Send the registration key to the e-mail address **license@rowley.co.uk**.

By return you will receive an **activation key**. To activate CrossWorks:

- Run the **CrossStudio** application.
- From the **Tools** menu, click **License Manager**.
- Click the **Activate Product** button.
- Type in or paste the returned activation key into the dialog and click **OK**.

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

Text conventions

Menus and user interface elements

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

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

Keyboard accelerators

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

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

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

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

Code examples and human interaction

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

hcl source-file

This means that the command consists of:

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

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

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

```
CrossWorks MSP430 Compiler Driver   Release 1.0.0  
Copyright (c) 1997-2004 Rowley Associates Ltd.
```

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.

Standard syntactic metalanguage

In a formal description of a computer language, it is often convenient to use a more precise language than English. This language-description language is referred to as a **metalanguage**. The metalanguage which will be used to describe the C language is that specified by British Standard 6154. A tutorial introduction to the standard syntactic metalanguage is available from the National Physical Laboratory.

The BS6154 standard syntactic metalanguage is similar in concept to many other metalanguages, particularly those of the well-known Backus-Naur family. It therefore suffices to give a very brief informal description here of the main points of BS6154; for more detail, the standard itself should be consulted.

- Terminal strings of the language—those built up by rules of the language—are enclosed in quotation marks.
- Non-terminal phrases are identified by names, which may consist of several words.
- When numbers are used in the text they will usually be decimal. When we wish to make clear the base of a number, the base is used as a subscript, for example 15_8 is the number 15 in base eight and 13 in decimal, $2F_{16}$ is the number 2F in hexadecimal and 47 in decimal.
- A sequence of items may be built up by connecting the components with commas.
- Alternatives are separated by vertical bars ('|').
- Optional sequences are enclosed in square brackets ('[' and ']').
- Sequences which may be repeated zero or more times are enclosed in braces ('{' and '}').
- Each phrase definition is built up using an equals sign to separate the two sides, and a semicolon to terminate the right hand side.

Requesting support and reporting problems

With software as complex as CrossWorks, it's almost inevitable that you'll need assistance at some point. Here are some pointers on what to do when you think you've found a problem.

Requesting help

If you need some help working with CrossWorks, please contact our support department by e-mail, ***support@rowley.co.uk***.

Reporting a bug

Should you have a problem with this product which you consider a bug, please report it by e-mail to our support department, ***bugs@rowley.co.uk***.

Support and suggestions

If you have any comments or suggestions regarding the software or documentation, please send these in an e-mail to ***support@rowley.co.uk*** or in writing to:

CrossWorks Customer Support
Rowley Associates Limited
Suite 4B/4C Drake House
Drake Lane
Dursley
Gloucestershire GL11 4HS
UNITED KINGDOM
Tel: +44 1453 549536
Fax: +44 1453 544068

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.

In this section

Activating CrossWorks

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

Managing support packages

Describes how to download, install and view CPU 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 project option inheritance works.

Building projects

Describes how to build the 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 find out how much memory your project takes and navigate around the files that make up the project.

Using the debugger

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

Low-level debugging

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

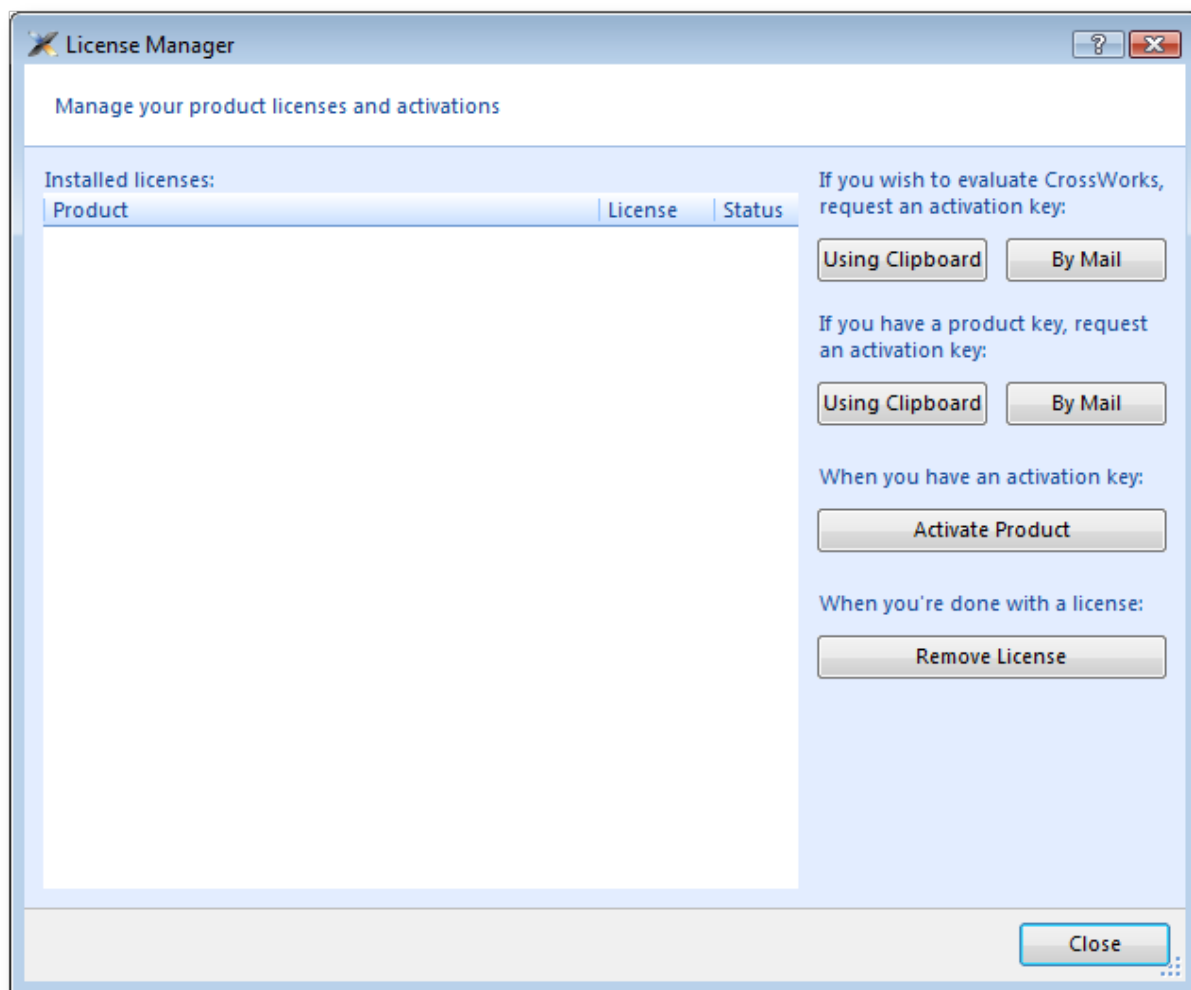
Debugging externally built applications

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

Activating CrossWorks

Each copy of CrossWorks must be registered and activated before it will build projects or download and debug applications. In this tutorial we are going to use CrossStudio's license manager window to request an evaluation activation key and then activate CrossWorks using it.

If you have already activated your copy of CrossWorks then you can skip this page and move on to the [next section](#).



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

To receive an evaluation activation key good for 30 days:

- Open the license manager by selecting the **Tools > License Manager** menu option.
- Click the **By Mail** button underneath the text "If you wish to evaluate CrossWorks...".
- Send the e-mail containing the registration key to the e-mail address **license@rowley.co.uk**.

Requesting an evaluation activation key (without default e-mail client)

To receive an evaluation activation key good for 30 days:

- Open the license manager by selecting the **Tools > License Manager** menu option.
- Click the **Using Clipboard** button underneath the text "If you wish to evaluate CrossWorks...", this will make CrossWorks copy the registration key into the clipboard.
- Compose a new e-mail addressed to **license@rowley.co.uk**.
- Paste the registration key into the body of the e-mail.
- Send the e-mail.

When we receive your registration key we will send an activation key back to the e-mail's reply address. You 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:

- Open the license manager by selecting the **Tools > License Manager** menu option.
- Click the **Activate Product** button.
- Enter the activation key you have received from us.
- Click **OK**.
- The new activation should now be visible in the **Evaluation Licenses** folder along with the expiry date, click **Close** to close the license manager window.

*Please note that if you request an activation key outside office hours, there maybe a delay processing the registration. If this is the case, you can continue the tutorial until you reach the **Building projects** section when you will need to have CrossWorks activated in order to build.*

Managing support packages

Before a project can be created, a CPU 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.

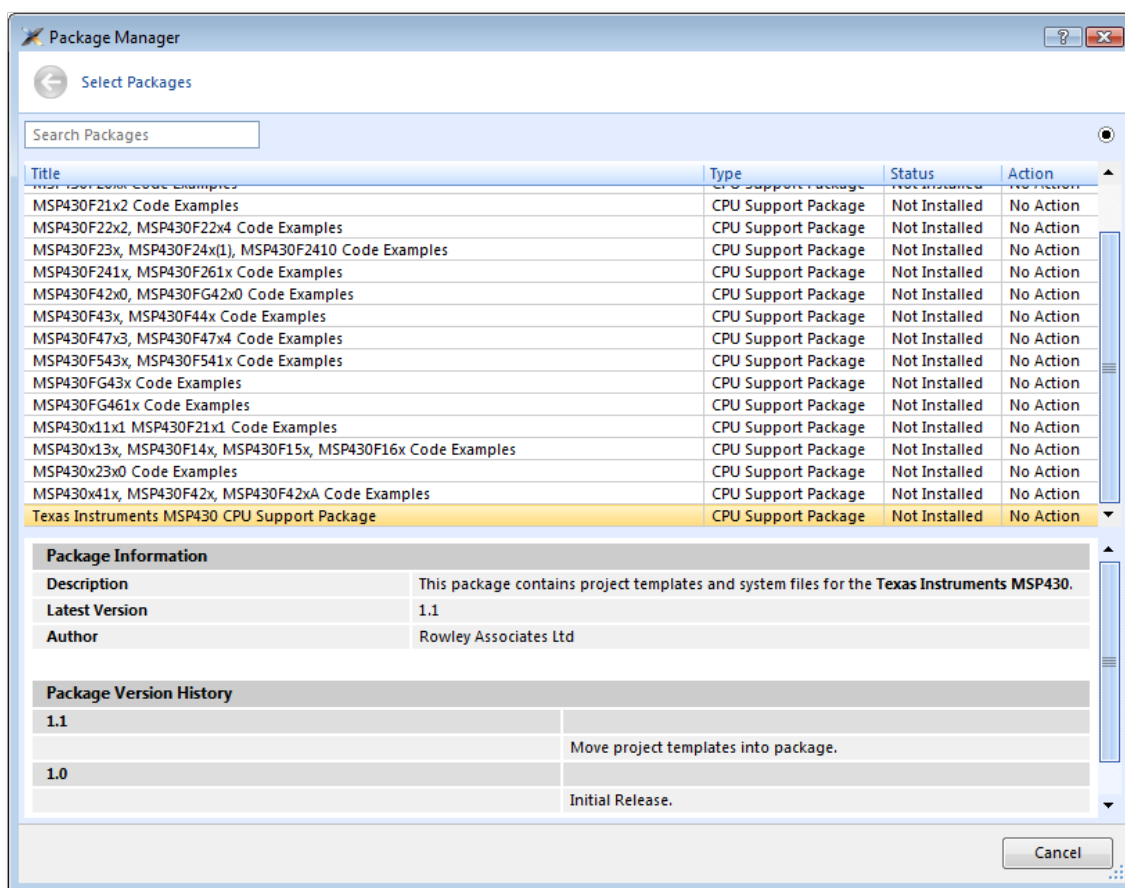
In this tutorial we are going to use CrossStudio's package manager window to download, install and use the **Texas Instruments MSP430 CPU Support Package**.

If you have already installed this support package then you can skip this page and move on to the [next section](#).

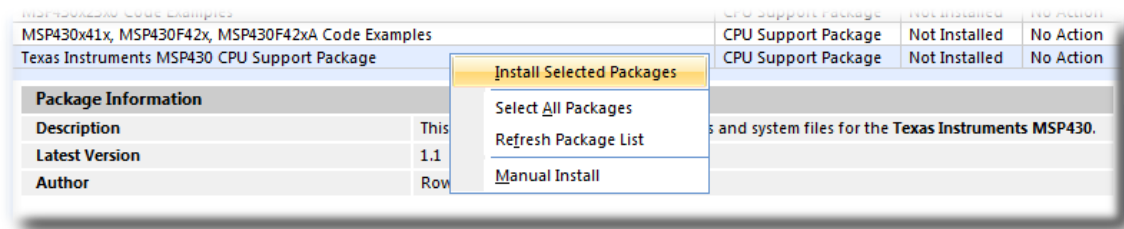
Downloading and installing a support package

To download and install a support package:

- Click **Tools > Package Manager** to view the current set of support packages available.
- Select the **Texas Instruments MSP430 CPU Support Package** entry.



- Right click on the entry and select **Install Selected Packages**.



- Click **Next** and you will be presented with a list of actions that the package manager is going to carry out.
- Click **Next** to download and install the support package.

Viewing installed support packages

To view the installed support packages:

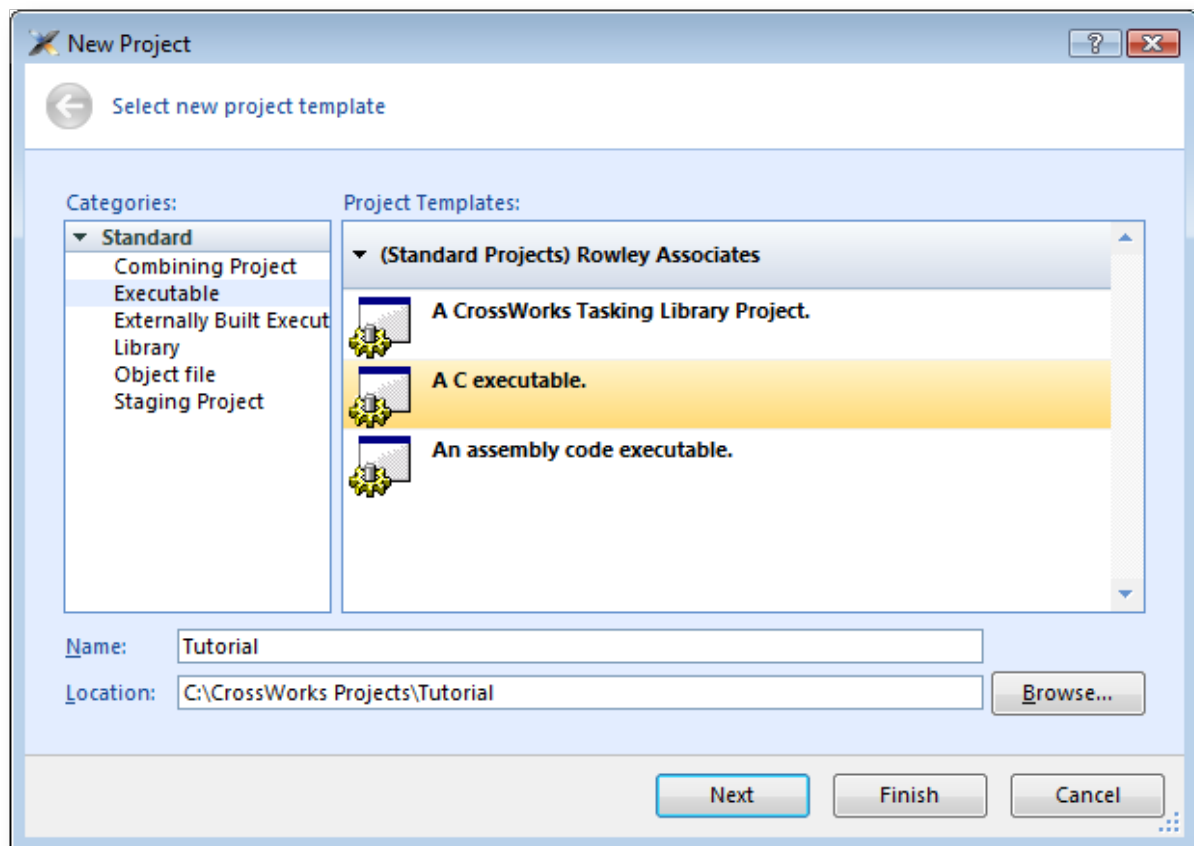
- Click **Tools > Show Installed Packages** to list the support packages you have installed on your system. You should see the **Texas Instruments MSP430 CPU Support Package** we've just installed listed.
- Click **Texas Instruments MSP430 CPU Support Package** to see the support package page. This page will provide more information on the support package and any links to documentation, example projects and system files included in the package.

Creating a project

To start developing an application, you create a new project. To create a new project, do the following:

- From the **File** menu, click **New** then **New Project...**

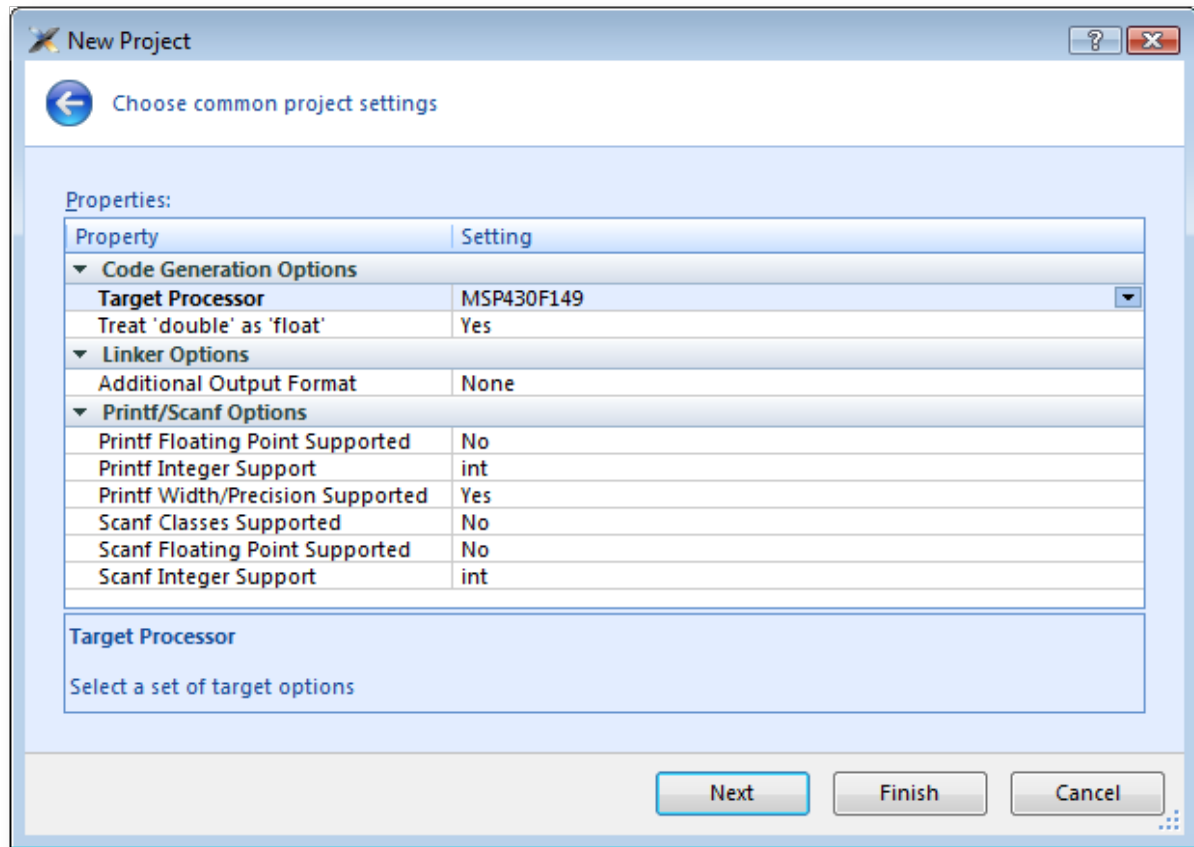
The **New Project** dialog appears. This dialog displays the set of project types and project templates.



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

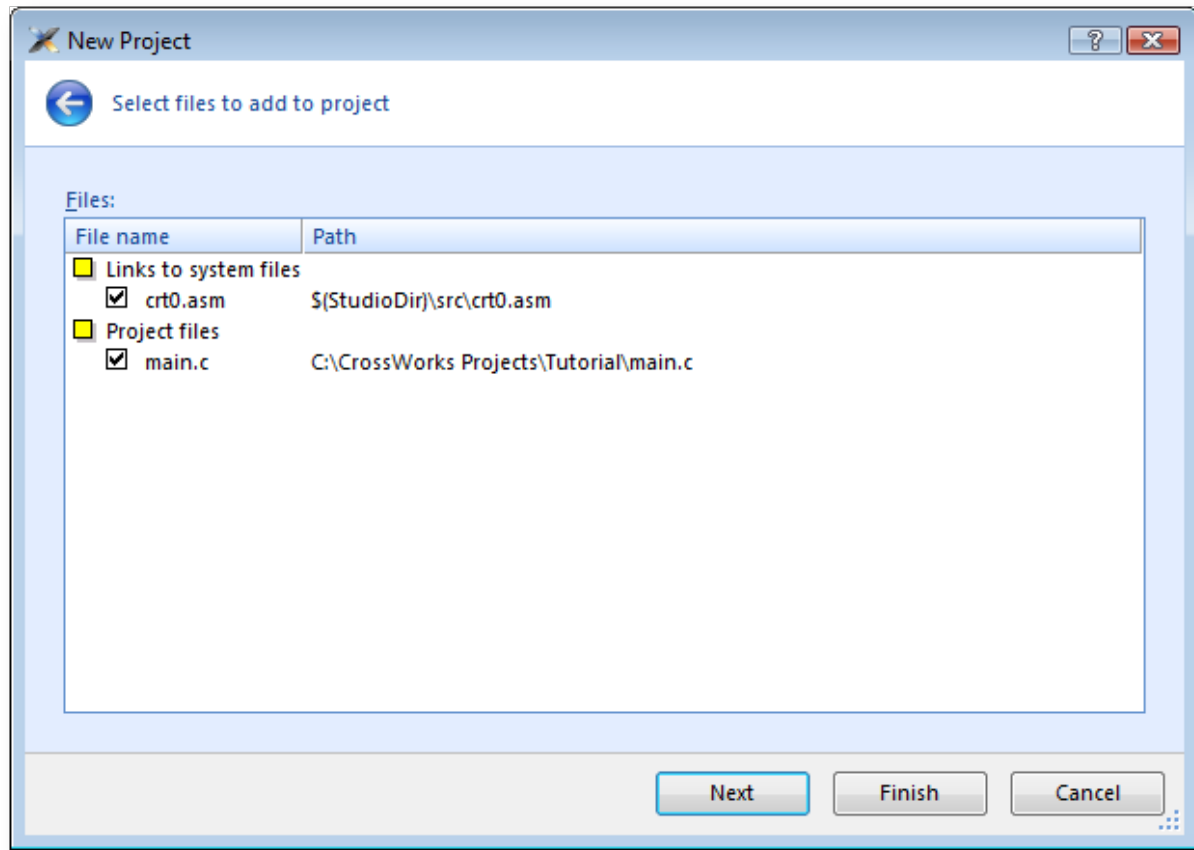
- Select the **Standard > Executable** project type in the **Categories** pane.
- Select the **A C Executable** icon in the **Project Templates** pane which selects the type of project to add.
- Type **Tutorial** in the **Name** edit box, which names the project.
- You can use the **Location** edit box or the **Browse** button to locate where you want the project to be created.
- Click **Next**.

Once created, the project setup wizard prompts you to set some common options for the project.



Here you can customise 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**. You can change these settings after the project is created using the Project Explorer.

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

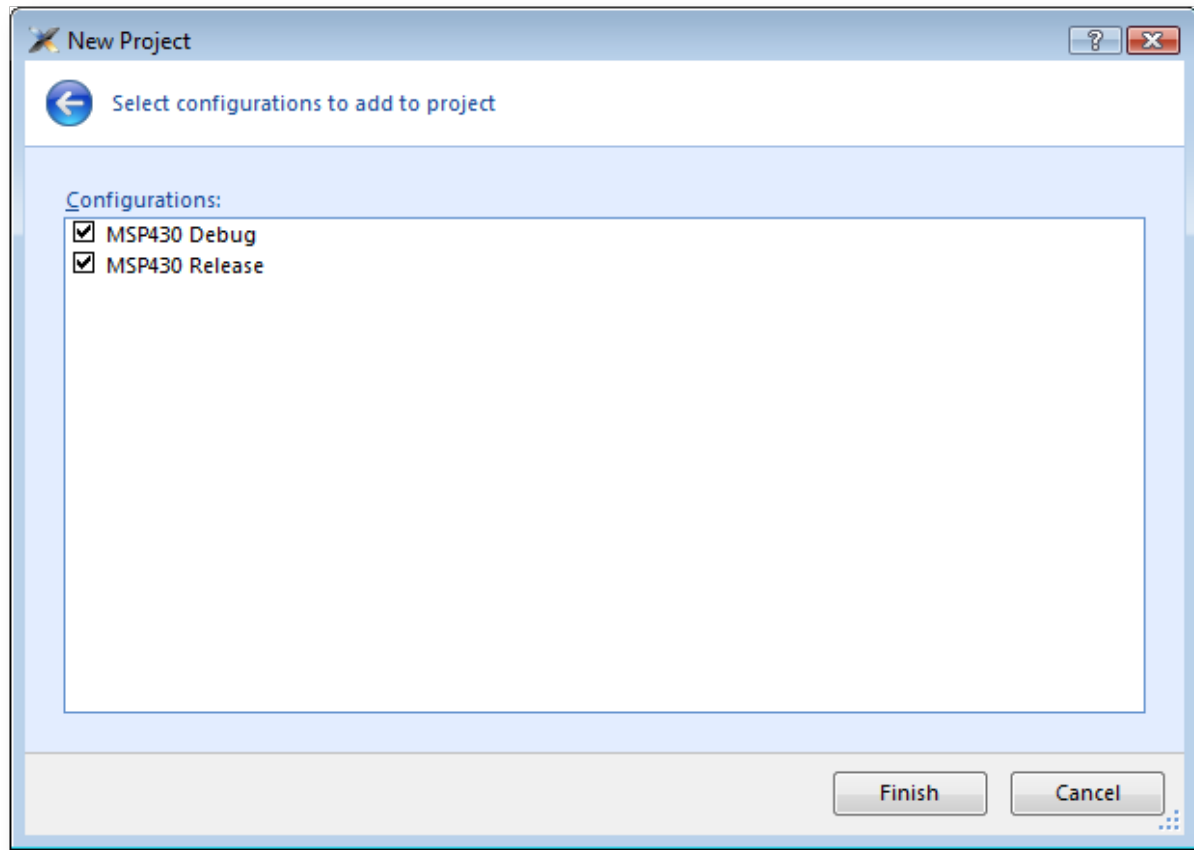


The **Links to system files** group shows the links that will be created in the project to CrossStudio system files. As these files are links they will, by default, be shared with other projects so modifying one will effect all projects containing similar links. To prevent accidental modification, these files are created as read-only. Should you wish to modify a shared file without effecting other projects you can do so by importing them into the project first. Importing a shared file will be demonstrated later in this tutorial. Project links are fully explained in [Project management](#).

The **Project files** group shows the files that will be copied into the project. As these files are copied to the project directory they can be modified without effecting 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.

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



Here you can specify the default configurations that will be added to the project. See [Project management](#) for more information on project configurations.

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

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

- From the **View** menu, click **Project Explorer**.

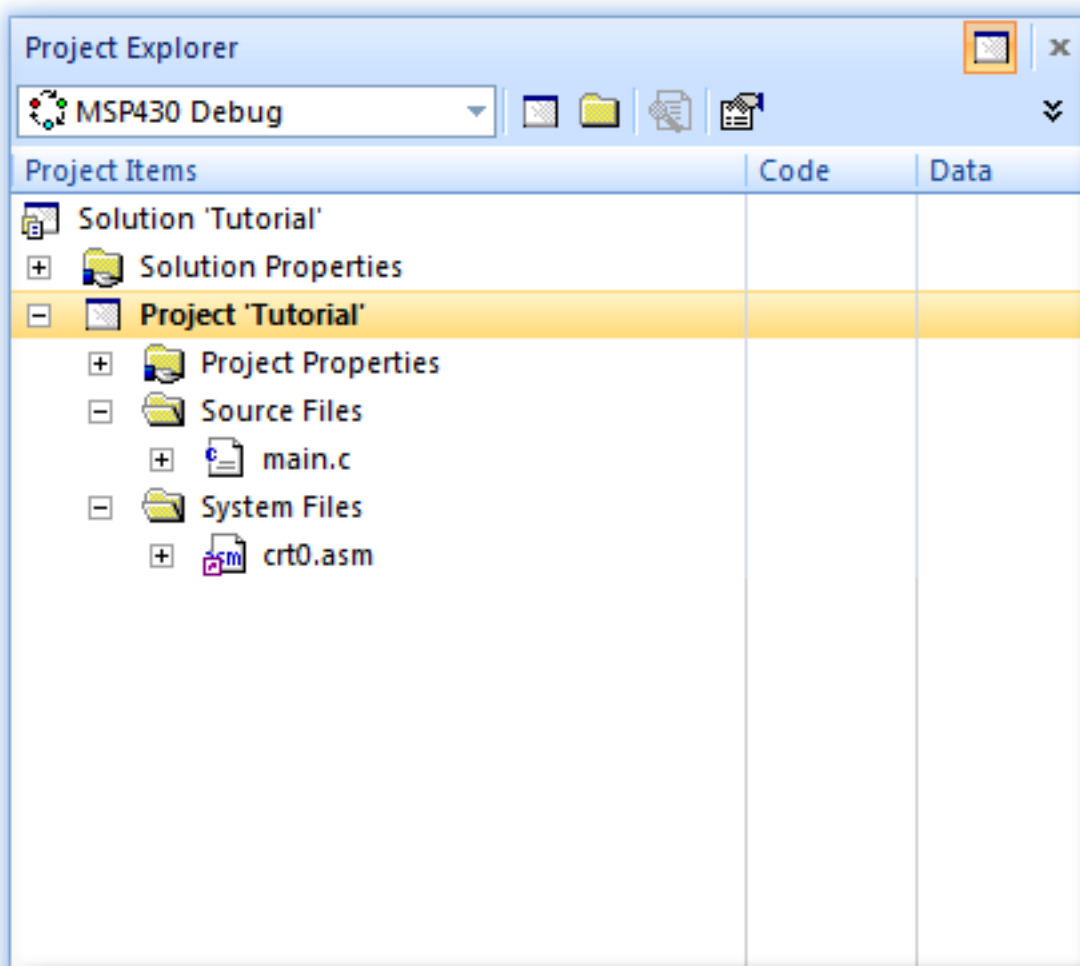
—or—

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

—or—

- Right click the tool bar area.
- From the popup menu, select **Project Explorer**.

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



You'll notice that the project name is shown in bold which indicates that it is the active project (and in our case, the only project). If you have more than one project then you can set the active project using the dropdown box on the build tool bar or the context menu of the project explorer.

The files are arranged into two groups:

- **Source Files** contains the main source files for your application which will typically be 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, yet are required when the project is built and run. In this case, the system files are `crt0.asm` which is the C runtime startup written in assembly code. Files which are stored outside of the project's home directory are shown by 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 together in the project explorer. You can create new folders and specify filters based on the file extension so that when you add a new file to the project it will be placed in the folder whose filter matches the file extension.

Managing files in a project

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

Adding an existing file to a project

We will add one of the tutorial files to the project. To add an existing file to the project, do the following:

- From the **File** menu, click **Add Existing File**.

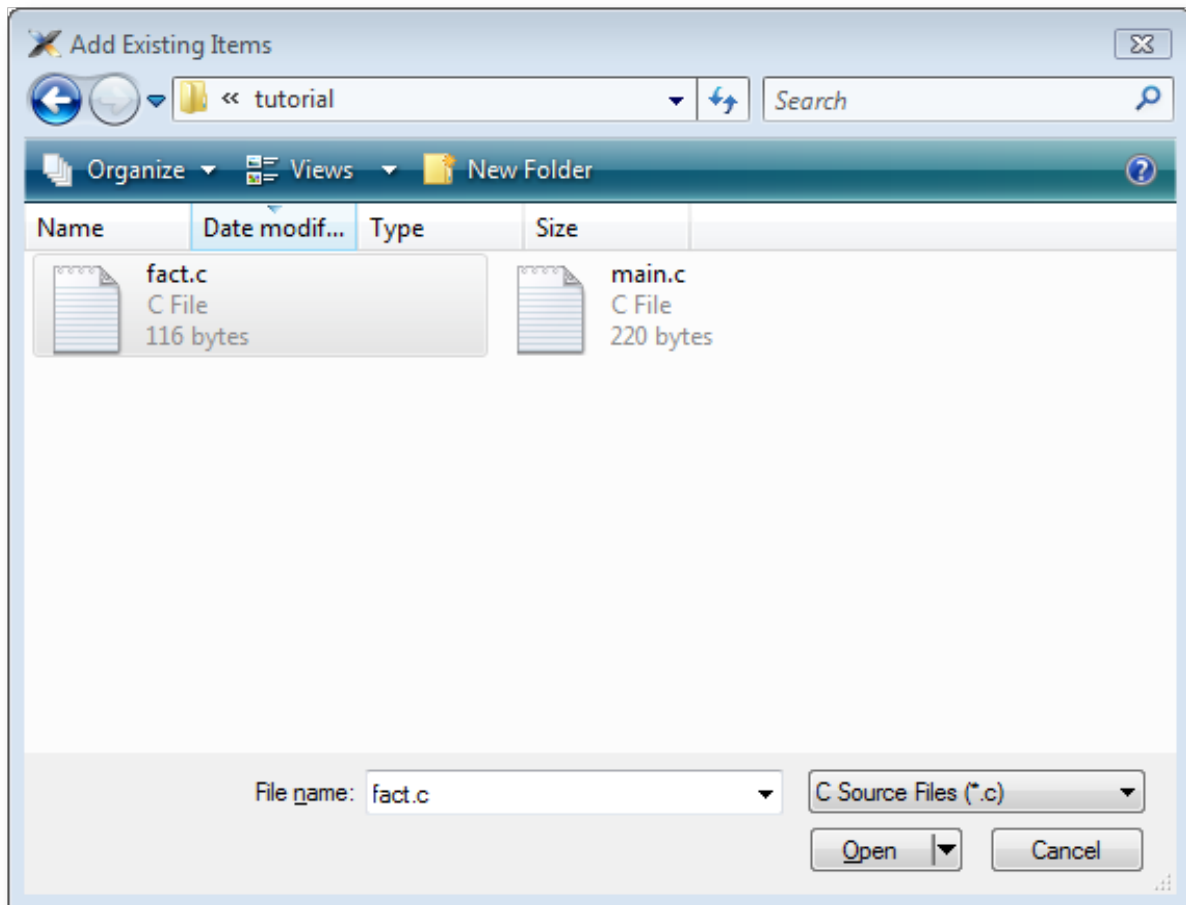
—or—

- Type **Ctrl+D**.

—or—

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

When you've done this, CrossStudio displays a standard file locator dialog. Navigate to the CrossStudio installation directory, then to the `tutorial` folder, select the `fact.c` file.

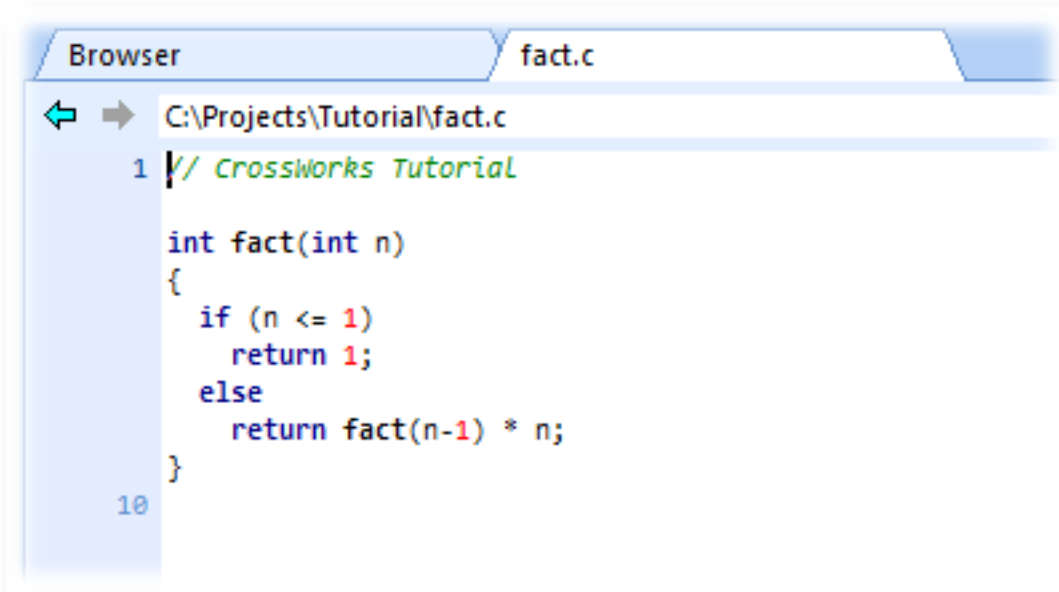


Now click **Open** to add the file to the project. The Project Explorer will show `fact.c` with a shortcut arrow because the file is not in the project's home directory. Rather than edit the file in the tutorial directory, we'll take a copy of it and put it into the project home directory:

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

The shortcut arrow disappears from the `fact.c` node which indicates that the file is now in our project home directory.

We can open a file for editing by double clicking the node in the Project Explorer. Double clicking `fact.c` brings it into the code editor:



Removing a file from a project

We don't need the `main.c` file that the new project wizard added to the project, so we will remove it. Click `main.c` in the Project Explorer and do one of the following:

- On the **Project Explorer** tool bar, click the **Delete** tool button

—or—

- From the **Edit** menu, click **Delete**.

—or—

- Type **Del**.

Alternatively, to remove `main.c` from the project using a context menu, do the following:

- In the **Project Explorer**, right click `main.c`.
- From the context menu, click **Remove**.

Adding a new file to a project

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

- From the **Project** menu, click **Add New File**.

—or—

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

—or—

- In the **Project Explorer**, right click the **Tutorial** node.
- From the context menu, click **Add New File**.

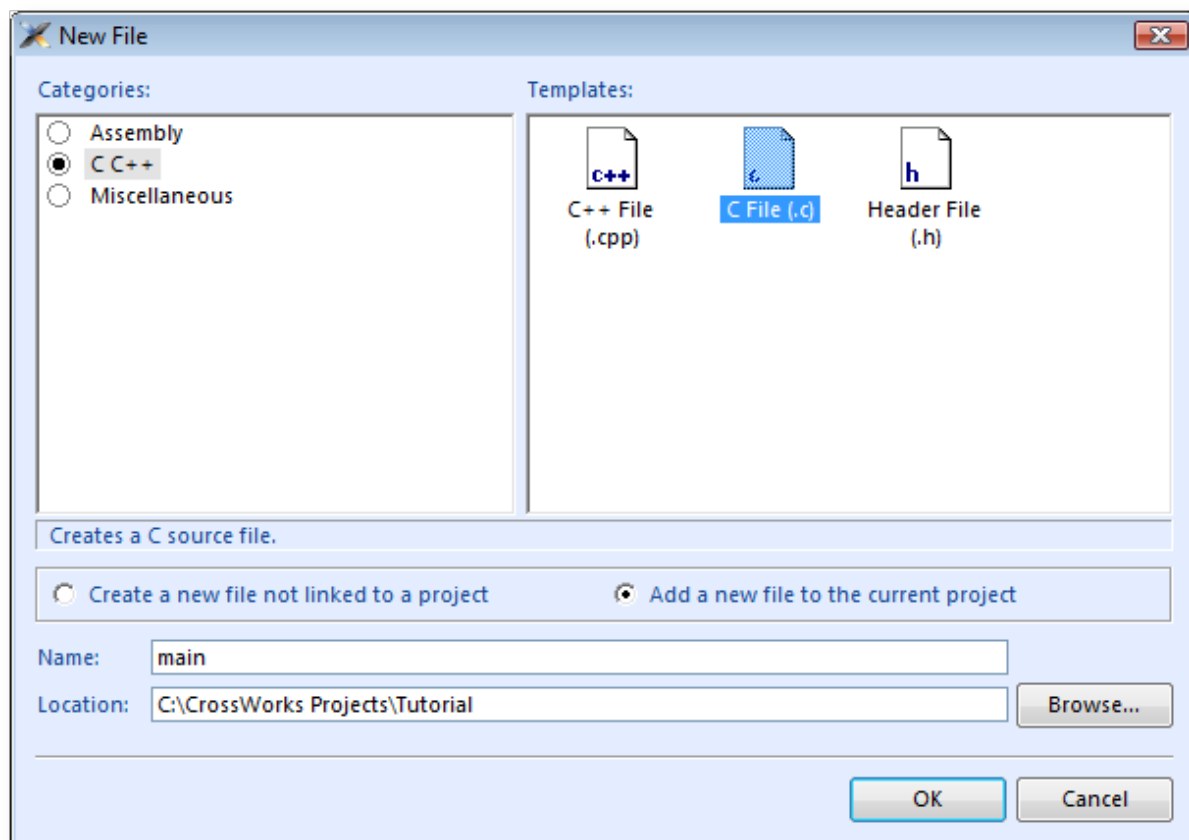
—or—

- Type **Ctrl+N**.

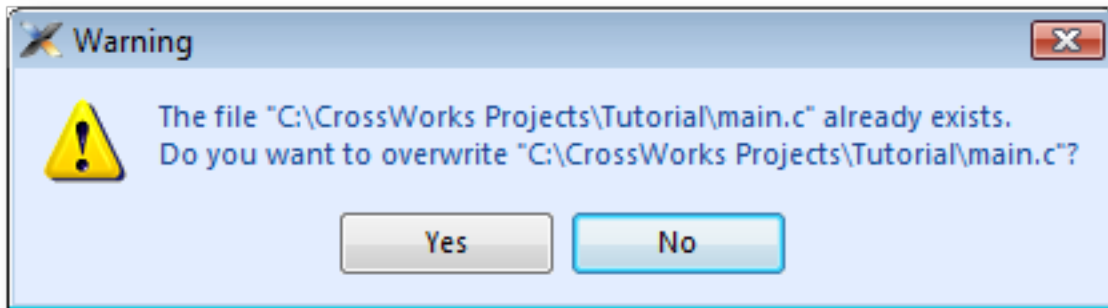
The **New File** dialog appears.

- Ensure that the **C File (.c)** icon is selected.
- In the **Name** edit box, type `main`.

The dialog box will now look like this:



Click **OK** to add the new file. Because `main.c` already exists on disk, you will be asked whether you wish to overwrite the file:

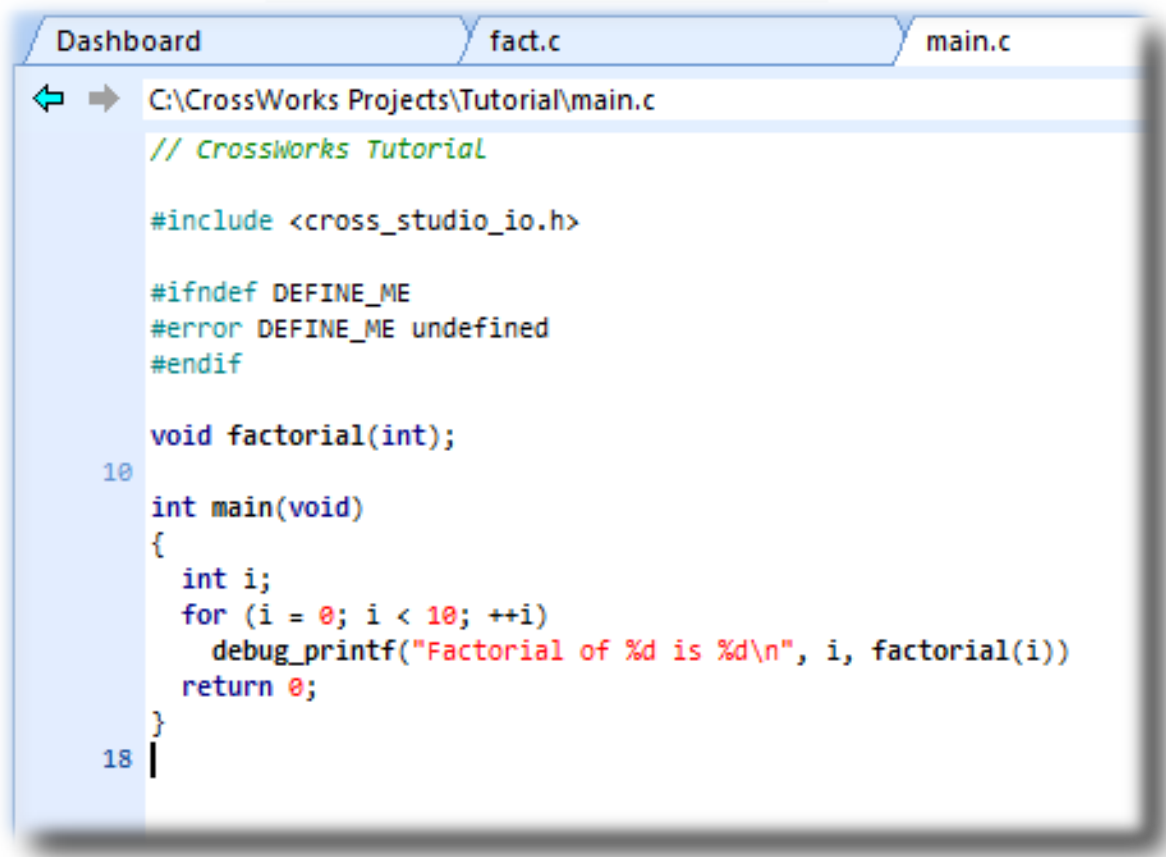


Click **Yes** to overwrite the file and continue with the tutorial.

CrossStudio opens an editor with the new file ready for editing. Rather than type in the program from scratch, we'll add it from a file stored on disk:

- From the **Edit** menu, click **Insert File** or type **Ctrl+K, Ctrl+I**.
- Using the file browser, 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

You have now created a simple project, and in this section we will set some options for the project.

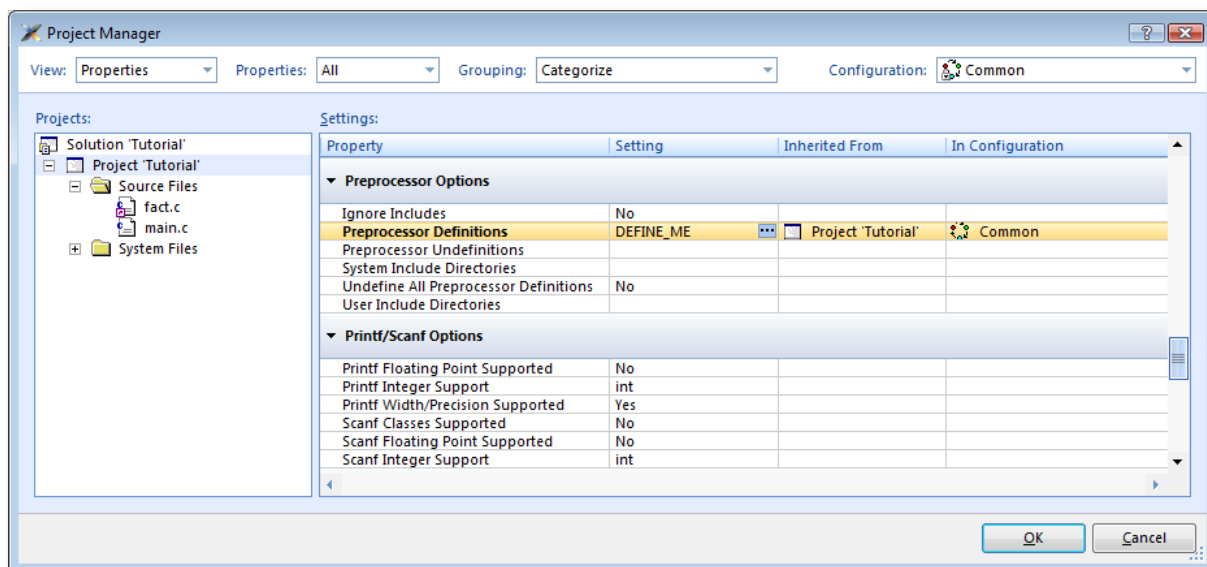
You can set project options on any node of the solution. That is, you can set options on a solution-wide basis, on a project-wide basis, on a project group basis, or on an individual file basis. For instance, options that you set on a solution are inherited by all projects in that solution, by all groups in each of those projects, and then 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) that node. The way that options are inherited provides a very 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 Tutorial project, this means that every file in the Tutorial project will inherit the definition. If however we were to add any further projects to the solution they would not inherit the definition, if we wanted to do that we would 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 Options** dialog appears.
- Click the **Configuration** dropdown and change to the **Common** configuration.
- Click the **Preprocessor Options > Preprocessor Definitions** property and add 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 which options are used when building a project (or anything in a project) in a given configuration. Because we have set the 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—these have all been inherited from the solution. If you modify a property when the project node is selected then you'll find that its value is highlighted because you have overridden the property value that was inherited from the solution.

You can restore the inherited value of a property by right clicking the property and selecting **Use Inherited Value** from the menu.

Next, we'll build the project.

Building projects

Now that the project is created and set up, it's time to build it. Unfortunately, there are some deliberate errors in the program which we need to correct.

Setting the build configuration

The first thing to do is set the active build configuration you want to use. To set the active build configuration, do the following:

- From the **Build** menu, click **Set Active Build Configuration** and select **MSP430 Debug**.

This signifies that we are going to use a build configuration that generates code with debug information and no optimisation so that it can be debugged. If we wanted to produce production code with no debug information and optimisation enabled we could use the **MSP430 Release** configuration however as we are going to use the debugger we shall stick with the **MSP430 Debug** configuration.

Building the project

To build the project, do the following:

- From the **Build** menu, click **Build Tutorial**.

—or—

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

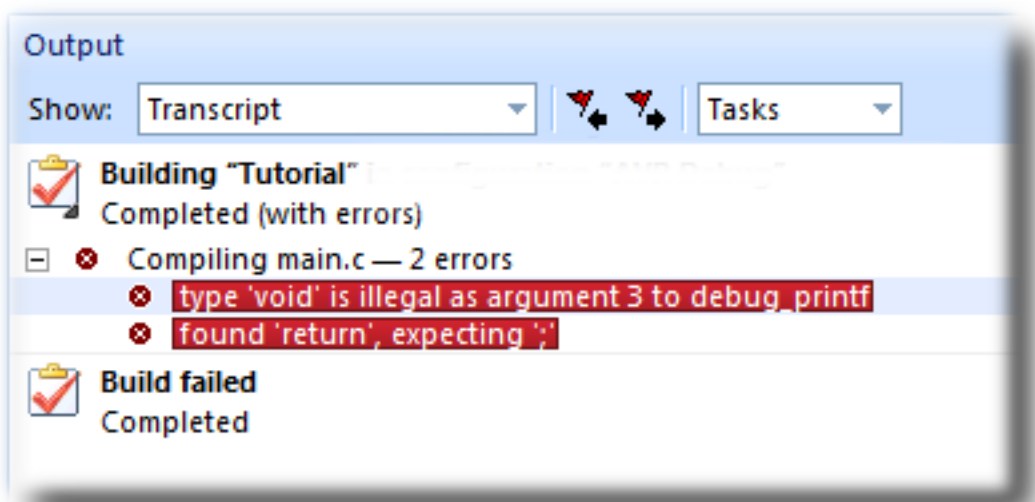
—or—

- Type **F7**.

Alternatively, to build the **Tutorial** project using a context menu, do the following:

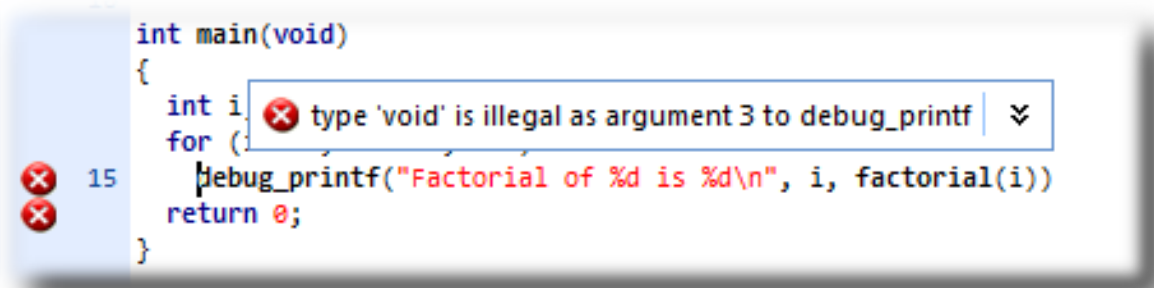
- In the **Project Explorer**, right click the **Tutorial** project node.
- Select **Build** from the context menu.

CrossStudio starts compiling the project files but finishes after detecting an error. The Output Window shows the Build Log which contains the errors found in the project:



Correcting compilation and linkage errors

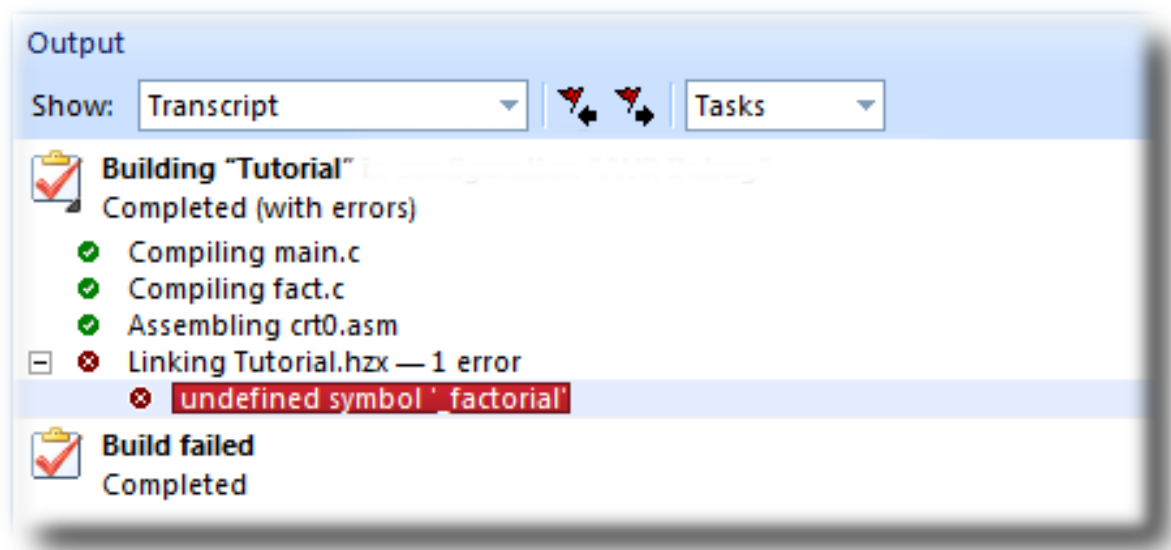
The file `main.c` contains two errors. After compilation, CrossStudio moves the cursor to the line containing the first reported error and displays the error message (You can change this behaviour by modifying the **Text Editor** > **Editing Options** > **Enable Popup Diagnostics** environment option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



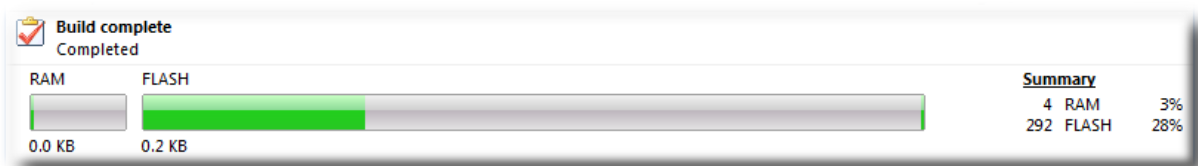
To correct the error, change the return type of `factorial` from `void` to `int` in its prototype.

To move the cursor to the line containing the next error, type **F4** or from the **Search** menu, click **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 indicates 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 **Previous Location** in the **Search** menu to move back through errors. Now that the errors are corrected, build the project again by pressing **F7**. The build log still shows that we have a problem.



The remaining error is a linkage error. Double click on `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 not match these, so don't worry if they don't.

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

Exploring projects

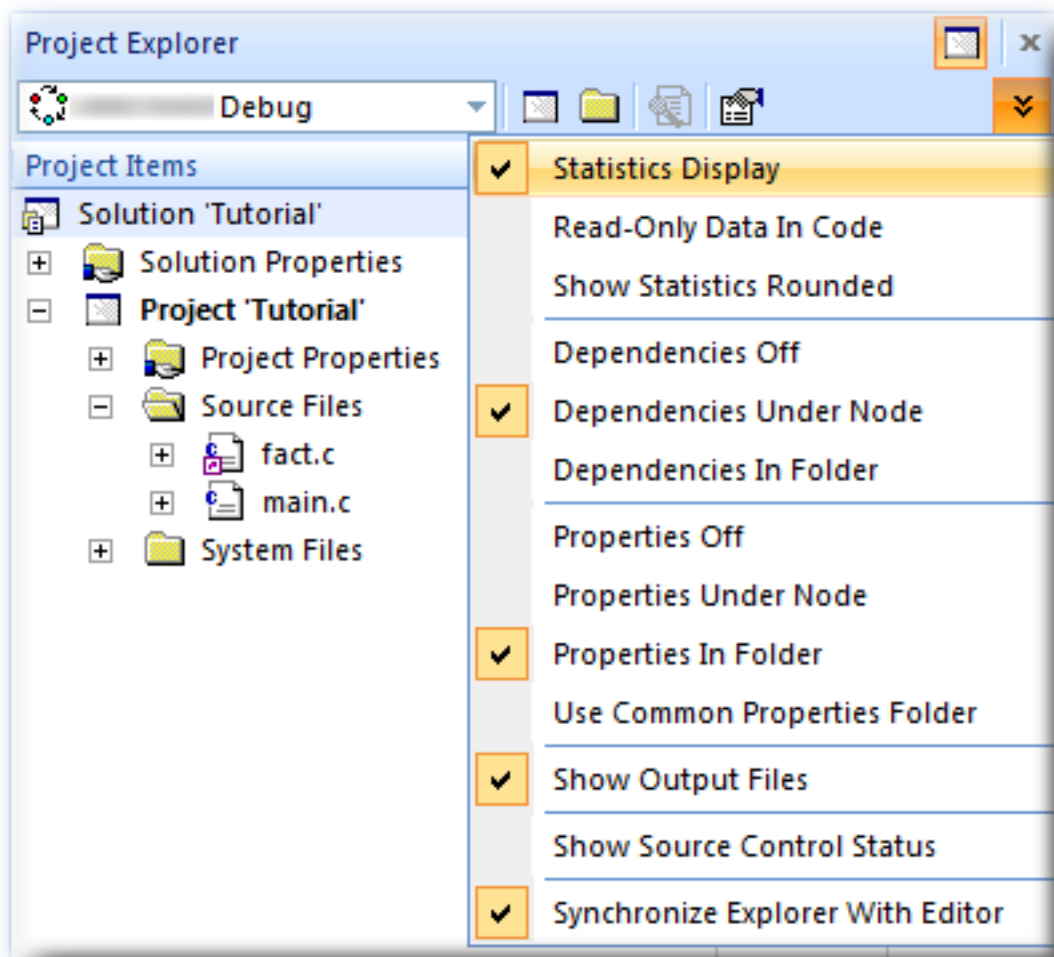
Now that the project has no errors and builds correctly, we can turn our attention to uncovering exactly how our application fits in memory and how to navigate around it.

Using Project Explorer features

The Project Explorer is the central focus for arranging your source code into projects, and it's a good place to show ancillary information gathered when CrossStudio builds your applications. This section will cover the features that 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 take up, and with small embedded microcontrollers this is especially true. The Project Explorer can display the code and data sizes for each project and individual source file that is successfully compiled. To do this, click the **Options** dropdown on the **Project Explorer** tool bar and make sure that **Statistics Display** is checked.



Once checked, the Project Explorer displays two additional columns, **Code** and **Data**.

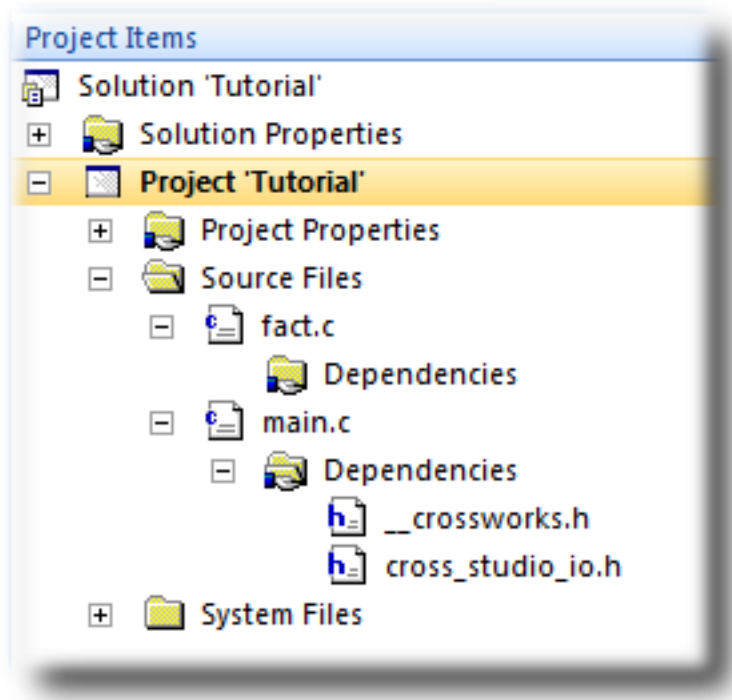
Project Items		Code	Data
Solution 'Tutorial'			
+	Solution Properties		
-	Project 'Tutorial'	2,148	28
+	Project Properties		
-	Source Files		
+	fact.c	80	
+	main.c	100	24
+	System Files		

The **Code** column displays the total code space required for the project and the **Data** column displays the total data space required. The code and data sizes for each C and assembly source file are *estimates*, but good estimates nonetheless. 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 before, 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 the project as a whole, you can also configure the Project Explorer to show the *dependencies* for a file. As part of the compilation process, CrossStudio finds and records the relationships between files—that is, it finds which files are dependent upon other files. CrossStudio uses these relationships when it comes to build the project again so that it does the minimum amount of work to bring the project up to date.

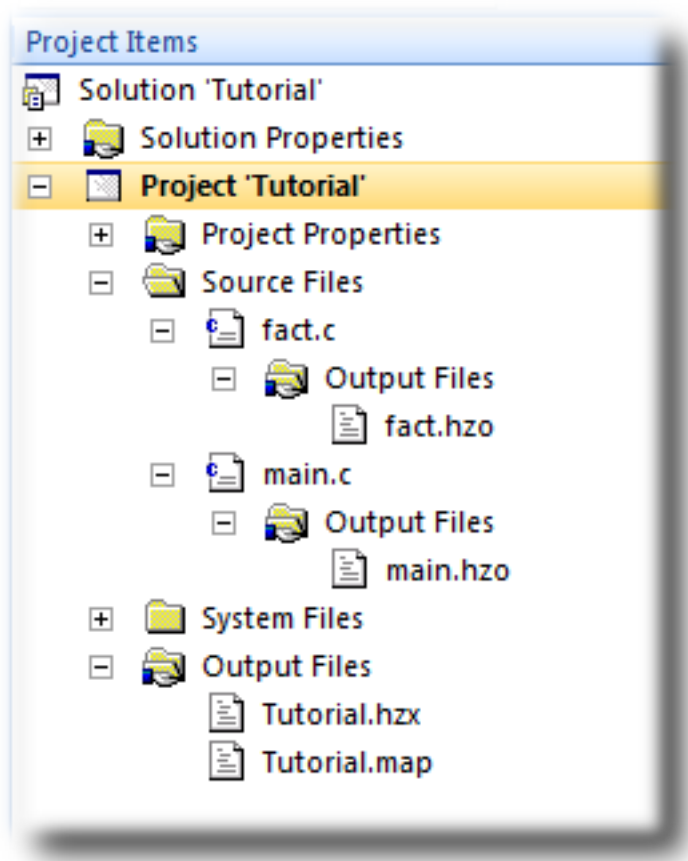
To show the dependencies for a project, click the **Options** button on the **Project Explorer** tool bar once again and ensure that either **Dependencies Under Node** or **Dependencies In Folder** are checked in the menu. Once checked, dependent files are shown as sub-nodes of the file which depends upon them.



In this case, `main.c` is dependent upon `cross_studio_io.h` because it includes it with a **#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 on them, so having dependencies turned on is an effective way of navigating to and summarising the files that a source file includes.

Output files

Another useful piece of information is knowing the output files when compiling and linking the application, CrossStudio can display this information too. To turn on output file display, click the **Options** button on the **Project Explorer** tool bar and ensure that **Show Output Files** is checked in the menu. Once checked, output files are shown in an **Output Files** folder underneath the node that generates them.



In the above figure, we can see that the object files `fact.hzo` and `main.hzo` are object files produced by compiling their corresponding source files; the map file `Tutorial.map` and the linked executable `Tutorial.hzx` 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 as described above, or you can use the Disassemble tool to do it.

To disassemble a project or file, do one of the following:

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

CrossStudio opens a new read-only editor and places a disassembled listing into it. If you change your project and rebuild it, causing a change in the object or executable file, the disassembly updates to keep the display up-to-date with the file on disk.

Using Memory Usage Window features

The **Memory Usage Window** can be used to display a graphical summary of how memory has been used in each memory segment of a linked application.

Displaying the Memory Usage Window

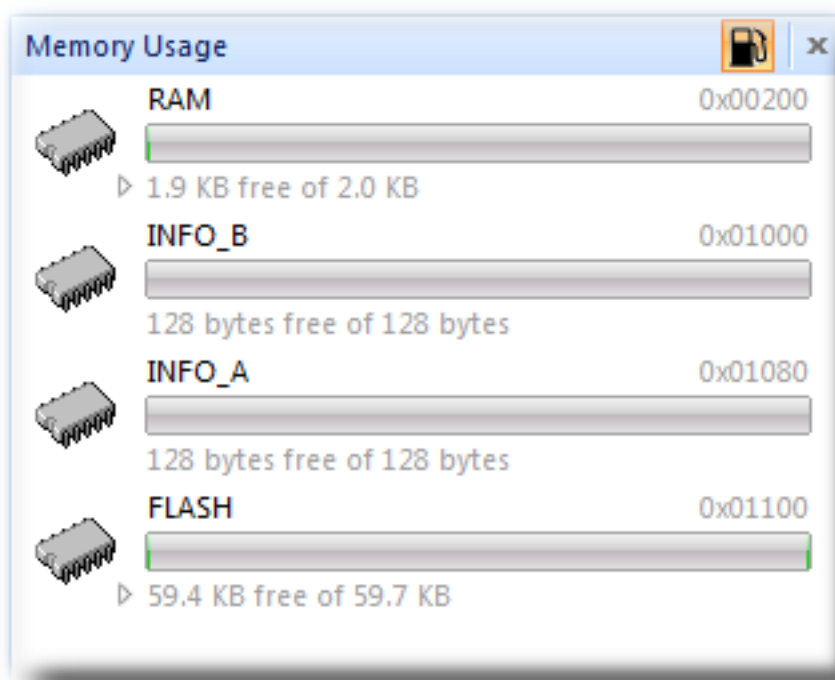
To display the **Memory Usage Window** if it is hidden, do one of the following:

- From the **View** menu, click **Memory Usage**.

—or—

- Type **Ctrl+Alt+Z**.

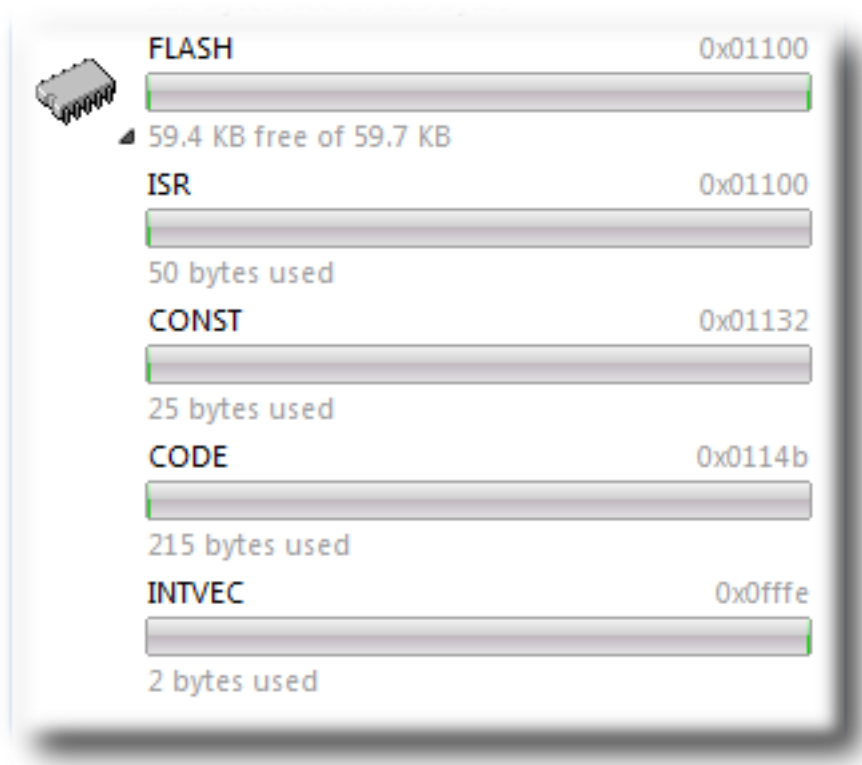
The **Tutorial** project shows this in the **Memory Usage Window**:



From this you can see:

- The **RAM** segment is located at 0200, is 2 KB in length and has 1.9 KB of unused memory.
- The **FLASH** segment is located at 1110, is 59.7 KB in length and has 59.4 KB of unused memory.

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



From this you can see that the the **CODE** section is located at and is 215 bytes in length.

Using Symbol Browser features

If you need 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**. The **Symbol Browser** 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.

Displaying the Symbol Browser

To display the **Symbol Browser** window if it is hidden, do one of the following:

- From the **Tools** menu, click **Symbol Browser**.

—or—

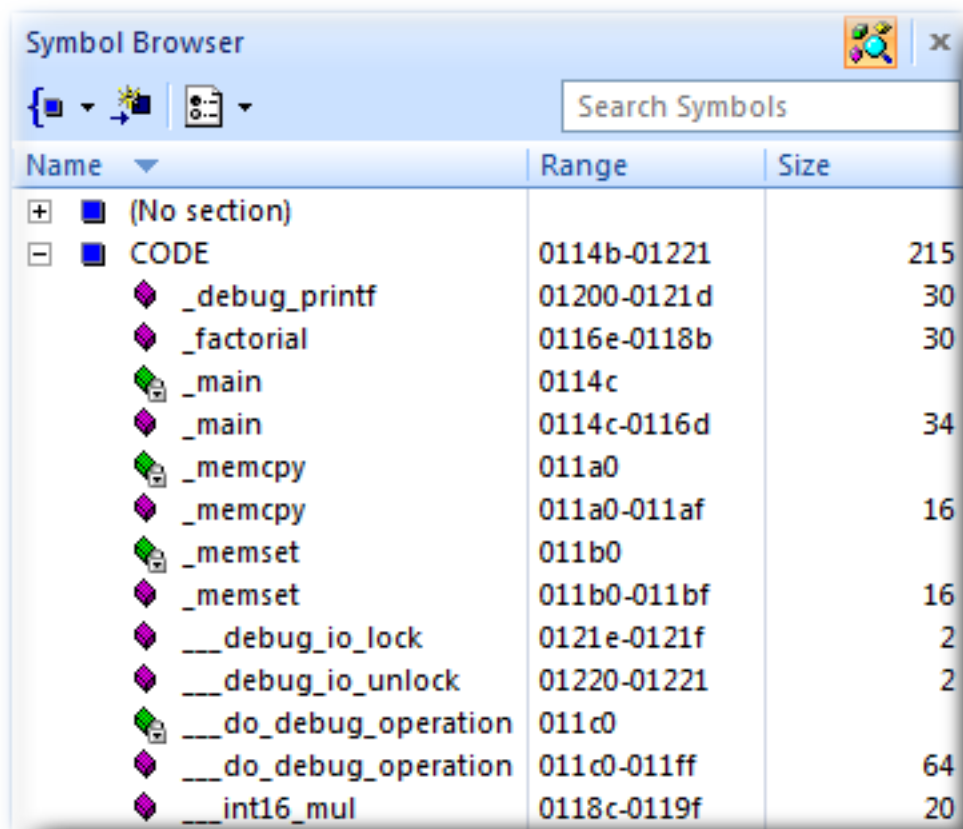
- Type **Ctrl+Alt+Y**.

Drilling down into the application

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

From this you can see that the **CODE** section is 215 bytes in size and is placed in memory between address 114B and 1221 inclusive. Similarly, the zeroed data section **UDATA0** is 2 bytes in size and is placed between 0202 and 0203, the **CONST** section that holds string constants and read-only data is 25 bytes in size between 1132 and 114A. You can click the header to order sections by their address by clicking on **Range** and by their size by clicking **Size**.

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



Name	Range	Size
(No section)		
CODE	0114b-01221	215
_debug_printf	01200-0121d	30
_factorial	0116e-0118b	30
_main	0114c	
_main	0114c-0116d	34
_memcpy	011a0	
_memcpy	011a0-011af	16
_memset	011b0	
_memset	011b0-011bf	16
__debug_io_lock	0121e-0121f	2
__debug_io_unlock	01220-01221	2
__do_debug_operation	011c0	
__do_debug_operation	011c0-011ff	64
__int16_mul	0118c-0119f	20

Here, we can see that **main** is 34 bytes in size and is placed in memory between addresses 114C and 116D inclusive and that **factorial** is 30 bytes and occupies addresses 116E through 118B. Just as in the Project Explorer, you can double click a function and CrossStudio moves the cursor to the line containing the definition of that function, so you can easily navigate around your application using the Symbol Browser.

Printing Symbol Browser contents

You can print the contents of the Symbol Browser by focusing the Symbol Browser window and selecting **Print** from the **File** menu, or **Print Preview** if you want to see what it will look like before printing. CrossStudio prints only the columns that you have selected for display, and prints items in the same order they are displayed in the

Symbol Browser, so you can choose which columns to print and how to print symbols by configuring the Symbol Browser display before you print.

We have touched on only some of the features that the Symbol Browser offers; to find out 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 using the features of CrossStudio to see how it performs.

Getting set up

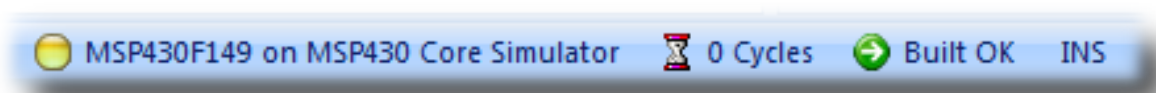
Before running your application, you need to select the target to run it on. The Targets window lists each target interface that is defined, as does the Targets menu, and you use these to connect CrossStudio to a target. For this tutorial, you'll be debugging on the simulator, not hardware, to simplify matters. To connect to the simulator, do one of the following:

- From the **Target** menu, click **Connect > MSP430 Core Simulator**.

—or—

- From the **View** menu, click **Targets** to focus the Targets window.
- In the Targets window, double click **MSP430 Core Simulator**.

After connecting, the connected target is shown in the status bar:



The color of the LED in the Target Status panel changes according to what CrossStudio and the target are doing:

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

Because the core simulator target can accurately count the cycles spent executing your application, the status bar shows the cycle counter panel. If you connect a target that cannot provide performance information, the cycle counter panel is hidden. Double clicking the Target Status panel will show the Targets window and double clicking the Cycle Counter panel will reset the cycle counter to zero.

Setting a breakpoint

CrossStudio will run a program until it hits a breakpoint. We'll place a breakpoint on the call to `debug_printf` in `main.c`. To set the breakpoint, Move the cursor to the line containing `debug_printf` and do one of the following:

- Click the **Toggle Breakpoint** from the **Debug** menu.

—or—

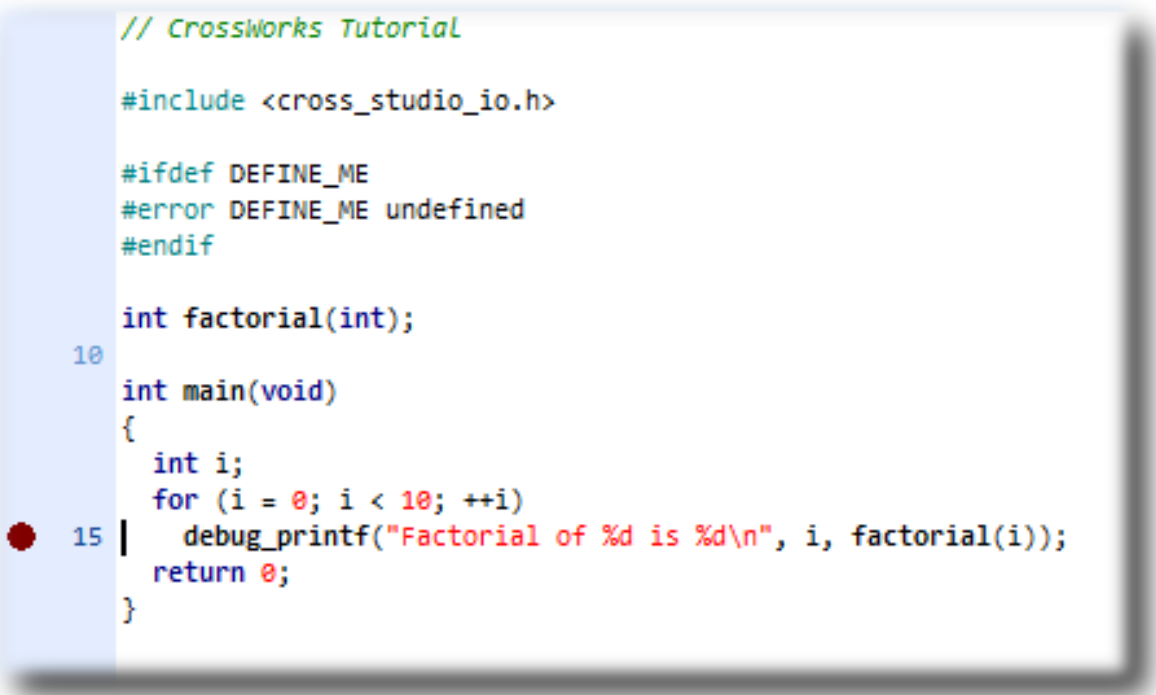
- On the **Build** tool bar, click the **Toggle Breakpoint** button —



—or—

- Type **F9**.

Alternatively, you can set a breakpoint without moving the cursor by clicking in the gutter of the line to set the breakpoint on.



The gutter displays an icon on lines where the 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 that you set are stored in the session file associated with the project which means that your breakpoints are remembered if you exit and re-run CrossStudio.

Starting the application

You can now start the program in one of these ways:

- From the **Debug** menu, click **Go**.

—or—

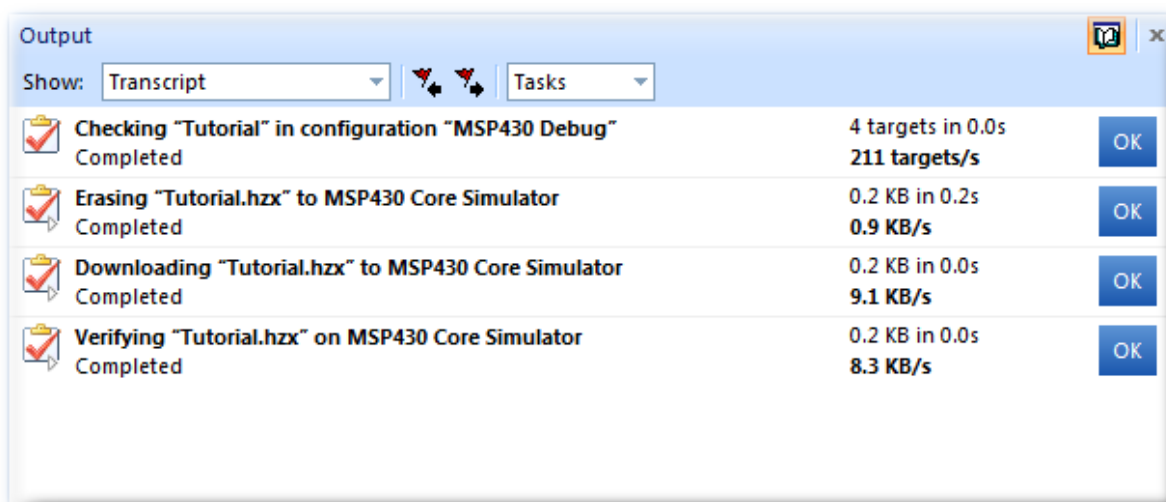
- On the **Build** tool bar, click the **Start Debugging** button —



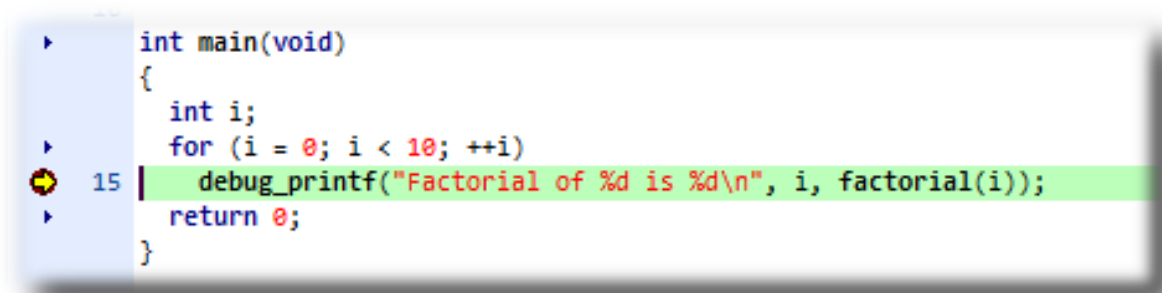
—or—

- Type **F5**.

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

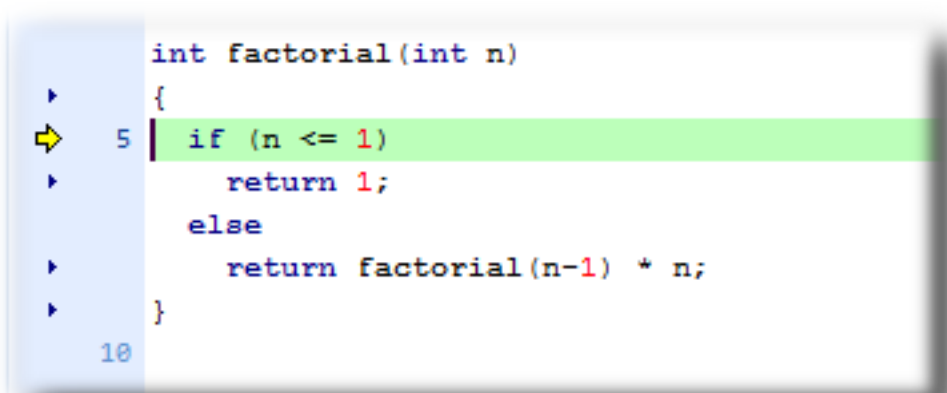


The program stops at our breakpoint and a yellow arrow indicates where the program is paused.



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

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

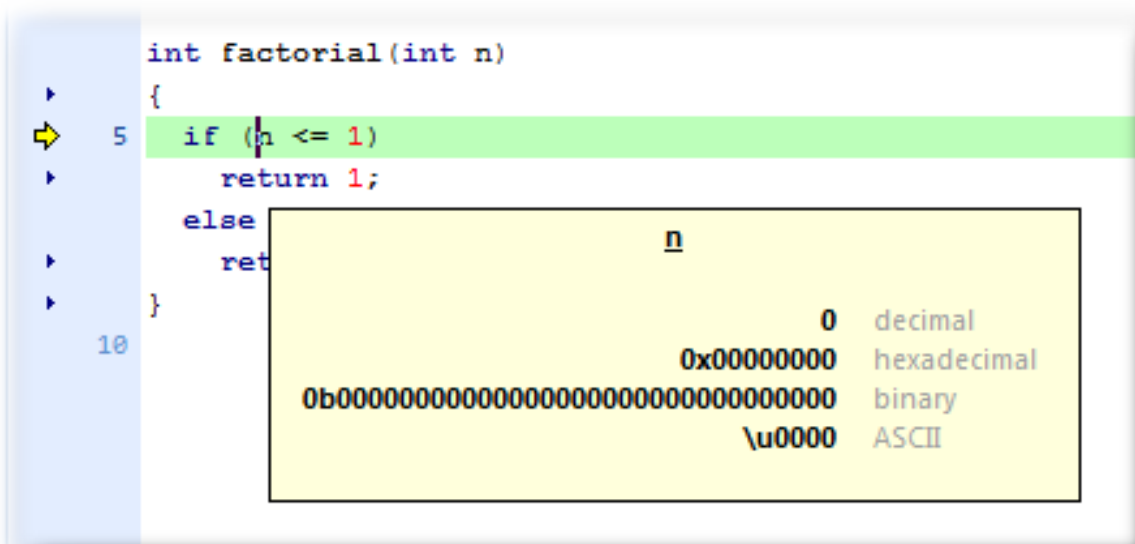


You can step out of a function by selecting **Debug > Step Out**, typing **Shift+F11** or clicking the **Step out** button on the **Debug** tool bar. You can also step to a specific statement using **Debug > Run To Cursor**. To restart your application to run to the next breakpoint use **Debug > Go**.

Note that when single stepping you may step into a function that the debugger cannot locate source code for. In this case 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 are may be cases in which the debugger cannot display the instructions, in these cases you will informed of this with a dialog and you should step out.

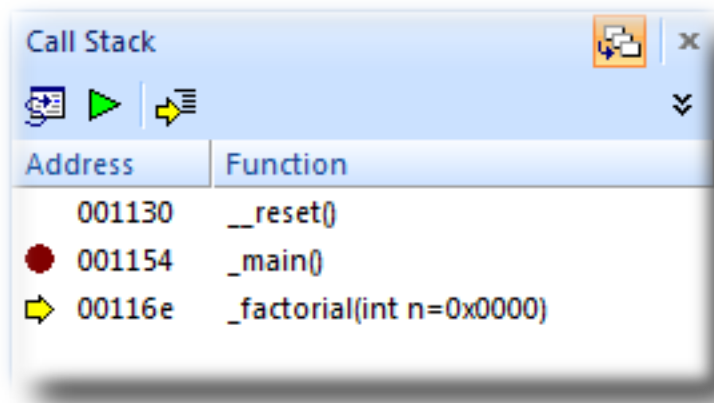
Inspecting data

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 pointer over a variable will show its value as a *data tip*:



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

The Call Stack window shows the function calls that have been made but have not yet finished, i.e. the active set of functions. To display the Call Window, select **Debug > Debug Windows > Call Stack**, or type **Ctrl+Alt+S**.



You can find out about the call stack window in the [Call stack window](#) section.

Program output

The tutorial application uses the function `debug_printf` to output a string to the Debug Console in the Output Window. The Debug Console appears automatically whenever something is written to it—pressing **F5** to continue program execution and you will notice that the Debug Console 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 all very well, but there are occasions where you need to look a little more closely into the way that your program executes to track down the causes of difficult-to-find bugs and CrossStudio provides the tools you need to do just this.

Setting up again

What we'll now do is run the sample application, but 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 **Stop** from the **Debug** menu, or clicking the **Stop Debugging** button on the **Debug** tool bar. Now run the program so that it stops at the first breakpoint again.

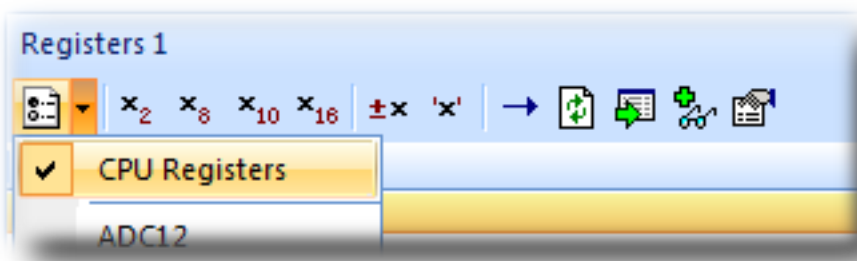
You can see the current processor state in the **Register** windows. To show the first registers window, do one of the following:

- From the **Debug** menu, click **Debug Windows, Registers** then **Registers 1**.

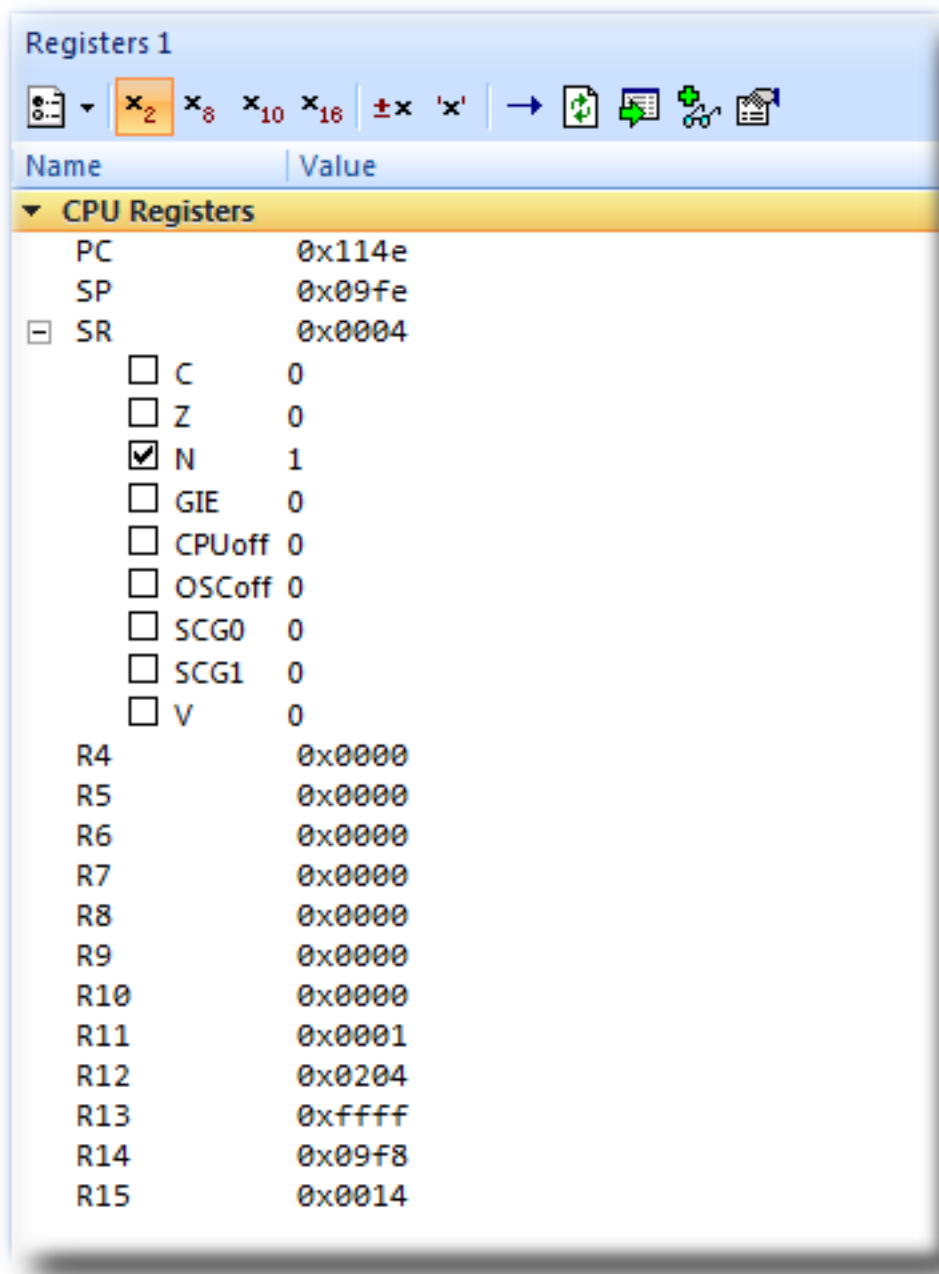
—or—

- Type **Ctrl+T, R, 1**.

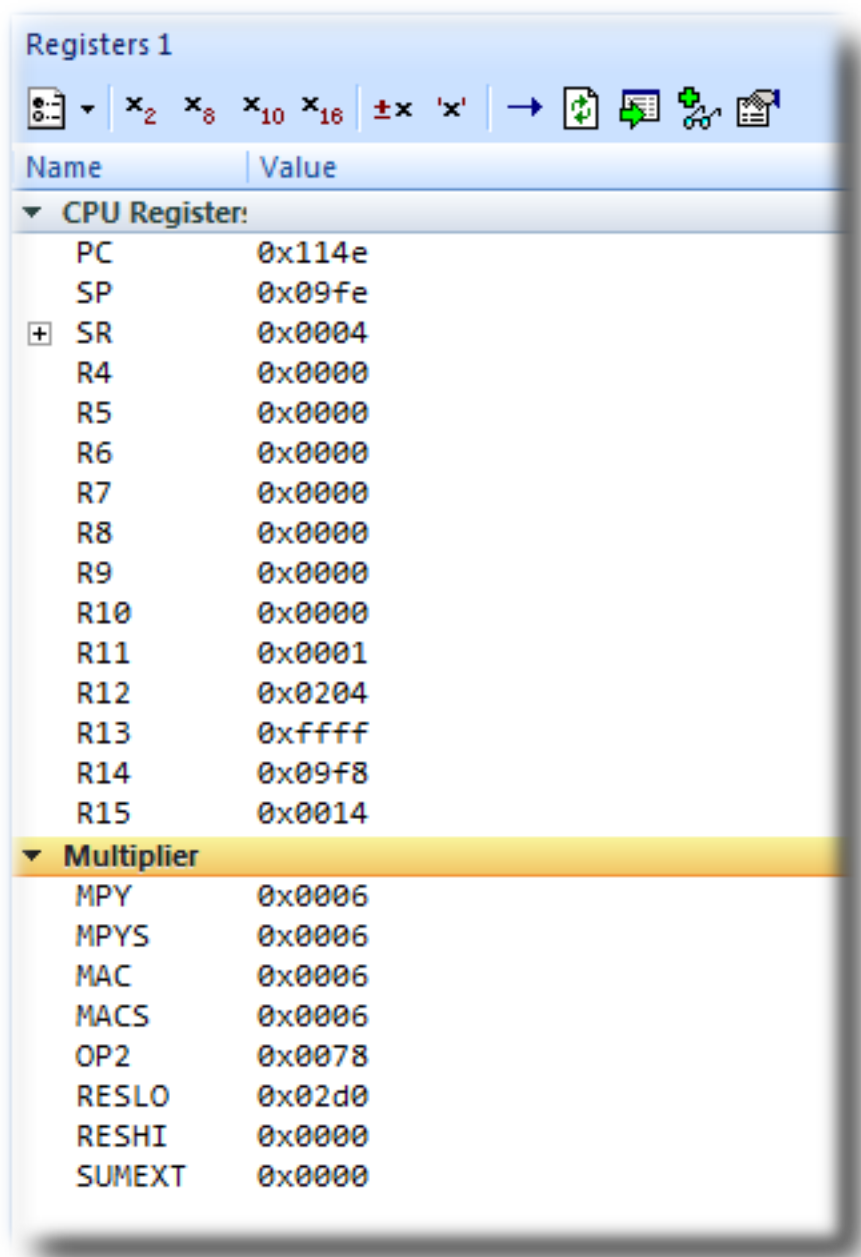
The registers window can be used to view CPU and peripheral registers, first we shall just look at the CPU registers. To do this select **CPU Registers** from the register window's **Groups** dropdown.



Your registers window will look something like this:



You can also use the registers window to display peripheral registers. To display the state of the target's hardware multiplier registers, select **Multiplier** from the **Groups** dropdown.

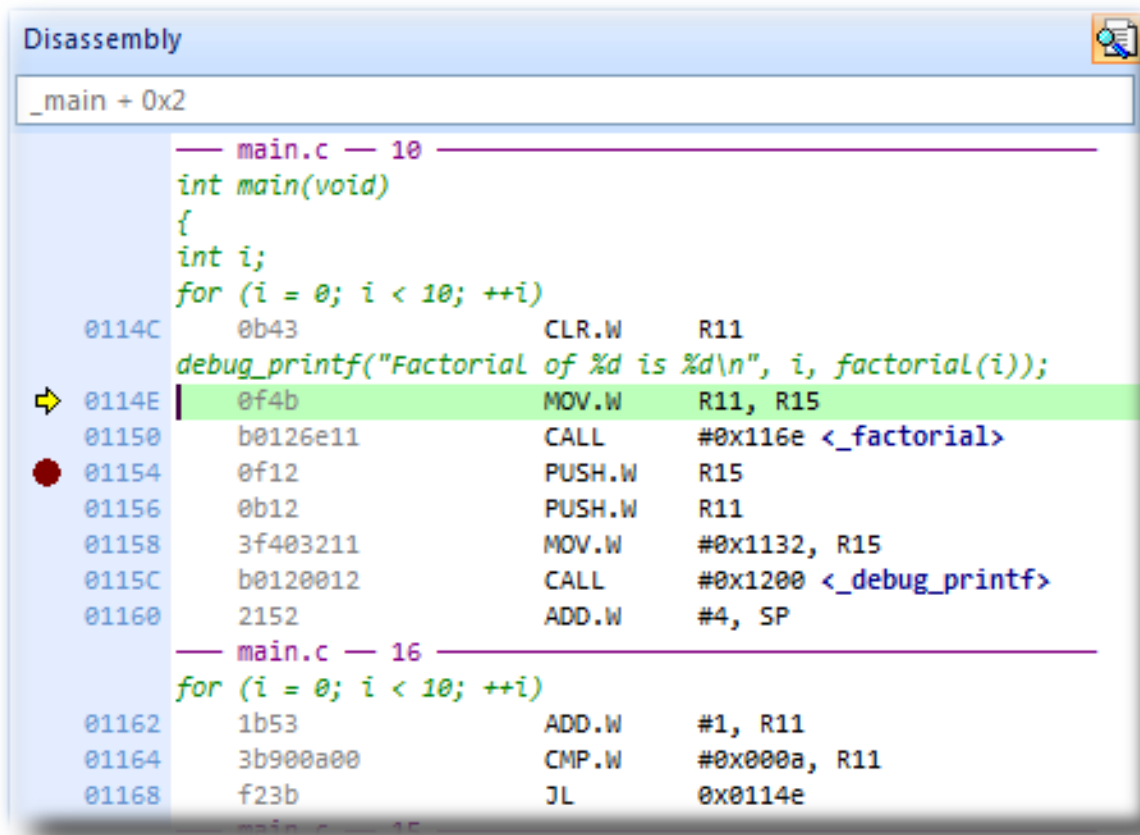


There are four register windows so you can open and display four sets of CPU and peripheral registers at the same time. You can configure which registers and peripherals to display in the Registers windows individually. As you single step the program, the contents of the Registers window updates automatically 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 when

available. When the disassembly window is focused, all single stepping is done one instruction at a time. The disassembly window also allows you to set breakpoints on individual instructions, you can do this by clicking in the gutter of the line containing the instruction you want to set the breakpoint on.



Stopping and starting debugging

You can stop debugging using **Debug > Stop**. If you wish to restart debugging without reloading the program then you can use **Debug > Debug From Reset**. Note that when you debug from reset no loading takes place so it is expected that your program is built in a way such that any resetting of data values is done as part of the program startup. You can also attach the debugger to a running target using the **Target > Attach Debugger**.

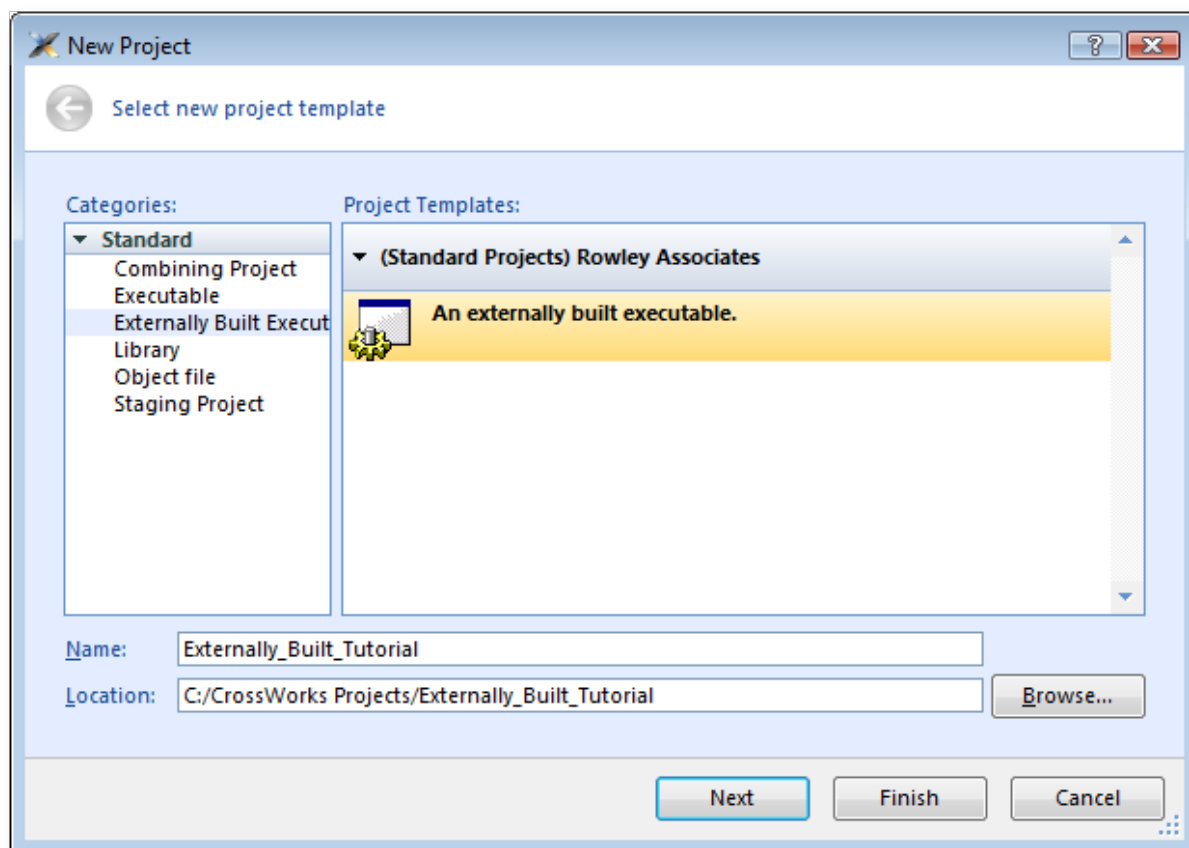
Debugging externally built applications

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

To start debugging an externally built application, you need to create a new externally built executable project. To do this, do the following:

- From the **File** menu, click **New** then **New Project...**

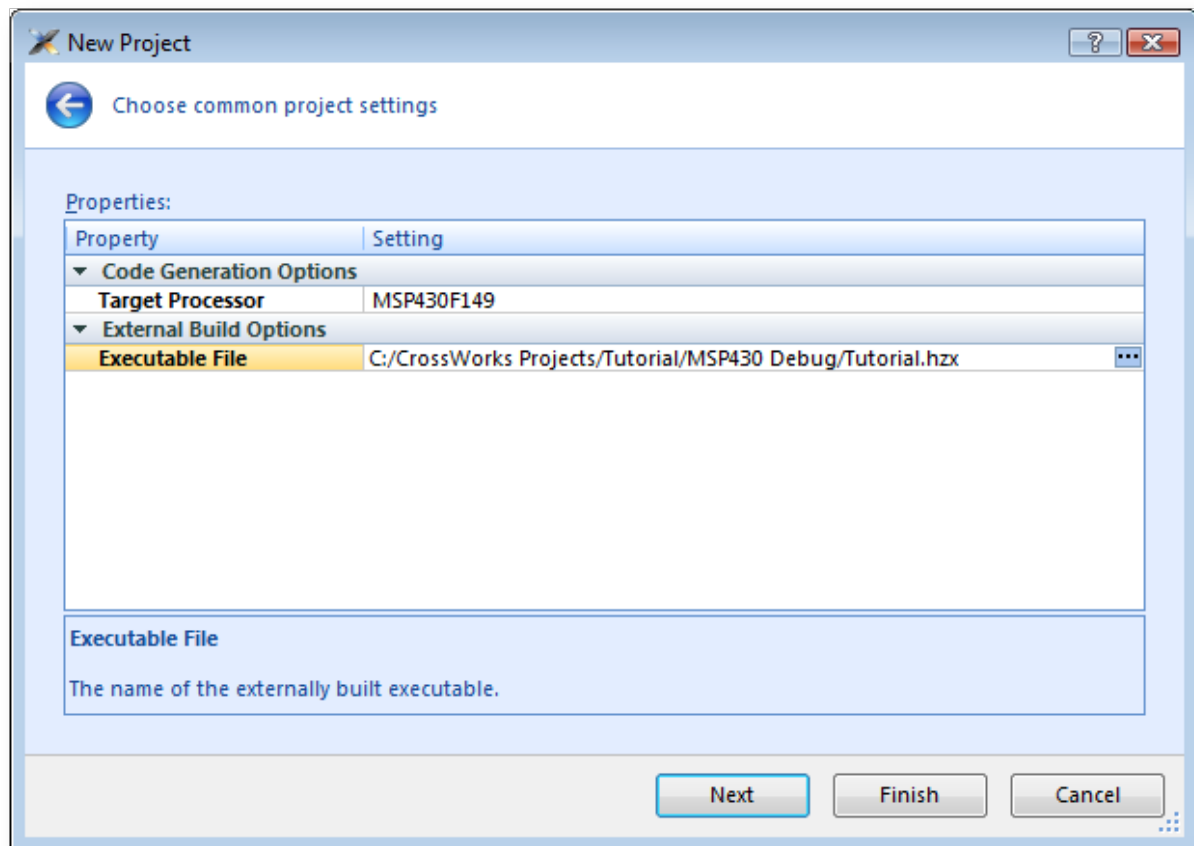
The **New Project** dialog appears. This dialog displays the set of project types and project templates.



We'll create an externally built executable project :

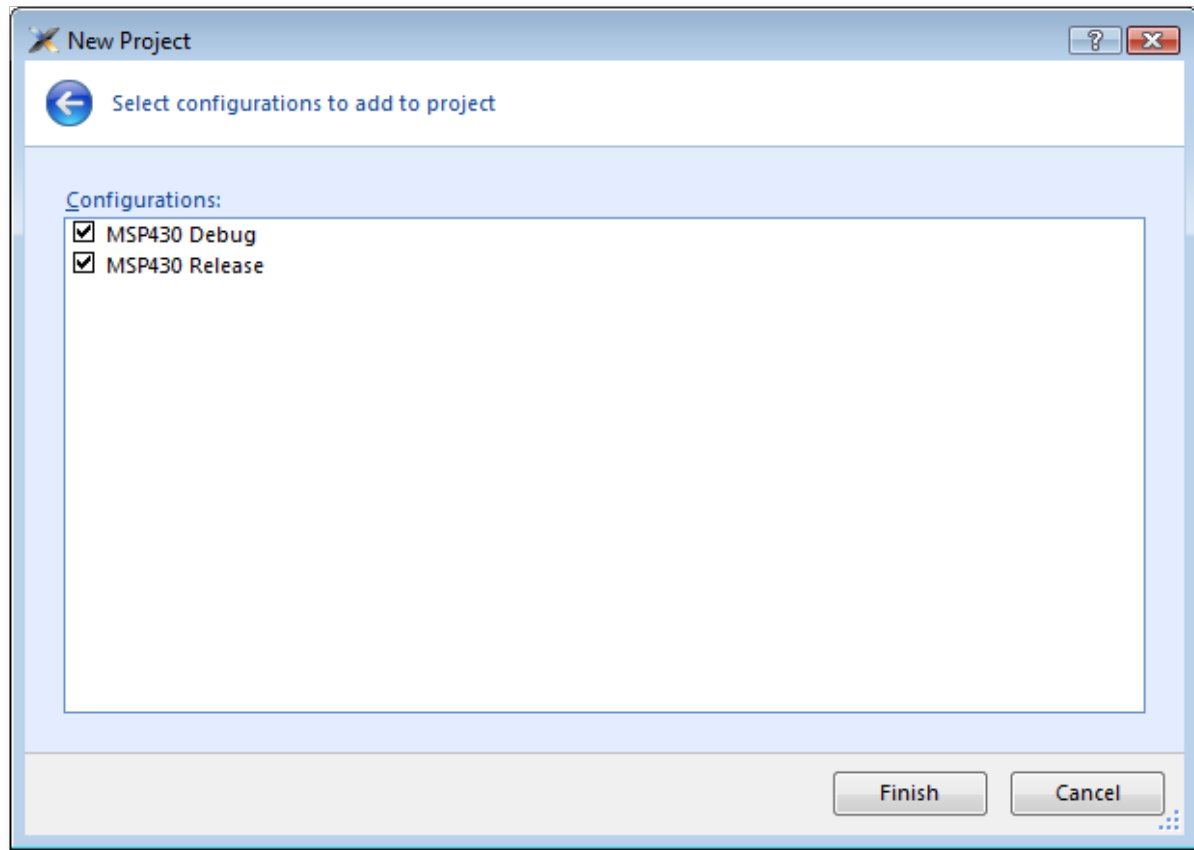
- Select the **Standard > Externally Built Executable** project type in the **Categories** pane.
- Select the **An externally built executable** icon in the **Project Templates** pane which selects the type of project to add.
- Type `Externally_Built_Tutorial` in the **Name** edit box, which names the project.
- You can use the **Location** edit box 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.



Enter the path to the *Tutorial.hzx* executable file we generated earlier in the **Executable File** field. For example, if the project was created in the *C:/CrossWorks Projects/Tutorial* directory and was built using the *MSP430 Debug* configuration, the path to the executable file will be *C:/CrossWorks Projects/Tutorial/MSP430 Debug/Tutorial.hzx*.

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 Reference

This section is a reference to the CrossStudio integrated development environment.

In this section

Overview

Contains an overview of CrossStudio and its layout.

Project management

Describes how to manage files and projects in CrossStudio.

Building projects

Describes how to build projects and correct errors in them.

Source code control

Describes how to set up your source code control provider so that it works seamlessly with CrossStudio.

Debug expressions

Describes the type and form of expression that CrossStudio can evaluate when debugging an application.

Source code editor

Describes CrossStudio's integrated code editor and how to get the most from it.

Section placement

Describes how your project is partitioned and placed into the target device's memory.

CrossStudio Windows

Describes each of CrossStudio's window individually.

MSP430 Target interfaces

Describes the target interfaces that CrossStudio provides to program and debug your application.

CrossStudio menu summary

Summarizes each of the menus presented in CrossStudio.

Overview

This section introduces the overall layout and operation of the CrossStudio integrated development environment.

CrossStudio standard layout

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

- **Title bar** Displays the name of the current file being edited and the active workspace.
- **Menu bar** Dropdown 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 or MDI view of multiple editors and the HTML viewer.
- **Docked windows** CrossStudio has many windows which can be docked to the left of, to the right of, or below the editing area. You can configure which windows are visible when editing and debugging. The figure shows the project explorer, targets window, and output window.
- **Status bar** At the bottom of the window, the status bar contains useful information about the current editor, build status, and debugging environment.

The title bar

CrossStudio's title bar displays the name of the active editor tab if in **Tabbed Document Workspace** mode or the active MDI window if in **Multiple Document Workspace** mode.

Title bar format

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

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

After the file name, 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 the CrossStudio's inbuilt debugger

[break]

The debugger is stopped at a breakpoint.

[autostep]

The debugger is single stepping the application without user interaction—this is called *autostepping*.

The **Target Status** panel in the status bar also shows CrossStudio's state—see [The status bar](#).

Configuring the title bar

You can configure whether the full path of the file or just its file name is shown in the title bar.

Displaying the full file path in the title bar

To display the full file path in the title bar, do the following:

- From the **Tools** menu, click **Options**.
- In the **Environment > User Interface Options** group, set the **File Name Caption Format** property to **Full Path Name**.

Displaying only the file name in the title bar

To display only the file name in the title bar, do the following:

- From the **Tools** menu, click **Options**.

- In the ***Environment > User Interface Options*** group, set the ***File Name Caption Format*** property to ***File Name Only***.

The menu bar

The menu bar contains dropdown menus for editing, building, and debugging your program. You can navigate menu items using the keyboard or using the mouse. You'll find a complete description of each menu and its contents in [CrossStudio menu summary](#).

Navigating menus using the mouse

To navigate menus using the mouse, do the following;

- Click the required menu title in the menu bar; the menu appears.
- Click the required menu item in the dropdown menu.

—or—

- Click and hold the mouse on the required menu title in the menu bar; the menu appears.
- Drag the mouse to the required menu item on the dropdown menu.
- Release the mouse.

Navigating menus using the keyboard

To navigate menus using the keyboard, do the following:

- Tap the **Alt** key which focuses the menu bar.
- Use the **Left** and **Right** keys to navigate to the required menu.
- Use the **Up** or **Down** key to activate the required menu
- Type **Alt** or **Esc** to cancel menu selection at any time.

Each menu on the menu bar has one letter underlined, its shortcut, so to activate the menu using the keyboard:

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

Once the menu has dropped down you can navigate it using the cursor keys:

- Use **Up** and **Down** to move up and down the menu.
- Use **Esc** to cancel a dropdown 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 menu item to activate that menu item.
- Type **Enter** to activate the selected menu item.

The 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 that contains a set of fixed panels and the other that is used for messages.

The message area

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

The 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 name and, if applicable, the processor that the target interface is connected to. The LED icon flashes green when programs are running, is solid red when stopped at a breakpoint, and is yellow when connected but not running a program. Double clicking this panel displays the Targets window and right clicking it brings up the Target menu.
Cycle count panel	Displays the number of processor cycles run by the executing program. This panel is only visible if the currently connected target supports performance counters which can report the total number of cycles executed. Double clicking this panel resets the cycle counter to zero, and right clicking this panel brings up the Cycle Count menu.
Insert/overwrite status	Indicates whether the current editor is in insert or overwrite mode. If the editor is in overwrite mode the OVR panel is highlighted otherwise it is dimmed.
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 READ panel is highlighted otherwise it is dimmed.
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 "Build OK" otherwise it contains the number of errors and warnings reported. Right clicking this panel displays the Build Log in the Output window.

Caret position	Indicates the cursor position of the current editor. For text editors, the caret position pane displays the line number and column number of the cursor; for binary editors it displays the address where the
Caps lock status	Indicates the Caps Lock state. If the Caps Lock is on, CAPS is highlighted, otherwise it is dimmed.
Num lock status	Indicates the Num Lock state. If the Num Lock is on, NUM is highlighted, otherwise it is dimmed.
Scroll lock status	Indicates the Scroll Lock state. If the Scroll Lock is on, SCR is highlighted, otherwise it is dimmed.
Time panel	Displays the current time.

Configuring the status bar panels

To configure which panels are shown on the status bar, do the following:

- From the **View** menu, click **Status Bar**.
- From the status bar menu, check the panels that you want displayed and uncheck the ones you want hidden.

—or—

- Right click on the status bar.
- From the status bar menu, check the panels that you want displayed and uncheck the ones you want hidden.

You can also select the panels to display from the **Tools > Options** dialog in the **Environment > More...** folder.

- From the **Tools** menu, click **Options**.
- In the tree view **Environment** folder, click **More...**
- In the **Status bar** group, check the panels that you want displayed and uncheck the ones you want hidden.

Hiding the status bar

To hide the status bar, do the following:

- From the **View** menu, click **Status Bar**.
- From the status bar menu, uncheck the **Status Bar** menu item.

—or—

- Right click on the status bar.
- From the status bar menu, uncheck the **Status Bar** menu item.

Showing the status bar

To show the status bar, do the following:

- From the **Tools** menu, click **Options**.
- In the tree view **Environment** folder, click **More...**
- In the **Status bar** group, check **(visible)**.

Showing or hiding the size grip

You can choose to hide or display the size grip when the CrossStudio main window is not maximized—the size grip is never shown in full screen mode or when maximized. To do this:

- From the **View** menu, click **Status Bar**.
- From the status bar menu, uncheck the **Size Grip** menu item.

—or—

- Right click on the status bar.
- From the status bar menu, uncheck the **Size Grip** menu item.

You can also choose to hide or display the size grip from the **Tools > Options** dialog in the **Environment > More...** folder.

- From the **Tools** menu, click **Options**.
- In the tree view **Environment** folder, click **More...**
- In the **Status bar** group, check or uncheck the **Size grip** item.

The editing workspace

The main area of CrossStudio is the editing workspace. This area contains files that are being edited by any of the editors in CrossStudio, and also the online help system's HTML browser.

You can organize the windows in the editing area either into tabs or as separate windows. In ***Tabbed Document Workspace*** mode, only one window is visible at any one time, and each of the tabs displays the file's name. In ***Multiple Document Workspace*** mode, many overlapping windows are displayed in the editing area.

By default, CrossStudio starts in ***Tabbed Document Workspace*** mode, but you can change at any time between the two.

Changing to Multiple Document Workspace mode

To change to Multiple Document Workspace mode, do the following:

- From the ***Window*** menu, click ***Multiple Document Workspace***.

Changing to Tabbed Document Workspace mode

To change to Tabbed Document Workspace mode, do the following:

- From the ***Window*** menu, click ***Tabbed Document Workspace***.

The document mode is remembered between invocations of CrossStudio.

Docking windows

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

Window groups

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

Clicking on the window icons in the window group's header changes the active window. Hovering over an icon in the header will display the dock window's title in a tool tip.

To dock a window to a different window group

- Show the window you wish to move.
- Right click the window group header containing the window to move.
- Choose **Move Window To > *place***, where *place* is the new dock position.

or

- Right click the window group header you want to dock the window in.
- Choose **Dock Other Window Here > *window***, where *window* is the window you wish to dock in the window group.

Project management

CrossWorks has a project system that enables you to manage the source files and build instructions of your solution. The **Project Explorer** and the **Properties Window** are the standard ways to edit and view your solution. You can also edit and view the project file which contains your solution using the text editor—this can be used for making large changes to the solution.

In this section

Overview

A summary of the features of the CrossStudio project system.

Creating a project

Describes how to create a project and add it to a solution.

Adding existing files to a project

Describes how to add existing files to a project, how filters work, and what folders are for.

Adding new files to a project

Describes how create and add new files to a project.

Importing files

Describes how to import a file into the project directory.

Removing a file, folder, project, or project link

Describes how to remove items from a project.

Project properties

Describes what properties are, how they relate to a project, and how to change them.

Project configurations

Describes what project build configurations are, to to create them, and how to use them.

Project dependencies and build order

Describes project dependencies, how to edit them, and how they are used to define the order projects build in.

Project macros

Describes what project macros are and what they are used for.

Related sections

Project explorer

Describes the project explorer and how to use it.

Project property reference

A complete reference to the properties used in the project system.

Project file format

Describes the XML format CrossStudio uses for project files.

Overview

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, load them onto the target ready for debugging with another.

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 move or share project file hierarchies on different computers.

The **Project Explorer** organizes your projects and files and provides quick access to the commands that operate on them. A tool bar at the top of the window offers quick access to commonly used commands for the item selected in the **Project Explorer**.

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 [CrossStudio Project File Format](#) for a description of the project file format.

Projects

The projects you create within a solution have a project type which CrossStudio uses to determine how to build the project. The project type is selected when you use the **New Project** dialog. The particular set of project types can vary depending upon the variant of CrossWorks you are using, however the following project types are standard to most CrossWorks variants:

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

Properties and configurations

Properties are data that are attached to project nodes. They are usually used in the build process for example to define C preprocessor symbols. You can have different values of the same property based on a configuration, for example you can change the value of a C preprocessor symbol for a release or a debug build.

Folders

Projects can contain folders which are used to group related files together. This grouping can be done using the file extension of the file or it can be done by explicitly creating a file within a folder. Note that folders do not map onto directories in the file store they are solely used to structure the project explorer display.

Files

The source files of your project can be placed either in folders or directly in the project. Ideally files placed in project should be relative to the project directory, however there are cases when 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 if the file is in the project directory. If a file is not in the project directory then 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 then the project system detects if the file is relative to the `$(StudioDir)` directory. If the file is relative to `$(StudioDir)` directory then the filename is defined using `$(StudioDir)`. If a file is not relative to the project directory or to `$(StudioDir)` then the full filename is used.

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

The project system uses the extension of the file to determine the appropriate build action to perform on the file. So

- a file with the extension `.c` will be compiled by a C compiler.
- a file with the extension `.s` or `.asm` will be compiled an assembler.
- a file with the extension `.cpp` or `.cxx` will be compiled by a C++ compiler.
- a file with the object file extension `.o` or `.hzo` will be linked.
- a file with the library file extension `.a` or `.hza` will be linked.
- a file with the extension `.xml` will be opened and its file type determined by the XML document type.
- other file extensions will not be compiled or linked.

You can modify this behaviour by setting the **File Type** property of the file with the **Common** configuration selected in the properties window which enables files with non-standard extensions to be compiled by the project system.

Solution links

You can create links to existing project files from a solution which enables you to create hierarchical builds. For example you could have a solution that builds a library together with a stub test driver executable. You can then link to this solution (by right clicking on the solution node of the project explorer and selecting **Add Existing Project**) to be able to use the library from a project in the current solution.

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 files you have opened in the editor and breakpoints you set in the breakpoint window.

Creating a project

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

To create a new project in an existing solution, do the following:

- From the **Project** menu, click **New** then **New Project...** to display the **New Project** wizard.
- In the **New Project** wizard, select the type of project you wish to create and where it will be placed.
- Ensure that the "Add the project to current solution" radio button is checked.
- Click **OK** to go to next stage of project creation or **Cancel** to cancel the creation.

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

Creating a new project in a new solution

To create a new project in a new solution, do the following:

- From the **File** menu, click **New** then **New Project...** to display the **New Project** dialog.
- In the **New Project** dialog, select the type of project you wish to create and where it will be placed.
- Click **OK**.

Adding existing files to a project

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

Adding existing files to the active project

You can add one or more files to the active project quickly using the standard **Open File** dialog.

To add existing files to the active project do one of the following:

- From the **Project** menu, select **Add Existing File...**

—or—

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

—or—

- Type **Ctrl+D**.

Using the **Open File** dialog, navigate to the directory containing the existing files, select the ones to add to the project, then click **OK**. The selected files are added to the folders whose filter matches the extension of the each of the files. If no filter matches a file's extension, the file is placed underneath the project node.

Adding existing files to any project

To add existing files a project without making it active:

- In the **Project Explorer**, right click on the project to add a new file to.
- From the popup menu, select **Add Existing File...**

The procedure for adding existing files is the same as above.

Adding existing files to a specific folder

To add existing files directly to a folder bypassing the file filter do the following:

- In the **Project Explorer**, right click on the folder to add a new file to.
- From the popup menu, select **Add Existing File...**

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

Adding new files to a project

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

Adding a new file to the active project

To add new files to the active project, do one of the following:

- From the **Project** menu, click **Add New File...**

—or—

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

—or—

- Type **Ctrl+N**.

Adding a new file to any project

To add a new file to a project without making it active, do one of the following:

- In the **Project Explorer**, right click on the project to add a new file to.
- From the popup menu, select **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 file name, 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.

Adding a new file to a specific folder

To add new files directly to a folder bypassing the file filter do the following:

- In the **Project Explorer**, right click on the folder to add a new file to.
- From the popup menu, select **Add New File...**

The new file is added to the folder without using any 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** tool button on the project explorer's toolbar. Removing a source file from a project does not remove it from disk.

Removing an item

To remove an item from the solution do one of the following:

- Click on the project item to remove from the **Project Explorer** tree view.
- On the **Project Explorer** toolbar, click the **Remove** button (or type **Delete**).

—or—

- Right click on the the project item to remove from the **Project Explorer** tree view.
- From the popup menu, click **Remove**.

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 using the properties window in conjunction with the project explorer. As you select an item in the project explorer the properties window will list the set of properties that are applicable.

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 the solution, project or individual file. By setting properties at the solution level you enable all files of the solution to use this property 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 a example particular source file may have warnings that you decide are allowable so you set the **Treat Warnings As Errors** to **No** for this 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

Aggregating properties

An aggregating property collects all of the values that are defined for it in the project hierarchy. For example when a C file is compiled the **Preprocessor Definitions** property will take all of the values defined at the file, project and solution level. 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.

There is a special configuration named **Common** that is always inherited by a configuration. The **Common** configuration enables property values to be set that will apply to all configurations that 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 it's almost certain that you want each configuration to inherit these values - so ensure that the **Common** configuration has been selected.

If the property is unique then it will use the one defined for the particular configuration. If the property isn't defined for this configuration then it uses an arbitrary one from the set of inherited configurations. If the property still isn't defined it uses the value for the **Common** configuration. If it still isn't defined then it tries the to find the value in the next 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

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

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

Project configurations

Project configurations are used to create different software builds for your projects. A configuration is used to define different project property values, for example the output directory of a compilation can be put into different directories which are dependent upon the configuration. By default when you create a solution you'll get some default project configurations created.

Selecting a configuration

You can set the configuration that you are building and debugging with using the combo box of the **Build** tool bar or the **Build > Set Active Build Configuration** menu option.

Creating a configuration

You can create your own configurations using **Build > Build Configurations** which will show the **Configurations** dialog. The **New** button will produce a dialog that allows you name your configuration. You can now specify which existing configurations your new configuration will inherit values from.

Deleteing a configuration

You can delete a configuration by selecting it and pressing the **Remove** button. Note that this operation cannot be undone or cancelled so beware.

Hidden configurations

There are some configurations that 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 specify if you want that configuration to be hidden.

Project 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 manage loading and deleting 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 then the build system will ensure that the library it is dependent upon is up to date. In the case of a dependent library then the output file of the library build is supplied as an input to the executable build so you don't have to worry about this.

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.

Making a project dependent upon another

To make one project dependent upon another, do the following:

- From the **Project** menu, click **Dependencies** to display the **Project Dependencies** dialog.
- From the **Project** dropdown, select the target project which depends upon other projects.
- In the **Depends Upon** list box, check the projects that the target project depends upon and uncheck the projects that it does not depend upon.

Some items in the **Depends Upon** list box may be disabled, which indicates that if the project were checked, a circular dependency would result. Studio prevents you from constructing circular dependencies using the **Project Dependencies** dialog.

Finding the project build order

To display the project build order, do the following:

- From the **Project** menu, click **Build Order** to display the **Project Dependencies** dialog with the **Build Order** tab selected.
- The projects build in order from top to bottom.

If your target supports loading of multiple projects, then 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 applications that use them, and you can ensure that this happens by making the application dependent upon the library. With this a dependency set, the library gets built before the application and loaded before the application.

Applications are deleted from a target in reverse build order, and as such applications are removed before the libraries that they depend upon.

Project macros

You can use macros to modify the way that the project system refers to files. Macros are divided into four classes:

System Macros

These are provided by the Studio application and are used to relay information from the environment, such as paths to common directories.

Global Macros

These macros are saved in the environment and are shared across all solutions and projects. Typically, you would set up paths to library or external items here.

Project Macros

These macros are saved in the project file as project properties and can define macro values specific to the solution/project they are defined in.

Build Macros

These macros are generated by the project system whenever a build occurs.

System macros

System macros are defined by CrossStudio itself and as such are readonly. System macros can be used in project properties, environment settings and to refer to files. [See](#) for the list of System macros.

Global macros

To define a global macro

- Select **Macros** from the **Project** menu.
- Click on the the **Global** tab.
- Set the macro using the syntax *name = replacement text*.

Global macros can be used in project properties, environment settings and to refer to files.

Project macros

To define a project macro

- Select **Macros** from the **Project** menu.
- Click on the **Project** tab.
- Select the solution or project the macro should apply to.
- Set the macro using the syntax *name = replacement text*.

Alternatively you can set the project macros from the properties window:

- Select the appropriate solution/project in the Project Explorer.
- In the properties window, select the **Macros** property in the **Build Options** group.
- Click on the the ellipsis button on the right.
- Set the macro using the syntax *name = replacement text*.

Project macros can be used in project properties only.

Build macros

Build macros are defined by the project system for a build of a given project node. [See](#) for the list of Build macros.

Using macros

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

Building projects

CrossStudio provides a facility to build projects in various configurations.

Build configurations and their uses

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

Configurations inherit properties from other configurations. This provides a single point of change for definitions that are common to 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 using **Build > Build Configurations**. Before you build, ensure that the appropriate configuration is set using **Project > Set Active Build Configuration** or alternatively the configuration box in the build tool bar. You should also ensure that the appropriate build properties are set in the properties window.

Building your applications

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. If you are copying files then 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 is using by setting the **Environment Properties > Build Settings > Show Build Information** property and the build commands themselves by setting the **Environment Properties > Build Settings > Echo Build Command** property.

You may have a solution that contains several projects that are dependent upon each. Typically you might have several executable project and some library projects. The **Project > Dependencies** dialog specifies the dependencies between projects and to see the affect those dependencies have 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 new folders titled Dependencies has appeared in the project explorer. These folder contains the list of newly generated files and the files that they where generated from. These files can be decoded and displayed in the editor by right clicking on the file and seeing if it supports the **View** operation.

If you have the symbols window displayed then it will be updated with the symbol and section information of all executable files that have been built in the solution.

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

The combination of configurations, properties with inheritance, dependencies and macros provides a very powerful build management system. However, these systems can become complicated. To enable you to understand the implications of changing build settings, right clicking a node in the project explorer and selecting **Properties** brings up a dialog that shows the macros and build steps that apply to that project node.

Building all projects

To build all projects in the solution, do one of the following:

- On the **Build** toolbar, click the **Build Solution** button.

—or—

- From the **Build** menu, select **Build Solution**.

—or—

- Type **Alt+F7**.

—or—

- Right click the solution in the **Project Explorer** window.
- From the menu, click **Build**.

Building a single project

To build a single project only, do one of the following:

- Select the required project in the **Project Explorer**.
- On the **Build** tool bar, click the **Build** tool button.

—or—

- Select the required project in the **Project Explorer**.
- From the the **Project** menu, click **Build**.

—or—

- Right-click on the required project in the **Project Explorer** window.
- From the menu, click **Build**.

Compiling a single file

To compile a single file, do one of the following:

- In the **Project Explorer**, right click the source file to compile.
- From the menu, click **Compile**.

—or—

- In the **Project Explorer**, click the source file to compile.
- From the **Build** menu, click **Compile**.

—or—

- In the **Project Explorer**, click the source file to compile.
- Type **Ctrl+F7**.

Correcting errors after building

The results of a build are displayed in the **Build Log** in the **Output** window. Errors are highlighted in red, and warnings are highlighted in yellow. Double-clicking an error, warning, or note will move the cursor to the appropriate source line.

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 show you visually the memory used by the application and a summary of each memory area.

Source control

CrossWorks has a **source control** integration capability that can be used on the files of a CrossWorks project. The capability is implemented by a number of different source control **providers**, however the set of functions that are provided by CrossWorks aim to be provider independent. The source control integration capability provides:

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

Configuring the source control system

The source control system you are using must be enabled using:

- Select **Tools | Options** menu item.
- Selecting the **Source Control** category in the options dialog.
- Setting **Source Control Provider** to the appropriate provider.
- Setting the source control provider specific options.
- Setting **Enable Source Code Control Integration** to **Yes**.

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

Connecting to the source control system

You must connect to the source control system for each different CrossWorks project you have. To connect to the source control system, do the following:

- From the **Project** menu, click **Source Control** then **Connect...**

This displays a source control system login dialog that enables you to specify your username, password and select the source control database to connect to. These details will be saved in the session file (the password is enciphered) so that you don't need to repeat this information each time the project is loaded.

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

After the login dialog has completed you will be presented with a dialog that you use to specify the local and remote roots. The local root can be selected using a directory browser and the remote root can be selected using the source control explorer. With both browsers you can create new directories if you are starting a new project or you don't have an existing project in source control.

Opening a project from source control

To fetch a project that is already in source control to a local directory:

- From the **Project** menu, click **Source Control** then **Open Solution From...** This will show the login dialog and then the source control explorer.
- You should select a CrossWorks project file **.hzip** using the files list of the source control explorer.
- The mappings dialog will then be shown and you should use this to specify the local root directory i.e. where you want the project to go.
- A dialog showing the list of files to get from source control will be shown and then after confirmation these files are fetched and the project file is loaded into CrossWorks.

Source control status

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

- A file node is selected by the project explorer.
- The source control status is displayed in the project explorer and the file node is visible in the project explorer.
- Before a recursive source control operation is used.
- After a source control operation has been used.

A file can be in one of the following states:

- Controlled - the file is in source control.
- Not Controlled - the file is not in source control.
- Checked Out - the file is checked out.
- Old - the file is older than the latest version in source control.
- Checked Out and Old - both of the above.

When the status is displayed in the project explorer if the file has been modified then the status is displayed in red. Note that if a file is not under the local root then it will not have a source control status.

You can reset any stored source control file status using the **Project | Source Control | Refresh Status** operation.

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 using the **Source Control** toolbar and also the right click menu of the text editor. All operations are available using the menu at **Project | Source Control** and on the **Project Explorer** right click menu. The operations are described in terms of the **Project Explorer** right click menu.

Adding files to source control

You can add a file in the project that is not in source control using:

- In the **Project Explorer**, right click on a file node.
- From the menu, click **Source Control** then **Add To Source Control**.
- Add a comment and select okay for the dialog box.

To add multiple files to the source control system do the following:

- In the **Project Explorer**, right click on a solution, project or folder.
- From the menu, click **Source Control** then **Add To Source Control(Recursive)**.
- The dialog box will show the list of files that can be added i.e. ones that have a status of **Not Controlled**.
- In the dialog you can uncheck the ones you don't want to add to source control, add a comment and okay the dialog box.

Checking files out

To check out a file in the project from source control, do the following:

- In the **Project Explorer**, right click on a file node.
- From the menu, click **Source Control** then **Check Out**.
- Add a comment and select okay for the dialog box.

To check out multiple files in the project from source control, do the following:

- In the **Project Explorer**, right click on a solution, project or folder.
- From the menu, click **Source Control** then **Check Out(Recursive)**.
- The dialog box will show the list of files that can be checked out i.e. ones that have a status of **Controlled**.
- In the dialog you can uncheck the ones you don't want to check out, add a comment and select okay for the dialog box.

Checking files in

To check in files in the project to source control, do the following:

- In the **Project Explorer**, right click on a file node.
- From the menu, click **Source Control** then **Check In**.

- Add a comment and select okay for the dialog box.

To check in multiple files in the project from source control, do the following:

- In the **Project Explorer**, right click on a solution, project or folder.
- From the menu, click **Source Control** then **Check In(Recursive)**.
- The dialog box will show the list of files that can be checked in.
- In the dialog you can uncheck the ones you don't want to check in, add a comment and select okay for the dialog box.

Undoing Check Outs

To undo a check out of a file in the project, do the following:

- In the **Project Explorer**, right click on a file node.
- From the menu, click **Source Control** then **Undo Check Out**.

To undo check out of multiple files in the project, do the following:

- In the **Project Explorer**, right click on a solution, project or folder.
- From the menu, click **Source Control** then **Undo Check Out(Recursive)**.
- The dialog box will show the list of files that can have undo check out in i.e. ones that have a status of **Checked Out**.
- In the dialog you can uncheck the ones you don't want to undo and select okay for the dialog box.

Get Latest Version

To get the latest version of a file in the project, do the following:

- In the **Project Explorer**, right click on a file node.
- From the menu, click **Source Control** then **Get Latest Version**.

To get the latest version of multiple files in the project, do the following:

- In the **Project Explorer**, right click on a solution, project or folder.
- From the menu, click **Source Control** then **Get Latest Version(Recursive)**.
- The dialog box will show the list of files that can have undo check out in i.e. ones that have a status of **Controlled**, **Checked Out** or **Old**.
- In the dialog you can uncheck the ones you don't want to get and select okay for the dialog box.

Showing the differences between files

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

- In the **Project Explorer**, right click on a file node.
- From the menu, click **Source Control** then **Show Differences**.

You can use an external diff tool if you have one installed in preference to the built-in CrossWorks diff tool. You define the diff command line CrossWorks generates using **Tools | Options | Source Control | Diff Command Line** - note that 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 containing the file in the project.
- **\$(remotefile)** The filename containing the latest version of the file in source control.
- **\$(localname)** A display name for \$(localfile).
- **\$(remotename)** A display name for \$(remotefile).

Merging files

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

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

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

- In the **Project Explorer**, right click the file node.
- From the menu, click **Source Control** then **Merge**.
- When the external tool has finished if \$(commonfile) has been modified then you will be asked if you want to overwrite the file in the project with \$(commonfile).

Source control explorer

By selecting the **Project | Source Control | Show Explorer..** menu a dialog is displayed that lists the directories and files that are in source control. This dialog is used for selecting the remote root directory and when **Project | Source Control | Open Solution From..** is selected.

You can use the directory side of the dialog to create new directories and to refresh the list if this is required by the source control provider.

Source control properties

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

Checked Out

If **Yes**, the file is checked out by you to the project location; if **No**, the file is not checked out.

Different

If **Yes**, the checked out file differs from the one held in the source control system; if **No**, they are identical.

File Path

The file path of the file in the source control system.

Local Revision

The revision number/name of the local file.

Old Version

If **Yes**, the file in the project location is an old version compared to the latest version in the source control system.

Provider Status

The source control provider status of the file.

Remote Revision

The revision number/name of the most recent version in source control.

Status

The source control status of the file.

Provider Specific Help

- [Visual SourceSafe Provider](#)
- [SourceOffSite Provider](#)
- [CVS Provider](#)
- [SVN Provider](#)

Breakpoint expressions

The debugger can set breakpoints by evaluating simple C like expressions. The simplest expression supported is a symbol name. If the symbol name is a function then a breakpoint occurs when the first instruction of the symbol is about to be executed. If the symbol name is a variable then a breakpoint occurs when the symbol has been (target specific) accessed, this is termed a data breakpoint. For example the expression

```
x
```

will breakpoint when x is accessed. You can use a [debug expression](#) as a breakpoint expression. For example

```
x[4]
```

will breakpoint when element 4 of the array x is accessed and

```
@sp
```

will breakpoint when the sp register is accessed.

Data breakpoints can be specified to occur when a symbol is accessed with a specific value using the == operator. The expression

```
x == 4
```

will breakpoint when x is accessed and it's value is 4. Similarly the operators <, <=, >, >=, ==, != can be used. For example

```
@sp <= 0x1000
```

will breakpoint when the register sp is accessed and it's value is less than or equal to 0x1000.

You can use the operator & to mask the value you wish to breakpoint on. For example

```
(x & 1) == 1
```

will breakpoint when x is accessed and it 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 it's value is between 2 and 14.

You can specify an arbitrary memory range using an array cast expression. For example

```
(char[256])(0x1000)
```

will breakpoint when the memory region 0x1000-0x10FF is accessed.

You can specify an inverse memory range using the ! operator. For example

```
!(char[256])(0x1000)
```

will breakpoint when the memory region other than 0x1000-0x10FF is accessed.

Debug expressions

The debugger can evaluate simple expressions that can be subsequently displayed in the watch window or as a tool-tip in the code editor.

The simplest expression is an identifier which 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 `!, ~, *, /, %, +, -, >>, <<, <, <=, >, >=, ==, !=, &, ^, |, &&, ||` are supported on numeric types.

The standard assignment operators `=, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=` are supported on number 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 sstructures) and the `->` works similarly.

The dereference operator (prefix `*`) is supported on pointers, the addressof (prefix `&`) and sizeof operators are supported.

The `addressof(filename, linenumber)` operator will return the address of the specified source code linenumber.

Function calling with parameters and return results.

Casting to basic pointer types is supported. For example `(unsigned char *)0x300` can be used to display the memory at a given location.

Casting to basic array types is supported. For example `(unsigned char[256])0x100` can be used to reference a memory region.

Operators have the precedence and associativity that one would expect of a C like programming language.

Basic editing

CrossStudio has a built-in editor which allows you to edit text, but has features that make it particularly well suited to editing code and is referred to as either the Text Editor or the Code Editor, based on its content.

You can open multiple code editors to view or edit source code in projects and 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 that 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 on a file in the **Project Explorer** or by right clicking on a file and selecting **Open** from the context menu.
- Using the **File > New** or **File > Open** commands.
- Right clicking in a source file and selecting a file from the **Open Include File** menu.

Elements of the code editor

The code editor is divided into several elements which are described here.

- **Code Pane** The area where you edit your code. You can set options that affect the behavior of text in the code pane as it relates to indenting, tabbing, dragging and dropping of text, and so forth. For more information, see General, All Languages, Text Editor, Options Dialog Box.
- **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. You can hide or display the Margin Indicator bar in General, Tools, Text Editor, Options dialog box.
- **Horizontal and vertical scroll bars** Allows you to scroll the code pane horizontally and vertically so that you can view the code that extends beyond the viewable edges of the code pane. You can hide or display the horizontal and vertical scroll bars using the General, Tools, Text Editor, Options dialog box.

Navigation

There are several ways to move around code editors:

- Mouse and cursor motion keys
- Bookmarks
- The **Go To Line** command
- The **Navigate Backward** and **Navigate Forward** buttons

Navigating with the mouse and keyboard

The most common way to navigate text is with the mouse and cursor motion keys:

- Click a location with the mouse.
- Use the arrow keys to move one character at a time, or the arrow keys in combination with the Ctrl key to move one word at a time.
- Use the scroll bars or scroll wheel on the mouse to move through the text.
- Use the **Home**, **End**, **PageUp**, and **PageDown** keys.
- Use **Ctrl+Up** and **Ctrl+Down** to scroll the view without moving the insertion point.

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

Keystroke	Description
Up	Moves the cursor up one line.
Down	Moves the cursor down one line.
Left	Moves the cursor left one character.
Right	Moves the cursor right one character.
Home	Moves the cursor to the first character on the line. Pressing Home a second time moves the cursor to the first column.
End	Moves the cursor to the end of the line.
PageUp	Moves the cursor up one page.
PageDown	Moves the cursor down one page.
Ctrl+Left	Moves the cursor left one word.
Ctrl+Right	Moves the cursor right one word.

Ctrl+Up	Moves the cursor to the previous function.
Ctrl+Down	Moves the cursor to the next function.
Ctrl+Home	Moves the cursor to the start of the document.
Ctrl+End	Moves the cursor to the end of the document.
Alt+Up	Move the insertion point to the top of the window.
Alt+Down	Move the insertion point to the bottom of the window.
Ctrl+Up	Scrolls the document up one line in the window without moving the caret.
Ctrl+Down	Scrolls the document down one line in the window without moving the caret.

Go to line

To move the cursor to a particular line number, do the following:

- Choose **Search > Go To Line** or type **Ctrl+G**.
- Enter the line number to move the cursor to.

Selecting Text

Selecting text with the keyboard

You can select text using the keyboard by using **Shift** with the navigation keys.

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

Selecting text with the mouse

- Move mouse cursor to the point in the document that you want to start selecting.
- Hold down left mouse button and drag mouse to mark selection.
- Release left mouse button to end selection.

Matching delimiters

The editor can find the matching partner for delimiter characters such as `()`, `[]`, `{}`, `<>`.

To match a delimiter

- Move cursor to the left of the delimiter character to be matched.
- Choose **Search > Go To Mate** or type **Ctrl+J**.

To select a delimited range

- Move cursor to the left of the delimiter character to be matched.
- Choose **Search > Select To Mate** or type **Ctrl+Shift+J**.

Bookmarks

To edit a document elsewhere and then return to your current location, add a bookmark. The bookmarks presented in this section are *temporary bookmarks* and their positions are not saved when the file is closed nor when the solution is closed.

Adding a bookmark

To add a temporary bookmark, move to the line you want to bookmark and do one of the following:

- On the **Text Edit** tool bar, click the **Toggle Bookmark** button.

—or—

- From the **Edit** menu, click **Bookmarks** then **Toggle Bookmark**.

—or—

- Type **Ctrl+F2**.

A temporary bookmark symbol appears next to the line in the indicator margin which shows that the bookmark has been set.

Moving through bookmarks

To navigate forward through temporary bookmarks, do one of the following:

- On the **Text Edit** tool bar, click the **Next Bookmark** button.

—or—

- From the **Edit** menu, click **Bookmarks** then **Next Bookmark**.

—or—

- Type **F2**.

The editor moves the cursor to the next bookmark set in the document. If there is no following bookmark, the cursor is moved to the first bookmark in the document.

To navigate backward through temporary bookmarks, do one of the following:

- On the **Text Edit** tool bar, click the **Previous Bookmark** button.

—or—

- From the **Edit** menu, click **Bookmarks** then **Previous Bookmark**.

—or—

- Type **Shift+F2**.

The editor moves the cursor to the previous bookmark set in the document. If there is no previous bookmark, the cursor is moved to the last bookmark in the document.

Moving to the first or last bookmark

To move to the first bookmark set in a document, do one of the following:

- From the **Edit** menu, click **Bookmarks** then **First Bookmark**.

—or—

- Type **Ctrl+K, F2**.

To move to the last bookmark set in a document, do one of the following:

- From the **Edit** menu, click **Bookmarks** then **Last Bookmark**.

—or—

- Type **Ctrl+K, Shift+F2**.

Removing bookmarks

To remove a temporary bookmark, move to the line you want to remove the bookmark from and do one of the following:

- On the **Text Edit** tool bar, click the **Toggle Bookmark** button.

—or—

- From the **Edit** menu, click **Bookmarks** then **Toggle Bookmark**.

—or—

- Type **Ctrl+F2**.

The temporary bookmark symbol disappears which shows that the bookmark has been removed.

To remove all temporary bookmarks set in a document, do the following:

- From the **Edit** menu, click **Bookmarks** then **Clear All Bookmarks**.

—or—

- Type **Ctrl+Shift+ F2**.

Changing text

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.

Adding text

The editor has two text input modes:

- **Insertion mode** As text is entered it is inserted at the current cursor position and any text to the right of the cursor is shifted along. A visual indication of insertion mode is that the cursor is a flashing line.
- **Overstrike mode** As text is entered it replaces any text to the right of the cursor. A visual indication of overstrike 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 set to insert mode. You can configure the cursor appearance in both insertion and overstrike modes using the **Tools > Options dialog** in the **Text Editor > General** pane.

Changing to insertion or overstrike mode

To toggle between insertion and overstrike mode, do the following:

- Press the **Insert** button to toggle between insert and overwrite mode.
- If overstrike mode is enabled, the **OVR** status indicator will be enabled and the overstrike cursor will be visible.

Adding or inserting text

To add or insert text, do the following:

- Either click somewhere in the document or move the cursor to the desired location.
- Enter the text.
- If your cursor is between existing characters, the text is inserted between them.

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

Deleting text

The text editor supports the following common editing keystrokes:

Key	Description

Backspace	Deletes one character to the left of the cursor
Delete	Deletes one character to the right of the cursor
Ctrl+Backspace	Deletes one word to the left of the cursor
Ctrl+Delete	Deletes one word to the right of the cursor

Deleting characters

To delete characters or a words in a line, do the following:

- Place the cursor immediately before the word or letter you want to delete.
- Press the **Delete** key as many times as needed to delete the characters or words.

—or—

- Place your cursor at the end of the letter or word you want to delete.
- Press the **Backspace** key as many times as needed to delete the characters or words.

Note You can double-click a word and then press **Delete** or **Backspace** to delete it.

Deleting lines or paragraphs

To delete text which spans more than a few characters, do the following:

- Highlight the text you want to delete by selecting it. You can select text by holding down the left mouse button and dragging over the text, or by using the **Shift** key with the either the arrow keys or the **Home**, **End**, **Page Up**, **Page Down** keys.
- Press **Delete** or **Backspace**.

Using the clipboard

Copying text

To copy the selected text to the clipboard, do one of the following:

- From the **Edit** menu, select **Copy**.

—or—

- Type **Ctrl+C**.

—or—

- Type **Ctrl+Ins**.

To append the selected text to the clipboard, do the following:

- From the **Edit** menu, click **Clipboard** then **Copy Append**.

To copy whole lines from the current editor and place them onto the clipboard

- Select **Edit | Clipboard | Copy Lines** menu item.

To copy whole lines from the current editor and append them onto the end of the clipboard

- Select **Edit | Clipboard | Copy Lines Append** menu item.

To copy bookmarked lines from the current editor place them onto the clipboard

- Select **Edit | Clipboard | Copy Marked Lines** menu item.

To copy bookmarked lines from the current editor and append them onto the end of the clipboard

- Select **Edit | Clipboard | Copy Marked Lines Append** menu item.

Cutting text

To cut the selected text to the clipboard, do one of the following:

- From the **Edit** menu, click **Cut**.

—or—

- Type **Ctrl+X**.

—or—

- Type **Shift+Del**.

To cut selected text from the current editor and append them onto the end of the clipboard

- Select **Edit | Clipboard | Cut Append** menu item.

To cut whole lines from the current editor and place them onto the clipboard

- Select **Edit | Clipboard | Cut Lines** menu item.

To cut whole lines from the current editor and append them onto the end of the clipboard

- Select **Edit | Clipboard | Cut Lines Append** menu item.

To cut bookmarked lines from the current editor and place them onto the clipboard

- Select **Edit | Clipboard | Cut Marked Lines** menu item.

To cut bookmarked lines from the current editor and append them onto the end of the clipboard

- Select **Edit | Clipboard | Cut Marked Lines Append** menu item.

Pasting text

To paste text into current editor from clipboard, do one of the following:

- From the **Edit** menu, click **Paste**.

—or—

- Type **Ctrl+V**.

—or—

- Type **Shift+Ins**.

To paste text into a new editor from clipboard, do the following:

- From the **Edit** menu, click **Clipboard** then **Paste As New Document**.

Clearing the clipboard

To clear the clipboard, do the following:

- From the **Edit** menu, click **Clipboard** then **Clear Clipboard**.

Drag and drop editing

You can select text and then drag and drop it in another location. You can drag text to a different location in the same text editor or to another text editor.

Dragging and dropping text

To drag and drop text, do the following:

- Select the text you want to move, either with the mouse or with the keyboard.
- Click on the highlighted text and keep the mouse button pressed.
- Move the mouse cursor to where you want to place the text.
- Release the mouse button to drop the text.

Dragging text moves it to the new location. You can copy the text to a new location by holding down the **Ctrl** key while moving the text: the mouse cursor changes to indicate a copy. Pressing the **Esc** key while dragging text will cancel a drag and drop edit.

Enabling drag and drop editing

To enable or disable drag and drop editing, do the following:

- From the **Tools** menu, click **Options**.
- Under **Text Editor**, click **General**.
- In the **Editing** section, check **Drag/drop editing** to enable drag and drop editing or uncheck it to disable drag and drop editing.

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 editing actions.

Undoing one edit

To undo one editing action, do one of the following:

- From the **Edit** menu, click **Undo**.

—or—

- On the **Standard** toolbar, click the **Undo** tool button.

—or—

- Type **Ctrl+Z** or **Alt+Backspace**.

Undoing multiple edits

To undo multiple editing actions, do the following:

- On the **Standard** toolbar, click the arrow next to the **Undo** tool button.
- From the menu, select the editing operations to undo.

Undoing all edits

To undo all edits, do one of the following:

- From the **Edit** menu, click **Advanced** then **Undo All**.

—or—

- Type **Ctrl+K, Ctrl+Z**.

Redoing one edit

To redo one editing action, do one of the following:

- From the **Edit** menu, click **Redo**.

—or—

- On the **Standard** toolbar, click the **Redo** tool button.

—or—

- Type **Ctrl+Y** or **Alt+Shift+Backspace**.

Redoing multiple edits

To redo multiple editing actions, do the following:

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

Redoing all edits

To redo all edits, do one of the following:

- From the **Edit** menu, click **Advanced** then **Redo All**.

—or—

- Type **Ctrl+K, Ctrl+Y**.

Indentation

The editor uses the **Tab** key to increase or decrease the indentation level. The indentation size can be altered in the editor's **Language Properties** window.

Changing indentation size

To change the indentation size, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.
- Set the **Indent Size** property for the required language.

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

Selecting tab or space fill when indenting

To enable or disable the use of tab characters when indenting, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.
- Set the **Use Tabs** property for the required language. Note that changing this setting does not add or remove existing tabs from files, the change will only effect new indents.

The editor can provide assistance 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 **Return** or **Enter** is pressed, the editor automatically moves the cursor to the indentation level of the previous line.
- **Smart** The editor analyses the source code to compute the appropriate indentation level for the line. The number of lines before the current cursor position that are analysed for context can be altered. The smart indent mode can be configured to either indent open and closing braces or the lines following the braces.

Changing indentation options

To change the indentation mode, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.
- Set the **Indent Mode** property for the required language.

To change whether opening braces are indented in smart indent mode, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.
- Set the **Indent Opening Brace** property for the required language.

To change whether closing braces are indented in smart indent mode, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.
- Set the **Indent Closing Brace** property for the required language.

Changing indentation context

To change number of previous line used for context in smart indent mode, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.
- Set the **Indent Context Lines** property for the required language.

File management

To create a file

- Select **File > New > New File** menu item.

Opening an existing document

To open an existing document, do one of the following:

- Click **File > Open...**
- Choose the file to open from the dialog and click **Open**.

—or—

- Type **Alt+O**.
- Choose the document to open from the dialog and click **Open**.

Opening multiple documents

To open multiple existing documents in the same directory, do one of the following

- Select **File > Open**.
- Choose multiple documents to open from the dialog. Hold down **Ctrl** key to add individual documents or hold down **Shift** to select a range of documents.
- Click **Open**.

Saving a document

To save a file, do one of the following:

- Select the document to save.
- From the **File** menu, click **Save**.

—or—

- Select the document to save.
- Type **Ctrl+S**.

—or—

- Select the document to save.
- Click the document icon in the document's title bar.
- From the popup menu, click **Save**.

Saving a document to a different name

To save a file, do one of the following:

- Select the document to save.
- From the **File** menu, click **Save As...**
- Enter the new file name and click **Save**.

—or—

- Select the document to save.
- Click the document icon in the document's title bar.
- From the popup menu, click **Save As...**
- Enter the new file name and click **Save**.

Printing a document

To print a document, do one of the following:

- Select editor to print.
- From the **File** menu, click **Print...**
- Select the printer to print to and click **OK**.

—or—

- Click the document icon in the document's title bar.
- From the popup menu, click **Print...**
- Select the printer to print to and click **OK**.

To insert a file at the current cursor position

- Select the editor to insert file into.
- Move the cursor to the required insertion point.
- Select **Edit > Insert File** menu item.
- Select file to insert.
- Click **Open** button.

To toggle a file's write permission

- Select the editor containing the file.
- Select **Edit > Advanced > Toggle Read Only**.

Find and replace

To find text in a single file

- Select **Edit | Find and Replace | Find...** menu item.
- Enter the string to be found in the **Find what** 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, the beginning or the end of line on either side 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.
- If the search should move up the document from the current cursor position rather than down the document, set the **Search up** option.
- Click **Find** button to find next occurrence of the string or click **Mark All** to bookmark all lines in the file containing the string.

To find text within a selection

- Select text to be searched.
- Select **Edit | Find and Replace | Find...** menu item.
- Enter the string to be found in the **Find what** 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, the beginning or the end of line on either side 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.
- If the search should move up the document from the current cursor position rather than down the document, set the **Search up** option.
- Click **Mark All** to bookmark all lines in the selection containing the string.

To find and replace text

- Select **Edit | Find and Replace | Replace...** menu item.
- Enter the string to be found in the **Find what** input.
- Enter the string to replace the found string with in the **Replace with** input. If the search string is a **Regular Expression** then the `\n` backreference can be used in the replace 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, the beginning or the end of line on either side 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.
- If the search should move up the document from the current cursor position rather than down the document, set the **Search up** option.
- Click **Find** button to find next occurrence of string and then **Replace** button to replace the found string with replacement string or click **Replace All** to replace all occurrences of the string without prompting.

To find text in multiple files

- Select **Edit | Find and Replace | Find in Files...** menu item.
- Enter the string to be found in the **Find what** input.
- Enter the wildcard to use to filter the files in the **In file types** input.
- Enter the folder to start search in the **In folder** 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, the beginning or the end of line on either side 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.
- If the search will be carried out in the root folder's sub-folders, set the **Look in subfolders** option.
- The output of the search results can go into two separate panes. If the output should go into the second pane, select **Output to pane 2** option.
- Click **Find** button.

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 modelled 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 that the CrossStudio editor supports.

Characters	Meaning
\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 character is 0, 1, 2, 7 8 or 9. A range can be negated using the '^' character, e.g. '[^a-z]' matches if character is anything other than a lower case alphabetic character.
\c	The literal character c. For example to match the character * you would use *.
\a	Match ASCII bell character.
\f	Match ASCII form feed character.
\n	Match ASCII line feed character.
\r	Match ASCII carriage return character.
\t	Match ASCII horizontal tab character.
\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.
{ <i>n</i> }	Match <i>n</i> occurrences of the preceding expression.
{ <i>n</i> ,}	Match at least <i>n</i> occurrences of the preceding expression.
{, <i>m</i> }	Match at most <i>m</i> occurrences of the preceding expression.
{ <i>n</i> , <i>m</i> }	Match at least <i>n</i> and at most <i>m</i> occurrences of the preceding expression.
^	Beginning of line.
\$	End of line.
\b	Word boundary.
\B	Non-word boundary.
(<i>e</i>)	Capture expression <i>e</i> .
\n	Backreference 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** check box must be set in the search and replace dialog. Once enabled, the regular expressions can be used in the **Find what** search string. The **Replace with** strings can use the "\n" backreference string to reference any captured strings.

"Find what" String	"Replace with" String	Description
u\w.d		Search for any length string containing one or more word characters beginning with the character 'u' and ending in the character 'd'.
^.*;\$		Search for any lines ending in a semicolon.
(typedef.\s+)(\S+);	\1TEST_\2;	Find C type definition and insert the string "TEST" onto the beginning of the type name.

Advanced editor features

Code Templates

The editor provides the ability to use code templates. A code template is a block of frequently used source code that can be inserted automatically by using a particular key sequence. A `|` character is used in the template to indicate the required position of the cursor after the template has been expanded.

To view code templates

- Select **Edit > Advanced > View Code Templates** menu item.

Code templates can either be expanded manually or automatically when the **Space** key is pressed.

To expand a code template manually

- Type a key sequence, for example the keys **c** followed by **b** for the comment block template.
- Select **Edit > Advanced > Expand Template** or type **Ctrl+J** to expand the template.

To expand the template automatically

- Ensure the **Expand Templates On Space** editor property is enabled.
- Type a key sequence, for example the keys **c** followed by **b** for the comment block template.
- Now type **Space** key to expand the template.

Editing Macros

The editor has a number of built-in macros for carrying out common editing actions.

To declare a type

- Select **Edit > Editing Macros > Declare Or Cast To** menu item for required type.

To cast to a type

- Select text in the editor containing expression to cast.
- Select **Edit > Editing Macros > Declare Or Cast To** menu item for required type cast.

To insert a qualifier

- Select **Edit > Editing Macros > Insert** menu item for required qualifier.

Tab Characters

The editor can either use tab characters or only use space characters to fill whitespace. The use of tabs or spaces when indenting can be specified in the editor's language properties. The editor can also add or remove tabs characters in blocks of selected text.

To replace spaces with tab characters in selected text

- Select text.
- Select **Edit > Advanced > Tabify Selection** menu item

To replace tab characters with spaces in selected text

- Select text.
- Select **Edit > Advanced > Untabify Selection** menu item

Changing Case

The editor can change the case of selected areas of text.

To change case of selected text to uppercase

- Select text.
- Select **Edit > Advanced > Make Selection Uppercase** menu item.

To change case of selected text to lowercase

- Select text.
- Select **Edit > Advanced > Make Selection Lowercase** menu item.

Commenting

The editor can add or remove language specific comment characters to areas of text.

To comment out an area of selected text

- Select text to comment out.
- Select **Edit > Advanced > Comment** menu item.

To uncomment an area of selected text

- Select text to remove comment characters from.
- Select **Edit > Advanced > Uncomment** menu item.

Indentation

The editor can increase or decrease the indentation level of an area of selected text.

To increase indentation of selected text

- Select text.
- Select **Edit > Advanced > Increase Line Indent** menu item.

To decrease indentation of selected text

- Select text.
- Select **Edit > Advanced > Decrease Line Indent** menu item.

Sorting

The editor can sort areas of selected text in ascending or descending ASCII order.

To sort selected lines into ascending order

- Select text to sort.
- Select **Edit > Advanced > Sort Ascending** menu item.

To sort selected lines into descending order

- Select text to sort.
- Select **Edit > Advanced > Sort Descending** menu item.

Text Transposition

The editor can transpose word or line pairs.

To transpose the word at the current cursor position with the previous word

- Select **Edit > Advanced > Transpose Words** menu item.

To transpose the current line with the previous line

- Select **Edit > Advanced > Transpose Lines** menu item.

Whitespace

To make whitespace visible

- Select **Edit > Advanced > Visible Whitespace** menu item.

Code templates

The editor provides the ability to use code templates. A code template is a block of frequently used source code that can be inserted automatically by using a particular key sequence. A `'|'` character is used in the template to indicate the required position of the cursor after the template has been expanded.

Editing code templates

To edit code templates, do the following:

- From the **Edit** menu, click **Advanced** then **View Code Templates**.

Code templates can either be expanded manually or automatically when the **Space** key is pressed.

Manually expanding a template

To expand a code template manually, do the following:

- Type a key sequence, for example the keys **c** followed by **b** for the comment block template.
- From the Edit menu, click **Advanced** then **Expand Template** or type **Ctrl+J** to expand the template.

Automatically expanding templates

To expand the template automatically, do the following:

- Ensure the **Expand Templates On Space** editor property is enabled.
- Type a key sequence, for example the keys **c** followed by **b** for the comment block template.
- Now type **Space** key to expand the template.

Linking and section placement

Executable programs consists of a number of program sections. Typically there will be program sections for code, initialised data and zero'd data. There will often be more than one code section and these will require placement at specific addresses in memory.

To describe how the program sections of your program are positioned in memory the CrossWorks project system uses a [memory map file](#) and a [section placement file](#). These files are both xml files and can be edited with the CrossWorks text editor. The memory map file specifies the start address and size of memory segments of the target. The section placement file specifies where to place program sections in the memory segments of the target. 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.

```
<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 list the sections that are to be placed in those segments. This is done using a memory segment name in the section placement file that matches a 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** section in the **FLASH** segment would look like.

```
<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 before **.text**.

The memory map file and section placement file to use for linkage can either be included as a part of the project or alternatively they can be specified the [linker properties](#) of the project.

You can create a new program section using the either the assembler or the compiler. For the C compiler this can be achieved using one of the **#pragma** directives. For example:

```
#pragma codeseg(".foo")
void foobar(void);
#pragma codeseg(default)
```

This will allocate **foobar** in the section called **.foo**. Alternatively you can specify the section names of the code, constant, data and zero'd data for an entire compilation unit using the Section Options of the [compiler properties](#) of the project.

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 right click menu on project explorer.

Sections containing code and constant data should have their **load** property set to be "Yes". There are sections that don't require any loading such as stack sections and zero'd data sections, these sections should have the **load** property set to "No".

You can specify that initialisation data is stored in the default program section using the .init directive and you can refer to the start and end of the section using the SFE and SFB directives. If for example you create a new data section called "IDATA2" you can store this in the program by putting the following into the startup code

```
data_init_begin2
.init
```

"IDATA2"

```
data_init_end2
```

You can then use these symbols to copy the stored section information into the data section using (an assembly coded version of)

```
memcpy(SFB(IDATA2), data_init_begin2, data_init_end2-data_init_end2)
```


CrossStudio Windows

This section is a reference to each of the windows in the CrossStudio environment.

In this section

Breakpoints window

Describes how to use the breakpoints window to manage breakpoints in a program.

Call stack window

Describes how to traverse the call stack to examine data and where function calls came from.

Clipboard ring window

Describes how to use the clipboard ring to make complex cut-and-pastes easier.

Execution counts window and Trace Window

Describes how to gather useful profiling statistics on your application on the simulator and targets that support execution profiling and tracing.

Globals window, Locals window, and Watch windows.

Describes how to examine your application's local and global variables and how to watch specific variables.

Memory windows

Describes how to look at target memory in raw hexadecimal form.

Register window

Describes how to examine processor registers and peripherals defined by the project's memory map file.

Threads window

Describes how CrossStudio can display thread-local data, tasks and objects when you run your application under a real-time operating system.

Help window

Describes how the CrossStudio help system works and how to get answers to your questions.

Output window

Describes the output window and the logs it contains.

Project explorer

Describes the project explorer and how to manage your projects.

Properties window

Describes the property window and how to change environment and project properties using it.

Source navigator

Describes how to use the Source Navigator to easily browse your project's functions, methods, and variable.

Symbol browser

Describes how you can use the Symbol browser to find out how much code and data your application requires.

Targets window

Describes how to manage your target connections by creating new ones, editing existing ones, and deleting unused ones.

Breakpoints window

The **Breakpoints** window manages the list of currently set breakpoints on the solution. Using the breakpoint 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 breakpoint window for the duration of the debug run but are not saved when you stop debugging.

When a breakpoint is hit then the matched breakpoint will be highlighted in the breakpoint window.

Breakpoints window layout

The **Breakpoints** window is divided into a tool bar and the main breakpoint display.

The Breakpoint tool bar

Button	Description
	Creates a new breakpoint using the New Breakpoint dialog.
	Toggles the selected breakpoint between enabled and disabled states.
	Removes the selected breakpoint.
	Moves the cursor to the statement that the selected breakpoint is set at.
	Deletes all breakpoints.
	Disables all breakpoints.
	Enables all breakpoints.
	Creates a new breakpoint group and makes it active.

The Breakpoints window display

The main part of the **Breakpoints** window displays the breakpoints that have been set and what 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 there is no executable code 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 display the **Breakpoints** window if it is hidden, do one of the following:

- From the **View** menu, click **Other Windows** then **Breakpoints**.

—or—

- From the **Debug** menu, click **Debug Windows** then **Breakpoints**.

—or—

- Type **Ctrl+Alt+B**.

—or—

- On the **Debug** tool bar, click the **Breakpoints** icon.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Other Windows** then **Breakpoints**.

Managing single breakpoints

You can manage breakpoints in the **Breakpoint** window.

Deleting a breakpoint

To delete a breakpoint, do the following:

- In the **Breakpoints** window, click the breakpoint to delete.
- From the **Breakpoints** window tool bar, click the **Delete Breakpoint** button.

Editing a breakpoint

To edit the properties of a breakpoint, do the following:

- In the **Breakpoints** window, right click the breakpoint to edit.
- From the popup menu, click **Edit Breakpoint**.
- Edit the breakpoint in the **New Breakpoint** dialog.

Enabling or disabling a breakpoint

To toggle the enable state of a breakpoint, do one of the following:

- In the **Breakpoints** window, right click the breakpoint to enable or disable.
- From the popup menu, click **Enable/Disable Breakpoint**.

—or—

- In the **Breakpoints** window, click the breakpoint to enable or disable.
- Type **Ctrl+F9**.

Managing 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.

Creating a new breakpoint group

To create a new breakpoint group, do one of the following:

- From the **Breakpoints** window tool bar, click the **New Breakpoint Group** button.

—or—

- From the **Debug** menu, click **Breakpoints** then **New Breakpoint Group**.

—or—

- Right click anywhere in the **Breakpoints** window.
- From the popup menu, click **New Breakpoint Group**.

In the **New Breakpoint Group Dialog**, enter the name of the breakpoint group.

Selecting a new active breakpoint group

When you create a breakpoint, it is added to the active breakpoint group. To make a group the active group, do the following:

- In the **Breakpoints** window, click the breakpoint group to make active.
- From the popup menu, click **Set as Active Group**.

Deleting a breakpoint group

To delete a breakpoint group, do the following:

- In the **Breakpoints** window, right click the breakpoint group to delete.
- From the popup menu, click the **Delete Breakpoint Group** button.

Enabling all breakpoints in a breakpoint group

You can enable all breakpoints within a group as a whole. To enable all breakpoints in a group, do the following:

- In the **Breakpoints** window, right click the breakpoint group to enable.
- From the popup menu, click **Enable Breakpoint Group**.

Disabling all breakpoints in a breakpoint group

You can disable all breakpoints within a group as a whole. To disable all breakpoints in a group, do the following:

- In the **Breakpoints** window, right click the breakpoint group to disable.
- From the popup menu, click **Disable Breakpoint Group**.

Managing all breakpoints

You can delete, enable, or disable all breakpoints.

Deleting all breakpoints

To delete all breakpoints, do one of the following:

- From the **Debug** menu, click **Breakpoints** then **Delete All Breakpoints**.

—or—

- From the **Breakpoints** window tool bar, click the **Delete All Breakpoints** button.

—or—

- Type **Ctrl+Shift+F9**.

Enabling all breakpoints

To enable all breakpoints, do one of the following:

- From the **Debug** menu, click **Breakpoints** then **Enable All Breakpoints**.

—or—

- From the **Breakpoints** window tool bar, click the **Enable All Breakpoints** button.

Disabling all breakpoints

To disable all breakpoints, do one of the following:

- From the **Debug** menu, click **Breakpoints** then **Disable All Breakpoints**.

—or—

- From the **Breakpoints** window tool bar, click the **Disable All Breakpoints** button.

Clipboard ring window

The code editor captures all **Cut** and **Copy** operations and stores the the cut or copied item on the **Clipboard Ring**. The clipboard ring stores the last 20 text items that were cut or copied, but you can configure the maximum number of items stored on the clipboard ring 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.

Showing the clipboard ring

To display the **Clipboard Ring** window if it is hidden, do one of the following:

- From the **View** menu, click **Clipboard Ring**.

—or—

- Type **Ctrl+Alt+C**.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Clipboard Ring**.

Pasting an item by cycling the clipboard ring

To paste from the clipboard ring, do the following:

- 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.
- Type **Ctrl+Shift+V** to paste the clipboard ring's current item to the current document.
- Repeatedly type **Ctrl+Shift+V** to cycle through the entries in the clipboard ring until you get to the one you want to permanently paste in the document. Each time you press **Ctrl+Shift+V**, the editor replaces the last entry you pasted from the clipboard ring so that you end up with only the last one you selected. The item you stop on then becomes the current item.
- Move to another location or cancel the selection. You can use **Ctrl+Shift+V** to paste the current item again or cycle the clipboard ring to a new item.

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

Pasting a specific item into a document

To paste an item on the clipboard ring directly into the current document, do one of the following:

- Move the cursor to the position where you want to paste the item into the document.
- Display the dropdown menu of the item to paste by clicking the arrow to its right.
- From the menu, click **Paste**.

—or—

- Make the item you want to paste the current item by clicking it.
- Move the cursor to the position where you want to paste the item into the document.
- Type **Ctrl+Shift+V**.

Pasting all items into a document

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

- From the **Edit** menu, click **Clipboard Ring** then **Paste All**.

—or—

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

—or—

- Type **Ctrl+R, Ctrl+V**.

Removing a specific item from the clipboard ring

To remove an item from the clipboard ring, do the following:

- Display the dropdown menu of the item to delete by clicking the arrow at the right of the item.
- From the menu, click **Delete**.

Removing all items from the clipboard ring

To remove all items from the clipboard ring, do one of the following:

- From the **Edit** menu, click **Clipboard Ring** then **Clear Clipboard Ring**.

—or—

- On the **Clipboard Ring** tool bar, click the **Clear Clipboard Ring** button.

—or—

- Type **Ctrl+R, Delete**.

Configuring the clipboard ring

To configure the clipboard ring, do the following:

- From the **Tools** menu, select **Options**.
- Under **Environment**, select **Even More...**
- Check **Preserve Contents** to save the content of the clipboard ring between runs, or uncheck it to start with an empty clipboard ring.
- Change **Maximum Items** to configure the maximum number of items stored on the clipboard ring.

Call stack window

The **Call Stack** window displays the list of function calls (stack frames) that are active at the point that program execution halted. When program 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 do 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](#).

Call Stack user interface

The **Call Stack** window is divided into a tool bar and the main breakpoint display.

Call Stack tool bar

Button	Description
	Moves the cursor to where the call to the selected frame was made.
	Sets the debugger context to the selected stack frame.
	Moves the debugger context down one stack to the called function
	Moves the debugger context up one stack to the calling function
	Selects the fields to display for each entry in the call stack.
	Sets the debugger context to the most recent stack frame and moves the cursor to the currently executing statement.

Call Stack display

The main part of the **Call Stack** window displays each unfinished function call (active stack frame) at the point that program execution halted. The most recent stack frame is displayed at the bottom of the list and the eldest 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 display the **Call Stack** window if it is hidden, do one of the following:

- From the **View** menu, click **Other Windows** then **Call Stack**.

—or—

- From the **Debug** menu, click **Debug Windows** then **Call Stack**.

—or—

- Type **Ctrl+Alt+S**.

—or—

- On the **Debug** tool bar, click the **Call Stack** icon.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Other Windows** then **Breakpoints**.

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** to display varying amounts of information for each stack frame. By default, CrossStudio displays all information.

Displaying or hiding parameter names

To display or hide the name of each parameter in the call stack, do the following:

- On the **Call Stack** tool bar, click the **Fields** button.
- From the dropdown menu, check or uncheck **Parameter Names**.

Displaying or hiding parameter values

To display or hide the value of each parameter in the call stack, do the following

- On the **Call Stack** tool bar, click the **Fields** button.
- From the dropdown menu, check or uncheck **Parameter Valuev**.

Displaying or hiding parameter types

To display or hide the type of each parameter in the call stack, do the following:

- On the **Call Stack** tool bar, click the **Fields** button.
- From the dropdown menu, check or uncheck **Parameter Types**.

Displaying or hiding file names and source line numbers

To display or hide the file name and source line number columns of each frame in the call stack, do the following:

- On the **Call Stack** tool bar, click the **Fields** button.
- From the dropdown menu, check or uncheck **Call Source Location**.

Displaying or hiding call addresses

To display or hide the call address of each frame in the call stack, do the following:

- On the **Call Stack** tool bar, click the **Fields** button.
- From the dropdown menu, check or uncheck **Call Address**.

Changing the debugger context

You can select the stack frame for the debugger context from the **Call Stack** window.

Selecting a specific stack frame

To move the debugger context to a specific stack frame, do one of the following:

- In the **Call Stack** window, double click the stack frame to move to.

—or—

- In the **Call Stack** window, click the stack frame to move to.
- On the **Call Stack** window's tool bar, click the **Switch To Frame** button.

—or—

- In the **Call Stack** window, right click the stack frame to move to.
- From the popup menu, select **Switch To Frame**.

The debugger moves the cursor to the statement where the call was made. If there is no debug information for the statement at the call location, CrossStudio opens a disassembly window at the instruction.

Moving up one stack frame

To move the debugger context up one stack frame to the calling function, do one of the following:

- On the **Call Stack** window's tool bar, click the **Up One Stack Frame** button.

—or—

- On the **Debug Location** tool bar, click the **Up One Stack Frame** button.

—or—

- Type **Alt++**.

The debugger moves the cursor to the statement where the call was made. If there is no debug information for the statement at the call location, CrossStudio opens a disassembly window at the instruction.

Moving down one stack frame

To move the debugger context down one stack frame to the called function, do one of the following:

- On the **Call Stack** window's tool bar, click the **Down One Stack Frame** button.

—or—

- On the **Debug Location** tool bar, click the **Down One Stack Frame** button.

—or—

- Type **Alt++**.

The debugger moves the cursor to the statement where the call was made. If there is no debug information for the statement at the call location, CrossStudio opens a disassembly window at the instruction.

Setting a breakpoint on a return to a function

To set a breakpoint on return to a function, do one of the following:

- In the **Call Stack** window, click the stack frame on the function to stop at when it is returned to.
- From the **Build** tool bar, click the **Toggle Breakpoint** button.

—or—

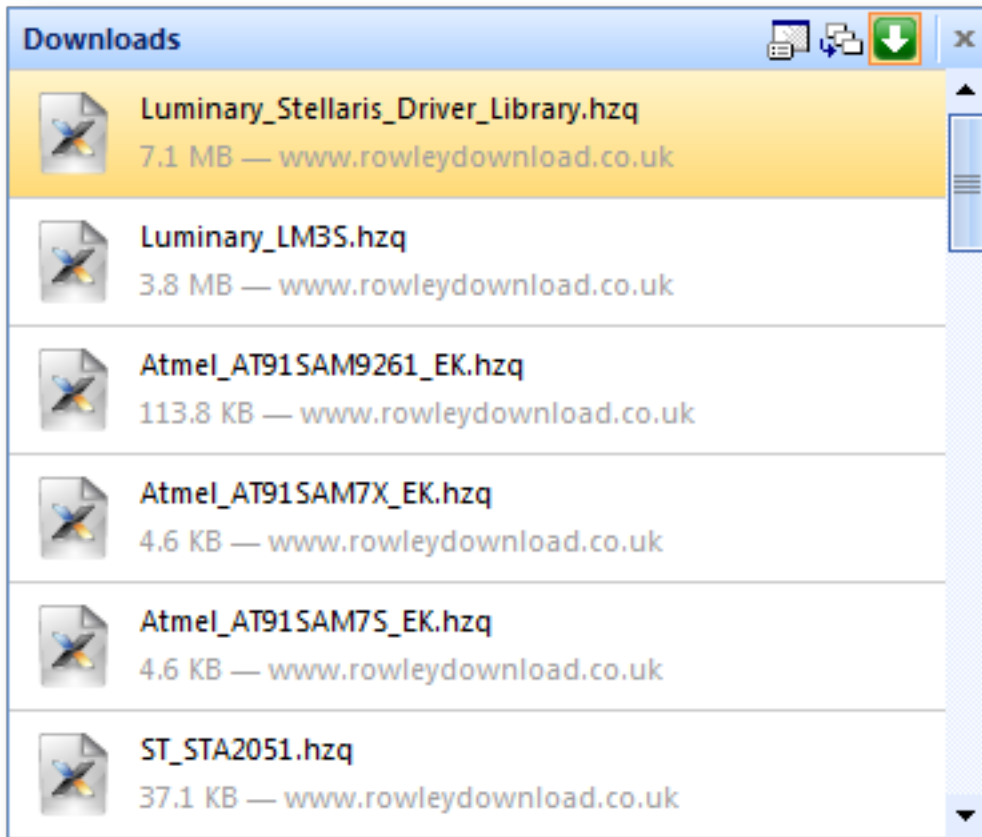
- In the **Call Stack** window, click the stack frame on the function to stop at when it is returned to.
- Type **F9**.

—or—

- In the **Call Stack** window, right click the function to stop at when it is returned to.
- From the popup menu, click **Toggle Breakpoint**.

Downloads window

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



To display the **Downloads Window**:

- Click **Tools > Downloads Window**.

Execution counts window

The Execution Counts window shows a list of source locations and the number of times those source locations have been executed. This window is only available for targets that support the collection of jump trace information.

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.

The counts window can be sorted by any column (counts, source file, or function name) by clicking on the appropriate column header. Double clicking on an entry will locate the source display to the appropriate source code location.

Find and replace window

The find and replace window allows you to carry out text search and replacement in the current document or in a range of specified files.

To open the find and replace window

- From **Search** menu, click **Find and Replace**.

—or—

- Type **Ctrl+Alt+F**.

To find text in a single file

- Select the



Find button on the tool bar.

- Enter the string to be found in the **Find what** 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, the beginning or the end of line on either side 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 or click **Bookmark All** to bookmark all lines in the file containing the string.

To find and replace text in a single file

- Select the



Replace button on the tool bar.

- Enter the string to be found in the **Find what** input.
- Enter the string to replace the found string with in the **Replace with** input. If the search string is a **Regular Expression** then the `\n` backreference can be used in the replace 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, the beginning or the end of line on either side 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 string and then **Replace** button to replace the found string with replacement string or click **Replace All** to replace all occurrences of the string without prompting.

To find text in multiple files

- Select the



Find In Files button on the tool bar.

- Enter the string to be found in the **Find what** input.
- Select whether you want to carry out the search in all the open documents, all the documents contained in the current project, all the documents in the current solution or all the files in a specified folder by selecting the appropriate option in the **Look in** input.
- If you have specified that you want to search in a specified folder select the folder you want to search in by entering the path in the **Folder** input and specify the type of files you want to search using the **Look in files matching** 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, the beginning or the end of line on either side 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 the 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

- Select the



Replace In Files button on the tool bar.

- Enter the string to be found in the **Find what** input.
- Enter the string to replace the found string with in the **Replace with** input. If the search string is a **Regular Expression** then the `\n` backreference can be used in the replace string to reference captured text.
- Select whether you want to carry out the search and replace in all the open documents, all the documents contained in the current project, all the documents in the current solution or all the files in a specified folder by selecting the appropriate option in the **Look in** input.
- If you have specified that you want to search in a specified folder select the folder you want to search in by entering the path in the **Folder** input and specify the type of files you want to search using the **Look in files matching** 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, the beginning or the end of line on either side 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 the occurrences of the string in the specified files.

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 that variables cannot be added to or deleted from the globals window.

Globals window user interface

The **Globals** window is divided into a tool bar and the main data display.

Globals tool bar

Button	Description
	Displays the selected item in binary.
	Displays the selected item in octal.
	Displays the selected item in decimal.
	Displays the selected item in hexadecimal.
	Displays the selected item as a signed decimal.
	Displays the selected item as a character or Unicode character.
	Sets the displayed range in the active memory window to the where the selected item is stored.
	Sorts the global variables alphabetically by name.
	Sorts the global 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 automatically updates to show the active stack frame and new variable values. Items that have changed since they were previously displayed are highlighted in red.

Showing the Globals window

To display the **Globals** window if it is hidden, do one of the following:

- Click **View > Other Windows > Globals**, or
- Click **Debug > Debug Windows > Globals**.

—or—

- Type **Ctrl+Alt+G**.

—or—

- Right click the tool bar and click **Other Windows > Globals**.

Changing display format

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

To change the display format of a global variable, do one of the following:

- Right click the item to change.
- From the popup menu, select the format to display the item in.

—or—

- Click the item to change.
- On the **Globals** window tool bar, select the format to display the item in.

Modifying global variable values

To modify the value of a global variable, do one of the following:

- 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**'.

—or—

- Right click the value of the global variable to modify.
- From the popup menu, select one of the operations to modify the global variable value.

CrossStudio Help and Assistance

CrossStudio provides context sensitive help with increasing detail:

Tool tips

When you move your mouse pointer over a tool button and keep it still, a small window appears with a very brief description of the tool button and its keyboard shortcut if it has one.

Status tips

In addition to tool tips, CrossStudio provides a longer description in the status bar when you hover over a tool button or when you move over a 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 which is available at all times. To go to the help information for a particular window or user interface element, do the following:

- Focus the appropriate element by clicking it.
- Choose **Help > CrossStudio Help** or type **F1**.

You can return to the **Welcome** page at any time:

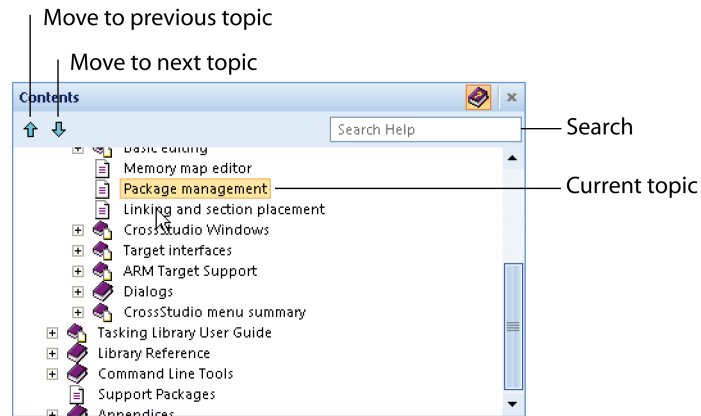
- Choose **Help > Welcome**.

Help from the text editor

The text editor is linked to the help system in a special way. If you place the cursor over a word and press **F1**, that word is looked up in the help system index and the most likely page displayed in the HTML browser—it's a great way to quickly find the reference help pages for functions provided in the library.

Browsing the documentation

The **Contents** window provides a list of all the topics present in the CrossWorks documentation and a way to search through it.



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 tool buttons in the Contents window tool bar, move the selected topic.

You can search the online documentation by typing a search phrase into the **Search** box in the **Contents** window toolbar.

To search the online documentation:

- Choose **Help > Search**.
- Enter your search phrase in the **Search** box and press **Return**.

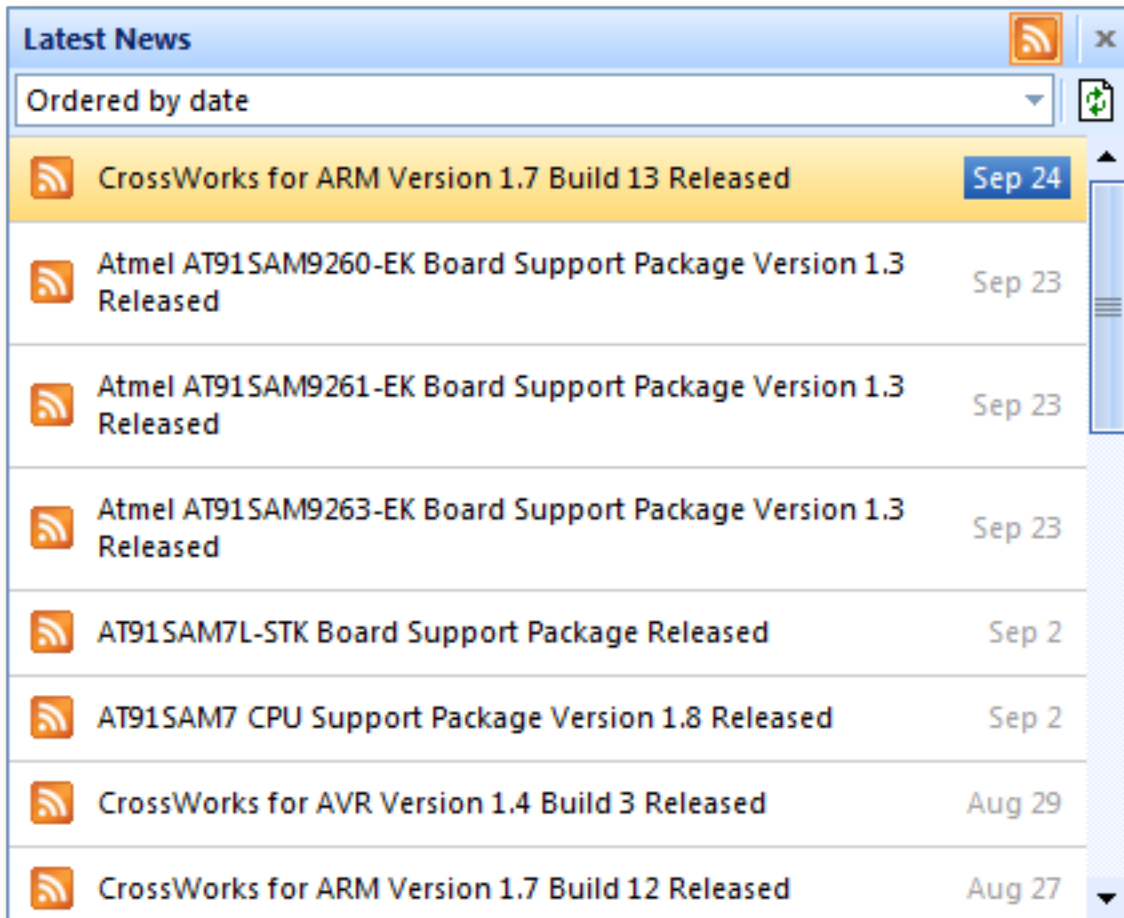
The search commences and the table of contents are replaced by links to matching pages, listed in order of relevance. To clear the search and return to the table of contents, click the clear icon in the **Search** box.

Immediate window

The **Immediate** window allows you to type in [debug expressions](#) and displays the results. The results are displayed in the format specified by the **Default Display Mode** which is set in the [Debugging Environment Options](#) dialog.

Latest news window

The **Latest News Window** displays a historical list of news articles from the Rowley Associates web site.



To display the **Latest News Window**:

- Click **Help > Latest News**.

Locals window

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

Locals window user interface

The **Locals** window is divided into a tool bar and the main data display.

Locals tool bar

Button	Description
	Displays the selected item in binary.
	Displays the selected item in octal.
	Displays the selected item in decimal.
	Displays the selected item in hexadecimal.
	Displays the selected item as a signed decimal.
	Displays the selected item as a character or Unicode character.
	Sets the displayed range in the active memory window to the where the selected item is stored.
	Sorts the local variables alphabetically by name.
	Sorts the local 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** tool bar items or select a new frame in the Call Stack window. When the program stops at a breakpoint or is stepped, the Locals window automatically updates to show the active stack frame. Items that have changed since they that were previously displayed are highlighted in red.

To display the **Locals** window:

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

Changing display format

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

To change the display format of a local variable, do one of the following:

- Right click the item to change.
- From the popup menu, select the format to display the item in.

—or—

- Click the item to change.
- On the Locals window tool bar, select the format to display the item in.

Modifying local variable values

To modify the value of a local variable, do one of the following:

- 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 popup menu, select one of the operations to modify the local variable value.

Memory window

The memory window shows the contents of the connected target's memory areas. The memory window does not show the complete address space of the target, instead you must enter both the start address and the number of bytes for the memory window to display. You can specify the start address and the size using [debugger expressions](#) which enables you to position the memory display at the start address of a variable or use a value in a register. You can also specify if 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. The memory window updates each time your program stops on a breakpoint or single step and whenever you traverse the call stack. If any values that were previously displayed have changed they will be displayed in red.

Memory window user interface

The **Memory** window is divided into a tool bar and the main data display.

Memory tool bar

Button	Description
Address	Start address to display, specified as a debugger expression .
Size	Number of bytes to display, specified as a debugger expression .
	Select binary display.
	Select octal display.
	Select unsigned decimal display.
	Select signed decimal display.
	Select hexadecimal display.
	Select byte display which also includes an ASCII display.
	Select 2 byte display.
	Select 4 byte display.
	Evaluate the address and size expressions and update the memory window.
	Move the data display up one line.
	Move the data display down one line.



Move the data display up by **Size** bytes.



Move the data display down by **Size** bytes.

Left click operations

The following operations are available with the left click of the mouse:

Action	Description
Single Click	First click selects the line, second click selects the displayed memory value. Once the memory value is selected it can be modified by typing in a new value. Note that the input radix is the same as the display radix i.e. 0x is not required to specify a hex number.

Right click menu operations

The following operations are available on the right click menu:

Action	Description
Auto Evaluate	Re-evaluate Address and Size each time the memory window is updated.
Set Number of Columns...	Set the number of columns to display - default is 8.
Access Memory By Display Width	Access memory in terms of the display width.
Export To Binary Editor	Create a binary editor with the current memory window contents.
Save As	Save the current memory window contents to a file. Supported file formats are Binary File , Motorola S-Record File , Intel Hex File , TI Hex File , and Hex File .
Load From	Load the current memory window from a file. Supported file formats are Binary File , Motorola S-Record File , Intel Hex File , TI Hex File , and Hex File .

Using the memory window

Showing the memory window

To display memory window *n* if it is hidden, do one of the following:

- Click **View > Other Windows > Memory > Memory *n***.

—or—

- Click **Debug > Debug Windows > Memory > Memory *n***.

—or—

- Type **Ctrl+T, M, n**.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Other Windows > Memory > Memory n**.

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 change the number of columns that are displayed.

You can change a value in the memory window by clicking the value to change and editing it as a text field. Note that when you modify memory values you need to prefix hexadecimal numbers with “0x”, binary numbers with “0b” and octal numbers with “0”.

Saving memory contents

You can save the displayed contents of the memory window to a file in various formats. Alternatively you can export the contents to a binary editor to work on them.

Saving memory

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, do the following:

- Selects the start address and number of bytes to save by editing the **Start Address** and **Size** fields in the memory window tool bar.
- Right click the main memory display.
- From the popup menu, select **Save As** then select the format from the submenu.

Exporting memory

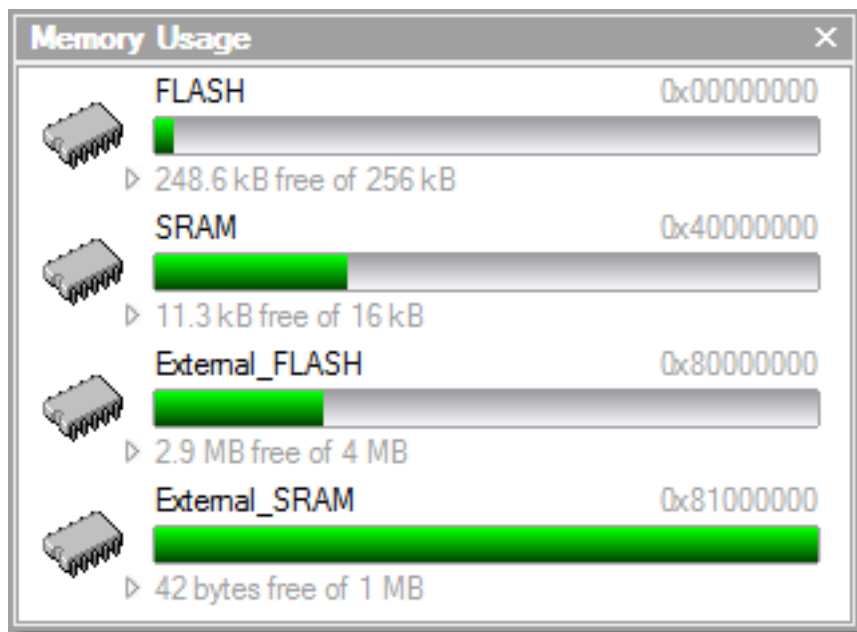
To export the current state of memory to a binary editor, do the following:

- Select the start address and number of bytes to save by editing the **Start Address** and **Size** fields in the memory window tool bar.
- Right click the main memory display.
- From the popup menu, select **Export to Binary Editor**.

Note that subsequent modifications in the binary editor will not modify memory in the target.

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 and the green coloured area represents the area of the segment that has code or data placed in it.

Showing the Memory Usage Window

To activate the **Memory Usage Window** if it is hidden, do one of the following:

- From the **View** menu, click **Memory Usage**.

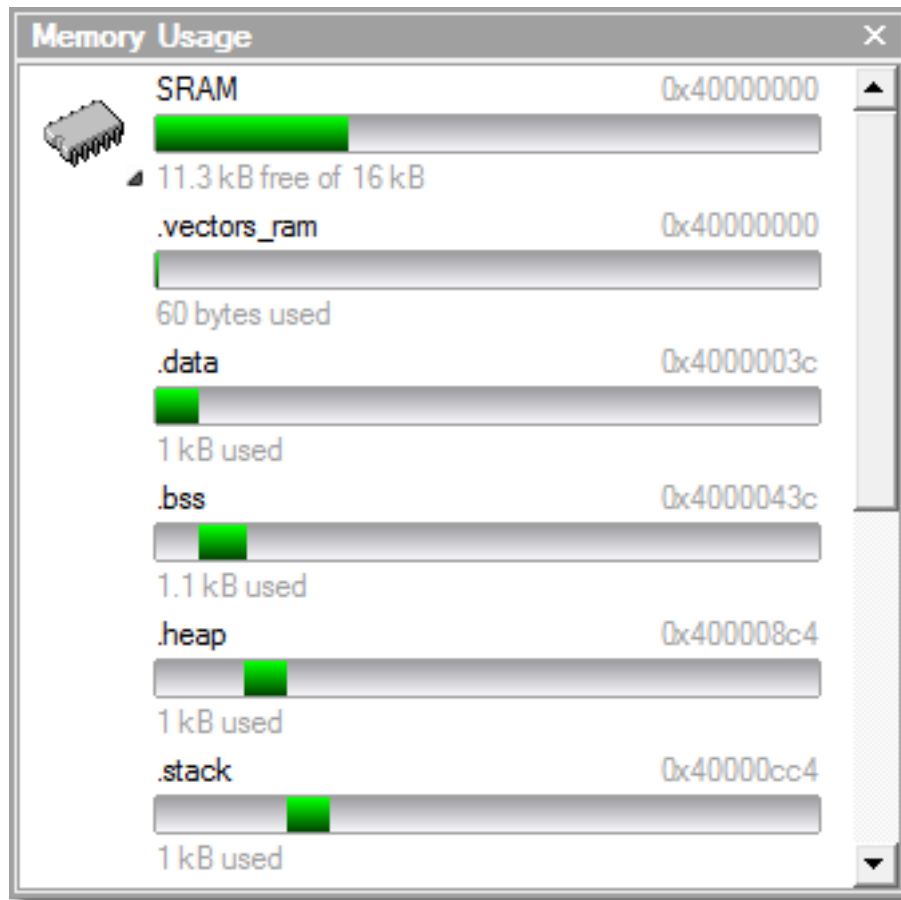
—or—

- Type **Ctrl+Alt+Z**.

The memory usage graph will only be visible if your current 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

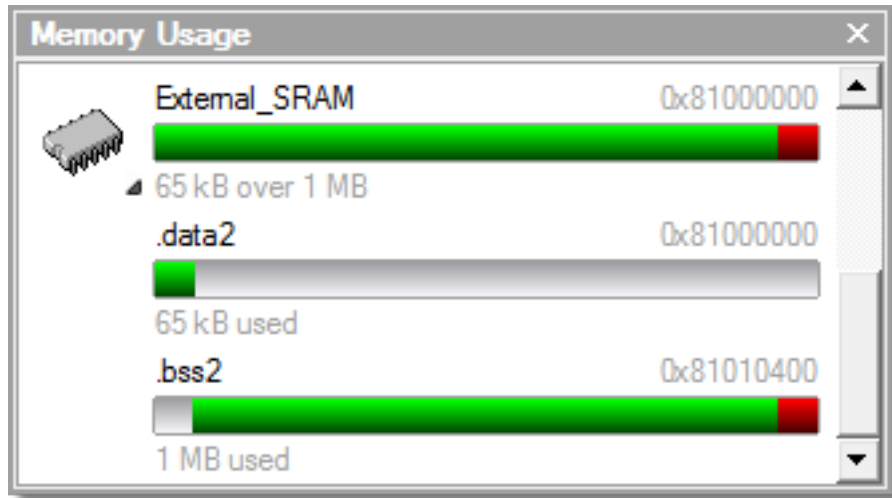
The **Memory Usage Window** can also be used to display graphically how program sections have been placed in memory. To display the program sections simply click on the memory segment to expand it or alternatively right click and select **Show Memory Sections** from the context menu.



Each bar represents an entire memory segment and the green coloured area represents the area of the segment that contains the program section.

Displaying Segment Overflow

The **Memory Usage Window** also displays segment overflows, i.e. when the total size of the program sections placed in a segment is larger than the segment size. When this happens the segment and section bars represents the total memory used, areas coloured green represent the code or data that has been placed within the segment and areas coloured red represent code or data that has been placed outside of 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, you should use the [Symbols Window](#).

Output window

The **Output** window contains logs and transcripts from various systems within CrossStudio. Most notably, it contains the [Build Log](#), [Find Log](#), [Source Navigator Log](#) and [Target Log](#).

Build Log

The build log contain the results of the last build, it is cleared automatically on each build. Errors detected by CrossStudio are shown in red, and warnings are shown yellow. Double-clicking an error or warning in the build log will open the offending file at the error position.

The command lines used to do the build can be echoed to the build log by setting the **Building > Echo Build Command Lines** environment option.

Find Log

The find log contains the results from a **Find in Files** operation, it is cleared automatically on each new search.

Each file containing a match is listed along with the matching line. Double-clicking a match will make the editor load the file and locate to the matching line.

Source Navigator Log

The Source Navigator log displays a list of the files that the source navigator has parsed and the time it has taken for each file to parse.

Target Log

The Target Log displays a trace of high-level loading and debug operations carried out on the target. In the case of downloading, uploading and verification it also displays the time it took to carry out each operation. The log is automatically cleared for each new download or debug session.

Output window user interface

The **Output** window is divided into a tool bar and the log display.

Output tool bar

Button	Description
	Tree view Shows the log as a tree view.
	Flat view Shows the log as a flat view.

Showing the Output window

To display the **Output** window if it is hidden, do one of the following:

- Click **View > Output** .

—or—

- Type **Ctrl+Alt+O**.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Output**.

Using the output window

Showing a specific log

To display a specific log, do one of the following:

- On the **Output** window tool bar, click the **Output Pane List**.
- From the list, click the log to display.

—or—

- Click **View > Logs** and select the log to display.

—or—

- Right click the tool bar area to display the **View** menu
- From the popup menu, click **Logs** and then the log to display.

Showing the Build Log

To display the build log in the output window, do one of the following:

- From the **Build** menu, click **Show Build Log**.

—or—

- Double click the **Target Status** panel in the status bar.

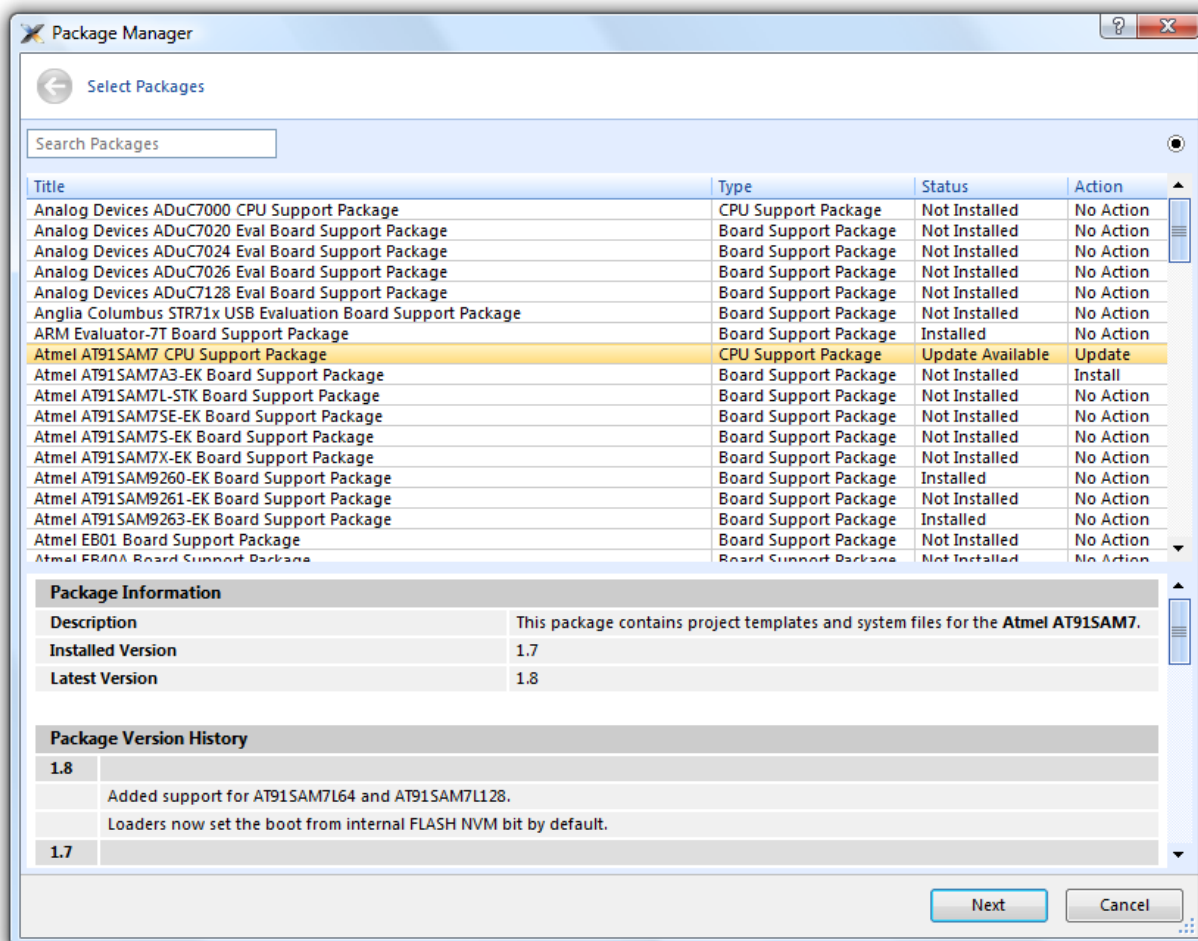
Showing the Target Log

To display the target log in the output window, do the following:

- Click **Target > Show Target Log**.

Package manager window

The **Package Manager Window** is used to manage the support packages installed on your system. It displays a list of the packages available, it also shows the packages you have installed and allows you to install, update, reinstall or remove them.



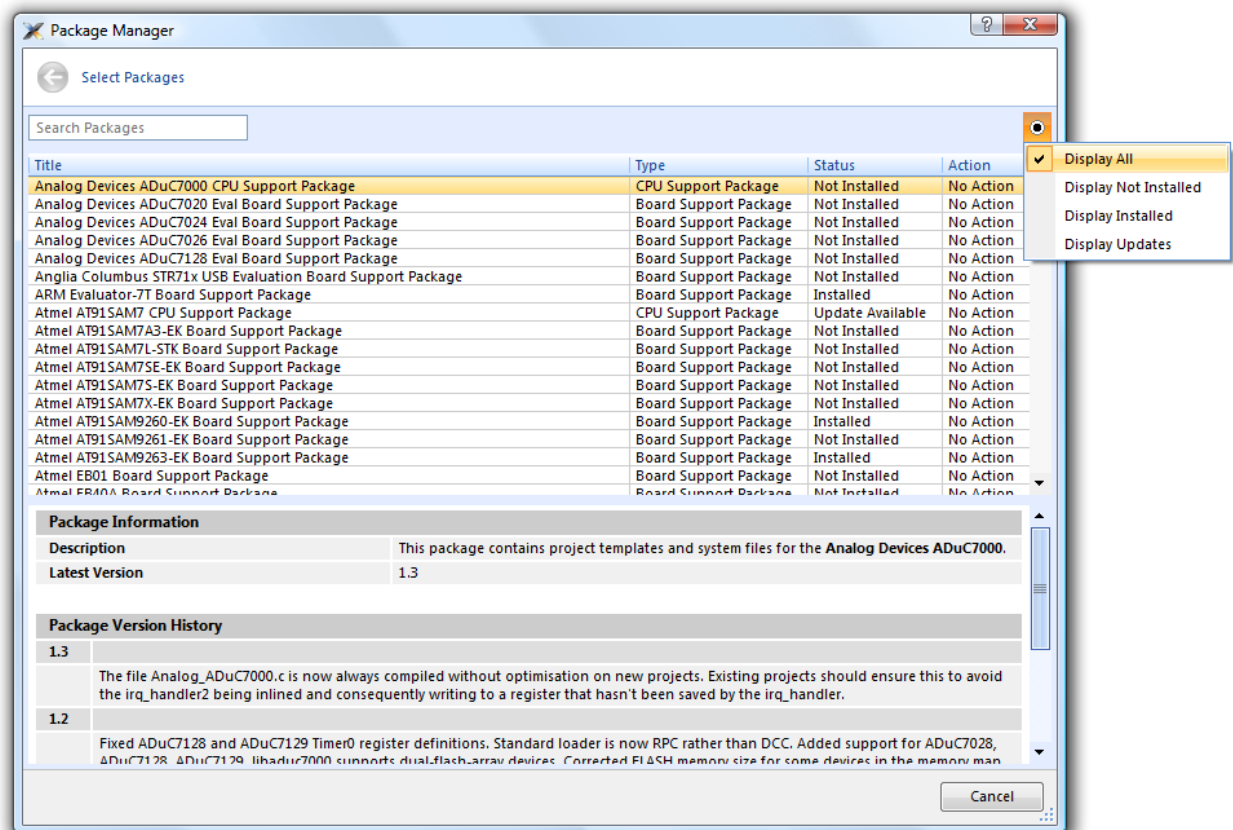
To display the **Package Manager Window**:

- Click **Tools > Package Manager**.

Filtering the package list

By default, the package manager displays a list of all available and installed packages. You can filter the packages that are displayed in a number of ways.

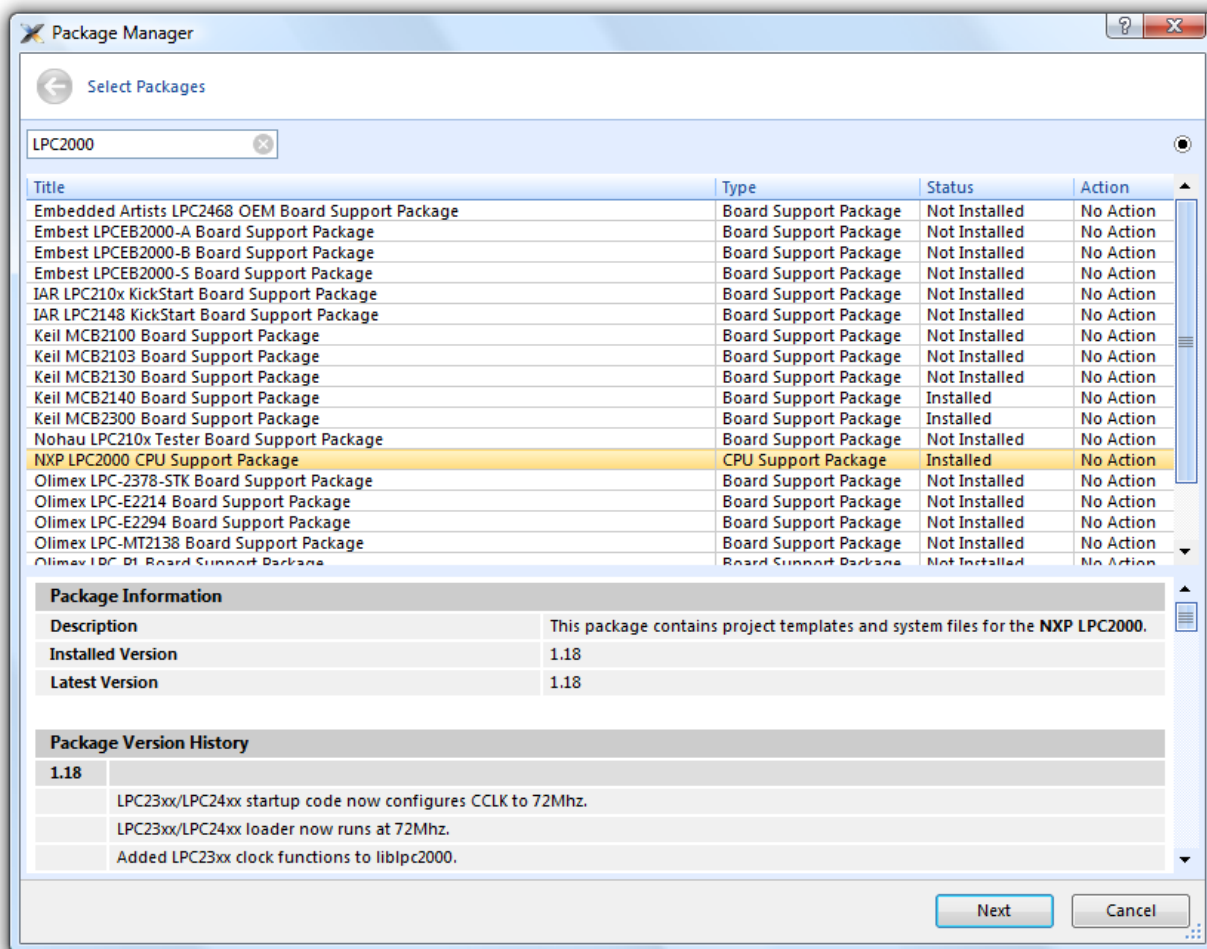
Firstly, you can filter by package status. To do this click on the **Options** button at the top right hand corner of the dialog.



From the options menu you can select the following display options:

- **Display All** - Display all packages irrespective of their status.
- **Display Not Installed** - Only display packages that have not been installed.
- **Display Installed** - Only display packages that have been installed.
- **Display Updates** - Only display packages that have been installed and that have updated versions available.

You can also filter the packages displayed by the text contained within the package title and documentation. To do this simply type the text into the **Search Packages** box at the top left hand 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 then unpacks the files contained within the package to their destination directory.

To install a package:

1. Click **Tools > Install Packages** (note that this is equivalent to clicking **Tools > Package Manager** and setting the status filter to **Display Not Installed**).
2. Select the package(s) you wish to install.
3. Right click and select **Install Selected Packages**.
4. Click **Next**, you will be presented with a list of actions the package manager is going to carry out.
5. Click **Next** and the package manager will install the package(s).
6. When the installation is complete click **Finish** to close the package manager.

Once installed, click **Tools > Show Installed Packages** to see more information on the installed packages.

Updating a package

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

To update a package:

1. Click **Tools > Update Packages** (note that this is equivalent to clicking **Tools > Package Manager** and setting the status filter to **Display Updates**).
2. Select the package(s) you wish to update.
3. Right click and select **Update Selected Packages**.
4. Click **Next**, you will be presented with a list of actions the package manager is going 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. Click **Tools > Remove Packages** (note that this is equivalent to clicking **Tools > Package Manager** and setting the status filter to **Display Installed**).
2. Select the package(s) you wish to remove.
3. Right click and select **Remove Selected Packages**.
4. Click **Next**, you will be presented with a list of actions the package manager is going 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. Click **Tools > Reinstall Packages** (note that this is equivalent to clicking **Tools > Package Manager** and setting the status filter to **Display Installed**).
2. Select the package(s) you wish to reinstall.
3. Right click and select **Reinstall Selected Packages**.
4. Click **Next**, you will be presented with a list of actions the package manager is going to carry out.
5. Click **Next** and the package manager will reinstall the package(s).
6. When the operation is complete click **Finish** to close the package manager.

Project explorer

The **Project Explorer** is the user interface on to the CrossWorks **Project System**. The project explorer organizes your projects and files and provides access to the commands that operate on them. A tool bar at the top of the window offers quick access to commonly used commands for the selected project node or the active project. A right click menu offers a larger set of commands and will work on the selected project node, ignoring the active project.

The selected project node affects the operations that 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 then each of the files in the folder are compiled.

You can select project nodes by clicking on 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 that you're currently editing.

Showing the Project Explorer

To activate the **Project Explorer** if it is hidden, do one of the following:

- From the **View** menu, click **Project Explorer**.

—or—

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

—or—

- On the **Standard** tool bar, click the **Project Explorer** icon.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Project Explorer**.

Left click operations

The following operations are available in the project explorer with the left click of the mouse:

Action	Description
Single Click	Select the project node. If the project node is already selected and is a solution, project or folder node then an rename editor is shown.
Double Click	Double click on a solution node or folder node will open/close the node. Double click on a project node and it is set as the active project. Double click on a file and it will be opened using the default editor for the file type.

Tool bar operations

The following operations are available on the tool bar:

Button	Description
	Adds a new file to the active project using the New File dialog.
	Adds existing files to the active project.
	Removes files, folders, projects, and links from the project.
	Creates a new folder in the active project.
	Drop down menu that provides a number of build operations.
	Disassembles the active project.
	Drop down menu that sets a variety of project explorer options
	Displays the properties dialog for the selected item.

Right click menu operations

The following operations are available on the right click menu:

Solution node menu entries	
(Batch) Build	Build all projects under the solution in the current/batch build configuration.
(Batch) Rebuild	Rebuild all projects under the solution in the current/batch build configuration.
(Batch) Clean	Remove all output and intermediate build files for the projects under the solution in the current/batch build configuration.
(Batch) Export Build	Create an editor with the build commands for the projects under the solution in the current/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.
Add To Favourites	Add the project file to the favourites window.
Paste	Paste a copied project into the solution.
Remove	Remove the link to another solution from the solution.

Rename	Rename the solution node.
Source Control Operations	Source control operations on the project file and recursive operations on all files in the solution.
Edit Solution As Text	Create an editor containing the project file.
Save Solution As	Change the filename of the project file - note that the saved project file is not reloaded.
Properties	Show the properties dialog with the solution node selected.
Project node menu entries	
(Batch) Build	Build the project in the current/batch build configuration.
(Batch) Rebuild	Build the project in the current/batch build configuration.
(Batch) Clean	Remove all output and intermediate build files for the project in the current/batch build configuration.
(Batch) Export Build	Create an editor with the build commands for the project in the current/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 the memory map file into the project.
Section Placement Commands	For Executable project types that don't have section placement files in the project and have the section placement file project property set there are commands to view the section placement file and to import the section placement file 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 properties dialog with the project node selected.
Folder node menu entries	
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/folder.
Copy	Copy the folder from the project/folder.
Paste	Paste a copied folder or file into the folder.
Remove	Remove the folder from the project/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.
File node menu entries	
Open	Edit the file with the default editor for the file type.
Open With	Edit the file with a selected editor, you can choose from the Binary Editor , Text Editor , and Web Browser .
Open in Windows Explorer	Create a windows explorer and locate to the file.
Compile	Compile the file.
Export Build	Create an editor with the build commands for the compile in the current build configuration.
Exclude From Build	Set the Exclude From Build property to be Yes for this project node in the current 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/folder.
Copy	Copy the file from the project/folder.
Remove	Remove the file from the project/folder.
Import	Import the file into the project.
Add To Favourites	Add the file to the favourites window.
Source Control Operations	Source control operations on the file.
Properties	Show the properties dialog with the file node selected.

Properties window

The properties window displays properties of the current focused CrossStudio object. Using the properties window you can set build properties of your project, modify the editor defaults and change target settings.

The properties window is organised as a set of key value pairs. As you select one of the keys then a help display explains the purpose of the property. Because the properties are numerous and can be specific to a particular product build you should consider this help to be the definitive help on the property.

You can organise the property display so that it is divided into categories or alternatively display it as a flat list that is sorted alphabetically.

The combo-box enables you to change the properties yourself and explains which properties you are looking at.

Some properties have actions associated with them - you can find these by right clicking on the property key. Most properties that represent filenames can be opened this way.

When the properties window is displaying project properties you'll find that some properties are displayed in **bold**. This means that the property value hasn't been inherited. If you wish to inherit rather than define such a property then on the right click context menu you'll find an action that enables you to inherit the property.

Register windows

The register windows can 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.

Register window user interface

The **Registers** window is divided into a tool bar and the main data display.

Register tool bar

Button	Description
	Displays the CPU, special function register, and peripheral register groups.
	Displays the selected item in binary.
	Displays the selected item in octal.
	Displays the selected item in decimal.
	Displays the selected item in hexadecimal.
	Displays the selected item as a signed decimal.
	Displays the selected item as a character or Unicode character.
	Force reading of a register ignoring the access property of the register.
	Update the selected register group.
	Sets the active memory window to the address and size of the selected register group.

Using the registers window

Both CPU registers and special function registers are shown in the main part of the Registers window. When the program stops at a breakpoint or is stepped, the Register windows automatically update to show the current values of the registers. Items that have changed since they were previously displayed are highlighted in red.

Showing the Registers window

To display register window *n* if it is hidden, do one of the following:

- From the **View** menu, click **Other Windows** then **Registers *n***.

—or—

- From the **Debug** menu, click **Debug Windows** then **Registers *n***.

—or—

- Type **Ctrl+T, R, *n***.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Other Windows** then **Registers *n***.

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 selected using the Call Stack window.
- The register state of the currently selected thread using the the Threads window.
- The register state that you have supplied using the **Debug > Locate** operation.

To display a group of CPU registers, do the following:

- On the Registers window tool bar, click the Groups button —



- From the dropdown menu, check the register groups to display and uncheck the ones to hide.

You can uncheck all CPU register groups to allow more space in the display for special function 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 that have been defined in the memory map file that 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, do the following:

- On the Registers window tool bar, click the Groups button —



- From the dropdown menu, check the register groups to display and uncheck 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 tool bar changes to show the item's display format.

To change the display format of a register, do one of the following:

- Right click the item to change.
- From the popup menu, select the format to display the item in.

—or—

- Click the item to change.
- On the Registers window tool bar, select the format to display the item in.

Modifying register values

To modify the value of a register, do one of the following:

- 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 popup menu, select one of the operations to modify the register value.

Modifying the saved register value of a function or thread may not be supported.

Script Console

The **Script Console** window allows you interactive access to the **JavaScript** interpreter and JavaScript classes that are built-in to 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.

load(filepath) loads and executes the JavaScript contained in <i>filepath</i> and returns a boolean success.

Source navigator window

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

Source navigator user interface

The **Source Navigator** window is divided into a tool bar and the main breakpoint display.

Source Navigator tool bar

Button	Description
	Sorts the objects alphabetically.
	Sorts the objects by type.
	Sorts the objects by access (public, protected, private).
	Groups objects by type (functions, classes, structures, variables).
	Move the cursor to the statement where the object is defined.
	Move the cursor to the statement where the object is declared. If more than one declaration exists, an arbitrary one is chosen.
	Manually re-parses any changed files in the project.

Source navigator display

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

CrossStudio displays these icons to the left of each object:

Icon	Description
	Structure or namespace A C or C++ structure or a C++ namespace.
	C++ class A C++ class.
	Private function A C++ member function that is declared private or a function that is declared with static linkage.
	Protected function A C++ member function that is declared protected .

	Public function A C++ member function that is declared public or a function that is declared with extern linkage.
	Private variable A C++ member variable that is declared private or a variable declared with static linkage.
	Protected variable A C++ member variable that is declared protected .
	Public variable A C++ member variable that is declared public or a variable that is declared with extern linkage.

Showing the Source Navigator window

To display the **Source Navigator** window if it is hidden, do one of the following:

- From the **View** menu, click **Source Navigator**.

—or—

- Type **Ctrl+Alt+N**.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Source Navigator**.

Using the source navigator

Parsing source files manually

To parse source files manually, do one of the following:

- From the **Tools** menu, click **Source Navigator** then **Refresh**.

—or—

- On the **Source Navigator** tool bar, click **Refresh**.

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

Grouping objects by type

You can group object 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 in the source browser by type, do one of the following:

- From the **Tools** menu, click **Source Navigator** then **Group By Type**.

—or—

- On the **Source Navigator** tool bar, click the arrow to the right of the **Cycle Grouping** button.
- From the dropdown menu, click **Group By Type**.

Symbol browser

The Symbol Browser window 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 as a whole. You can use the symbol browser to **drill down** to see how big each part of your program is and where it's placed. The way that symbols are sorted and grouped is saved between runs. When you rebuild an application, CrossStudio automatically updates the symbol browser so you can see the effect your changes are making to the memory layout of your program.

Symbol browser user interface

The symbol browser is divided into a tool bar and the main symbol display.

Symbol Browser tool bar

Button	Description
	Groups symbols by source file name.
	Groups symbols by symbol type (equates, functions, labels, sections, and variables)
	Groups symbols by the section that they are defined in.
	Moves the cursor to the statement that defined the symbol.
	Chooses the columns to display in the symbol browser.

Symbol Browser display

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

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

	Public Label A public data symbol, defined relative to a section.
	Section A program section.

Symbol browser columns

You can choose to display the following fields against 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, that the code or data item covers. 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, then **Size** is calculated as the difference between the start and end address of the range. If a symbol has no range, the size column is left blank.

Section

The section in which the symbol is defined. If the symbol is not defined within a section, the **Section** column is left 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 left blank.

Showing the Symbol Browser window

To display the **Symbol Browser** window if it is hidden, do one of the following:

- From the **View** menu, click **Symbol Browser**.

—or—

- Type **Ctrl+Alt+Y**.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Symbol Browser**.

Configuring the Symbol Browser

Choosing fields to display

Initially the **Range** and **Size** columns are shown in the symbol browser. You can select which columns to display using the **Field Chooser** on the **Symbol Browser** tool bar.

To select the fields to display in the Symbol Browser, do one of the following:

- Click the **Field Chooser** button on the **Symbol Browser** tool bar.
- Check the fields that you wish to display and uncheck the fields that you wish to hide.

—or—

- From the **Tools** menu, select **Symbol Browser** then **Fields**.
- Check the fields that you wish to display and uncheck the fields that you wish to hide.

Grouping symbols by section

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

To group symbols by section, do the following:

- On the **Symbol Browser** tool bar, click the arrow next to the **Cycle Grouping** tool button.
- From the popup menu, click **Group By Section**.

—or—

- From the **Tools** menu, click **Symbol Browser** then **Group By Section**.

The **Cycle Grouping** tool button icon will change to indicate that the symbol browser is now grouping symbols by section.

Grouping symbols by type

When you group symbols by type, each symbol is grouped underneath the type of symbol that it is. 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 a symbol that is defined by a high-level code sequence.
- A **Variable** is defined by a high-level data declaration.
- A **Label** is a symbol that is defined by an assembly language module. **Label** is also used when high-level modules are compiled with debugging information turned off.

To group symbols by source type, do the following:

- On the **Symbol Browser** tool bar, click the arrow next to the **Cycle Grouping** tool button.
- From the popup menu, click **Group By Type**.

—or—

- From the **Tools** menu, click **Symbol Browser** then **Group By Type**.

The **Cycle Grouping** tool button icon will change to indicate that the symbol browser is now grouping symbols by type.

Grouping symbols by source file

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

To group symbols by source file, do one of the following:

- On the **Symbol Browser** tool bar, click the arrow next to the **Cycle Grouping** tool button.
- From the popup menu, click **Group By Source File**.

—or—

- From the **Tools** menu, click **Symbol Browser** then **Group By Source File**.

The **Cycle Grouping** tool button icon will change to indicate that the symbol browser is now grouping symbols by source file.

Sorting symbols alphabetically

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

To group symbols alphabetically, do one of the following:

- On the **Symbol Browser** tool bar, click the arrow next to the **Cycle Grouping** tool button.
- From the popup menu, click **Sort Alphabetically**.

—or—

- From the **Tools** menu, click **Symbol Browser** then **Sort Alphabetically**.

The **Cycle Grouping** tool button icon will change to indicate that the symbol browser is now grouping symbols alphabetically.

Filtering, finding, and watching symbols

When you’re dealing with big projects with hundreds, or even thousands, of symbols, a way to filter the display of those symbols and drill down to the ones you need is very useful. The symbol browser provides an editable combo box in the toolbar which you can use to specify the symbols you’d like displayed. The symbol browser uses “*” 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. One thing to note is that 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.

Finding symbols with a common prefix

To display symbols that start with a common prefix, do the following:

- Type the required prefix 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 as it is implied.

Finding symbols with a common suffix

To display symbols that end with a common suffix, do the following:

- 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.

Jumping to the definition of a symbol

Once 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 the definition of a symbol using the **Go To Definition** tool button.

To go to the definition of a symbol, do one of the following:

- Select the symbol from the list of symbols.
- On the **Symbol Browser** tool bar, click **Go To Definition**.

—or—

- Right click the symbol in the list of symbols.
- From the popup menu, click **Go To Definition**.

Adding symbol to watch and memory windows

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, do the following:

- In the **Symbol Browser**, right click on the the symbol you wish to add to the watch window.
- From the popup menu, click **Add To Watch**.

To add a symbol to the memory window, do the following:

- In the **Symbol Browser**, right click on the the symbol you wish to add to the memory window.
- From the popup menu, click **Locate Memory**.

Working with the Symbol Browser

Here are a few common ways to use the symbol browser:

What function takes up the most code space or what takes the most data space?

- Show the symbol browser by selecting **Symbol Browser** from the **Tools** menu.
- Group symbols by type by choosing **Symbol Browser > Group By Type** from the **Tools** menu.
- Make sure that the **Size** field is checked in **Symbol Browser > Fields** on the **Tools** menu.
- Ensure that the filter on the symbol browser tool bar is empty.
- Click on the **Size** field in the header to sort by data size.
- Read off the sizes of variables under the **Variable** group and functions under the **Functions** group.

What's the overall size of my application?

- Show the symbol browser by selecting **Symbol Browser** from the **Tools** menu.
- Group symbols by section by choosing **Symbol Browser > Group By Section** from the **Tools** menu.
- Make sure that the **Range** and **Size** fields are checked in **Symbol Browser > Fields** on the **Tools** menu.
- Read off the section sizes and ranges of each section in the application.

Targets window

The targets window (and associated menu) displays the set of target interfaces that 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.

You can connect, disconnect, and reconnect to a target system. You can also reset and load programs using the target window. If you load a program using the target window and you need to debug it then you will have to use the **Debug > Attach Debugger** operation.

Targets window layout

Targets tool bar

Button	Description
	Connects the selected target interface.
	Disconnects the connected target interface.
	Reconnects the connected target interface.
	Resets the connected target interface.
	Displays the properties of the selected target interface.

Showing the Targets window

To display the **Targets** window if it is hidden, do one of the following:

- From the **View** or **Target** menu, click **Targets**.

—or—

- Type **Ctrl+Alt+T**.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Targets**.

Managing target connections

Connecting a target

To connect a target, do one of the following:

- In the **Targets** window, double click the target to connect.

—or—

- From the **Target** menu, click the target to connect.

—or—

- In the **Targets** window, click the target to connect.
- On the **Targets** window tool bar, click the **Connect** button

—or—

- In the **Targets** window, right click the target to connect.
- From the popup menu, click **Connect**

—or—

- In the **Targets** window, click the target to connect.
- Type **Ctrl+T, C**.

Disconnecting a target

To disconnect a target, do one of the following:

- From the **Target** menu, click **Disconnect**

—or—

- On the **Targets** window tool bar, click the **Disconnect** button

—or—

- Type **Ctrl+T, D**.

—or—

- Right click the connected target in the **Targets** window
- From the popup menu, click **Disconnect**.

Alternatively, connecting a different target will automatically disconnect the current target connection.

Reconnecting a target

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 as it forces CrossStudio to resynchronize with the target.

To reconnect a target, do one of the following:

- From the **Target** menu, click **Reconnect**.

—or—

- On the **Targets** window tool bar, click the **Reconnect** button.

—or—

- Type **Ctrl+T, E**.

—or—

- In the **Targets** window, right click the target to reconnect.
- From the popup menu, click **Reconnect**.

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, do the following:

- From the **View** menu, click **Properties Window**.
- In the **Properties Window**, click **Environment Properties** from the combo box.
- In the **Target Settings** section, set the **Enable Auto Connect** property to **Yes** to enable automatic connection or to **No** to disable automatic connection.

Creating a new target interface

To create a new target interface, do the following:

- From the targets window's context menu, click **New Target Interface**. A new menu will be displayed containing the types of target interface that may be created.
- 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 on the target, 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, do the following:

- Select a target.
- Select the **Properties** option from the target's context menu.

Restoring default target definitions

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 and therefore should be used with care.

To restore the default target definitions, do the following:

- Select **Restore Default Targets** from the targets window context menu.
- Click **Yes** when prompted if you want to restore the default targets.

Importing and exporting target definitions

You can import and export your target interface definitions, this maybe useful if you make a change to the default set of target definitions and want to share them with another user or use them on another machine.

To export the current set of target interface definitions:

- Select **Export Target Definitions To XML** from the targets window context menu.
- Select the location and file name of the file you want to save the target definitions to and click **Save**.

To import an existing set of target interface definitions:

- Select **Import Target Definitions From XML** from the targets window context menu.
- Select the file you want to load the target definitions from and click **Open**.

Controlling target connections

Resetting the target

Reset of the target is typically handled automatically by the system when you start debugging. However, the target may be manually reset using the **Targets** window.

To reset the connected target, do one of the following:

- On the **Targets** window tool bar, click the **Reset** button.

—or—

- From the **Target** menu, click **Reset**

—or—

- Type **Ctrl+T, S**.

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, do the following:

- In the **Targets** window, right click the selected target.

- From the popup menu, click **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 a arbitrary programs held on disk using the **Targets** window.

To verify a target's contents against a program, do the following:

- In the **Targets** window, right click the selected target.
- From the popup menu, click **Verify File**.
- From the **Verify File** menu, select the type of file to verify.
- 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 it does for downloading, described above.

Erasing target memory

Usually, erasing target memory is done automatically CrossStudio downloads a program, but you can erase a target's memory manually.

To completely erase target memory, do the following:

- In the **Targets** window, right click the target to erase.
- From the popup menu, click **Erase All**.

To erase part of target memory, do the following:

- In the **Targets** window, right click the target to erase.
- From the popup menu, click **Erase Range**.

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 open the terminal emulator window

- From **View** menu, click **Terminal Emulator**.

—or—

- Type **Ctrl+Alt+M**.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Terminal Emulator**.

To use the terminal emulator window

- Set the required **terminal emulator properties** by clicking the



icon on the tool bar or selecting **Properties** from the right click context menu.

- Connect the terminal emulator to the communications port by clicking the



icon on the tool bar or selecting **Connect** from the right click context menu.

Once connected any input in the terminal emulator window will be transmitted to the communications port, any data received from the communications port will be displayed on the terminal.

Connection may be refused if the communication port is in use by another application or if the port doesn't exist.

To disconnect the terminal emulator window

- Disconnect the communications port by clicking the



icon on the tool bar or selecting **Disconnect** from the right click context menu, this will release the communications port so it can be used by other applications.

Tool bar operations

The following operations are available on the tool bar:

Button	Description
	Connects the terminal emulator to the communications port.
	Disconnects the terminal emulator from the communications port.
	Displays the terminal emulator properties.

Right click menu operations

The following operations are available on the right click menu:

Action	Description
Connect	Connects the terminal emulator to the communications port.
Disconnect	Disconnects the terminal emulator from the communications port.
Copy	Copies selected text to the clipboard.
Paste	Paste text from the clipboard to the communications port.
Clear	Clear the terminal.
Send Control	Send a control code to the communications port
Send File	Send a file down the communications port.
Properties	Displays the terminal emulator properties.

Terminal emulator properties

The following properties can be set for the terminal emulator:

Property	Description
Backscroll Buffer Lines	The number of lines you can see when you scroll backward.
Baud Rate	The baud rate used when transmitting and receiving data.
Data Bits	The number of data bits to use when transmitting and receiving data.
Local Echo	If set to Yes , displays every character before sending it to the remote computer.
Parity	The parity used when transmitting and receiving data.

Port	The communications port to use. On Windows this refers to the communication port name, e.g. COM1, COM2, etc. On Linux this refers to the path to the serial port device driver, e.g. /dev/ttyS0, /dev/ttyS1, etc.
Stop Bits	The number of stop bits to use when transmitting data.

Supported control codes

The terminal supports a very limited set of control codes. The supported control codes are as follows:

Control code	Description
<BS>	Backspace.
<LF>	Line feed.
<ESC>[{attr1};...;{attrn}m	Set display attributes. Note that the attributes 2-Dim, 5-Blink, 7-Reverse and 8-Hidden are not supported.

Threads window

The threads window displays the set of executing contexts on the target processor structured as a set of queues. The threads window is populated using the threads script which is a JavaScript program store in a file that has a file type property of "Threads Script" (or is called threads.js) and is in the project that is being debugged.

On start debugging the threads script is loaded and the **function init()** is called that will determine the columns that are displayed in the threads window.

When the application stops on a breakpoint the **function update()** is called which creates 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 on one of the entries in the threads window the debugger is located to it's saved execution context - you can put the debugger back into the default execution context using **Show Next Statement**.

Writing the threads script

The threads script controls the threads window using the **Threads** object.

The methods **Threads.setColumns** and **Threads.setSortByNumber** can be called from the **function init()**.

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

The above example creates the named columns **Name**, **Priority**, **State** and **Time** in the threads window and makes the **Time** column be sorted numerically rather than alphabetically.

If you don't supply the **function init()** in the threads script then the threads window will create 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 that are added to this window will go under this. If you don't call this then the 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 the entries in the order 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 then each field can be accessed using the 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 } };
```

Then 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 then the debugger, and consequently CrossStudio, will become unresponsive and you will need to kill CrossStudio using a task manager. so 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. The **getregs** function should return the array of registers for example.

```

function getregs(x)
{
    return Debug.evaluate("((struct task*")+x+")->registers");
}

```

If you use thread local storage then by implementing the **gettls(x)** function you can return an expression for the debugger to evaluate when the base address of the thread local storage is accessed for example.

```

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

```

See [threads.js](#) for the threads script used with CTL.

Trace window

The trace window displays historical information on the instructions executed by the target. 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 right click context 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 on 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.

Watch window

The watch window provides a means to evaluate expressions and display the values of those expressions. Typically expressions are just the name of the variable to be displayed, but can be considerably more complex see [debugger expressions](#). Note that the 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.

Watch window user interface

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

Watch tool bar

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

Right clicking on a watch item shows a context menu that has some actions that are not available on the toolbar.

Watch menu

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 watch value by one.
	Decrement watch value by one.
	Negate the watch value.
	Invert the watch value.
	View the properties of the watch value.

You can view details of the watch item using the properties window.

Watch Properties

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) then you can open the structure see its contents.

The display is updated each time the debugger locates to source code. So it will update each time your program stops on a breakpoint or single step and whenever you traverse the call stack. Items that have changed since they that were previously displayed are highlighted in red.

Showing the Watch window

To display watch window *n* if it is hidden, do one of the following:

- From the **View** menu, click **Other Windows** then **Watch *n***.

—or—

- From the **Debug** menu, click **Debug Windows** then **Watch *n***.

—or—

- Type **Ctrl+T, W, *n***.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Other Windows** then **Watch *n***.

Adding or changing items to watch

You can add a new expression to be watched by clicking and typing into the last entry in the watch window. You can change the expression that is watched by clicking the expression entry and editing its contents.

Changing display format

When you select a variable in the main part of the display, the display format button highlighted on the Watch window tool bar changes to show the item's display format.

To change the display format of a local variable, do one of the following:

- Right click the item to change.
- From the popup menu, select the format to display the item in.

—or—

- Click the item to change.
- On the Watch window tool bar, select the format to display the item in.

The selected display format will then be used for all subsequent displays and will be recorded when the debug session stops.

For C programs the interpretation of pointer types can be changed by right clicking and selecting from the popup menu. A pointer can be interpreted as:

- a null terminated ASCII string.
- an array.
- an integer.
- dereferenced.

Modifying watched values

To modify the value of a local variable, do one of the following:

- 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 popup menu, select one of the operations to modify the variable's value

MSP430 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 on the targets window, see [Targets Window](#).

Before you can use a target interface, you must **connect** to it. You can only connect to a single target interface at any one time.

All target interfaces have a set of properties. The properties provide information on the connected target and allow you to configure the target interface. The following sections describe the general properties that are common to all target interfaces and the properties specific to each target interface type.

In this section

MSP430 Debug Capabilities

Details the debug capabilities that CrossWorks for MSP430 makes available when connected to MSP430 targets.

CrossConnect for MSP430

Describes the Rowley CrossConnect for MSP430 target interface. This target interface is applicable to CrossConnect for ARM when used with a suitable ARM 20 pin to MSP430 14 pin JTAG convertor. The convertor should map the ARM NTRST signal to the MSP430 TEST signal and the ARM NSRST, TDI, TMS, TCK, TDO, VCC and GND signals map to the equivalent MSP430 signals.

Flash emulation tool

Describes the TI Flash Emulation Tool (MSP-FET430PIF) interface.

TI MSP430 DLL Interface

Describes the MSP430.dll Interface which is used to connect to a variety of devices including the TI MSP-FET430UIF and Olimex MSP430-JTAG-TINY.

MSP430 core simulator

Describes the core simulator provided by CrossStudio which you can use to develop software without access to hardware.

MSP430 Target Debug

The debug capabilities that CrossWorks for MSP430 provides are implemented using the MSP430 Enhanced Emulation Module (EEM). At the time of writing [slaa393](#) is the only publically available document that details the capabilities of the EEM. You should refer to this document if you want to know about the debug capabilities of a particular device.

Software Breakpoints

The Connection section of the target properties window has the **Implement Software Breakpoints** property. If this property is set to Yes then the opcode 0x4343 (mov.b r3, r3) is treated as a software breakpoint instruction, and as such when you set a code breakpoint this opcode is written to the address you want execution to break at. Using this feature provides an unlimited number of code breakpoints but will use one hardware breakpoint (EEM combination trigger) to implement the software breakpoint.

EEM Triggers

CrossWorks for MSP430 implements [breakpoint expressions](#) using EEM triggers. Breakpoint expressions provide a simple interface for setting complex breakpoints. Additionally using the breakpoint dialog(s) and breakpoint property window with CrossWorks for MSP430 you can specify:

- the EEM trigger type using the **Breakpoint Trigger Type**.
- the EEM reaction (Stop, Trace or Sequencer) using the **Action**.
- the EEM combination trigger number using the **Use Hardware Breakpoint**. You should only need to use this if you are using the Trigger Sequencer or the MSP430F5xx variable watch capability.

EEM State Storage

The **Debug | Control | State Storage Control** dialog allows you to configure the state storage block of the EEM and the **Debug | Debug Windows | EEM State Storage** window allows you to see the contents of the EEM state storage block. You can specify that breakpoints are to be used for state storage control and or stored into the storage block by setting their breakpoint property **Action** to be **Trace**. To implement a real-time trace on a variable you can set a data breakpoint on a variable with the breakpoint property **Action** set to **Trace** and set the state storage control to **Store Mode: Triggers**. You can refresh the **EEM State Storage** window without stopping the processor by right clicking in the window and selecting **Refresh**.

Variable Watch (MSP430F5xx only)

On MSP430F5xx devices you can configure the state storage block to store variable values to fixed EEM storage locations and do instruction tracing with the remaining storage locations. For example you can set the state storage locations 0 and 1 to contain the data watch values corresponding to the EEM combination triggers 0 and 1. If you want to do this then you need to assign the appropriate EEM trigger combination number to the data breakpoint. Note that the EEM combination trigger number 0 is used to implement software breakpoints so if you want to use this feature then you should disable software breakpoints.

EEM Trigger Sequencer

The EEM trigger sequencer enables EEM combination triggers 4-7 to be used as inputs to the trigger sequencer and optionally the EEM combination trigger 3 to be used to reset the trigger sequencer. The breakpoints you want as inputs to the sequencer should have the EEM combination trigger numbers 4-7 assigned to them and should have the breakpoint property **Action** set to **Sequencer**. You can set up the trigger sequencer using the **Debug | Control | Sequencer Control** dialog and you can use the **Debug | Debug Windows | EEM Trigger Sequencer** window to see the current state of the trigger sequencer. The trigger sequencer starts in **State0** and executes the specified action when it enters **State3**. On each state there can be two transitions (**A** and **B**) to any of the other states.

Clock Control

When you are connected to a target you can use the Clock Control dialog from the **Debug | Control | Clock Control** menu to specify the clock behaviour when the CPU stops on a breakpoint. The desired clock control settings are programmed when you start debugging they cannot be changed when you are debugging. The settings that are reported by the Clock Control dialog are specific to a particular device.

[slaa393](#) examples.

Break on Write to Address

Using the **New Data Breakpoint** dialog the breakpoint expression **wLoopCounter==50** will break when **wLoopCounter** is written with the value **50**.

Break on Write to Register

Using the **New Data Breakpoint** dialog the breakpoint expression **@sp<=0x09A0** will break when the stackpointer is written with a value less than **0x09A0**.

Break on Write to Flash

Using the **New Data Breakpoint** dialog the breakpoint expression **(char[0xF000])0x1000** and set the "Breakpoint Trigger Type" to "Write" will break when a write is made to flash memory.

Break on Access of Invalid Memory

Using the **New Data Breakpoint** dialog the breakpoint expression **(char[0x400])0x0C00** and set the "Breakpoint Trigger Type" to "No IFetch" will break when an access is made to BSL memory.

Break if Fetch is Out of Allowed Area

Using the **New Data Breakpoint** dialog the breakpoint expression **!(char[0xF000])0x1000** and set the "Breakpoint Trigger Type" to "IFetch" will break when a fetch is made from outside of flash memory.

CrossConnect Target Interface

Connection

Property	Description
Enable Regulator enableRegulator – Boolean	Specifies whether the regulator is enabled when the target is connected.
Hardware Breakpoint Reserve numHardwareBreakpointsReserve – Integer	Number of hardware breakpoints to reserve before software breakpoints are used.
Implement Software Breakpoints enableSoftwareBreakpoints – Boolean	Implement software breakpoints using the first hardware breakpoint.
Release JTAG releaseJTAG – Boolean	Release the JTAG signals when "Reset" or "Start Without Debugging" are used and on "Stop Debugging".

Current

Property	Description
Operating Voltage operatingVoltage – String	Returns the operating voltage of the target.
Serial Number connectedSerialNumber – String	The serial number of the currently connected CrossConnect.
Version firmwareVersion – String	The firmware revision of the currently connected CrossConnect.

Loader

Property	Description
Enable BSL Segment Erase/Write unlockBSL – Boolean	BSL Segments can be erased/written.
Enable Fast Flashing fast_flash_writes – Boolean	Enable block-mode fast flashing for devices that support it.
Enable Info SegmentA Erase/Write unlockSegmentA – Boolean	Info SegmentA can be erased/written.

Erase All <code>loaderEraseAll</code> – Boolean	If set to Yes , all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to No , only the areas of FLASH containing the program being downloaded will be erased.
No Load Sections <code>noLoadSections</code> – StringList	Names of (loadable) sections not to load.

Target

Property	Description
Connection <code>Connection</code> – String	The connection to use.
Device Type <code>device_id</code> – String	The detected type of the currently connected target device.

MSP430 Flash Emulation Tool Target Interface

Connection

Property	Description
Enable Regulator enableRegulator – Boolean	Specifies whether the regulator is enabled when the target is connected.
Hardware Breakpoint Reserve numHardwareBreakpointsReserve – Integer	Number of hardware breakpoints to reserve before software breakpoints are used.
Implement Software Breakpoints enableSoftwareBreakpoints – Boolean	Implement software breakpoints using the first hardware breakpoint.
Parallel Port portName – String	The parallel port connection to use to connect to target.
Parallel Port Address portAddress – String	The base address of the currently connected parallel port.
Parallel Port Sharing portSharing – Boolean	Specifies whether sharing of the parallel port with other device drivers or programs is permitted.
Release JTAG releaseJTAG – Boolean	Release the JTAG signals when "Reset" or "Start Without Debugging" are used and on "Stop Debugging".

Loader

Property	Description
Enable BSL Segment Erase/Write unlockBSL – Boolean	BSL Segments can be erased/written.
Enable Info SegmentA Erase/Write unlockSegmentA – Boolean	Info SegmentA can be erased/written.
Erase All loaderEraseAll – Boolean	If set to Yes , all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to No , only the areas of FLASH containing the program being downloaded will be erased.
No Load Sections noLoadSections – StringList	Names of (loadable) sections not to load.

Target

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

MSP430 DLL Target Interface

Connection

Property	Description
Hardware Breakpoint Reserve numHardwareBreakpointsReserve – Integer	Number of hardware breakpoints to reserve before software breakpoints are used.
Implement Software Breakpoints enableSoftwareBreakpoints – Boolean	Implement software breakpoints using the first hardware breakpoint.
Is USB Device isUsb – Boolean	The device is connected by USB.
Release JTAG releaseJTAG – Boolean	Release the JTAG signals when "Reset" or "Start Without Debugging" are used and on "Stop Debugging".
Spy-Bi-Wire/JTAG Supported supportsSpyBiWire – Boolean	Spy-Bi-Wire/JTAG is supported.
Target Driver DLL Path targetDllPath – FileName	Specifies the path to use to load the MSP430 target driver DLL.
Use Spy-Bi-Wire useSpyBiWire – Boolean	Connect using Spy-Bi-Wire if available for target.
Vcc (mV) vcc – Integer	Set the desired supply voltage (specified in millivolts).
Version version – String	MSP430 target driver DLL version number.

Loader

Property	Description
Enable BSL Segment Erase/Write unlockBSL – Boolean	BSL Segments can be erased/written.
Enable Info SegmentA Erase/Write unlockSegmentA – Boolean	Info SegmentA can be erased/written.
Erase All loaderEraseAll – Boolean	If set to Yes , all of the FLASH memory on the target will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to No , only the areas of FLASH containing the program being downloaded will be erased.

No Load Sections

noLoadSections – StringList

Names of (loadable) sections not to load.

Target

Property	Description
Connection Connection – String	The connection to use.
Device Type device_id – String	The detected type of the currently connected target device.

MSP430 Core Simulator Target Interface

Diagnostic

Property	Description
Trace Buffer Size traceBufferSize – Integer	Set the number of trace entries that are recorded.

Loader

Property	Description
No Load Sections noLoadSections – StringList	Names of (loadable) sections not to load.

Peripherals

Property	Description
Enable Peripheral File enablePeripheralFile – Boolean	Enable JavaScript simulation of peripherals.
Interrupt poll interval interruptPollInterval – Integer	Poll for interrupts every N cycles.

Peripheral File
peripheralFile – String

JavaScript file that simulates peripherals which contains the following functions:

- **reset()** — this is called on reset and should reset all state.
- **pollForInterrupts(elapsedCycleCount)** — this is called to check for interrupts. It is passed the elapsed number of cycles since the last time it was called. It should return -1 if no interrupt is outstanding or the interrupt number if an interrupt is outstanding.
- **loadPeripheral(address, size, debug)** — this is called when a memory read is made to the peripheral memory region and should return a value. The address and size of the memory access are supplied as the first two parameters. If the access is for debug purposes the third parameter is set to 1.
- **storePeripheral(address, size, value)** — this is called when a memory write is made to the peripheral memory region.

Simulator

Property	Description
Simulate BSL simulateBSL – Boolean	Simulate the 5xx/6xx BSL device startup.

Target

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

Debug file search editor

When a program is built with debugging enabled the debugging information contains paths describing where the source files that went into the program are located in order to allow the debugger to find them. If a program or libraries linked into the program are being run on a different machine to the one they were compiled on or if the source files have moved since the program was compiled, the debugger will be unable to find the source files.

In this situation the simplest way to help CrossStudio find the moved source files is to add the directory containing the source file to one of its source file search paths. Alternatively, if CrossStudio cannot find a source file it will prompt you for its location and record its new location in its source file map.

Debug source file search paths

The debug source file search paths can be used to help the debugger locate source files that are no longer in the same location as 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 set in the current project session and does not apply to all projects.
- **The global search path** This path is set system wide and applies to all projects.

The project session search path is checked before the global search path.

To view and edit the debug search paths

- From the **Debug** menu, click **Edit Search Paths**

Debug source file map

If a source file cannot be found whilst 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 is simply a mapping between the original file name and its new location. 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 for the file has been recorded 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

- From the **Debug** menu, click **Edit Search Paths**

To remove individual entries from the debug source file map

- From the **Debug** menu, click **Edit Search Paths**

- Right click on the mapping you want to delete
- From the context menu, click **Delete Mapping**

To remove all entries from the debug source file map

- From the **Debug** menu, click **Edit Search Paths**
- Select **Delete All Mappings** from the context menu

Environment Options Dialog

The environment options dialog enables you to modify options that apply to all uses of a CrossWorks installation.

Building Options	Options that are applicable to program building.
Debugging Options	Options that are applicable to debugging.
Environment Options	Options that are applicable to the whole CrossWorks environment.
Source Control Options	Options that are applicable to source control.
Text Editor Options	Options that are applicable to text editing.
Languages Options	Options that are applicable to text formatting of supported programming languages.
Windows Options	Options that are applicable to window display.

Building Environment Options

Build Options

Property	Description
Automatically Build Before Debug Environment/Build/Build Before Debug – Boolean	Enables auto-building of a project before downloading if it is out of date.
Build Macros Environment/Macros/Global Macros – StringList	Build macros that are shared across all solutions and projects e.g. paths to library files.
Confirm Debugger Stop Environment/Build/Confirm Debugger Stop – Boolean	Present a warning when you start to build that requires the debugger to stop.
Echo Build Command Lines Environment/Build/Show Command Lines – Boolean	Selects whether build command lines are written to the build log.
Echo Raw Error/Warning Output Environment/Build/Show Unparsed Error Output – Boolean	Selects whether the unprocessed error and warning output from tools is displayed in the build log.
Find Error After Building Environment/Build/Find Error After Build – Boolean	Moves the cursor to the first diagnostic after a build completes with errors.
Keep Going On Error Environment/Build/Keep Going On Error – Boolean	Build doesn't stop on error.
Save Project File Before Building Environment/Build/Save Project File On Build – Boolean	Selects whether to save the project file prior to build.
Show Build Information Environment/Build/Show Build Information – Boolean	Show build information.
Toolchain Root Directory Environment/Build/Tool Chain Root Directory – String	Specifies where to find the toolchain (compilers etc).

Window Options

Property	Description
Show Build Log On Build Environment/Show Transcript On Build – Boolean	Show the build log when a build starts.

Debugging Environment Options

Breakpoint Options

Property	Description
Clear Disassembly Breakpoints On Debug Stop Environment/Debugger/Clear Disassembly Breakpoint – Boolean	Clear Disassembly Breakpoints On Debug Stop
Initial Breakpoint Is Set Environment/Debugger/Set Initial Breakpoint – Enumeration	Specify when the initial breakpoint should be set
Set Initial Breakpoint At Environment/Debugger/Initial Breakpoint – String	An initial breakpoint to set if no other breakpoints exist

Debugging Options

Property	Description
TLS Expression Environment/Debugger/TLS Expression – String	Default expression the debugger evaluates to get the base of Thread Local Storage

Display Options

Property	Description
Close Disassembly On Mode Switch Environment/Debugger/Close Disassembly On Mode Switch – Boolean	Close Disassembly On Mode Switch
Data Tips Display a Maximum Of Environment/Debugger/Maximum Array Elements Displayed – IntegerRange	Selects the maximum number of array elements displayed in a datatip.
Default Display Mode Environment/Debugger/Default Variable Display Mode – Enumeration	Selects the format that data values are shown in.
Display Floating Point Number In Environment/Debugger/Floating Point Format Display – Custom	The printf format directive used to display floating point numbers.
Maximum Backtrace Calls Environment/Debugger/Maximum Backtrace Calls – IntegerRange	Selects the maximum number of calls when backtracing.

Prompt To Display If More Than Environment/Debugger/Array Elements Prompt Size – IntegerRange	The array size to display with prompt.
Show CPU Registers In Locals Window Environment/Debugger/Locals Display Registers – Boolean	Specify whether the locals window should display CPU registers
Show Labels In Disassembly Environment/Debugger/Disassembly Show Labels – Boolean	Show Labels In Disassembly
Show Source In Disassembly Environment/Debugger/Disassembly Show Source – Boolean	Show Source In Disassembly
Show char * as null terminated string Environment/Debugger/Display Char Ptr As String – Boolean	Show char * as null terminated string
Source Path Environment/Debugger/Source Path – StringList	Global search path to find source files.

Extended Data Tips Options

Property	Description
ASCII Environment/Debugger/Extended Tooltip Display Mode/ASCII – Boolean	Selects ASCII extended datatips.
Binary Environment/Debugger/Extended Tooltip Display Mode/Binary – Boolean	Selects Binary extended datatips.
Decimal Environment/Debugger/Extended Tooltip Display Mode/Decimal – Boolean	Selects Decimal extended datatips.
Hexadecimal Environment/Debugger/Extended Tooltip Display Mode/Hexadecimal – Boolean	Selects Hexadecimal extended datatips.
Octal Environment/Debugger/Extended Tooltip Display Mode/Octal – Boolean	Selects Octal extended datatips.
Unsigned Decimal Environment/Debugger/Extended Tooltip Display Mode/Unsigned Decimal – Boolean	Selects Unsigned Decimal extended datatips.

Target Options

Property	Description
Reset Target When Restarting Environment/Debugger/Restart Resets Target – Boolean	Specify whether the target should be reset when restarting the debug session
Step Using Hardware Step Environment/Debugger/Step Using Hardware Step – Boolean	Step using hardware single stepping rather than setting breakpoints

Window Options

Property	Description
Clear Debug Terminal On Run Environment/Clear Debug Terminal On Run – Boolean	Clear the debug terminal automatically when a program is run.
Hide Output Window On Successful Load Debugging/Hide Transcript On Successful Load – Boolean	Hide the Output window when a load completes without error.
Show Debug Terminal Environment/Show Debug Terminal – Enumeration	Show or focus the debug terminal when input or output happens.
Show Target Log On Load Debugging/Show Transcript On Load – Boolean	Show the target log when a load starts.

IDE Environment Options

Browser Options

Property	Description
Text Size Environment/Browser/Text Size – Enumeration	Sets the text size of the integrated HTML and help browser.
Underline Hyperlinks In Browser Environment/Browser/Underline Web Links – Boolean	Enables underlining of hypertext links in the integrated HTML and help browser.

Directory Options

Property	Description
Package Directory Environment/Package/Destination Directory – String	Specifies the directory packages are installed to.
Quick Open Directories Environment/General/Quick Open Directories – StringList	Specifies additional directories to search when using Open From Project or Quick Open.

File Search Options

Property	Description
Files To Search Find In Files/File Type – StringList	The wildcard used to match files in Find In Files searches.
Find History Find In Files/Find History – StringList	The list of strings recently used in searches.
Folder History Find In Files/Folder History – StringList	The set of folders recently used in file searches.
Match Case Find In Files/Match Case – Boolean	Whether the case of letters must match exactly when searching.
Match Whole Word Find In Files/Match Whole Word – Boolean	Whether the whole word must match when searching.
Replace History Find In Files/Replace History – StringList	The list of strings recently used in searches.
Search Dependencies Find In Files/Search Dependencies – Boolean	Controls searching of dependent files.

Search In Find In Files/Context – Enumeration	Where to look to find files.
Use Regular Expressions Find In Files/Use RegExp – Boolean	Whether to use a regular expression or plain text search.

Internet Options

Property	Description
Check For Latest News Environment/Internet/RSS Update – Boolean	Specifies whether to enable downloading of the Latest News RSS feeds.
Check For Packages Environment/Internet/Check Packages – Boolean	Specifies whether to enable downloading of the list of available packages.
Check For Updates Environment/Internet/Check Updates – Boolean	Specifies whether to enable checking for software updates.
Enable Connection Debugging Environment/Internet/Enable Debugging – Boolean	Controls debugging traces of internet connections and downloads.
External Web Browser Environment/External Web Browser – FileName	The path to the external web browser to use when accessing non-local files.
HTTP Proxy Host Environment/Internet/HTTP Proxy Server – String	Specifies the IP address or hostname of the HTTP proxy server. If empty, no HTTP proxy server will be used.
HTTP Proxy Port Environment/Internet/HTTP Proxy Port – IntegerRange	Specifies the HTTP proxy server's port number.
Maximum Download History Items Environment/Internet/Max Download History Items – IntegerRange	The maximum amount of download history kept in the downloads window.

Print Options

Property	Description
Bottom Margin Environment/Printing/Bottom Margin – IntegerRange	The page's bottom margin in millimetres.
Left Margin Environment/Printing/Left Margin – IntegerRange	The page's left margin in millimetres.

Page Orientation Environment/Printing/Orientation – Enumeration	The page's orientation.
Page Size Environment/Printing/Page Size – Enumeration	The page's size.
Right Margin Environment/Printing/Right Margin – IntegerRange	The page's right margin in millimetres.
Top Margin Environment/Printing/Top Margin – IntegerRange	The page's top margin in millimetres.

Startup Options

Property	Description
Allow Multiple CrossStudios Environment/Permit Multiple Studio Instances – Boolean	Allow more than one CrossStudio to run at the same time.
Load Most-Recent Solution At Startup Environment/Startup Action – Boolean	Enables loading the previous session's solution on startup.
New Project Directory Environment/General/Solution Directory – String	The directory where projects are created.
Project Templates File Environment/General/Project Templates – String	The project templates file.
Splash Screen Environment/Splash Screen – Enumeration	How to display the splash screen on startup.

Status Bar Options

Property	Description
(Visible) Environment/Status Bar – Boolean	Show or hide the status bar.
Show Build Status Pane Environment/General/Status Bar/Show Build Status – Boolean	Show or hide the Build pane in the status bar.
Show Caps Lock Status Pane Environment/General/Status Bar/Show Caps Lock – Boolean	Show or hide the Insert/Overwrite pane in the status bar.

Show Caret Position Pane	
Environment/General/Status Bar/Show Caret Pos – Boolean	Show or hide the Caret Position pane in the status bar.
Show Insert/Overwrite Status Pane	
Environment/General/Status Bar/Show Insert Mode – Boolean	Show or hide the Insert/Overwrite pane in the status bar.
Show Num Lock Status Pane	
Environment/General/Status Bar/Show Num Lock – Boolean	Show or hide the Num Lock pane in the status bar.
Show Read-Only Status Pane	
Environment/General/Status Bar/Show Read Only – Boolean	Show or hide the Read Only pane in the status bar.
Show Scroll Lock Status Pane	
Environment/General/Status Bar/Show Scroll Lock – Boolean	Show or hide the Scroll Lock pane in the status bar.
Show Size Grip	
Environment/General/Status Bar/Show Size Grip – Boolean	Show or hide the status bar size grip.
Show Target Pane	
Environment/General/Status Bar/Show Target – Boolean	Show or hide the Target pane in the status bar.
Show Time Pane	
Environment/General/Status Bar/Show Time – Boolean	Show or hide the Time pane in the status bar.

User Interface Options

Property	Description
Application Main Font Environment/Application Main Font – Font	The font to use for the user interface as a whole.
Application Monospace Font Environment/Application Monospace Font – Font	The fixed-size font to use for the user interface as a whole.
Document Grouping Environment/General/Document Grouping – Enumeration	Specifies how documents are grouped
Document Title Format Environment/General/Document Title Format – Enumeration	Specifies how documents are shown in the header
Error Display Timeout Environment/Error Display Timeout – IntegerRange	The minimum time, in seconds, that errors are shown for in the status bar.

Errors Are Displayed Environment/Error Display Mode – Enumeration	How errors are reported in CrossStudio.
File Size Display Units Environment/Size Display Unit – Enumeration	How to display sizes of items in the user interface. SI defines 1kB=1000 bytes, IEC defines 1kiB=1024 bytes, Alternate SI defines 1kB=1024 bytes.
Number File Names in Menus Environment/Number Menus – Boolean	Number the first nine file names in menus for quick keyboard access.
Show Large Icons In Toolbars Environment/General/Large Icons – Boolean	Show large or small icons on toolbars.
Show Window Selector On Tab Environment/Show Selector – Boolean	Use the window selector on Next and Previous Window commands activated from the keyboard.
User Interface Theme Environment/General/Skin – Enumeration	The theme that CrossStudio uses.
Window Menu Contains At Most Environment/Max Window Menu Items – IntegerRange	The maximum number of windows appearing in the Windows menu.

Programming Language Environment Options

Assembly Language Settings

Property	Description
Column Guide Columns Text Editor/Indent/Assembly Language/ Column Guides – String	The columns that guides are drawn for.
Indent Closing Brace Text Editor/Indent/Assembly Language/ Close Brace – Boolean	Indent the closing brace of compound statements.
Indent Context Text Editor/Indent/Assembly Language/ Context Lines – IntegerRange	The number of lines to use for context when indenting.
Indent Mode Text Editor/Indent/Assembly Language/ Indent Mode – Enumeration	How to indent when a new line is inserted.
Indent Opening Brace Text Editor/Indent/Assembly Language/Open Brace – Boolean	Indent the opening brace of compound statements.
Indent Size Text Editor/Indent/Assembly Language/Size – IntegerRange	The number of columns to indent a code block.
Tab Size Text Editor/Indent/Assembly Language/Tab Size – IntegerRange	The number of columns between tabstops.
Use Tabs Text Editor/Indent/Assembly Language/Use Tabs – Boolean	Insert tabs when indenting.
User-Defined Keywords Text Editor/Indent/Assembly Language/ Keywords – StringList	Additional identifiers to highlight as keywords.

C and C++ Settings

Property	Description
Column Guide Columns Text Editor/Indent/C and C++/Column Guides – String	The columns that guides are drawn for.

Indent Closing Brace	
Text Editor/Indent/C and C++/Close Brace	Indent the closing brace of compound statements.
– Boolean	
Indent Context	
Text Editor/Indent/C and C++/Context Lines	The number of lines to use for context when indenting.
– IntegerRange	
Indent Mode	
Text Editor/Indent/C and C++/Indent Mode	How to indent when a new line is inserted.
– Enumeration	
Indent Opening Brace	
Text Editor/Indent/C and C++/Open Brace	Indent the opening brace of compound statements.
– Boolean	
Indent Size	
Text Editor/Indent/C and C++/Size	The number of columns to indent a code block.
– IntegerRange	
Tab Size	
Text Editor/Indent/C and C++/Tab Size	The number of columns between tabstops.
– IntegerRange	
Use Tabs	
Text Editor/Indent/C and C++/Use Tabs	Insert tabs when indenting.
– Boolean	
User-Defined Keywords	
Text Editor/Indent/C and C++/Keywords	Additional identifiers to highlight as keywords.
– StringList	

Default Settings

Property	Description
Column Guide Columns	
Text Editor/Indent/Default/Column Guides	The columns that guides are drawn for.
– String	
Indent Closing Brace	
Text Editor/Indent/Default/Close Brace	Indent the closing brace of compound statements.
– Boolean	
Indent Context	
Text Editor/Indent/Default/Context Lines	The number of lines to use for context when indenting.
– IntegerRange	
Indent Mode	
Text Editor/Indent/Default/Indent Mode	How to indent when a new line is inserted.
– Enumeration	

Indent Opening Brace	
Text Editor/Indent/Default/Open Brace – Boolean	Indent the opening brace of compound statements.
Indent Size	
Text Editor/Indent/Default/Size – IntegerRange	The number of columns to indent a code block.
Tab Size	
Text Editor/Indent/Default/Tab Size – IntegerRange	The number of columns between tabstops.
Use Tabs	
Text Editor/Indent/Default/Use Tabs – Boolean	Insert tabs when indenting.
User-Defined Keywords	
Text Editor/Indent/Default/Keywords – StringList	Additional identifiers to highlight as keywords.

Java Settings

Property	Description
Column Guide Columns	
Text Editor/Indent/Java/Column Guides – String	The columns that guides are drawn for.
Indent Closing Brace	
Text Editor/Indent/Java/Close Brace – Boolean	Indent the closing brace of compound statements.
Indent Context	
Text Editor/Indent/Java/Context Lines – IntegerRange	The number of lines to use for context when indenting.
Indent Mode	
Text Editor/Indent/Java/Indent Mode – Enumeration	How to indent when a new line is inserted.
Indent Opening Brace	
Text Editor/Indent/Java/Open Brace – Boolean	Indent the opening brace of compound statements.
Indent Size	
Text Editor/Indent/Java/Size – IntegerRange	The number of columns to indent a code block.
Tab Size	
Text Editor/Indent/Java/Tab Size – IntegerRange	The number of columns between tabstops.
Use Tabs	
Text Editor/Indent/Java/Use Tabs – Boolean	Insert tabs when indenting.
User-Defined Keywords	
Text Editor/Indent/Java/Keywords – StringList	Additional identifiers to highlight as keywords.

Source Control Environment Options

Configuration Options

Property	Description
Enable Source Control Integration Environment/Source Code Control/Enabled – Boolean	Source Control integration enabled.
Source Control Provider Environment/Source Code Control/Provider – Enumeration	The source control provider to use.

External Tools

Property	Description
Diff Command Line Environment/Source Code Control/ DiffCommand – StringList	The diff command line
Merge Command Line Environment/Source Code Control/ MergeCommand – StringList	The merge command line

Preference Options

Property	Description
Inhibit Add Dialog Environment/Source Code Control/ InhibitAddDialog – Boolean	Inhibits the dialog when adding a file to source control.
Inhibit Check In Dialog Environment/Source Code Control/ InhibitCheckinDialog – Boolean	Inhibits the dialog when checking in a file to source control.
Inhibit Check Out Dialog Environment/Source Code Control/ InhibitCheckoutDialog – Boolean	Inhibits the dialog when checking out a file from source control.
Inhibit Check Out On Edit Dialog Environment/Source Code Control/ InhibitCheckoutOnEditDialog – Boolean	Inhibits the check out on edit dialog.
Inhibit Get Latest Dialog Environment/Source Code Control/ InhibitGetLatestDialog – Boolean	Inhibits the dialog when updating the local version from the source control version.

Inhibit Undo Check Out Dialog

Environment/Source Code Control/

InhibitUndoCheckoutDialog – Boolean

Inhibits the dialog when undoing a checkout to source control.

Text Editor Environment Options

Cursor Fence Options

Property	Description
Bottom Margin Text Editor/Margins/Bottom – IntegerRange	The number of lines in the bottom margin.
Keep Cursor Within Fence Text Editor/Margins/Enabled – Boolean	Enable margins to fence and scroll around the cursor.
Left Margin Text Editor/Margins/Left – IntegerRange	The number of characters in the left margin.
Right Margin Text Editor/Margins/Right – IntegerRange	The number of characters in the right margin.
Top Margin Text Editor/Margins/Top – IntegerRange	The number of lines in the right margin.

Editing Options

Property	Description
Allow Drag and Drop Editing Text Editor/Drag Drop Editing – Boolean	Enables dragging and dropping of selections in the text editor.
Bold Popup Diagnostic Messages Text Editor/Bold Popup Diagnostics – Boolean	Displays popup diagnostic messages in bold for easier reading.
Check Spelling Text Editor/Spell Checking – Boolean	Enable spell checking in comments.
Column-mode Tab Text Editor/Column Mode Tab – Boolean	Tab key moves to the next textual column using the line above.
Confirm Modified File Reload Text Editor/Confirm Modified File Reload – Boolean	Display a confirmation prompt before reloading a file that has been modified on disk.
Copy Action Text Editor/Copy Action – Enumeration	What Copy copies when nothing is selected.
Copy On Mouse Select Text Editor/Copy On Mouse Select – Boolean	Automatically copy text to clipboard when marking a selection with the mouse.
Cut Action Text Editor/Cut Action – Enumeration	What Cut cuts when nothing is selected.
Diagnostic Cycle Mode Text Editor/Diagnostic Cycle Mode – Enumeration	Iterates through diagnostics either from most severe to least severe or in reported order.

Edit Read-Only Files Text Editor/Edit Read Only – Boolean	Allow editing of read-only files.
Enable Popup Diagnostics Text Editor/Enable Popup Diagnostics – Boolean	Enables on-screen diagnostics in the text editor.
Enable Virtual Space Text Editor/Enable Virtual Space – Boolean	Permit the cursor to move into locations that do not currently contain text.
Expand Templates On Space Text Editor/Auto Expand Templates – Boolean	Enables automatic expansion of templates when the space key is pressed.
Numeric Keypad Editing Text Editor/Numeric Keypad Enabled – Boolean	Selects whether the numeric keypad plus and minus buttons copy and cut text.
Paste On Mouse Middle Button Text Editor/Paste On Mouse Middle Button – Boolean	Paste text from clipboard when mouse middle button is pressed.
Undo And Redo Behavior Text Editor/Undo Mode – Enumeration	How Undo and Redo group your typing when it is undone and redone.

Find And Replace Options

Property	Description
Case Sensitive Matching Text Editor/Find/Match Case – Boolean	Enables or disables the case sensitivity of letters when searching.
Find History Text Editor/Find/History – StringList	The list of strings recently used in searches.
Regular Expression Matching Text Editor/Find/Use RegExp – Boolean	Enables regular expression matching rather than plain text matching.
Replace History Text Editor/Replace/History – StringList	The list of strings recently used in replaces.
Whole Word Matching Text Editor/Find/Match Whole Word – Boolean	Enables or disables whole word matching when searching.

International

Property	Description
Default Text File Encoding Text Editor/Default Codec – Enumeration	The encoding to use if not overridden by a project property or file is not in a known format.

Save Options

Property	Description

Backup File History Depth Text Editor/Backup File Depth – IntegerRange	The number of backup files to keep when saving an existing file.
Delete Trailing Space On Save Text Editor/Delete Trailing Space On Save – Boolean	Deletes trailing whitespace from each line when a file is saved.
Tab Cleanup On Save Text Editor/Cleanup Tabs On Save – Enumeration	Cleans up tabs when a file is saved.

Visual Appearance

Property	Description
Context Bar Text Editor/Context Bar – Enumeration	Show or hide the context bar below the tabs.
Font Text Editor/Font – Font	The font to use for text editors.
Hide Cursor When Typing Text Editor/Hide Cursor When Typing – Boolean	Hide or show the I-beam cursor when you start to type.
Highlight Cursor Line Text Editor/Highlight Cursor Line – Boolean	Enable or disable visually highlighting the cursor line.
Horizontal Scroll Bar Text Editor/HScroll Bar – Enumeration	Show or hide the horizontal scroll bar.
Insert Caret Style Text Editor/Insert Caret Style – Enumeration	How the caret is displayed with the editor in insert mode.
Line Numbers Text Editor/Line Number Mode – Enumeration	How often line numbers are displayed in the margin.
Mate Matching Mode Text Editor/Mate Matching Mode – Enumeration	Controls when braces, brackets, and parentheses are matched.
Overwrite Caret Style Text Editor/Overwrite Caret Style – Enumeration	How the caret is displayed with the editor in overwrite mode.
Show Diagnostic Icons In Gutter Text Editor/Diagnostic Icons – Boolean	Enables display of diagnostic icons in the icon gutter.
Show Icon Gutter Text Editor/Icon Gutter – Boolean	Show or hide the left-hand gutter containing breakpoint, bookmark, and optional diagnostic icons.
Show Mini Toolbar Text Editor/Mini Toolbar – Boolean	Show the mini toolbar when selecting text with the mouse.
Use I-beam Cursor Text Editor/Ibeam cursor – Boolean	Show an I-beam or arrow cursor in the text editor.

Vertical Scroll Bar

Text Editor/VScroll Bar – Enumeration

Show or hide the vertical scroll bar.

Windows Environment Options

Call Stack Options

Property	Description
Show Call Address Environment/Call Stack/Show Call Address – Boolean	Enables the display of the call address in the call stack.
Show Call Source Location Environment/Call Stack/Show Call Location – Boolean	Enables the display of the call source location in the call stack.
Show Parameter Names Environment/Call Stack/Show Parameter Names – Boolean	Enables the display of parameter names in the call stack.
Show Parameter Types Environment/Call Stack/Show Parameter Types – Boolean	Enables the display of parameter types in the call stack.
Show Parameter Values Environment/Call Stack/Show Parameter Values – Boolean	Enables the display of parameter values in the call stack.

Clipboard Ring Options

Property	Description
Maximum Items Held In Ring Environment/Clipboard Ring/Max Entries – IntegerRange	The maximum number of items held on the clipboard ring before they are recycled.
Preserve Contents Between Runs Environment/Clipboard Ring/Save – Boolean	Save the clipboard ring across CrossStudio runs.

Outline Window Options

Property	Description
Group #define Directives Windows/Outline/Group Defines – Boolean	Group consecutive #define and #undef preprocessor directives.
Group #if Directives Windows/Outline/Group Ifs – Boolean	Group lines contained between #if, #else, and #endif preprocessor directives.
Group #include Directives Windows/Outline/Group Includes – Boolean	Group consecutive #include preprocessor directives.

Group Top-Level Declarations Windows/Outline/Group Top Level Items – Boolean	Group consecutive top-level variable and type declarations.
Group Visibility Windows/Outline/Group Visibility – Boolean	Group class members by public, protected, and private visibility.
Refresh Outline and Preview Windows/Outline/Preview Refresh Mode – Enumeration	How the Preview pane refreshes its contexts.

Project Explorer Options

Property	Description
Add Filename Replace Macros Environment/Project Explorer/Filename Replace Macros – StringList	Macros (system and global) used to replace the start of a filename on project file addition.
Color Project Nodes Environment/Project Explorer/Color Nodes – Boolean	Show the project nodes colored for identification in the Project Explorer.
Output Files Folder Environment/Project Explorer/Show Output Files – Boolean	Show the build output files in an Output Files folder in the project explorer.
Read-Only Data In Code Environment/Project Explorer/Statistics Read-Only Data Handling – Boolean	Configures whether read-only data contributes to the Code or Data statistic.
Show Dependencies Environment/Project Explorer/Dependencies Display – Enumeration	Controls how the dependencies are displayed.
Show File Count on Folder Environment/Project Explorer/Count Files – Boolean	Show the number of files contained in a folder as a badge in the Project Explorer.
Show Properties Environment/Project Explorer/Properties Display – Enumeration	Controls how the properties are displayed.
Show Statistics Rounded Environment/Project Explorer/Statistics Format – Boolean	Show exact or rounded sizes in the project explorer.
Source Control Status Column Environment/Project Explorer/Show Source Code Control Status – Boolean	Show the source control status column in the project explorer.
Statistics Column Environment/Project Explorer/Statistics Display – Boolean	Show the code and data size columns in the Project Explorer.

Synchronize Explorer With Editor Environment/Project Explorer/Sync Editor – Boolean	Synchronizes the Project Explorer with the document being edited.
Use Common Properties Folder Environment/Project Explorer/Common Properties Display – Boolean	Controls how common properties are displayed.

Properties Window Options

Property	Description
Properties Displayed Environment/General/Properties Displayed – Enumeration	Set how the properties are displayed.
Show Property Details Environment/General/Property View Details – Boolean	Show or hide the property description.

Windows Window Options

Property	Description
Buffer Grouping Environment/Windows/Grouping – Enumeration	How the files are grouped or listed in the windows window.
Show File Path as Tooltip Environment/Windows/Show Filename Tooltips – Boolean	Show the full file name as a tooltip when hovering over files in the Windows window.
Show Line Count and File Size Environment/Windows/Show Sizes – Boolean	Show the number of lines and size of each file in the windows list.

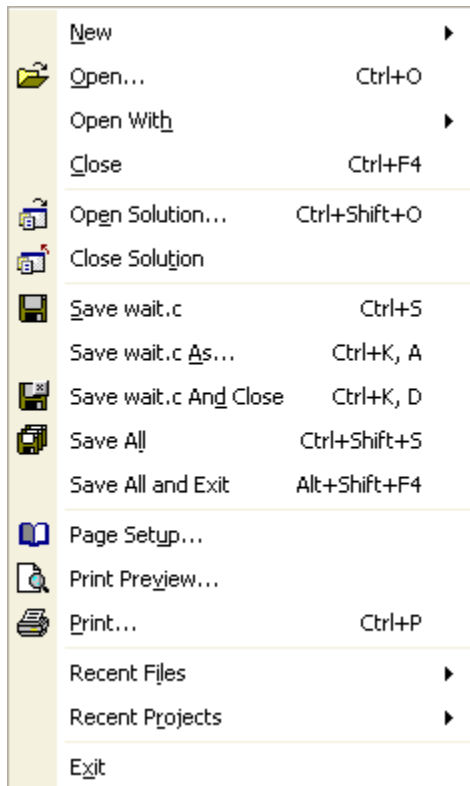
CrossStudio menu summary

The following sections describe each menu and each menu item.

File menu

The **File** menu provides commands to create, open, and close files, and to print them.

The File menu



File commands

Menu command	Keystroke	Description
New		Displays the New menu.
Open	Ctrl+O	Opens an existing file for editing.
Open With		Displays the Open With menu.
Close	Ctrl+F4	Closes the active editor. If you have made changes to the file, CrossStudio prompts you to save the file.

Open Solution	Ctrl+Shift+O	Opens an existing solution for editing. If you already have an open solution, CrossStudio will close it before opening the new solution and, if you have made changes to any of the files in your solution, you are prompted to save each of them.
Close Solution		Closes the current solution. If you have made changes to any of the files in your solution, you are prompted to save each of them.
Save <i>file</i>	Ctrl+S	Saves the contents of the active editor to disk. If it is a new file without a name, CrossStudio opens a file browser for you to choose where to save the file and what to call it.
Save <i>file</i> As...	Ctrl+K, A	Saves the contents of the active editor to disk using a different name. CrossStudio opens a file browser for you to choose where to save the file and what to call it. After saving, the editor is set to edit the newly saved file, not the previous file.
Save <i>file</i> And Close	Ctrl+K, D	Saves the contents of the active editor to disk and then closes the editor. If it is a new file without a name, CrossStudio opens a file browser for you to choose where to save the file and what to call it.
Save All	Ctrl+Shift+S	Saves all edited files to disk. For each new file without a name, CrossStudio opens a file browser for you to choose where to save the file and what to call it. Cancelling a save at any time will return you to CrossStudio without saving the remainder of the files.

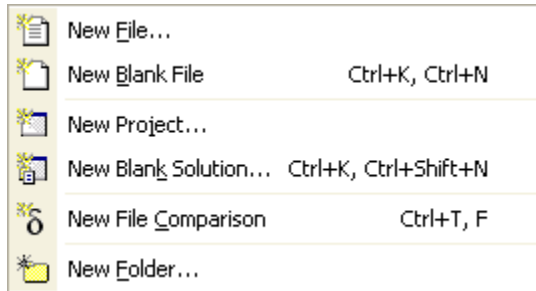
Save All And Exit	Alt+Shift+F4	Saves all edited files to disk and then exits CrossStudio. For each new file without a name, CrossStudio opens a file browser for you to choose where to save the file and what to call it. Cancelling a save at any time will return you to CrossStudio without exiting.
Page Setup...		Steps into the next statement or instruction and enters C functions and assembly language subroutines. If a breakpoint is hit when stepping, the debugger immediately stops at that breakpoint.
Print Preview...		Opens the Print Preview dialog and shows the document as it will appear when it is printed.
Recent Files		Opens the Recent Files menu which contains a list of files that have been recently opened, with the most recently opened file first in the list. You can configure the number of files retained in the Recent Files menu in the Environment Options dialog. You can clear the list of recent files by selecting Clear Recent Files List from the Recent Files menu.
Recent Projects		Opens the Recent Projects menu which contains a list of projects that have been recently opened, with the most recently opened project first in the list. You can configure the number of projects retained in the Recent Projects menu in the Environment Options dialog. You can clear the list of recent projects by selecting Clear Recent Projects List from the Recent Projects menu.

Exit	Alt+F4	Saves all edited files, closes the solution, and exits CrossStudio. For each new file without a name, CrossStudio opens a file browser for you to choose where to save the file and what to call it. Cancelling a save at any time will return you to CrossStudio without exiting.

New menu

The **New** menu provides commands to create files and folders.

The New menu



New menu commands

Menu command	Keystroke	Description
New File...		Creates a new file using the New File dialog
New Blank File	Ctrl+K, Ctrl+N	Creates a new, unnamed document.
New Project...		Creates a new project using the New Project dialog.
New Blank Solution	Ctrl+K, Ctrl+Shift+N	Creates a new solution containing no projects.
New File Comparison	Ctrl+T, F	Creates a new file comparison window.
New Folder...		Creates a new folder underneath the currently selected item in the Project Explorer .

Edit menu

The **Edit** menu provides commands to edit files.

The Edit menu

	Undo	Ctrl+Z
	Redo	Ctrl+Y
	Cut	Ctrl+X
	Copy	Ctrl+C
	Paste	Ctrl+V
	Delete	Del
	Clipboard	▶
	Clipboard Ring	▶
	Select All	Ctrl+A
	Insert File	Ctrl+K, Ctrl+I
	Expand Template	Ctrl+J
	Editing Macros	▶
	Selection	▶
	Bookmarks	▶
	Format	▶
	Advanced	▶

Edit menu commands

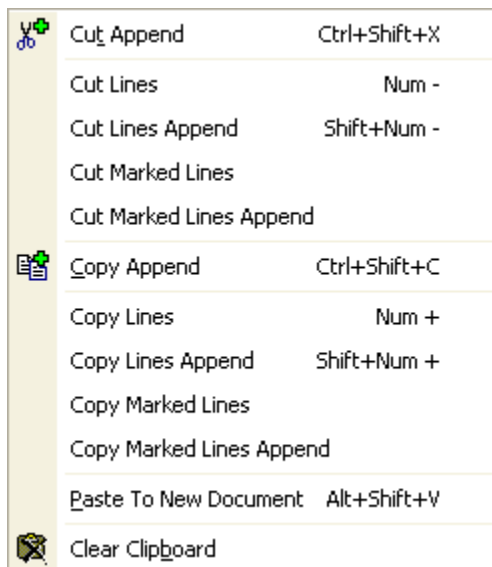
Menu command	Keystroke	Description
Undo	Ctrl+Z —or— Alt+Backspace	Undoes the last editing action.
Redo	Ctrl+Y —or— Alt+Shift+Backspace	Redoes the last undone editing action.
Cut	Ctrl+X —or— Shift+Delete	Cuts the selected text to the clipboard. If no text is selected, cuts the current line to the clipboard.
Copy	Ctrl+C —or— Ctrl+Insert	Cuts the selected text to the clipboard. If no text is selected, copies the current line to the clipboard.

Paste	Ctrl+V —or— Shift+Insert	Pastes the clipboard into the document.
Delete	Delete	Deletes the selection. If no text is selected, deletes the character to the right of the cursor.
Clipboard		Displays the Clipboard menu.
Clipboard Ring		Displays the Clipboard Ring menu.
Select All	Ctrl+A	Selects all text or items in the document.
Insert File	Ctrl+K, Ctrl+I	Inserts a file into the document at the cursor position.
Expand Template	Ctrl+J	Forces expansion of a template.
Editing Macros		Displays the Editing Macros menu.
Selection		Displays the Edit Selection menu. See Edit Selection menu .
Bookmarks		Displays the Bookmarks menu.
Format		Displays the Formatting menu.
Advanced		Displays the Advanced Editing menu.

Clipboard menu

The **Clipboard** menu provides commands to edit files using the clipboard.

The Clipboard menu



Clipboard menu commands

Menu command	Keystroke	Description
Cut Append	Ctrl+Shift+X	Cuts the selected text and appends it to the clipboard. If no text is selected, cuts and appends the current line to the clipboard.
Cut Lines	Num -	Converts the selection to complete lines then cuts the selected text lines them to the clipboard. If no text is selected, cuts and appends the current line to the clipboard.
Cut Lines Append	Shift+Num -	Converts the selection to complete lines then cuts the selected text lines and appends them to the clipboard. If no text is selected, cuts and appends the current line to the clipboard.
Cut Marked Lines		Cuts all bookmarked lines in the current document to the clipboard.

Cut Marked Lines Append		Cuts all bookmarked lines in the current document and appends them to the clipboard.
Copy Append		Copies the selected text and appends it to the clipboard. If no text is selected, copies and appends the current line to the clipboard.
Copy Lines		Converts the selection to complete lines then copies the selected text lines them to the clipboard. If no text is selected, copies and appends the current line to the clipboard.
Copy Lines Append		Converts the selection to complete lines then copies the selected text lines and appends them to the clipboard. If no text is selected, copies and appends the current line to the clipboard.
Copy Marked Lines		Copies all bookmarked lines in the current document to the clipboard.
Copy Marked Lines Append		Copies all bookmarked lines in the current document and appends them to the clipboard.
Paste to New Document	Alt+Shift+V	Creates a new, unnamed document and pastes the clipboard into it.
Clear Clipboard		Clears the contents of the clipboard.

Clipboard Ring menu

The **Clipboard Ring** menu provides commands to edit files using the clipboard ring.

The Clipboard Ring menu

	Paste All	Ctrl+R, Ctrl+V
	Cycle Clipboard Ring	Ctrl+Shift+V
	Clear Clipboard Ring	Ctrl+R, Del
	Clipboard Ring	Ctrl+Alt+C

Clipboard Ring menu commands

Menu command	Keystroke	Description
Paste All	Ctrl+Shift+X	Pastes the contents of the clipboard ring to the current document.
Cycle Clipboard Ring	Num -	Cycles the clipboard ring.
Clear Clipboard Ring	Ctrl+R, Del	Clears the contents of the clipboard ring.
Clipboard Ring	Ctrl+Alt+C	Displays the Clipboard Ring window. See Clipboard Ring .

Macros menu

The **Macros** menu provides additional commands to record and play key sequences as well as provide some fixed macros.

The Macros menu

	Play Recording	Ctrl+Shift+P
	Start Recording	Ctrl+Shift+R
	Pause/Resume Recording	
	Stop Recording	
	Cancel Recording	
	Insert Hard Tab	Ctrl+Q, Tab
	Declare Or Cast To "char"	Ctrl+Q, Ctrl+C
	Declare Or Cast To "short"	Ctrl+Q, Ctrl+S
	Declare Or Cast To "int"	Ctrl+Q, Ctrl+I
	Declare Or Cast To "long"	Ctrl+Q, Ctrl+L
	Declare Or Cast To "void"	Ctrl+Q, Ctrl+V
	Insert "const"	Ctrl+Q, Ctrl+K
	Insert "volatile"	Ctrl+Q, Ctrl+O
	Insert "extern"	Ctrl+Q, Ctrl+X

Macros menu commands

Menu command	Keystroke	Description
Play Recording	Ctrl+Shift+P	Plays the last recorded keyboard macro.
Start Recording	Ctrl+Shift+R	Starts recording a keyboard macro.
Pause/Resume Recording		Temporarily pauses a recording a keyboard macro. If already paused, recommences recording of the keyboard macro.
Stop Recording		Stops recording a keyboard macro and saves it. Note that when recording has commenced, the keystroke to stop recording the keyboard macro is Ctrl+Shift+R.

Cancel Recording		Cancels recording without changing the current keyboard macro.
Insert Hard Tab	Ctrl+Q, Tab	Inserts a tab character into the document even if the document's language settings inserts tabs as spaces.
Declare Or Cast to <i>type</i>		If there is a selection, parentheses are placed around the selection and that expression is cast to <i>type</i> . If there is no selection, <i>type</i> is inserted into the document.
Insert <i>keyword</i>		Inserts <i>keyword</i> into the document, followed by a space.

Edit Selection menu

The **Edit > Selection** menu provides commands to operate on the selection.

The Edit Selection menu

	Tabify	Ctrl+K, Tab
	Untabify	Ctrl+K, Space
	Make Uppercase	Ctrl+Shift+U
	Make Lowercase	Ctrl+U
	Switch Case	Alt+Shift+U
	Comment	Ctrl+/*
	Uncomment	Ctrl+Shift+/*
	Increase Line Indent	Tab
	Decrease Line Indent	Shift+Tab
	Align Left	Ctrl+K, Ctrl+J, L
	Center	Ctrl+K, Ctrl+J, C
	Align Right	Ctrl+K, Ctrl+J, R
	Sort Ascending	
	Sort Descending	

Edit Selection menu commands

Menu command	Keystroke	Description
Tabify	Ctrl+K, Tab	Convert space characters in the selection to tabs according to the tab settings for the language.
Untabify	Ctrl+K, Space	Convert tab characters in the selection to spaces according to the tab settings for the language.
Make Uppercase	Ctrl+Shift+U	Convert the letters in the selection to uppercase. If there is no selection, CrossStudio converts the character to the right of the cursor to uppercase and moves the cursor right one character.

Switch Case	Ctrl+U	Switches the letter case of letters in the selection; that is, uppercase characters become lowercase, and lowercase become uppercase. If there is no selection, CrossStudio switches the letter case of the character to the right of the cursor and moves the cursor right one character.
Comment	Ctrl+/ 	Prefixes lines in the selection with language-specific comment characters. If there is no selection, CrossStudio comments the cursor line.
Uncomment	Ctrl+Shift+/ 	Removes the prefixed from lines in the selection that contains language-specific comment characters. If there is no selection, CrossStudio uncomments the cursor line.
Increase Line Indent	Tab	Increases the line indent of the selection. If there is no selection, the cursor is moved to the next tab stop by inserting spaces or a tab character according to the tab settings for the document.
Decrease Line Indent	Shift+Tab	Decreases the line indent of the selection. If there is no selection, the cursor is moved to the previous tab stop.
Align Left	Ctrl+K, Ctrl+J, L	Aligns all text in the selection to the leftmost non-blank character in the selection.
Align Center	Ctrl+K, Ctrl+J, C	Centers all text in the selection between the leftmost and rightmost non-blank characters in the selection.
Align Right	Ctrl+K, Ctrl+J, R	Aligns all text in the selection to the rightmost non-blank character in the selection.

Sort Ascending		Sorts the selection into ascending lexicographic order.
Sort Descending		Sorts the selection into descending lexicographic order.

Bookmarks menu

The **Bookmarks** menu provides commands to drop and find temporary bookmarks.

The Bookmarks menu

	<u>T</u> oggle Bookmark	Ctrl+F2
	<u>N</u> ext Bookmark	F2
	<u>P</u> revious Bookmark	Shift+F2
	<u>F</u> irst Bookmark	Ctrl+K, F2
	<u>L</u> ast Bookmark	Ctrl+K, Shift+F2
	<u>C</u> lear All Bookmarks	Ctrl+Shift+F2

Bookmarks menu commands

Menu command	Keystroke	Description
Toggle Bookmark	Ctrl+F2	Inserts or removes a bookmark on the cursor line.
Next Bookmark	F2	Moves the cursor to the next bookmark in the document. If there is no following bookmark, the cursor is placed at the first bookmark in the document.
Previous Bookmark	Shift+F2	Moves the cursor to the previous bookmark in the document. If there is no previous bookmark, the cursor is placed at the last bookmark in the document.
First Bookmark	Ctrl+K, F2	Moves the cursor to the first bookmark in the document.
Last Bookmark	Ctrl+K, Shift+F2	Moves the cursor to the last bookmark in the document.
Clear All Bookmarks	Ctrl+Shift+F2	Removes all bookmarks from the document.

Advanced menu

The **Advanced** menu provides additional commands to edit your document.

The Advanced menu

	Undo All	Ctrl+K, Ctrl+Z
	Redo All	Ctrl+K, Ctrl+Y
	Transpose Words	Ctrl+K, Ctrl+T
	Transpose Lines	Ctrl+K, T
	Scroll To Top	Ctrl+G, Ctrl+T
	Scroll To Middle	Ctrl+G, Ctrl+M
	Scroll To Bottom	Ctrl+G, Ctrl+B
	Toggle Read Only	Ctrl+K, Ctrl+R
	Visible Whitespace	Ctrl+Shift+8

Advanced menu commands

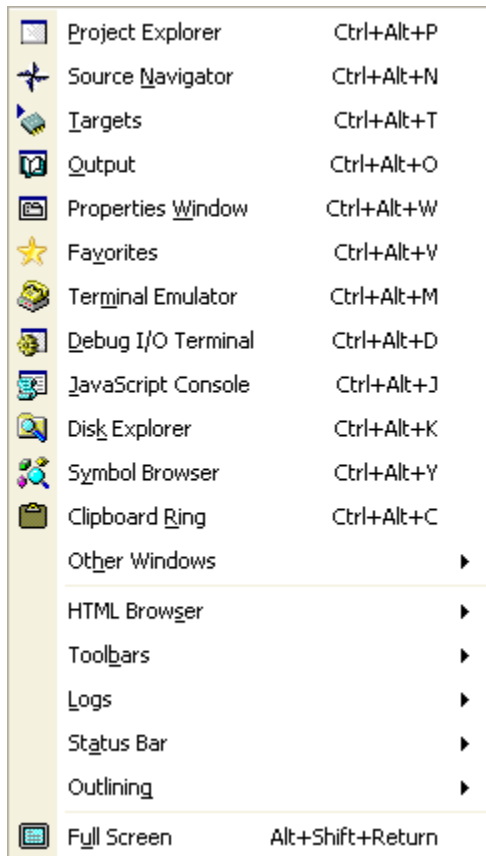
Menu command	Keystroke	Description
Undo All	Ctrl+K, Ctrl+Z	Undoes all editing actions in the document.
Redo All	Ctrl+K, Ctrl+Y	Redoes all editing actions in the document.
Transpose Words	Ctrl+K, Ctrl+T	Swaps the word at the cursor position with the preceding word.
Transpose Lines	Ctrl+K, T	Swaps the cursor line with the preceding line.
Scroll To Top	Ctrl+G, Ctrl+T	Moves the cursor line to the top of the window.
Scroll To Middle	Ctrl+G, Ctrl+M	Moves the cursor line to the middle of the window.
Scroll To Bottom	Ctrl+G, Ctrl+B	Moves the cursor line to the bottom of the window.
Toggle Read Only	Ctrl+K, Ctrl+R	Toggles the read only bit of the document.

Visible Whitespace	Ctrl+Shift+8	Toggles the document display between non-visible whitespace and visible whitespace where tabs and spaces are shown with special characters.

View menu

The **View** menu provides commands to control the way that windows and their contents are seen within CrossStudio.

The View menu



View menu commands

Menu command	Keystroke	Description
Project Explorer	Ctrl+Alt+P	Activates the Project Explorer . See Project Explorer .
Source Navigator	Ctrl+Alt+N	Activate the Source Navigator . See Source Navigator.
Targets	Ctrl+Alt+T	Activates the Targets window. See Target Window .
Output	Ctrl+Alt+O	Activates the Output window. See Output Window .

Properties Window	Ctrl+Alt+W	Activates the Properties window. See Properties Window .
Favorites	Ctrl+Alt+V	Activates the Favorites window. See Favorites Window.
Terminal Emulator	Ctrl+Alt+M	Activates the Terminal Emulator window. See Terminal Emulator Window.
Debug Console	Ctrl+Alt+D	Activates the Debug Console window.
JavaScript Console	Ctrl+Alt+J	Activates the JavaScript Console window.
Symbol Browser	Ctrl+Alt+Y	Activates the Symbol Browser window. See Symbol Browser .
Clipboard Ring	Ctrl+Alt+C	Activates the Clipboard Ring window.
Other Windows		Displays the Other Windows menu.
HTML Browser		Displays the HTML Browser menu.
Logs		Displays the Logs menu.
Status Bar		Displays the Status Bar menu.
Outlining		Displays the Outlining menu.
Full Screen	Alt+Shift+Return	Activates the Full Screen workspace.

Other Windows menu

The **Other Windows** menu provides commands to activate additional windows in CrossStudio.

The Other Windows menu

	Breakpoints	Ctrl+Alt+B
	Call Stack	Ctrl+Alt+S
	Locals	Ctrl+Alt+L
	Globals	Ctrl+Alt+G
	Threads	Ctrl+Alt+D
	Registers 1	Ctrl+T, R, 1
	Registers 2	Ctrl+T, R, 2
	Registers 3	Ctrl+T, R, 3
	Registers 4	Ctrl+T, R, 4
	Watch 1	Ctrl+T, W, 1
	Watch 2	Ctrl+T, W, 2
	Watch 3	Ctrl+T, W, 3
	Watch 4	Ctrl+T, W, 4
	Memory 1	Ctrl+T, M, 1
	Memory 2	Ctrl+T, M, 2
	Memory 3	Ctrl+T, M, 3
	Memory 4	Ctrl+T, M, 4
	Execution Trace	
	Execution Counts	

Other Windows commands

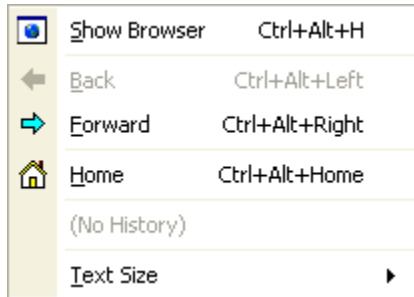
Menu command	Keystroke	Description
Breakpoints	Ctrl+Alt+B	Activates the Breakpoints window. See Breakpoints Window .
Call Stack	Ctrl+Alt+S	Activates the Call Stack window. See Call Stack Window .
Locals	Ctrl+Alt+L	Activates the Locals window. See Locals Window .
Globals	Ctrl+Alt+G	Activates the Globals window. See Globals Window .

Threads	Ctrl+Alt+D	Activates the Threads window. See Threads Window .
Registers 1	Ctrl+T, R, 1	Activates the first Register window. See Register Windows .
Registers 2	Ctrl+T, R, 2	Activates the second Register window. See Register Windows .
Registers 3	Ctrl+T, R, 3	Activates the third Register window. See Register Windows .
Registers 4	Ctrl+T, R, 4	Activates the fourth Register window. See Register Windows .
Watch 1	Ctrl+T, W, 1	Activates the first Watch window. See Watch Windows .
Watch 2	Ctrl+T, W, 2	Activates the second Watch window. See Watch Windows .
Watch 3	Ctrl+T, W, 3	Activates the third Watch window. See Watch Windows .
Watch 4	Ctrl+T, W, 4	Activates the fourth Watch window. See Watch Windows .
Memory 1	Ctrl+T, M, 1	Activates the first Memory window. See Memory Windows .
Memory 2	Ctrl+T, M, 2	Activates the second Memory window. See Memory Windows .
Memory 3	Ctrl+T, M, 3	Activates the third Memory window. See Memory Windows .
Memory 4	Ctrl+T, M, 4	Activates the fourth Memory window. See Memory Windows .
Execution Trace		Activates the Execution Trace window. See Execution Trace Window .
Execution Counts		Activates the Execution Counts window. See Execution Count Window .

Browser menu

The **Browser** menu provides commands nagivate through the browser history.

The Browser menu



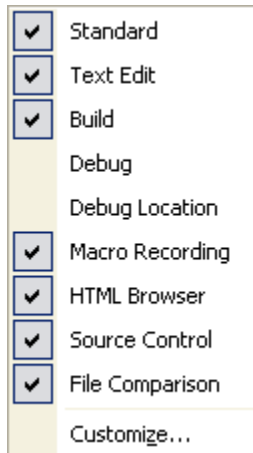
Browser commands

Menu command	Keystroke	Description
Show Browser	Ctrl+Alt+H	Activates the Browser window.
Back	Ctrl+Alt+Left	Displays the previous page in the browser history.
Forward	Ctrl+Alt+Right	Displays the following page in the browser history.
Home	Ctrl+Alt+Home	Displays the home page.
Text Size		Displays the Browser Text Size menu.

Toolbars menu

The **Toolbars** menu provides commands to display or hide CrossStudio tool bars.

The Toolbars menu



Toolbar menu commands

Menu command	Keystroke	Description
Standard		Displays the Standard tool bar.
Text Edit		Displays the Text Edit tool bar.
Build		Displays the Build tool bar.
Debug		Displays the Debug tool bar.
Debug Location		Displays the Debug Location tool bar.
Macro Recording		Displays the Macro Recording tool bar.
HTML Browser		Displays the HTML Browser tool bar.
Source Control		Displays the Source Control tool bar.
File Comparison		Displays the File Comparison tool bar.
Customize...		Displays the Toolbar Configuration dialog.

Search menu

The **Search** menu provides commands to search in files.

The Search menu

	Find...	Ctrl+F
	Find in Files...	Ctrl+Shift+F
	Replace...	Ctrl+H
	Replace in Files...	Ctrl+Shift+H
	Find Next	F3
	Find Previous	Shift+F3
	Find Selected Text	Ctrl+F3
	Find and Mark All	Alt+Shift+F3
	Go To Line...	Ctrl+G, Ctrl+L
	Go To Mate	Ctrl+]
	Next Location	F4
	Previous Location	Shift+F4
	Next Function	Ctrl+PgDown
	Previous Function	Ctrl+PgUp
	Case Sensitive Matching	Ctrl+K, Ctrl+F, C
	Whole Word Matching	Ctrl+K, Ctrl+F, W
	Regular Expression Matching	Ctrl+K, Ctrl+F, X

Search menu commands

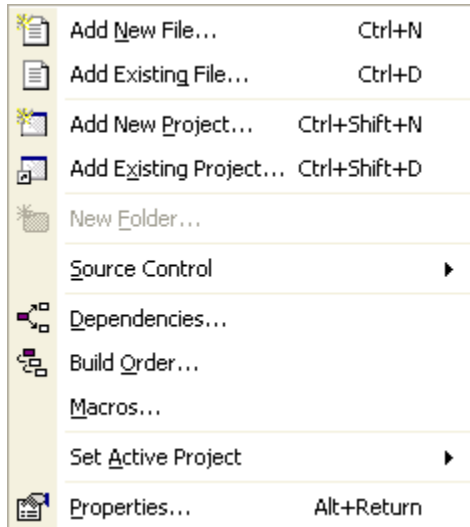
Menu command	Keystroke	Description
Find	Ctrl+F	Searches documents for strings.
Find in Files	Ctrl+Shift+F	Searches for a string in multiple files.
Replace	Replace	Replaces text with different text.
Replace in Files	Ctrl+Shift+H	Replaces text with different text in multiple files.
Find Next	F3	Searches for the next occurrence of the specified text.
Find Previous	Shift+F3	Searches for the previous occurrence of the specified text.

Find Selected Text	Ctrl+F3	Searches for the next occurrence of the selection.
Find and Mark All	Alt+Shift+F3	Searches the document for all occurrences of the specified text and marks them with bookmarks.
Go To Line	Ctrl+G, Ctrl+L	Moves the cursor to a specified line in the document.
Go To Mate	Ctrl+]	Moves the cursor to the bracket, parenthesis, or brace that matches the one at the cursor.
Next Location	F4	Moves the cursor to the line containing the next error or tag.
Previous Location	Shift+F4	Moves the cursor to the line containing the previous error or tag.
Next Function	Ctrl+PgDn	Moves the cursor to the declaration of the next function.
Previous Function	Ctrl+PgUp	Moves the cursor to the declaration of the previous function.
Case Sensitive Matching	Ctrl+K, Ctrl+F, C	Enables or disables the case sensitivity of letters when searching.
Whole Word Matching	Ctrl+K, Ctrl+F, W	Enables or disables whole word matching when searching.
Regular Expression Matching	Ctrl+K, Ctrl+F, X	Enables or disables expression matching rather than plain text matching.

Project menu

The **Project** menu provides commands to manipulate the project.

The Project menu



Project menu commands

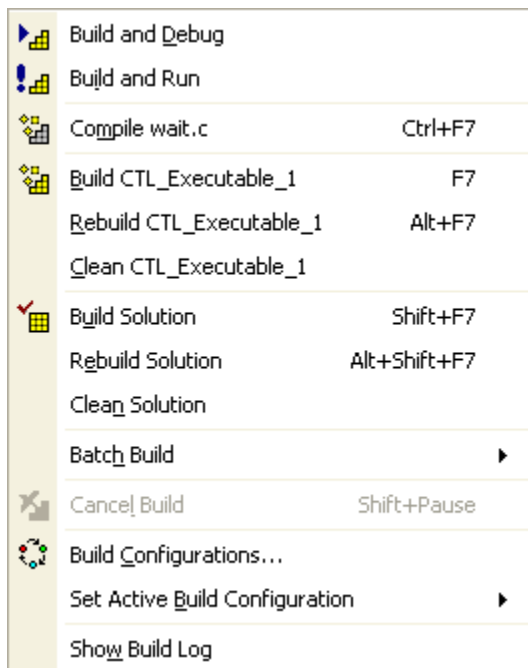
Menu command	Keystroke	Description
Add New File...	Ctrl+N	Adds a new file to the active project.
Add Existing File...	Ctrl+D	Adds an existing file to the active project.
Add New Project...	Ctrl+Shift+N	Adds a new project to the solution.
Add Existing Project	Ctrl+Shift+D	Adds a link to an existing project to the solution.
New Folder...		Adds a new folder to the current project or folder.
Source Control		Displays the Source Control menu.
Dependencies...		Displays the Project Dependencies dialog to alter project dependencies.

Build Order...		Displays the Build Order tab of the Project Dependencies dialog.
Macros...		Displays the Project Macros dialog to edit the macros defined in a project.
Set Active Project		Displays a menu which allows you to select the active project.
Properties	Alt+Return	Displays the Project Properties dialog for the current project item.

Build menu

The **Build** menu provides commands to build projects and solutions.

The Build menu



Build menu commands

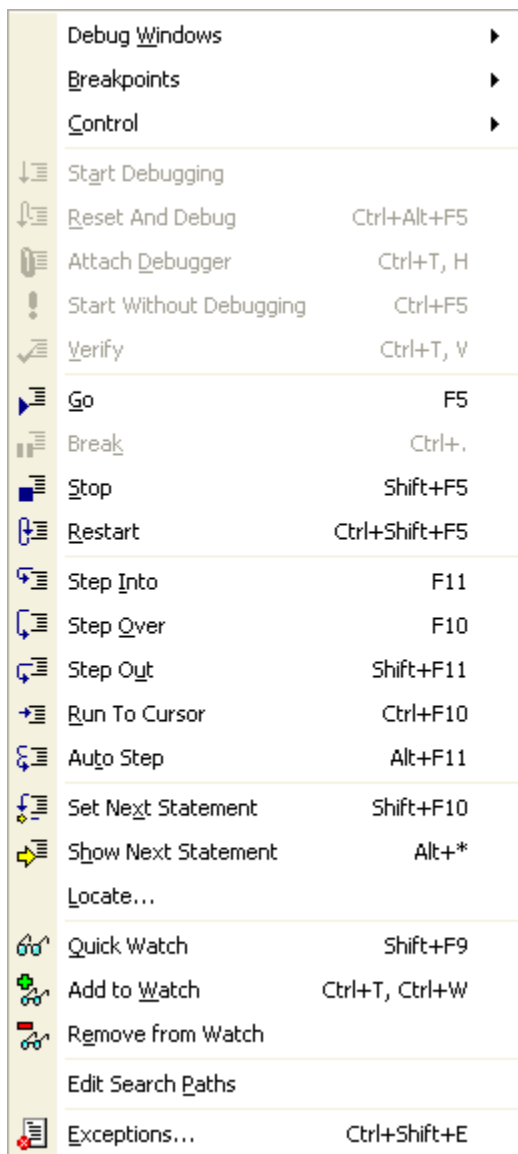
Menu command	Keystroke	Description
Build and Debug		Builds the active project and starts debugging it.
Build and Run		Builds the active project and runs it without debugging.
Compile <i>file</i>	Ctrl+F7	Compiles the selected project file.
Build <i>project</i>	F7	Builds the active project.
Rebuild <i>project</i>	Alt+F7	Rebuilds the active project.
Clean <i>project</i>		Removes all output and temporary files generated by the active project.
Build Solution	Shift+F7	Builds all projects in the solution.

Rebuild Solution	Alt+Shift+F7	Rebuilds all projects in the solution.
Clean Solution		Removes all output and temporary files generated by all projects in the solution.
Batch Build		Displays the Batch Build menu.
Cancel Build	Shift+Pause	Stops any build in progress.
Build Configurations...		Displays the Build Configurations dialog.
Set Active Build Configuration		Displays a menu which allows you to select the active build configuration.
Show Build Log		Displays the Build Log in the Output window.

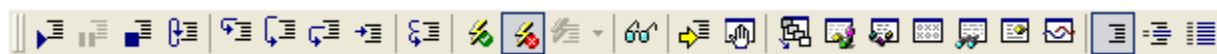
Debug menu

The **Debug** menu provides commands to download, run, and debug your application. You can find common debug actions as tool buttons on the **Debug** toolbar.

The Debug menu



The Debug toolbar



Debug commands

Menu command	Keystroke	Description

Debug Windows		Displays the Debug Windows menu. See Debug Windows menu .
Breakpoints		Displays the Breakpoints menu. See Breakpoints menu .
Control		Displays the Debug Control menu. See Debug Control menu .
Start Debugging	F5	Downloads the program to the selected target interface and starts running the program under control of the debugger.
Reset and Debug	Ctrl+Alt+F5	Resets the selected target interface without downloading the project and starts running the program.
Attach Debugger	Ctrl+T, H	Attaches the debugger to the program running on the selected target interface.
Start Without Debugging	Ctrl+F5	Downloads the program to the selected target interface and starts running the program without the debugger.
Go	F5	Continues running the program until a breakpoint is hit or a hardware exception is raised.
Break	Ctrl+.	Stops the program running and returns control to the debugger.
Stop	Shift+F5	Stops debugging the program and returns to the editing workspace.
Restart	Ctrl+Shift+F5	Resets the selected target interface and starts debugging the program.

Step Into	F11	Steps into the next statement or instruction and enters C functions and assembly language subroutines. If a breakpoint is hit when stepping, the debugger immediately stops at that breakpoint.
Step Over	F10	Steps over the next statement or instruction without entering C functions and assembly language subroutines. If a breakpoint is hit when stepping, the debugger immediately stops at that breakpoint.
Step Out	Shift+F11	Steps out of the function or subroutine by executing up to the instruction following the call to the current function or subroutine. If a breakpoint is hit when stepping, the debugger immediately stops at that breakpoint.
Run To Cursor	Ctrl+F10	Runs the program to the statement or instruction the cursor is at. If a breakpoint is hit when stepping, the debugger immediately stops at that breakpoint.
Auto Step	Alt+F11	Animates program execution by running the program and updates all debugger windows after each statement or instruction executed.
Set Next Statement	Shift+F10	Sets the program counter to the statement or instruction that the cursor is on. Note that doing this may lead to unpredictable or incorrect execution of your program.

Show Next Statement	Alt+*	Displays the source line or instruction associated with the program counter. You can use this to show the execution point after navigating through files.
Locate...		
Quick Watch	Shift+F9	Opens a viewer on the variable or expression at the cursor position. If no text is selected, CrossStudio opens a viewer using the word at the cursor position as the expression. If some text is selected, CrossStudio opens a viewer using the selected text as the expression.
Add To Watch	Ctrl+T, Ctrl+W	Adds the variable or expression at the cursor position to the last activated watch window. If no text is selected, CrossStudio adds the word at the cursor position to the watch window. If some text is selected, CrossStudio adds the selected text as the expression to the watch window.
Remove From Watch		Removes the variable or expression at the cursor position to the last activated watch window. If no text is selected, CrossStudio removes any expression matching the word at the cursor position from the watch window. If some text is selected, CrossStudio removes any expression matching the selected text from the watch window.
Edit Search Paths		Opens the Debug Search File dialog. See Debug Search File Dialog .
Exceptions		Opens the Exceptions dialog.

Debug Control menu

The **Debug Control** menu provides commands to control how you debug your program. The **Debug Control** menu is a submenu of the **Debug** menu.

The Debug Control menu

	Source Mode	Ctrl+T, S
	Interleaved Mode	Ctrl+T, I
	Assembly Mode	Ctrl+T, A
	Toggle Debug Mode	Ctrl+F11
	Enable Interrupt Processing	Ctrl+T, N
	Disable Interrupt Processing	Ctrl+T, X
	Start Cycle Counter	
	Pause Cycle Counter	
	Reset Cycle Counter	

Debug Control commands

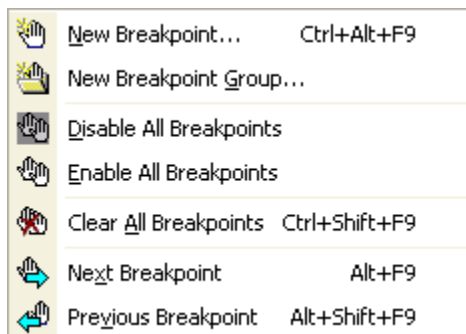
Menu command	Keystroke	Description
Source Mode	Ctrl+T, S	Switches the debugger into Source Debugging mode where code is stepped a statement at a time.
Interleaved Mode	Ctrl+T, I	Switches the debugger into Interleaved Debugging mode where code is stepped one instruction at a time and source code is intermixed with the generated assembly code.
Assembly Mode	Ctrl+T, A	Switches the debugger into Assembly Debugging mode where code is stepped one instruction at a time with a simple disassembly of memory.
Toggle Debug Mode	Ctrl+F11	Toggles between Source Debugging and Interleaved Debugging modes.
Enable Interrupt Processing	Ctrl+T, N	Enables global interrupts in the processor by writing to the appropriate register.

Disable Interrupt Processing	Ctrl+T, X	Disables global interrupts in the processor by writing to the appropriate register.
Start Cycle Counter		Restarts the cycle counter after it has been paused.
Pause Cycle Counter		Pauses the cycle counter so that it does not increment and count cycles even though code executes.
Reset Cycle Counter		Resets the cycle counter to zero.

Breakpoint menu

The **Breakpoint** menu provides commands to create, modify, and remove breakpoints. The **Breakpoint** menu is a submenu of the **Debug** menu.

The Breakpoint menu



Breakpoint commands

Menu command	Keystroke	Description
New Breakpoint...	Ctrl+Alt+F9	Activates the New Breakpoint menu which allows you to create complex breakpoints on code or data. See Breakpoints Window .
New Breakpoint Group...		Creates a new breakpoint group in the Breakpoints window. You can manage breakpoints individually or as a group.
Disable All Breakpoints		Disables all breakpoints so that they are never hit.
Enable All Breakpoints		Enables all breakpoints so that they can be hit.
Clear All Breakpoints	Ctrl+Shift+F9	Removes all breakpoints set in the Breakpoints window.
Next Breakpoint	Alt+F9	Selects the next breakpoint in the Breakpoint window and moves the cursor to the statement or instruction associated with that breakpoint.

Previous Breakpoint	Alt+Shift+F9	Selects the previous breakpoint in the Breakpoint window and moves the cursor to the statement or instruction associated with that breakpoint.

Debug Windows menu

The **Debug Windows** menu provides commands to activate debugging windows. The **Debug Windows** menu is a submenu of the **Debug** menu.

The Debug Windows menu

	Breakpoints	Ctrl+Alt+B
	Call Stack	Ctrl+Alt+S
	Locals	Ctrl+Alt+L
	Globals	Ctrl+Alt+G
	Threads	Ctrl+Alt+D
	Registers 1	Ctrl+T, R, 1
	Registers 2	Ctrl+T, R, 2
	Registers 3	Ctrl+T, R, 3
	Registers 4	Ctrl+T, R, 4
	Watch 1	Ctrl+T, W, 1
	Watch 2	Ctrl+T, W, 2
	Watch 3	Ctrl+T, W, 3
	Watch 4	Ctrl+T, W, 4
	Memory 1	Ctrl+T, M, 1
	Memory 2	Ctrl+T, M, 2
	Memory 3	Ctrl+T, M, 3
	Memory 4	Ctrl+T, M, 4
	Execution Trace	
	Execution Counts	

Debug Windows commands

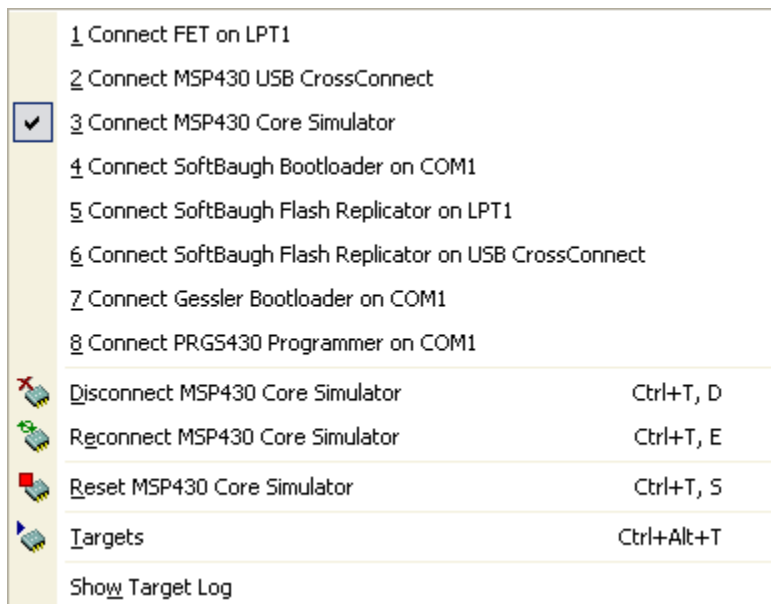
Menu command	Keystroke	Description
Breakpoints	Ctrl+Alt+B	Activates the Breakpoints window. See Breakpoints Window .
Call Stack	Ctrl+Alt+S	Activates the Call Stack window. See Call Stack Window .
Locals	Ctrl+Alt+L	Activates the Locals window. See Locals Window .
Globals	Ctrl+Alt+G	Activates the Globals window. See Globals Window .

Threads	Ctrl+Alt+D	Activates the Threads window. See Threads Window .
Registers 1	Ctrl+T, R, 1	Activates the first Register window. See Register Windows .
Registers 2	Ctrl+T, R, 2	Activates the second Register window. See Register Windows .
Registers 3	Ctrl+T, R, 3	Activates the third Register window. See Register Windows .
Registers 4	Ctrl+T, R, 4	Activates the fourth Register window. See Register Windows .
Watch 1	Ctrl+T, W, 1	Activates the first Watch window. See Watch Windows .
Watch 2	Ctrl+T, W, 2	Activates the second Watch window. See Watch Windows .
Watch 3	Ctrl+T, W, 3	Activates the third Watch window. See Watch Windows .
Watch 4	Ctrl+T, W, 4	Activates the fourth Watch window. See Watch Windows .
Memory 1	Ctrl+T, M, 1	Activates the first Memory window. See Memory Windows .
Memory 2	Ctrl+T, M, 2	Activates the second Memory window. See Memory Windows .
Memory 3	Ctrl+T, M, 3	Activates the third Memory window. See Memory Windows .
Memory 4	Ctrl+T, M, 4	Activates the fourth Memory window. See Memory Windows .
Execution Trace		Activates the Execution Trace window. See Execution Trace Window .
Execution Counts		Activates the Execution Counts window. See Execution Count Window .

Target menu

The **Target** menu provides commands to manipulate the project.

The Target menu



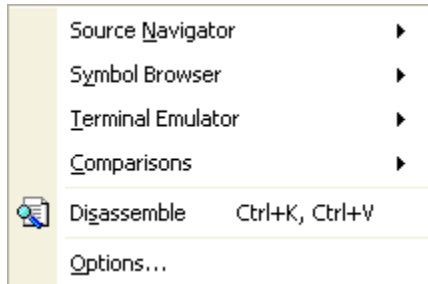
Target menu commands

Menu command	Keystroke	Description
Connect <i>interface</i>		Connects the selected target interface.
Disconnect <i>interface</i>	Ctrl+T, D	Disconnects the connected target interface.
Reconnect <i>interface</i>	Ctrl+T, E	Reconnects the connected target interface.
Reset <i>interface</i>	Ctrl+T, S	Resets the target connected to the selected target interface.
Targets	Ctrl+Alt+T	Activates the Target window. See Target Window .
Show Target Log		Activates the Target Log in the Output window.

Tools menu

The **Tools** menu provides setup and configuration of CrossStudio.

The Tools menu



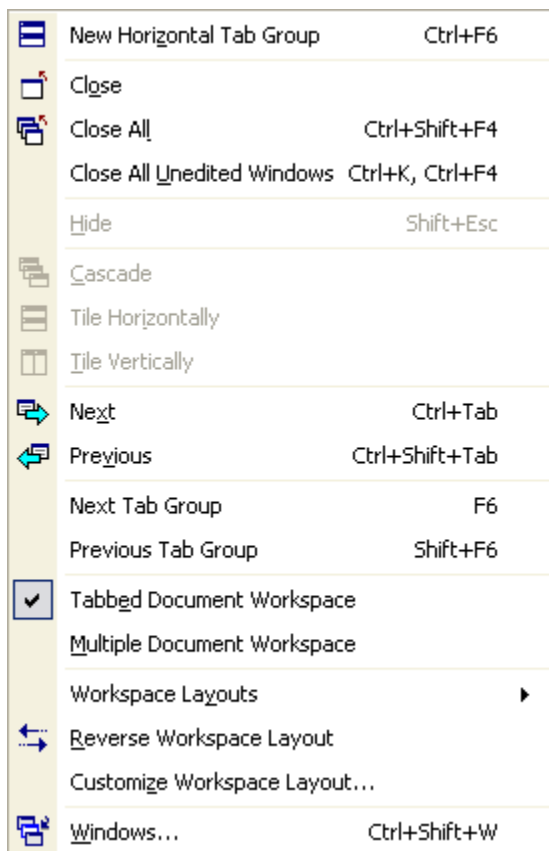
Tools menu commands

Menu command	Keystroke	Description
Source Navigator		Displays the Source Navigator configuration menu.
Symbol Browser		Displays the Symbol Browser configuration menu.
Terminal Emulator		Displays the Terminal Emulator configuration menu.
Comparisons		Displays the Comparisons menu.
Disassemble	Ctrl+K, Ctrl+V	Disassembles the selected project item.
Options...		Displays the Environment Options dialog.

Window menu

The **Window** menu provides commands to control windows within CrossStudio.

The Window menu



Window menu commands

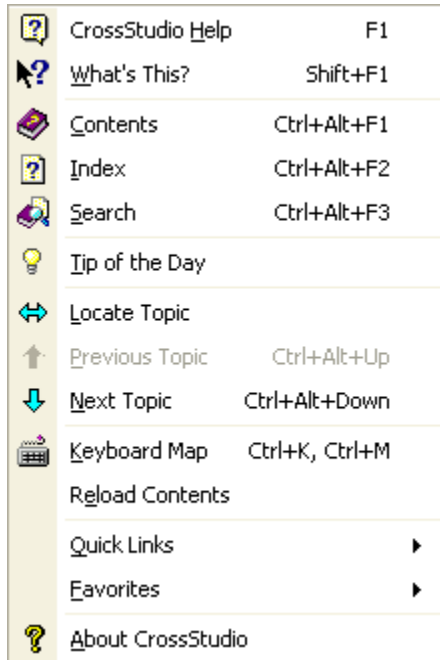
Menu command	Keystroke	Description
New Horizontal Tab Group	Ctrl+F6	Splits the tab group in two at the active tab and creates two tab groups in tabbed document workspace mode.
Close	Ctrl+F4	Closes the active document window.
Close All	Ctrl+Shift+F4	Closes all document windows.
Close All Unedited Windows	Ctrl+K, Ctrl+F4	Closes all document windows that have not been changed.
Hide <i>window</i>		Hides the focused dock window.

Cascade		Cascades windows in multiple document interface mode.
Tile Horizontally		Tiles windows horizontally in multiple document interface mode.
Next	Ctrl+Tab	Activates the next window in the tab group or window stack.
Previous	Ctrl+Shift+Tab	Activates the previous window in the tab group or window stack.
Next Tab Group	F6	Activates the next tab group in tabbed document interface mode.
Previous Tab Group	Shift+F6	Activates the next tab group in tabbed document interface mode.
Tabbed Document Workspace		Enables the tabbed document workspace.
Multiple Document Workspace		Enables the multiple document workspace.
Workspace Layouts		Displays the Workspace Layout menu which allows selection of various workspace layouts.
Reverse Workspace Layout		Reverses the left and right dock areas.
Customize Workspace Layout...		Displays the Document Workspace Layout dialog.
Windows...		Displays the Windows dialog.

Help menu

The **Help** menu provides access to online help for CrossStudio.

The Help menu



Help menu commands

Menu command	Keystroke	Description
CrossStudio Help	F1	Displays online help for the focused GUI element.
What's This	Shift+F1	Enters What's This? mode which provides a short description of each GUI element.
Contents	Ctrl+Alt+F1	Activates the Contents window.
Index	Ctrl+Alt+F2	Activates the Index window.
Search	Ctrl+Alt+F3	Activates the Search window.
Tip of the Day		Activates the Tip of the Day window.
Locate Topic		Locates the help page displayed by the browser in the Contents window.

Previous Topic	Ctrl+Alt+Up	Moves to the previous topic in the Contents window and updates the browser.
Next Topic	Ctrl+Alt+Down	Moves to the next topic in the Contents window and updates the browser.
Keyboard Map	Ctrl+K, Ctrl+M	Displays the Keyboard Map dialog.
Quick Links		Displays the Quick Links menu which contains useful shortcuts to the online manual.
Favorites		Displays the web pages from the Favorites window.
About CrossStudio		Displays information on CrossStudio, the license agreement, and activation status.

C Compiler Reference

CrossWorks C is a faithful implementation of the ANSI and ISO standards for the programming language C. This manual describes the C language as implemented by the CrossWorks C compiler.

In this section

Command line options

Describes the command line options accepted by the CrossWorks compiler.

Functions

Describes the different types of functions supported by the CrossWorks compiler, such as interrupt functions and top-level functions.

Strings

Describes the additional string types which make programming more comfortable in some application domains

Pragmas

Describes the specific pragmas recognized by the CrossWorks compiler.

Section reference

Describes the sections that the CrossWorks compiler uses.

Preprocessor

Describes the symbols defined by the preprocessor.

Customizing runtime behavior

Describes how to modify and extend the runtime behavior of CrossWorks applications.

Diagnostics

Describes the diagnostics reported by the preprocessor and compiler.

Extension summary

Summarises the extensions provided by the CrossWorks compiler.

Command line options

This section describes the command line options accepted by the CrossWorks C compiler.

-D (Define macro symbol)

Syntax

-D name

-D name = value

Description

You can define preprocessor macros using the **-D** option. The macro definitions are passed on to the respective language compiler which is responsible for interpreting the definitions and providing them to the programmer within the language.

The first form above defines the macro **name** but without an associated replacement value, and the second defines the same macro with the replacement value **value**.

Setting this in CrossStudio

To define preprocessor macros for a project:

- Select the project in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **Preprocessor Definitions** property.

To define preprocessor macros for a particular file:

- Select the file in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **Preprocessor Definitions** property.

The **Preprocessor Definitions** property is a semicolon-separated list of macro definitions, for example **"name1=value1;name2=value2"**. Clicking the button at the right of the property displays the **Preprocessor Definitions** dialog which will allow you to easily edit the definitions.

Example

The following defines two macros, **SUPPORT_FLOAT** with a value of 1 and **LITTLE_ENDIAN** with no replacement value.

```
-DSUPPORT_FLOAT=1 -DLITTLE_ENDIAN
```

-g (Generate debugging information)

Syntax

-g

Description

The **-g** option instructs the compiler to generate debugging information (line numbers and data type information) for the debugger to use.

Setting this in CrossStudio

To set include debugging information for all files in a project:

- Select the project in the **Project Explorer**.
- In the **General Options** group set the **Include Debug Information** property to **Yes** or **No** as appropriate.

To set include debugging information for a particular file (**not recommended**):

- Select the file in the **Project Explorer**.
- In the **General Options** group set the **Include Debug Information** property to **Yes** or **No** as appropriate.

-I (Define user include directories)

Syntax

-I directory

In order to find include files the compiler driver arranges for the compilers to search a number of standard directories. You can add directories to the search path using the **-I** switch which is passed on to each of the language processors.

Setting this in CrossStudio

To set the directories searched for include files for a project:

- Select the project in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **User Include Directories** property.

To set the directories searched for include files for a particular file:

- Select the file in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **User Include Directories** property.

Example

To tell the compiler to search the directories **../include** and **../lib/include** for included files when compiling **file.c** use the following:

```
hcl -I../include -I../lib/include file.c
```

You can specify more than one include directory by separating each directory component with either a comma or semicolon, so the following command lines have the same effect as the one above.

```
hcl -I../include,../lib/include file.c  
hcl -I../include;../lib/include file.c
```

See Also

[Exclude Standard Include Directories \(-I-\)](#)

-J (Define system include directories)

Syntax

-J directory

The **-J** option adds **directory** to the end of the list of directories to search for source files included (using triangular brackets) by the **#include** preprocessor command.

Example

For example, to tell the compiler to search the directories **../include** and **../lib/include** for included system files when compiling **file.c** and writing the output to **file.hzo** you could use the following:

```
hcc -J../include -J../lib/include file.c -o file.hzo file.c
```


-mmpy (Enable hardware multiplier)

Syntax

-mmpy

Description

This option instructs the compiler to generate code that can use the MSP430 hardware multiplier. By default, the hardware multiplier is not used and all integer and floating-point multiplications are carried out by software loops.

When using the hardware multiplier, the compiler ensures that no interrupts occur during the time the multiplier is in use. Global interrupts are disabled during a multiplication to prevent, for instance, an interrupt being taken immediately after the multiplication is complete but before the result has been loaded which could possibly corrupt the result of the multiplication. Because interrupts are disabled during hardware-assisted multiplication, interrupt latency is increased—if you wish to have the lowest possible interrupt latency, then do not enable the hardware multiplier and use soft multiplication instead.

The CrossWorks compiler generates inline code to use the hardware multiplier for 16-bit multiplications and calls out-of-line subroutines for all other multiplications. The runtime library also uses the hardware multiplier to accelerate multiplication of floating-point values.

Setting this in CrossStudio

To use the hardware multiplier for a project:

- Select the project in the **Project Explorer**.
- In the **Compiler Options** group set the **Use Hardware Multiplier** property to **Yes**.

It is not possible to set the **Use Hardware Multiplier** property on a per-file basis.

Special notes

There is no means to prevent a non-maskable interrupt from occurring, so you must be very careful not to use the hardware multiplier in any NMI interrupt service routines.

See also

[-mmpyinl \(Enable inline hardware multiplier\)](#)

-mmpyintl (Enable inline hardware multiplier)

Syntax

-mmpyintl

Description

This option instructs the compiler to generate inline code that can use the MSP430 hardware multiplier. By default, the hardware multiplier is not used and all integer and floating-point multiplications are carried out by software loops.

When using the hardware multiplier, the compiler ensures that no interrupts occur during the time the multiplier is in use. Global interrupts are disabled during a multiplication to prevent, for instance, an interrupt being taken immediately after the multiplication is complete but before the result has been loaded which could possibly corrupt the result of the multiplication. Because interrupts are disabled during hardware-assisted multiplication, interrupt latency is increased—if you wish to have the lowest possible interrupt latency, then do not enable the hardware multiplier and use soft multiplication instead.

The CrossWorks compiler generates inline code to use the hardware multiplier for 16-bit multiplications and calls out-of-line subroutines for all other multiplications. The runtime library also uses the hardware multiplier to accelerate multiplication of floating-point values.

Setting this in CrossStudio

To use the hardware multiplier for a project:

- Select the project in the **Project Explorer**.
- In the **Compiler Options** group set the **Use Hardware Multiplier** property to **Inlines**.

It is not possible to set the **Use Hardware Multiplier** property on a per-file basis.

Special notes

There is no means to prevent a non-maskable interrupt from occurring, so you must be very careful not to use the hardware multiplier in any NMI interrupt service routines.

See also

[-mmpy \(Enable hardware multiplier\)](#)

-msd (Treat double as float)

Syntax

-msd

Description

This option directs the compiler to treat **double** as **float** and not to support 64-bit floating point arithmetic.

Setting this in CrossStudio

To treat **double** as **float** for a project:

- Select the project in the **Project Explorer**.
- In the **Compiler Options** group set the **Treat 'double' as 'float'** property to **Yes**.

It is not possible to set the **Treat 'double' as 'float'** property on a per-file basis

-o (Set output file name)

Syntax

-o filename

Description

The **-o** option instructs the compiler to write its object file to **filename**.

-O (Optimize code generation)

Syntax

-O[level]

Description

Optimize at level **level**. **level** must be between -9 and +9. Negative values of **level** optimize code space at the expense of speed, whereas positive values of **level** optimize for speed at the expense of code space. The '+' sign for positive optimization levels is accepted but not required.

The exact strategies used by the compiler to perform the optimization will vary from release to release and are not described here.

-Or- (Disable register allocation)

Syntax

-Or-

Description

The **-Or-** option disables all allocation of values and addresses to processor registers.

Setting this in CrossStudio

To disable all allocations of values and addresses to processor register for a project:

- Select the project in the **Project Explorer**.
- In the **Optimization Options** group set the **Register Allocation** property to **None**.

To disable all allocations of values and addresses to processor register for a particular file:

- Select the file in the **Project Explorer**.
- In the **Optimization Options** group set the **Register Allocation** property to **None**.

See Also

[-Orl \(Register allocation of locals\)](#), [-Org \(Register allocation of locals and global addresses\)](#)

-Org (Register allocation of locals and global addresses)

Syntax

-Org

Description

The **-Org** option enables allocation of local variables and addresses of global variables and functions to processor registers for the lifetime of a function. This form of register allocation will always reduce code size but may reduce execution speed for some paths through the function.

Setting this in CrossStudio

To enable allocation of local variables and addresses of global variables and functions to processor registers for a project:

- Select the project in the **Project Explorer**.
- In the **Optimization Options** group set the **Register Allocation** property to **Locals and Globals**.

To enable allocation of local variables and addresses of global variables and functions to processor registers for a particular file:

- Select the file in the **Project Explorer**.
- In the **Optimization Options** group set the **Register Allocation** property to **Locals and Globals**.

See Also

[-Or- \(Disable register allocation\)](#), [-Orl \(Register allocation of locals\)](#)

-Orl (Register allocation of locals)

Syntax

-Orl

Description

The **-Orl** option enables allocation of local variables to processor registers for the lifetime of a function. Register allocation of locals to processor registers will always reduce code size and increase execution speed.

Setting this in CrossStudio

To enable allocation of local variables to processor registers for a project:

- Select the project in the **Project Explorer**.
- In the **Optimization Options** group set the **Register Allocation** property to **Locals Only**.

To enable allocation of local variables to processor registers for a particular file:

- Select the file in the **Project Explorer**.
- In the **Optimization Options** group set the **Register Allocation** property to **Locals Only**.

See Also

[-Or-](#) (Disable register allocation), [-Org](#) (Register allocation of locals and global addresses)

-Rc (Set default code section name)

Syntax

-Rc, name

Description

The **-Rc** command line option sets the name of the default code section that the compiler emits code into. If no other options are given, the default name for the section is **CODE**.

You can control the name of the code section used by the compiler within a source file using the [codeseg pragma](#) or by using CrossStudio to set the **Code Section Name** property of the file or project.

Setting this in CrossStudio

To set the default code section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Code Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Code Section Name** property.

Example

The following command line option instructs the compiler to use the name **RAMCODE** as the default code section name and to initially generate all code into that section.

```
-Rc, RAMCODE
```

-Rd (Set default initialised data section name)

Syntax

-Rd, name

Description

The **-Rd** command line option sets the name of the default data section that the compiler emits initialized data into. If no other options are given, the default name for the section is **IDATA0**.

You can control the name of the data section used by the compiler within a source file using the [dataseg pragma](#) or by using CrossStudio to set the **Data Section Name** property of the file or project.

Setting this in CrossStudio

To set the default data section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Data Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Data Section Name** property.

Example

The following command line option instructs the compiler to use the name **NVDATA** as the default initialised section name and to initially generate all initialised data into that section.

-Rd, NVDATA

-Rk (Set default read-only data section name)

Syntax

-Rk, name

Description

The **-Rk** command line option sets the name of the default data section that the compiler emits read-only data into. If no other options are given, the default name for the section is **CONST**.

You can control the name of the read-only data section used by the compiler within a source file using the [constseg pragma](#) or by using CrossStudio to set the **Constant Section Name** property of the file or project.

Setting this in CrossStudio

To set the default constant section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Constant Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Constant Section Name** property.

Example

The following command line option instructs the compiler to use the name **ROMDATA** as the default read-only data section name and to initially generate all read-only data into that section.

-Rk, ROMDATA

-Rv (Set default vector section name)

Syntax

-Rv, name

Description

The **-Rv** command line option sets the name of the default vector table section that the compiler emits interrupt vectors into. If no other options are given, the default name for the section is **INTVEC**.

You can control the name of the vector table section used by the compiler within a source file using the [vectorseg pragma](#) or by using CrossStudio to set the **Vector Section Name** property of the file or project.

Setting this in CrossStudio

To set the default interrupt vector section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Vector Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Vector Section Name** property.

Example

The following command line option instructs the compiler to use the name **IVDATA** as the default vector section name and to initially generate all interrupt vectors into that section.

-Rv, IVDATA

-Rz (Set default zeroed data section name)

Syntax

-Rz, name

Description

The **-Rz** command line option sets the name of the default zeroed data section that the compiler emits uninitialized data into. If no other options are given, the default name for the section is **UDATA0**. Uninitialised data in **UDATA0** is set to zero on program startup.

You can control the name of the zeroed data section used by the compiler within a source file using the [zeroedseg pragma](#) or by using CrossStudio to set the **Zeroed Section Name** property of the file or project.

Setting this in CrossStudio

To set the default zeroed section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Zeroed Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Zeroed Section Name** property.

Example

The following command line option instructs the compiler to use the name **ZDATA** as the default zeroed data section name and to initially generate all uninitialised into that section.

-Rz, ZDATA

-V (Version information)

Syntax

-V

Description

The **-V** switch instructs the compiler to display its version information.

-w (Suppress warnings)

Syntax

-w

Description

This option instructs the compiler not to issue any warnings.

Setting this in CrossStudio

To suppress warnings for all files in a project:

- Select the project in the **Project Explorer**.
- In the **General Options** group set the **Suppress Warnings** property to **Yes**.

To suppress warnings for a particular file:

- Select the file in the **Project Explorer**.
- In the **General Options** group set the **Suppress Warnings** property to **Yes**.

-we (Treat warnings as errors)

Syntax

-we

Description

This option directs the compiler to treat all warnings as errors.

Setting this in CrossStudio

To suppress warnings for all files in a project:

- Select the project in the **Project Explorer**.
- In the **General Options** group set the **Treat Warnings as Errors** property to **Yes**.

To suppress warnings for a particular file:

- Select the file in the **Project Explorer**.
- In the **General Options** group set the **Treat Warnings as Errors** property to **Yes**.

Functions

This section describes the way in which code is generated and the models that the CrossWorks compiler uses.

Interrupt functions

It's common for embedded systems to be **real time systems** which need to process information as it arrives and take some action immediately. Processors provide **interrupts** specifically for this, where normal program execution is suspended whilst an interrupt service routine is executed, finally returning to normal program execution when the interrupt is finished.

The MSP430 supports many interrupt sources; these are chip-specific and you can find the exact interrupt sources from each processor's data sheet.

You define an interrupt function just like a standard C function, but in addition you tell the compiler that it is an interrupt function and optionally which vectors to use. The compiler generates the correct return sequence for the interrupt and saves any registers that are used by the function. Note that the name of the interrupt function is not significant in any way.

Initializing a single interrupt vector

```
void handle_timer_interrupt(void) __interrupt[TIMER_VECTOR]
{
    /* Handle interrupt here */
}
```

This constructs an interrupt function called **handle_timer_interrupt** and initializes **TIMER_VECTOR** in the processor's interrupt vector table to point to **handle_timer_interrupt**.

Initializing a multiple interrupt vectors

```
void
handle_spurious_interrupt(void) __interrupt[UART0_RX_VECTOR,
                                           UART0_TX_VECTOR,
                                           ACCVIO_VECTOR]
{
    /* Handle interrupt here */
}
```

This constructs an interrupt function called **handle_spurious_interrupt** and initializes the three vectors **UART0RX_VECTOR**, **UART0_TX_VECTOR**, and **ACCVIO_VECTOR** in the processor's interrupt vector table to point to **handle_spurious_interrupt**.

Simple interrupt handler

```
void handle_pluggable_interrupt(void) __interrupt
{
    /* Handle interrupt here */
}
```

This constructs an interrupt function called **handle_pluggable_interrupt** but does not initialize the interrupt vector table. This style of interrupt function is useful when you plug different interrupt routines into a RAM-based table to dynamically change interrupt handlers when the application runs.

Monitor functions

In embedded systems it's common for access to critical system structures to be protected by disabling and the enabling interrupts so that interrupt service routines are not executed during the update. You can write your own code to do this using the `__disable_interrupt` and `__set_interrupt` intrinsic functions like this:

```
void update_critical_resource(void)
{
    // Disable interrupts and save previous interrupt enable state
    unsigned state = __disable_interrupt();

    // Update your critical resource here...
    task_list = task_list->next; // just an example

    // Restore interrupt state on entry
    __set_interrupt(state);
}
```

If you disabled and enabled interrupts using `__disable_interrupt` and `__enable_interrupt`, rather than using `__disable_interrupt` and `__set_interrupt` as above, calling the function with interrupts disabled would re-enable interrupts on return which is usually not what you want. If you write your code in the same fashion as above you can call the function and be sure that it's run with interrupts disabled and that on return the interrupt enable state is as it was before the call.

Because this type of function is so common, CrossWorks provides the `__monitor` keyword. Using `__monitor` the example above becomes:

```
void update_critical_resource(void) __monitor
{
    // Update your critical resource here...
    task_list = task_list->next; // just an example
}
```

The compiler generates better code using the `__monitor` compares to using the `__disable_interrupt` and `__set_interrupt` intrinsic functions.

Top-level functions

Usually the compiler saves and restores registers in a function according to the calling convention and, in almost all cases, this is exactly what you want. However, there are some cases where it's just not necessary to save registers on entry to a function as it is a **top-level function** and will not be called directly from code. The compiler can't easily detect these cases so you can point them out using the `__toplevel` attribute.

The most common function, **main**, is a good example of a top-level function: it's only called by the runtime startup code, runs, and usually never terminates in an embedded system. As such, CrossWorks automatically marks **main** as a top-level function which instructs the code generator not to save and restore registers on entry and exit because their values are not required.

Another good example is top-level task functions when you're using the CrossWorks tasking library; in this case, you can safely declare all your task functions with the top-level attribute because none of their registers are unimportant on entry and exit. Using the top-level attribute in this way will reduce the stack requirement of the task.

Example

```
void task1(void *p) __toplevel
{
    // task code
}

void task2(void *p) __toplevel
{
    // task code
}

void main(void)
{
    ctl_task_run(&task1Task, 1, task1, 0, "task1", sizeof(task1Stack)/sizeof(unsigned),
task1Stack);
    ctl_task_run(&task2Task, 1, task2, 0, "task2", sizeof(task2Stack)/sizeof(unsigned),
task2Stack);
}
```

Strings

The CrossWorks compiler supports additional string types which make programming more comfortable in some application domains.

Code-space strings

Harvard machines such as the Atmel AVR and Dallas Semiconductor MAXQ require special compiler support for addressing data held in code space—and so CrossWorks provides the `__code` keyword to store data in code space rather than data space. This does, however, lead to some inconvenient programming when dealing with constant string data because each string needs to be named and stored into code space using `__code`. The CrossWorks compiler offers a solution using the `C` qualifier to store strings into code space rather than data space. The type of a `C`-qualified string is `'__code const char *'`.

Example

Without using `C`-qualified strings you would write:

```
void sign_on(void)
{
    const __code char message[] = "Tynadyne wiper widget, v1.0";
    printf_c(message);
}
```

Using the CrossWorks extension you can write:

```
void sign_on(void)
{
    printf_c(C"Tynadyne wiper widget, v1.0");
}
```

GSM 03.38 strings

If you are writing SIM Application Toolkit (SAT) applications for a smart card, you'll need to encode some of your string literals according to the GSM 03.38 specification for SMS strings. Because encoding strings this way is difficult and error prone, CrossWorks C provides a means to encode them when compiling using the **G** specification:

Example

```
void displayHello(void)
{
    const char *gsmHello = G"Hello, World";
    gsmDisplayText(gsmHello, NORMAL_PRIORITY_AUTO_CLEAR);
}
```

The characters in the GSM 03.38 (SMS) alphabet are:

```
@ £ $ ¥ è é ù ì ò Ç Ø ø Å å _ Æ
æ ß É ! " # ¤ % & ' ( ) * + , -
. / 0 1 2 3 4 5 6 7 8 9 : ; < =
> ? ¡ A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z Ä Ö Ñ
Ü Š ž a b c d e f g h i j k l m
n o p q r s t u v w x y z ä ö ñ
ü à
```

The 03.38 SMS alphabet also includes space, carriage return, and line feed so you can use `'\r'` and `'\n'` escapes in G-encoded strings.

Pragmas

This section describes the pragmas recognized by the CrossWorks C compiler.

In this section

#pragma codeseg(segment)

Set the section name used for code.

#pragma dataseg(segment)

Set the section name used for initialized data.

#pragma constseg(segment)

Set the section name used for read-only data.

#pragma zeroedseg(segment)

Set the section name used for uninitialised, zeroed data.

#pragma vectorseg(segment)

Set the section name used for interrupt vector tables.

#pragma linklib(library)

Automatically link a library.

#pragma codeseg

Synopsis

#pragma codeseg(" name " | *default*)

Description

The **codeseg** pragma sets the name of the section that the compiler emits code into. If the argument to the **codeseg** pragma is a string, a section of that name is created and the compiler emits code for all function definitions following the pragma into that section. If the argument to **codeseg** is the reserved word **default**, the compiler selects the default code section name.

The default code section name, if no other directives have been given, is **CODE**. You can change the default code section name for the whole compilation unit by using the **-Rc** command line option or by setting the **Code Section Name** property of the file or project in the CrossStudio development environment.

#pragma dataseg

Synopsis

#pragma dataseg(" name " | **default**)

Description

The **dataseg** pragma sets the name of the section that the compiler emits initialized data into. If the argument to the **dataseg** pragma is a string, a section of that name is created and the compiler emits initialized data for all following initialized statics or externals into that section. If the argument to **dataseg** is the reserved word **default**, the compiler selects the default data section name.

The default data section name, if no other directives have been given, is **IDATA0**. You can change the default data section name for the whole compilation unit by using the **-Rd** command line option or by setting the **Data Section Name** property of the file or project in the CrossStudio development environment.

#pragma constseg

Synopsis

#pragma constseg("name" | *default*)

Description

The **constseg** pragma sets the name of the section that the compiler emits read-only data into. If the argument to the **constseg** pragma is a string, a section of that name is created and the compiler emits all following read-only data into that section. If the argument to **zeroedseg** is the reserved word **default**, the compiler selects the default read-only data section name.

The default read-only data section name, if no other directives have been given, is **CONST**. You can change the default read-only data section name for the whole compilation unit by using the **-Rk** command line option or by setting the **Constant Section Name** property of the file or project in the CrossStudio development environment.

#pragma vectorseg

Synopsis

#pragma vectorseg("name" | *default*)

Description

The **vectorseg** pragma sets the name of the section that the compiler emits interrupt vector tables into. If the argument to the **vectorseg** pragma is a string, a section of that name is created and the compiler emits all interrupt vector tables into that section. If the argument to **vectorseg** is the reserved word **default**, the compiler selects the default interrupt vector section name.

The default interrupt vector section name, if no other directives have been given, is **INTVEC**. You can change the default interrupt vector section name for the whole compilation unit by using the **-Rv** command line option or by setting the **Vector Section Name** property of the file or project in the CrossStudio development environment.

#pragma zeroedseg

Synopsis

#pragma zeroedseg("name" | **default**)

Description

The **zeroedseg** pragma sets the name of the section that the compiler emits uninitialized data into. If the argument to the **zeroedseg** pragma is a string, a section of that name is created and the compiler emits uninitialized data for all following uninitialized statics or externals into that section. If the argument to **zeroedseg** is the reserved word **default**, the compiler selects the default uninitialised data section name.

The default zeroed data section name, if no other directives have been given, is **UDATA0**. You can change the default zeroed data section name for the whole compilation unit by using the **-Rz** command line option or by setting the **Zeroed Section Name** property of the file or project in the CrossStudio development environment.

#pragma linklib

Synopsis

#pragma linklib("name ")

Description

The **linklib** pragma requests the linker to automatically include a library when an object file is included into a link.

Example

The following requests that **libtcp.hza** is automatically linked into the application:

```
#pragma linklib("tcp")
```

Type-based enumerations

CrossWorks offers *type-based enumerations*, an extension to the ISO standard to set the size of enumeration types. You can use type-based enumerations to select the base type for your enumeration. Using type-based enumeration you can reduce the size of your application by using enumerations that match the size of the underlying data rather than using the default `int`-based enumeration.

Syntax

`enum [base-type]`

Where *base-type* is either a plain, **signed**, or **unsigned** variant of **char**, **int**, **long**, or **long long**.

Example

Use an 8-bit unsigned character to define an enumeration that maps onto a single byte and map that onto a byte at location `10016`:

```
enum unsigned char T0CN_t {
    M0  = 1<<0,
    M1  = 1<<1,
    CT  = 1<<2,
    GATE = 1<<3,
    TR0  = 1<<4,
    TF0  = 1<<5,
    T0M  = 1<<6,
    ET0  = 1<<7
};

enum T0CN_t T0CN @ 0x100;
```

Section reference

The CrossWorks C compiler separates generated code and data into sections so that they can be individually placed by the linker. It's the linker's job to combine, and make contiguous, sections of the same name from multiple object files.

You can change the sections that the compiler uses for individual data objects or functions using appropriate pragmas. The default section names used by the compiler are:

CODE

Contains code generated for functions. You can change the default code section name using the **-Rc** command line option—see [-Rc \(Set default code section name\)](#).

IDATA0

Contains static initialized data. You can change the default initialized data section name using the **-Rd** command line option—see [-Rd \(Set default initialised data section name\)](#).

UDATA0

Contains static zeroed (uninitialized) data. You can change the default zeroed section name using the **-Rz** command line option—see [-Rz \(Set default zeroed data section name\)](#).

CONST

Contains read-only constant data. You can change the default constant section name using the **-Rk** command line option—see [-Rk \(Set default read-only data section name\)](#).

INTVEC

Contains interrupt vector tables. You can change the default interrupt vector section name using the **-Rv** command line option—see [-Rv \(Set default vector section name\)](#).

Preprocessor

The C preprocessor provides a number of useful facilities which extend the underlying compiler. For instance, the preprocessor is responsible for finding header files in **#include** directives and for expanding the macros set using **#define**.

In many implementations the C preprocessor is a separate program from the C compiler. However, the CrossWorks C compiler has an integrated preprocessor which makes compilations faster.

Preprocessor predefined symbols

The C preprocessor defines the following macro names:

`__DATE__`

The date of translation of the program unit in the form "Mmm dd yy".

`__FILE__`

The name of the current source file. `__FILE__` expands to a string constant.

`__LINE__`

The line number of the current source line in the current source file. `__LINE__` expands to an integer constant.

`__STDC__`

The integer constant 1 as *CrossWorks C* conforms to the ISO/IEC 9899 standard. The integer constant 0 denotes that the implementation does not conform to the relevant standard.

`__STDC_HOSTED__`

The integer constant 0 as *CrossWorks C* is not a hosted implementation. The integer constant 1 denotes that the implementation is a hosted implementation.

`__STDC_VERSION__`

The integer constant 199409L as *CrossWorks C* conforms to ISO/IEC 9899:1990 with the changes required by ISO/IEC 9899/AMD1:1995. For standard C compilers conforming to ISO/IEC 9899:1999, this constant is 199901L.

`__TIME__`

The time of translation of the program unit. This expands to a string constant of the form "hh:mm:ss".

The following macro names are **not** defined by *CrossWorks C* as the implementation is still in the process of being upgraded to the 1999 standard.

`__STDC_IEC_599__`

`__STDC_IEC_599_COMPLEX__`

`__STDC_ISO_10646__`

Data representation

All data items are held in the native byte order of the MSP430 processor. The plain character type is signed by default. The floating-point types ***float*** and ***double*** are implemented as 32-bit and 64-bit IEEE floating-point.

Data Type	Size in bytes	Alignment in bytes
<i>char</i>	1	1
<i>signed char</i>	1	1
<i>unsigned char</i>	1	1
<i>int</i>	2	2
<i>unsigned int</i>	2	2
<i>short</i>	2	2
<i>unsigned short</i>	2	2
<i>long</i>	4	2
<i>unsigned long</i>	4	2
<i>long long</i>	8	2
<i>unsigned long long</i>	8	2
<i>float</i>	4	2
<i>double</i> (compiled with <i>-msd</i>)	4	2
<i>double</i>	8	2
<i>long double</i>	8	2
<i>type *</i> (pointer)	2	2
<i>enum</i> (enumeration)	2	2

External naming convention

CrossWorks makes a distinction between the low-level symbol names used for C objects and the names of the C objects themselves. The CrossWorks compiler always prepends an underscore character '_' to the name of any externally visible C function or variable when constructing its low-level symbol name.

Example

For instance, an external variable declared at the C level like this:

```
extern int var;
```

will be accessible at the assembly level like this:

```
mov.w #1, _var
```

Register usage

The compiler partitions the MSP430 general purpose registers into two sets.

The registers in the first set, **R12** through **R15**, are used for parameter passing and returning function results and are not preserved across functions calls.

The registers in the second set, **R4** through **R11**, are used for registerized variables, working storage, and temporary results and must be preserved across function calls.

Parameter passing

The compiler uses the scratch registers to pass values to the called routine for all parameters of simple data type. If there are not enough scratch registers to hold all parameter data to be passed to the called routine, the excess data are passed on the stack.

Simple data types which require more than a single word of storage are passed in register pairs or register quads. The register requirement for the basic data types are:

- The eight-bit and 16-bit type **char**, **int**, **short**, enumerations, and any pointer type require one register.
- The 32-bit types **long** and **float** (and **double** if compiled with **double** equivalent to **float**) require two registers.
- The 64-bit types **long long** and **double** require four registers.

Allocation of the scratch registers for function calls proceeds in a left-to-right fashion, starting with register **R15** and progressing in reverse order to **R12**. The compiler tries to fit each parameter into the scratch registers and, if it can, allocates those registers to the incoming parameter. If the parameter requires more scratch registers than are free, it is noted and is passed on the stack. All parameters which are passed on the stack are pushed in reverse order.

Returning values

The compiler uses the scratch registers to return values to the caller.

- The eight-bit and 16-bit type ***char***, ***int***, ***short***, enumerations, and any pointer type are returned in ***R15***.
- The 32-bit types ***long*** and ***float*** (and ***double*** if compiled with ***double*** equivalent to ***float***) are returned in the register pair ***R15:R14***, with ***R15*** holding the most significant word of the result and ***R14*** the least significant word.
- The 64-bit types ***long long*** and ***double*** are returned in the register quad ***R15:R14:R13:R12*** with ***R15*** holding the most significant word of the result and ***R12*** the least significant word.

Examples

This section contains some examples of the calling convention in use.

Example #1

Consider the function prototype:

```
void fun1(int u, int v)
```

Reading from left to right, the parameter **u** is passed in register **R15** and **v** is passed in **R14**. The scratch registers **R12** and **R13** are not used to pass parameters and can be used in **fun1** without needing to be preserved.

Example #2

Consider the function prototype:

```
void fun1(int u, long v, int w)
```

The parameter **u** is passed in register **R15**. Because **v** requires two registers to hold its value it is passed in the register pair **R14:R13** with **R14** holding the high part of **v** and **R13** the low part. The final parameter **w** is passed in **R12**.

Example #2

Consider the function prototype:

```
void fun1(int u, long v, int w, int x)
```

The parameter **u** is passed in register **R15**. Because **v** requires two registers to hold its value, it is passed in the register pair **R14:R13** with **R14** holding the high part of **v** and **R13** the low part. Parameter **w** is passed in **R12**. As all scratch registers are now used, **x** is placed onto the stack.

Example #3

Consider the function prototype:

```
void fun1(int u, long v, long w)
```

The parameter **u** is passed in register **R15**. Because **v** requires two registers to hold its value it is passed in the register pair **R14:R13** with **R14** holding the high part of **v** and **R13** the low part. When considering **w**, there is only one free scratch register left, which is **R12**. The compiler cannot fit **w** into a single register and therefore places the argument onto the stack—the compiler does not split the value into two and pass half in a register and half on the stack.

Example #4

Consider the function prototype:


```
void fun1(int u, long v, long w, int x, int y)
```

The parameter **u** is passed in register **R15**. Because **v** requires two registers to hold its value it is passed in the register pair **R14:R13** with **R14** holding the high part of **v** and **R13** the low part. When considering **w**, there is only one free scratch register left, which is **R12**. The compiler cannot fit **w** into the single register **R12** and therefore places the argument onto the stack. When considering **x**, the compiler sees that **R12** is unused and so passes **x** in **R12**. All scratch registers are used when considering **y**, so the argument is placed onto the stack. The parameters **w** and **x** are pushed onto the stack before the call and are pushed in reverse order, with **y** pushed before **w**.

This example shows two parameters, **w** and **y**, that are passed to the called routine on the stack, but they are separated by a parameter **x** that is passed in a register.

Customizing runtime behavior

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, and both 32-bit and 64-bit floating point. 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, and how to use the available hardware to the best of its ability.

In this section

Floating point implementation

Describes your options when compiling applications that require floating point.

Customizing putchar

Describes how to customize **putchar** to direct output from **printf**, **puts**, and so on.

Extending I/O library functions

Describes how to implement your own input and output functions using CrossWorks features.

Floating-point implementation

CrossWorks C allows you to choose whether the **double** data type uses the IEC 60559 32-bit or 64-bit format. The following sections describe the details of why you would want to choose a 32-bit **double** rather than a 64-bit **double** in many circumstances.

Why choose 32-bit doubles?

Many users are surprised when using **float** variables exclusively that sometimes their calculations are compiled into code that calls for **double** arithmetic. They point out that the C standard allows **float** arithmetic to be carried out only using **float** operations and not to automatically promote to the **double** data type of classic K&R C.

This is valid point. However, upon examination, even the simplest calculations can lead to **double** arithmetic. Consider:

```
// Compute sin(2x)
float sin_two_x(float x)
{
    return sinf(2.0 * x);
}
```

This looks simple enough. We're using the **sinf** function which computes the sine of a **float** and returns a **float** result. There appears to be no mention of a **double** anywhere, yet the compiler generates code that calls **double** support routines—but why?

The answer is that the constant **2.0** is a **double** constant, not a **float** constant. That is enough to force the compiler to convert both operands of the multiplication to **double** format, perform the multiplication in **double** precision, and then convert the result back to **float** precision. To avoid this surprise, the code should have been written:

```
// Compute sin(2x)
float sin_two_x(float x)
{
    return sinf(2.0F * x);
}
```

This uses a single precision floating-point constant **2.0F**. It's all too easy to forget to correctly type your floating-point constants, so if you compile your program with **double** meaning the same as **float**, you can forget all about adding the **'F'** suffix to your floating point constants.

As an aside, the C99 standard is very strict about the way that floating-point is implemented and the latitude the compiler has to rearrange and manipulate expressions that have floating-point operands. The compiler cannot second-guess user intention and use a number of useful mathematical identities and algebraic simplifications because in the world of IEC 60559 arithmetic many algebraic identities, such as $x * 1 = x$, do not hold when x takes one of the special values NaN, infinity, or negative zero.

More reasons to choose 32-bit doubles

Floating-point constants are not the only silent way that **double** creeps into your program. Consider this:

```
void write_results(float x)
{
    printf("After all that x=%f", x);
}
```

Again, no mention of a **double** anywhere, but **double** support routines are now required. The reason is that ISO C requires that **float** arguments are promoted to **double** when they are passed to the non-fixed part of variadic functions such as **printf**. So, even though your application may never mention **double**, **double** arithmetic may be required simply because you use **printf** or one of its near relatives.

If, however, you compile your code with 32-bit doubles, then there is no requirement to promote a **float** to a **double** as they share the same internal format.

Why choose 64-bit doubles?

If your application requires very accurate floating-point, more precise than the seven decimal digits supported by the **float** format, then you have little option but to use **double** arithmetic as there is no simple way to increase the precision of the **float** format. The **double** format delivers approximately 15 decimal digits of precision.

Customizing putchar

To use the standard output functions **putchar**, **puts**, and **printf**, you need to customize the way that characters are written to the standard output device. These output functions rely on a function **__putchar** that outputs a character and returns an indication of whether it was successfully written.

The prototype for **__putchar** is

```
int __putchar(int ch);
```

Sending all output to the CrossStudio virtual terminal

You can send all output to the CrossStudio virtual terminal by supplying the following implementation of the **__putchar** function in your code:

```
#include <__cross_studio_io.h>
```

```
int __putchar(int ch)
{
    return debug_putchar(ch);
}
```

This hands off output of the character **ch** to the low-level debug output routine, **debug_putchar**.

Whilst this is an adequate implementation of **__putchar**, it does consume stack space for an unnecessary nested call and associated register saving. A better way of achieving the same result is to define the low-level symbol for **__putchar** to be equivalent to the low-level symbol for **debug_putchar**. To do this, we need to instruct the linker to make the symbols equivalent.

To do this using the HCC environment:

- Select the project node in the **Project Explorer**.
- Display the **Properties Window**.
- Enter the text “**-D__putchar=debug_putchar**” into the **Additional Options** property of the **Linker Options** group. Note that there are three leading underscores in **__putchar** and a single leading underscore in **debug_putchar** because the C compiler automatically prepends an underscore to all global symbols.

To do this using the GCC environment:

- Select the project node in the **Project Explorer**.
- Display the **Properties Window**.
- Enter the text “**__putchar=debug_putchar**” into the **Linker > Linker Symbol Definitions** property of the **Linker Options** group.

Sending all output to another device

If you need to output to a physical device, such as a UART, the following notes will help you:

- If the character cannot be written for any reason, **putchar** must return EOF.

- 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).

Extending I/O library functions

The standard functions that perform input and output are the `printf` and `scanf` functions. These functions convert between internal binary and external printable data. In some cases, though, you need to read and write formatted data on other channels, such as other RS232 ports. This section shows how you can extend the I/O library to best implement these function.

Classic custom printf-style output

Assume that we need to output formatted data to two UARTs, numbered 0 and 1, and we have a functions `uart0_putc` and `__uart1_putc` that do just that and whose prototypes are:

```
int uart0_putc(int ch);
int uart1_putc(int ch);
```

These functions return a positive value if there is no error outputting the character and EOF if there was an error.

Using a classic implementation, you would use `sprintf` to format the string for output and then output it:

```
void
uart0_printf(const char *fmt, ...)
{
    char buf[80], *p;
    va_list ap;
    va_start(ap, fmt);
    vsnprintf(buf, sizeof(buf), fmt, ap);
    for (p = buf; *p; ++p)
        uart0_putc(*p);
    va_end(ap);
}
```

We would, of course, need an identical routine for outputting to the other UART. This code is portable, but it requires an intermediate buffer of 80 characters. On small systems, this is quite an overhead, so we could reduce the buffer size to compensate. Of course, the trouble with that means that the maximum number of characters that can be output by a single call to `printf_uart0` is also reduced. What would be good is a way to output characters to one of the UARTs without requiring an intermediate buffer.

CrossWorks printf-style output

CrossWorks provides a solution for just this case by using some internal functions and data types in the CrossWorks library. These functions and types are define in the header file `<__vfprintf.h>`.

The first thing to introduce is the `__printf_t` type:

```
typedef struct
{
    int is_string;
    size_t charcount;
    size_t maxchars;
    union
    {
        char *string;
        int (*output_fn)(int);
    }
}
```

```
    } u;
} __printf_t;
```

This type is used by the library functions to direct what the formatting routines do with each character they need to output. The **is_string** member discriminates the union **u** and directs whether the character should be appended to the string pointed to by **string** or output using **output_fn**. The member **charcount** counts the number of characters currently output, and **maxchars** defines the maximum number of characters output by the formatting routine **__vfprintf**.

We can use this type and function to rewrite **uart0_printf**:

```
int
uart0_printf(const char *fmt, ...)
{
    int n;
    va_list ap;
    __printf_t iod;
    va_start(ap, fmt);
    iod.is_string = 0;
    iod.maxchars = INT_MAX;
    iod.u.output_fn = uart0_putc;
    n = __vfprintf(&iod, fmt, ap);
    va_end(ap);
    return n;
}
```

This function has no intermediate buffer: when a character is ready to be output by the formatting routine, it calls the **output_fn** function in the descriptor **iod** to output it immediately. The maximum number of characters isn't limited as the **maxchars** member is set to **INT_MAX**. if you wanted to limit the number of characters output you can simply set the **maxchars** member to the appropriate value before calling **__vfprintf**.

We can adapt this function to take a UART number as a parameter:

```
int
uart_printf(int uart, const char *fmt, ...)
{
    int n;
    va_list ap;
    __printf_t iod;
    va_start(ap, fmt);
    iod.is_string = 0;
    iod.maxchars = INT_MAX;
    iod.u.output_fn = uart ? uart1_putc : uart0_putc;
    n = __vfprintf(&iod, fmt, ap);
    va_end(ap);
    return n;
}
```

Now we can use:

```
uart_printf(0, "This is uart %d\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 = __vfscanf((__scanf_t *)&iod, (const unsigned char *)fmt, a);
    va_end(a);
    return n;
}
```

Using this template, we can add functions to do additional formatted input from other UARTs or devices, just as we did for formatted output.

Diagnostics

This section is a reference to each of the windows in the CrossStudio environment.

In this section

Preprocessor warning messages

Describes the warnings reported by the preprocessor.

Preprocessor error messages

Describes the errors reported by the preprocessor.

Compiler warning messages

Describes the warnings reported by the compiler.

Compiler error messages

Describes the errors reported by the compiler.

Pre-processor warning messages

These warning messages come from the pre-processing pass of the compiler. Although the compiler and pre-processor are integrated into the same executable, it is worth distinguishing the pre-processor warning messages from those generated by the compiler proper.

bad digit 'digit' in number

When evaluating a pre-processor expression the pre-processor encountered a malformed octal, decimal, or hexadecimal number.

bad token 'token' produced by

A bad pre-processing token has been produced when using the token pasting operator `##`. This error is extremely unlikely to occur in your code.

character constant taken as not signed

Characters with ASCII codes greater than 127 are treated as unsigned numbers by the pre-processor.

end of file inside comment

The pre-processor came to the end of file whilst processing a comment. This is usually an error: comments cannot extend across source files.

multi-byte character constant undefined

Multi-byte character constants are not supported by the pre-processor when evaluating expressions.

no newline at end of file

The last character in the file is not a new line. Although this isn't an error, it may help portability of your code if you include a new line at the end of your file.

syntax error in #if/#endif

There's a general problem in the way you've used the `#if` or `#endif` control

unknown pre-processor control 'control'

The pre-processor control `# control` isn't a valid ANSI pre-processor control. Usually this is caused by a spelling error.

undefined escape '\char' in character constant

When evaluating a pre-processor expression the pre-processor encountered an escape sequence `\ char` which isn't defined by the ANSI standard.

wide character constant undefined

Wide character constants are not supported by the pre-processor when evaluating expressions.

Pre-processor error messages

These error messages come from the pre-processing pass of the compiler. Although the compiler and pre-processor are integrated into the same executable, it is worth distinguishing the pre-processor error messages from those generated by the compiler proper.

is not followed by a macro parameter

The # concatenation operator must be followed by a macro parameter name.

occurs at border of replacement

The ## operator cannot be placed at the end of a line.

#defined token is not a name

The token defined immediately after #define is not a valid pre-processor identifier.

#defined token 'token' can't be redefined

You cannot redefine a number of standard tokens such as __LINE__ and __STDC__. The token you're trying to redefine is one of these.

bad ?: in #if/#elif

There is an error parsing the ternary ?: operator in an expression. This is usually caused by mismatched parentheses or forgetting one of the ? or : separators.

bad operator 'operator' in #if/#elif

The operator operator is not allowed in pre-processor expressions.

bad syntax for 'defined'

The `defined` standard pre-processor function does not conform to the syntax `defined( name )`.

can't find include file 'file'

The include file `file` can't be found in any of the directories specified in compilation.

disagreement in number of macro arguments to 'name'

The macro `name` has been invoked with either too few or too many actual arguments according to its formal argument list.

duplicate macro argument 'name'

The macro argument `name` has been given twice in the argument list of a `#define` pre-processor control.

end of file in macro argument list

The pre-processor encountered the end of file whilst processing the argument list of a macro.

illegal operator * or & in #if/#elif

The pointer dereference operator `*` and the address-of operator `&` cannot be used in pre-processor expressions.

insufficient memory

The pre-processor has run out of memory. This is a very unlikely error message, but if it does occur you should split up the file you are trying to compile into several smaller files.

macro redefinition of 'name'

The macro `name` has been defined twice with two different definitions. This usually occurs when two header files are included into a C source file and macros in the header files clash.

pre-processor internal error: cause

The pre-processor has found an internal inconsistency in its data structures. It would help us if you could submit a bug report and supporting files which demonstrate the error.

stringified macro argument is too long

The stringified macro argument is longer than 512 characters. This error is unlikely to occur in user code and it isn't practical to show an example of this failure here.

syntax error in #ifdef/#ifndef

The pre-processor found an error when processing the `#ifdef` or `#ifndef` controls. This is usually caused by extra tokens on the pre-processor control line.

syntax error in #include

The pre-processor found an error when processing the file to include in an `#include` directive. The usual cause of this is that the file name isn't enclosed in angle brackets or quotation marks, or that the trailing quotation mark is missing.

syntax error in macro parameters

The syntax of the comma-separated list of macro parameters in a `#define` pre-processor control is not correct. This can occur for a number of reasons, but most common is incorrect punctuation.

undefined expression value

The pre-processor encountered an error when evaluating an expression which caused the expression to be undefined. This is caused by dividing by zero using the division or modulus operators.

unterminated string or character constant

A string is not terminated at the end of a line.

Compiler warning messages

'function' is a non-ANSI definition

You have declared the **main** entry point using an old-style function definition. **main** should be an ANSI-prototyped function. The compiler only reports this warning when extra-picky ANSI warnings are enabled.

Old-style function definitions, although valid, should not be used because they are a common source of errors and lead to code which is less efficient than a prototyped function. A function which takes no parameters should be declared with the parameter list (**void**).

'type' is a non-ANSI type

The `long` type and its unsigned variant are not supported by ANSI C. The CrossWorks C compiler supports this type as you would expect.

'type' used as an lvalue

You used an object of type **type** as an lvalue. Assigning through an uncast, dereferenced `void *` pointer is an error and will result in this error.

empty declaration

A declaration does not declare any variables (or types in the case of `typedef`).

empty input file

The input file contains no declarations and no functions. A C file which contains only comments will raise this warning.

local 'type name' is not referenced

The local declared variable **name** is not referenced in the function.

Compiler error messages

'*number*' is an illegal array size

Array sizes must be strictly positive -- the value *number* is either zero or negative.

'*number*' is an illegal bit-field size

The size of a bit field be within the range 0 to $8 * \text{sizeof}(\text{int})$. In many cases this means that long data items cannot be used in bit field specifications.

'*type*' is an illegal bit-field type

The type of a bit field must be either an unsigned integer or a signed integer, and *type* is neither of these. Note that enumeration types cannot be used in bit fields.

'*type*' is an illegal field type

You cannot declare function types in structures or unions, only pointers to functions.

(' expected

An opening parenthesis was expected after the built-in function `__typechk`.

addressable object required

An addressable object is required when applying the address-of operator '&'. In particular, you can't take the address of a simple constant.

assignment to const identifier '*name*'

You cannot assign to const-qualified identifiers.

assignment to const location

You cannot assign through a pointer to a const-qualified object nor can you assign to members of const-qualified structures or unions.

bad hexadecimal escape sequence '*\xchar*'

The character *char* which is part of a hexadecimal escape sequence isn't a valid hexadecimal digit.

'break' not inside loop or switch statement

You have placed a break statement in the main body of a function without an enclosing for, do-while, while-do, or switch statement. This usually happens when you edit the code and remove a trailing close brace by mistake.

cannot initialize undefined '*type*'

You cannot initialize an undefined structure or union type. Undefined structure or union types are usually used to construct recursive data structures or to hide implementation details. They are introduced using the

syntax `struct tag ;` or `union tag ;` which makes the type's structure tag known to the compiler, but not the structure or union's size. Usually, this error indicates that an appropriate header file has not been included.

case label must be a constant integer expression

The value in a case label must be known to the compiler at compilation time and cannot depend upon runtime values. If you need to make multi-way decisions using runtime values then use a set of `if-else` statements.

'case' not inside 'switch'

You have placed a case label outside a switch statement in the body of a function. This usually happens when you edit the code and remove a trailing close brace by mistake.

cast from '*type*₁' to '*type*₂' is illegal

Casting between *type*₁ and *type*₂ makes no sense and is illegal according to the ANSI specification. Casting, for instance, between an integer and a structure is disallowed, as is casting between structures.

cast from '*type*₁' to '*type*₂' is illegal in constant expression

Casting the pointer type *type*₁ to type *type*₂ is not allowed in a constant expression as the pointer value cannot be known at compile-time.

conflicting argument declarations for function '*name*'

You have declared the function *name* with an inconsistent prototype, such as changing the type of a parameter or not matching the number of parameters.

'continue' not inside loop statement

You have placed a continue statement in the main body of a function without an enclosing for, do-while, or while-do statement. This usually happens when you edit the code and remove a trailing close brace by mistake.

declared parameter '*name*' is missing

In an old-style function definition, the parameter name is declared but is missing from the function.

'default' not inside 'switch'

You have placed a default case label outside a switch statement in the body of a function. This usually happens when you edit the code and remove a trailing close brace by mistake.

duplicate case label '*number*'

You have given two or more case labels the same value -- all case labels must have distinct values.

duplicate declaration for '*name*' previously declared at *pos*

You have used the name *name* to declare two objects in the same scope with identical names.

duplicate field name '*name*' in '*type*'

The field name *name* has already been used in the structure or union type *type*.

empty declaration

You have started a declaration but haven't defined an object with it. This usually occurs when declaring structure or union types as the syntax is commonly misunderstood.

expecting an enumerator identifier

You must use only identifiers in enumeration types, and the enumeration type must have at least one element.

expecting an identifier

You have not constructed an old-style parameter list correctly.

extra 'default' cases in 'switch'

You have supplied more than one default case label in a switch statement. A switch statement can have either no default case label or a single default case label.

extraneous identifier '*name*'

You have given more than one identifier in a declaration. This usually happens when you forget a comma.

extraneous old-style parameter list

You have given an old-style parameter list when declaring a pointer to a function.

extraneous return value

You have provided an expression to return from a void function -- void functions cannot return values.

field name expected

You have not used an identifier to select a field name after '.' or '->'.

field name missing

You have not provided a field name in a structure or union declaration.

found '*type*', expected a function

You have tried to call something which is not a function.

frame exceeds *size* bytes

The size of the stack frame has exceeded *size* bytes which means that the application will not work at runtime.

illegal character '*char*'

The character *char* isn't valid in a C program. For example, '\$' isn't used in C.

illegal character '\0ooo'

The compiler has found a non-printable character which isn't valid in a C program.

illegal expression

You have not constructed an expression correctly. This can happen for many reasons and it is impractical to list them all here.

illegal formal parameter types

You cannot specify a formal parameter as void, only as a pointer to void.

illegal initialization for '*name*'

You cannot initialize the function *name* -- functions can't be initialized.

illegal initialization for parameter '*name*'

You cannot initialise parameter *name* in a formal parameter list.

illegal initialization of 'extern *name*'

You cannot initialise the value of external variables.

illegal return type; found '*type*₁' expected '*type*₂'

The type *type*₁ of the expression returned is not compatible with the declared return type of the function which is *type*₂.

illegal return type '*type*'

You have declared a function with return type *type* which is a function or an array -- a function cannot return an array nor a function.

illegal statement termination

You have not correctly terminated a statement.

illegal type '*type*'

You can use const and volatile only on types which are not already declared const or volatile.

illegal type '*type*' in switch expression

You can only use simple types in switch expressions -- expressions which are compatible with int.

illegal type '*type*[]'

You cannot declare arrays of functions, only arrays of function pointers,

illegal use of incomplete type '*type*'

The return type *type* of a function must be known before it can be called or declared as a function -- that is, you cannot use incomplete types in function declarations.

illegal use of type name '*name*'

The name *name* is a typedef and cannot be used in an expression.

initializer must be constant

The value you have used to initialize a variable is not constant and is only known at run time -- a constant which is computable at compile time is required.

insufficient number of arguments to '*name*'

You have not provided enough arguments to the function *name*.

integer expression must be constant

The value you have used in a bit field width, in specifying the size of an array, or defining the value of an enumeration is not constant -- a constant which is computable at compile time is required.

invalid *type* field declarations

Structure and union field declarations must start with a type name.

invalid floating constant '*string*'

The string *string* is a invalid floating-point constant.

invalid hexadecimal constant '*string*'

The string *string* is a invalid hexadecimal constant.

invalid initialization type; found '*type*₁' expected '*type*₂'

The type *type*₁ which you have used to initialize a variable is not compatible with the type *type*₂ of the variable.

invalid octal constant '*string*'

The string *string* is a invalid octal constant.

invalid operand of unary &; '*name*' is declared register

You cannot take the address of register variables and name is declared as a register variable.

invalid storage class '*class*' for '*type name*'

You have mis-declared the storage class for the variable *name* of type *type*. For example, you cannot declare global objects auto, nor can you declare parameters static or external.

invalid type argument '*type*' to 'sizeof'

You cannot apply **sizeof** to an undefined type or to a function.

invalid type specification

The combination of type qualifiers and size specifiers isn't valid.

invalid use of '*keyword*'

You can't use **register** or **auto** at global level.

invalid use of 'typedef'

You can only use typedef to define plain types without a storage class.

left operand of . has incompatible type '*type*'

The operand to the left of the '.' isn't of structure type.

left operand of -> has incompatible type '*type*'

The operand to the left of the '->' isn't a pointer to a structure type.

lvalue required

A lvalue is required. An lvalue is an object which can be assigned to.

**missing '
missing "**

A string constant hasn't been closed correctly.

missing *type* tag

A structure tag is sometimes required if an undefined structure is used on its own without a typedef.

missing { in initialization of '*type*'

Types with nested structures or unions must be initialised correctly with structures delimited with '{' and '}'.

missing array size

When declaring an array without an initialiser, you must give explicit array sizes.

missing identifier

Your declaration is missing an identifier which defines what is being declared.

missing label in goto

An identifier is required immediately after goto.

missing name for parameter *number* to function '*name*'

Only function prototypes can have anonymous parameters; for ANSI-style function declarations all parameters must be given names.

missing parameter type

An ANSI-style function declaration requires that all parameters are typed in the function prototype.

operand of unary *operator* has illegal type '*type*'

The operand's type isn't compatible with the unary operator *operator*.

operands of *operator* have illegal types '*type*₁' and '*type*₂'

The types of the left and right operands to the binary operator *operator* are not allowed.

overflow in value for enumeration constant '*name*'

When declaring *name* as an enumeration constant, the integer value of that constant exceeds the maximum integer value.

redeclaration of '*name*'**redeclaration of '*name*' previously declared at *position***

The identifier *name* has already been used in this scope for another purpose and cannot be used again.

redefinition of '*name*' previously defined at *position*

You have redefined the initialisation of the identifier *name* -- only one definition of the identifier's value is allowed.

redefinition of label '*name*' previously defined at *position*

You have redefined the label *name* with the same name in the function. Labels are global to the function, not local to a block.

size of '*type*' exceeds *size* bytes

The size of the type *type* is greater than *size* bytes. The compiler cannot construct data items larger than *size* bytes for this processor.

size of '*type*[]' exceeds *size* bytes

The size of the array of *type* is greater than *size* bytes. The compiler cannot construct data items larger than *size* bytes for this processor.

'sizeof' applied to a bit field

You cannot use `sizeof` with a bit field as `sizeof` returns the number of bytes used for an object, whereas a bit field is measured in bits.

too many arguments to '*name*'

You have provided too many arguments to the function *name*.

too many errors

The compiler has stopped compilation because too many errors have been found in your program. Correct the errors and then recompile.

too many initializers

You have provided more initializers for an array or a structure than the compiler expected. Check the bracketing for nested structures.

type error in argument *n* to *name*; '*type*' is illegal

The *n*th actual argument to the function *name* is of type *type* and is not compatible with the *n*th formal argument given in the prototype for *name*. You should check that the formal and actual parameters on a function call match.

type error in argument *n* to *name*; found '*type* 1' expected '*type* 2'

The *n*th actual argument to the function *name* is of type *type1* and is not compatible with the *n*th formal argument given of type *type2* in the prototype for *name*. You should check that the formal and actual parameters on a function call match.

undeclared identifier '*name*'

You have used the identifier *name* but it has not been previously declared.

undefined label '*name*'

You have used the label *name* in a goto statement but no label in the current function has been defined with that name.

undefined size for '*type name*'

You have declared the variable *name* using the type *type*, but the size of type is not yet known. This occurs when you define a union or structure with a tag but do not define the contents of the structure and then use the tag to define a variable.

undefined size for field '*type name*'

You have declared the field *name* using the type *type*, but the size of type is not yet known. This occurs when you define a union or structure with a tag but do not define the contents of the structure and then use the tag to define a field.

undefined size for parameter '*type name*'

You have declared the parameter *name* using the type *type*, but the size of type is not yet known. This occurs when you define a union or structure with a tag but do not define the contents of the structure and then use the tag to define a parameter.

undefined static '*type name*'

You have declared the static function *name* which returns type *type* in a prototype, but have not defined the body of the function.

unknown enumeration '*name*'

You have not defined the enumeration tag name but have used it to defined an object.

unknown field '*name*' of '*type*'

You have accessed the field *name* of the structure or union type *type*, but a field of that name is not declared within that structure.

unknown size for type '*type*'

You have tried to use an operator where the size of the type *type* must be known to the compiler.

unrecognized declaration

The compiler can't recognise the declaration syntax you have used. This is usually caused by misplacing a comma in a declarator.

unrecognized statement

The compiler can't recognise the start of a statement. A statement must start with one of the statement keywords or must be an expression. This is usually caused by misplacing a semicolon.

user type check error: found '*type 1*' expected '*type 2*'

The operand to the `__typechk` intrinsic function is incorrect. This occurs when you provide a parameter to a the macro function which checks its expected parameters using `__typechk`. You should check the types of parameters you pass to the macro routine

Extensions summary

This section summarises the extensions the ISO standard provided by the CrossWorks C compiler . It does **not** cover any extensions to the library.

Compiler

- [Code-space strings](#)
- [Type-based enumerations](#)
- [GSM 03.38 strings](#)
- [Interrupt functions](#)
- [Monitor functions](#)
- [Top-level functions](#)
- Binary constants

Preprocessor

- `#warning` directive

Tasking Library User Guide

This section describes the CrossWorks Tasking Library which will be subsequently referred to as CTL. CTL provides a multi-priority, preemptive, task switching and synchronisation facility. Additionally CTL provides timer, interrupt service routine and memory block allocation support.

In this section

Overview

Describes the principles behind the CTL.

Tasks

Describes how to create CTL tasks, turn the main program into a task and manage tasks.

Event sets

Describes what a CTL event set is and how it can be used.

Semaphores

Describes what a CTL semaphore is and how it can be used.

Mutexes

Describes what a CTL mutex is and how it can be used.

Message queues

Describes what a CTL message queue is and how it can be used.

Byte queues

Describes what a CTL byte queue is and how it can be used.

Global interrupts control

Describes how you can use CTL functions to enable and disable global interrupts.

Timer support

Describes the timer facilities that CTL provides.

Interrupt service routine support

Describes how to write interrupt service routines that co-exist with CTL.

Memory block allocation support

Describes how you can use CTL to allocate fixed sized memory blocks.

Task Scheduling

An example of task scheduling.

MSP430 Implementation

Implementation details for the MSP430 architecture.

Revisions

CTL revisions.

Related sections

<ctl.h> - CTL functions

The reference for each of the functions and variables defined by the CTL.

Threads window

A scriptable debugger window that displays the threads of a running program together with their state.

Overview

The CTL enables your application to have multiple tasks. You will typically use a task when you have some algorithmic or protocol processing that may suspend its execution whilst other activities occur. For example you may have a protocol processing task, a user interface task and a data acquisition task.

Each task has its own stack which is used to store local variables and function return information. The task stack is also used to store the CPU execution context when the task isn't executing. The CPU execution context of a task varies between machine architectures. It is typically the subset of the CPU register values which enable a task to be descheduled at any point during its execution.

The process of changing the CPU registers from one task to another is termed task switching. Task switching occurs when a CTL function is called, either from a task or from an interrupt service routine (ISR) and there is a runnable task which has a higher priority than the executing task.

Task switching also occurs when there is a runnable task of the same priority as the executing task which has exceeded its time slice period. If you have more than one runnable task of the same priority then the next task (modulo priority) after the executing task is selected. This is sometimes called round robin scheduling.

There is a single task list kept in priority sorted order. The task list is updated when tasks are created, deleted and have their priority changed. The task list is traversed when a CTL function is called that could change the execution state of a task. When the task list is modified or traversed global interrupts are disabled. Consequently the interrupt disable period is dependent on the number of tasks in the task list, the priority of the task affected by (and the type of) the CTL operation.

If you require a simply deterministic (sometimes called real-time) system then you should ensure that each task has a unique priority. The task switching will always select the highest priority task that is runnable.

CTL has a pointer to the executing task. There must always be a task executing, if there isn't then a CTL error is signalled. Typically there will be an idle task that loops and perhaps puts the CPU into a power save mode.

When a task switch occurs global interrupts will be enabled. So you can safely call the tasking library functions with interrupts disabled.

Task synchronisation and resource allocation

The CTL provides several mechanisms to synchronise execution of tasks, to serialise resource access and to provide high level communication.

- **Event Sets** — An event set is a word sized variable which tasks can wait for specific bits (events) to be set to 1. Events can be used for synchronisation and to serialise resource access. Events can be set by interrupt service routines.
- **Semaphores** — A semaphore is a word size variable which tasks can wait for to be non-zero. Semaphores can be used for synchronisation and to serialise resource access. Semaphores can be signalled by interrupt service routines.

- **Mutexes** — A mutex is a structure that can be used to serialise resource access. Unlike semaphores mutexes cannot be used by interrupt service routines, but they do provide extra features that make mutexes preferable to semaphores for serialising resource access.
- **Message Queues** — A message queue is a structure that enables tasks to post and receive data. Message queues are used to provide a buffered communication mechanism. Messages can be sent by interrupt service routines.
- **Byte Queues** — A byte queue is a specialisation of a message queue i.e. it is a message queue where the messages are one byte in size. Byte queues can be sent by interrupt service routines.
- **Interrupt enable/disable** — The tasking library provides functions that enable and disable the global interrupt enables state of the processor. These functions can be used to provide a time critical mutual exclusion facility.

Note that all waits on task synchronization objects are priority based i.e. the highest priority task waiting will be scheduled first.

Timer support

If your application can provide a periodic timer interrupt then you can use the timer facility of the CTL. This facility enables time slicing of equal priority tasks, allows tasks to delay and provides a timeout capability when waiting for something. The timer is a software counter that is incremented by your timer interrupt. The counter is typically a millisecond counter, you can change the amount the timer is incremented to reduce the interrupt frequency.

Memory allocation support

The CTL provides a simple memory block allocator that can be used in situations where the standard C malloc and free functions are either too slow or may block the calling task.

C library support

The CTL provides the functions required of the CrossWorks C library for multi-threading.

Tasks

Each task has a corresponding task structure which contains the following information

- When the task isn't executing a pointer to the stack containing the execution context.
- The priority of the task, the lowest priority is 0 the highest is 255.
- The state of the task - is the task runnable or waiting for something.
- A pointer to the next task.
- If the task is waiting for something, the details of what it is waiting for.
- Thread specific data such as errno.
- A pointer to a null terminated string that names the task for debugging purposes.

You allocate task structures by declaring them as C variables.

```
CTL_TASK_t mainTask;
```

You create the first task using the `ctl_task_init` function which turns the main program into a task. This function takes a pointer to the task structure that represents the main task, it's priority and a name as parameters.

```
ctl_task_init(&mainTask, 255, "main");
```

This function must be called before any other CrossWorks tasking library calls are made. The priority (second parameter) must be between 0 (the lowest priority) and 255 (the highest priority). It is advisable to create the first task with the highest priority which enables the main task to create other tasks without being descheduled. The name should point to a zero terminated ASCII string for debug purposes.

You can create other tasks using the function `ctl_task_run` which initialises a task structure and may cause a context switch. You supply the same arguments as `task_init` together with the function that the task will run and the memory that the task will use as its stack.

The function that a task will run should take a `void *` parameter and not return any value.

```
void task1Fn(void *parameter)
{
    // task code goes in here
    ...
}
```

The `parameter` value is supplied to the function by the `ctl_task_run` call. Note when a task function returns the `ctl_task_die` function is called which terminates the task.

You have to allocate the stack for the task as an C array of unsigned.

```
unsigned task1Stack[64];
```

The size of the stack you need depends on the CPU type (the number of registers that have to be saved), the function calls that the task will make and (depending upon the CPU) the stack used for interrupt service routines. Running out of stack space is common problem with multi-tasking systems and the error behaviour is often misleading. It is recommended that you initialise the stack to known values so that you can check the stack with the CrossWorks debugger if problems occur.

```
memset(task1Stack, 0xba, sizeof(task1Stack));
```

Your `ctl_task_run` function call should look something like this.

```
ctl_task_run(&task1Task,
            12,
            task1Fn,
            0,
            "task1",
            sizeof(task1Stack) / sizeof(unsigned),
            task1Stack,
            0);
```

The first parameter is a pointer to the task structure. The second parameter is the priority (in this case 12) the task will start executing at. The third parameter is a pointer to the function to execute (in this case `task1Fn`). The fourth parameter is the value that is supplied to the task function (in this case zero). The fifth parameter is a null terminated string that names the task for debug purposes. The sixth parameter is the size of the stack in words. The seventh parameter is the pointer to the stack. The last parameter is for systems that have a separate call stack and is the number of words to reserve for the call stack.

You can change the priority of a task using the `ctl_task_set_priority` function call which takes a pointer to a task structure and the new priority as parameters and returns the old priority.

```
old_priority = ctl_task_set_priority(&mainTask, 255); // lock scheduler
...
ctl_task_set_priority(old_priority);
```

If you want to enable time slicing then you need to set the `ctl_timeslice_period` variable before any task scheduling occurs.

```
ctl_timeslice_period = 100; // time slice period of 100 ms
```

If you want finer control over the scheduling of tasks then you can call `ctl_task_reschedule`.

```
ctl_task_reschedule();
```

Example

The following example turns main into a task and creates another task. The main task ultimately will be the lowest priority task that switches the CPU into a power save mode when it is scheduled - this satisfies the requirement of always having a task to execute and enables a simple power saving system to be implemented.

```
#include <ctl.h>

void task1(void *p)
{
    // task code, on return task will be terminated
}

static CTL_TASK_t mainTask, task1Task;
static unsigned task1Stack[64];

int
main(void)
{
    // Turn myself into a task running at the highest priority.
```



```
ctl_task_init(&mainTask, 255, "main");

// Initialise the stack of task1.
memset(task1Stack, 0xba, sizeof(task1Stack));

// Make another task ready to run.
ctl_task_run(&task1Task, 1, task1, 0, "task1", sizeof(task1Stack) / sizeof(unsigned),
task1Stack, 0);

// Now all the tasks have been created go to lowest priority.
ctl_task_set_priority(&mainTask, 0);

// Main task, if activated because task1 is suspended, just
// enters low power mode and waits for task1 to run again
// (for example, because an interrupt wakes it).
for (;;)
{
    // Go into low power mode
    sleep();
}
```

Note that initially the main task is created at the highest priority whilst it creates the other tasks, it then changes its priority to the lowest task. This technique can be used when multiple tasks are created to enable all of the tasks to be created before they start to execute.

Note the usage of **sizeof** when passing the stack size to **ctl_task_run**.

Event sets

An event set is a means to synchronise tasks with other tasks and interrupt service routines. An event set contains a set of events (one per bit) which tasks can wait to become set (value 1). When a task waits on an event set the events it is waiting for are matched against the current values—if they match then the task can still execute. If they don't match, the task is put on the task list together with details of the event set and the events that the task is waiting for.

You allocate an event set by declaring it as C variable

```
CTL_EVENT_SET_t e1;
```

An **CTL_EVENT_SET_t** is a synonym for an **unsigned** type. Thus, when an **unsigned** is 16 bits an event set will contain 16 events and when it is 32 bits an event set will contain 32 events.

You can initialise an event set using the [ctl_events_init](#) function.

```
ctl_events_init(&e1, 0);
```

Note that initialisation should be done before any tasks can use an event set.

You can set and clear events of an event set using the [ctl_events_set_clear](#) function.

```
ctl_events_set_clear(&e1, (1<<0), (1<<15));
```

This example will set the bit zero event and clear the bit 15 event. If any tasks are waiting on this event set the events they are waiting on will be matched against the new event set value which could cause the task to become runnable.

You can wait for events to be set using the [ctl_events_wait](#) function. You can wait for any of the events in an event set to be set (**CTL_EVENT_WAIT_ANY_EVENTS**) or all of the events to be set (**CTL_EVENT_WAIT_ALL_EVENTS**). You can also specify that when events have been set and have been matched that they should be automatically reset (**CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR** and **CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR**). You can associate a timeout with a wait for an event set to stop your application blocking indefinitely.

```
ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e1, (1<<15), CTL_TIMEOUT_NONE, 0);
```

This example waits for bit 15 of the event set pointed to by **e1** to become set.

```
if (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e1, (1<<15), CTL_TIMEOUT_DELAY, 1000)==0)
{
    // timeout occurred
}
```

This example uses a timeout and tests the return result to see if the timeout occurred.

You can use the [ctl_events_pulse](#) function to set and then clear events. You can use this to wake up multiple threads and reset the events atomically.

Task synchronisation with an ISR example

The following example illustrates synchronising a task with a function called from an ISR.

```

CTL_EVENT_SET_t e1;
CTL_TASK_s t1;

void ISRfn()
{
    // do work, and then...
    ctl_events_set_clear(&e1, (1<<0), 0);
}

void task1(void *p)
{
    while (1)
    {
        ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e1, (1<<0), CTL_TIMEOUT_NONE, 0);
        ...
        ctl_events_set_clear(&e1, 0, (1<<0));
    }
}

```

Task synchronisation with more than one ISR example

The following example illustrates synchronising a task with functions called from two interrupt service routines.

```

CTL_EVENT_SET_t e1;
CTL_TASK_s t1;

void ISRfn1(void)
{
    // do work, and then...
    ctl_events_set_clear(&e1, (1<<0), 0);
}

void ISRfn2(void)
{
    // do work, and then...
    ctl_events_set_clear(&e1, (1<<1), 0);
}

void task1(void *p)
{
    for (;;)
    {
        unsigned e;
        e = ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR, &e1, (1<<0) | (1<<1),
CTL_TIMEOUT_NONE, 0);
        if (e & (1<<0))
        {
            // ISRfn1 completed
        }
        else if (e & (1<<1))
        {
            // ISRfn2 completed
        }
        else
        {
            // error
        }
    }
}

```

Resource serialisation example

The following example illustrates resource serialisation of two tasks.

```

CTL_EVENT_SET_t e1;

void task1(void)
{
    for (;;)
    {
        ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR, &e1, (1<<0),
        CTL_TIMEOUT_NONE, 0);
        // resource has now been acquired
        ctl_events_set_clear(&e1, (1<<0), 0);
        // resource has now been released
    }
}

void task2(void)
{
    for (;;)
    {
        ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR, &e1, (1<<0),
        CTL_TIMEOUT_NONE, 0);
        // resource has now been acquired
        ctl_events_set_clear(&e1, (1<<0), 0);
        // resource has now been released
    }
}
....
void main(void)
{
    ...
    ctl_events_init(&e1, (1<<0));
    ...
}

```

Note that **e1** is initialised with the event set without this neither task would acquire the resource.

Semaphores

A semaphore is a counter which tasks can wait for to be non-zero. When a semaphore is non-zero and a task waits on it then the semaphore value is decremented and the task continues execution. When a semaphore is zero and a task waits on it then the task will be suspended until the semaphore is signalled. When a semaphore is signalled and no tasks are waiting for it then the semaphore value is incremented. When a semaphore is signalled and tasks are waiting then one of the tasks is made runnable.

You allocate a semaphore by declaring it as a C variable

```
CTL_SEMAPHORE_t s1;
```

An **CTL_SEMAPHORE_t** is a synonym for an **unsigned** type, so the maximum value of the counter is dependent upon the word size of the processor (either 16 or 32 bits).

You can initialise a semaphore using the [ctl_semaphore_init](#) function.

```
ctl_semaphore_init(&s1, 1);
```

Note that initialisation should be done before any tasks can use a semaphore.

You can signal a semaphore using the [ctl_semaphore_signal](#) function.

```
ctl_semaphore_signal(&s1);
```

The highest priority task waiting on the semaphore pointed at by **s1** will be made runnable by this call. If no tasks are waiting on the semaphore then the semaphore value is incremented.

You can wait for a semaphore with an optional timeout using the [ctl_semaphore_wait](#) function.

```
ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
```

This example will block the task if the semaphore is zero, otherwise it will decrement the semaphore and continue execution.

```
if (ctl_semaphore_wait(&s1, CTL_TIMEOUT_ABSOLUTE, ctl_get_current_time()+1000)==0)
{
    // timeout occurred
}
```

This example uses a timeout and tests the return result to see if the timeout occurred.

Task synchronisation in an interrupt service routine

The following example illustrates synchronising a task with a function called from an interrupt service routine.

```
CTL_SEMAPHORE_t s1;

void ISRfn()
{
    // do work
    ctl_semaphore_signal(&s1);
}
```

```

void task1(void *p)
{
    while (1)
    {
        ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
        ...
    }
}

```

Resource serialisation

The following example illustrates resource serialisation of two tasks.

```

CTL_SEMAPHORE_t s1=1;

void task1(void)
{
    for (;;)
    {
        ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
        /* resource has now been acquired */
        ...
        ctl_semaphore_signal(&s1);
        /* resource has now been released */
    }
}

void task2(void)
{
    for (;;)
    {
        ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
        /* resource has now been acquired */
        ...
        ctl_semaphore_signal(&s1);
        /* resource has now been released */
    }
}

int
main(void)
{
    ...
    ctl_semaphore_init(&s1, 1);
    ...
}

```

Note that **s1** is initialised to one, without this neither task would acquire the resource.

Mutexes

A mutex is a structure that can be used to serialise resource access. Tasks can lock and unlock mutexes. A mutex holds a lock count that enables the same task to recursively lock the mutex. Tasks must ensure that the number of unlocks matches the number of locks. When a mutex is locked if another task tries to lock the mutex this task waits until the mutex becomes unlocked. The priority of the task that has locked the mutex is raised to the highest priority of the tasks that are waiting to lock the mutex, this mechanism prevents what is often called **priority inversion**. Note that mutexes cannot be used by interrupt service routines.

You allocate a mutex by declaring it as a C variable

```
CTL_MUTEX_t mutex;
```

You can initialise a mutex using the [ctl_mutex_init](#) function.

```
ctl_mutex_init(&mutex);
```

Note that initialisation should be done before any tasks can use a mutex.

You can lock a mutex with an optional timeout using the [ctl_mutex_lock](#) function.

```
ctl_mutex_lock(&mutex, CTL_TIMEOUT_NONE, 0);
```

You can unlock a (locked by the calling task) mutex using the [ctl_mutex_unlock](#) function.

```
ctl_mutex_unlock(&mutex);
```

Resource serialisation

The following example illustrates resource serialisation of two tasks.

```
CTL_MUTEX_t mutex;

void fn1(void)
{
    ctl_mutex_lock(&mutex, CTL_TIMEOUT_NONE, 0);
    ...
    ctl_mutex_unlock(&mutex);
}

void fn2(void)
{
    ctl_mutex_lock(&mutex, CTL_TIMEOUT_NONE, 0);
    ...
    fn1();
    ...
    ctl_mutex_unlock(&mutex);
}

void task1(void)
{
    for (;;)
    {
        fn2()
    }
}
```

```
void task2(void)
{
    for (;;)
    {
        fn1();
    }
}

int
main(void)
{
    ...
    ctl_mutex_init(&mutex);
    ...
}
```

Note that task1 locks the mutex twice by calling fn2() which then calls fn1().

Message queues

A message queue is a structure that enables tasks to post and receive messages. A message is a generic (void) pointer and as such can be used to send data that will fit into a pointer type (2 or 4 bytes depending upon processor word size) or can be used to pass a pointer to a block of memory. The message queue has a buffer that enables a number of posts to be completed without receives occurring. The buffer keeps the posted messages in a fifo order so the oldest message is received first. When the buffer isn't full a post will put the message at the back of the queue and the calling task continues execution. When the buffer is full a post will block the calling task until there is room for the message. When the buffer isn't empty a receive will return the message from the front of the queue and continue execution of the calling task. When the buffer is empty a receive will block the calling task until a message is posted.

Initialisation

You allocate a message queue by declaring it as a C variable

```
CTL_MESSAGE_QUEUE_t m1;
```

A message queue is initialised using the [ctl_message_queue_init](#) function.

```
void *queue[20];  
...  
ctl_message_queue_init(&m1, queue, 20);
```

This example uses an 20 element array for the message queue. Note that the array is a `void *` which enables pointers to memory or (cast) integers to be communicated via a message queue.

Posting

You can post a message to a message queue with an optional timeout using the [ctl_message_queue_post](#) function.

```
ctl_message_queue_post(&m1, (void *)45, CTL_TIMEOUT_NONE, 0);
```

This example posts the integer 45 onto the message queue.

You can post multiple messages to a message queue with an optional timeout using the [ctl_message_queue_post_multi](#) function.

```
if (ctl_message_queue_post_multi(&m1, 4, messages, CTL_TIMEOUT_ABSOLUTE,  
    ctl_get_current_time()+1000) != 4)  
{  
    // timeout occurred  
}
```

This example uses a timeout and tests the return result to see if the timeout occurred.

If you want to post a message and you don't want to block (e.g from an interrupt service routine) you can use the [ctl_message_queue_post_nb](#) function (or [ctl_message_queue_post_multi_nb](#) if you want to post multiple messages)

```
if (ctl_message_queue_post_nb(&m1, (void *)45)==0)
{
    // queue is full
}
```

This example tests the return result to see if the post failed.

Receiving

You can receive a message with an optional timeout using the [ctl_message_queue_receive](#) function.

```
void *msg;
ctl_message_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
```

This example receives the oldest message in the message queue.

You can receive multiple messages from a message queue with an optional timeout using the [ctl_message_queue_receive_multi](#) function.

```
if (ctl_message_queue_multi_receive(&m1, 4, msgs, CTL_TIMEOUT_DELAY, 1000) != 4)
{
    // timeout occurred
}
```

This example uses a timeout and tests the return result to see if the timeout occurred.

If you want to receive a message and you don't want to block (e.g from an interrupt service routine) you can use the [ctl_message_queue_receive_nb](#) function (or [ctl_message_queue_receive_multi_nb](#) if you want to receive multiple messages).

```
if (ctl_message_queue_receive_nb(&m1, &msg)==0)
{
    // queue is empty
}
```

Example

The following example illustrates usage of a message queue to implement the producer-consumer problem.

```
CTL_MESSAGE_QUEUE_t m1;
void *queue[20];

void task1(void)
{
    ...
    ctl_message_queue_post(&m1, (void *)i, CTL_TIMEOUT_NONE, 0);
    ...
}

void task2(void)
{
    void *msg;
    ...
    ctl_message_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
    ...
}

int
```

```
main(void)
{
    ...
    ctl_message_queue_init(&m1, queue, 20);
    ...
}
```

Advanced Usage

You can associate event flags with a message queue that are set (and similarly cleared) when the message queue is not full and not empty using the function [ctl_message_queue_setup_events](#).

Using this you can wait (for example) for messages to arrive from multiple message (or byte) queues.

```
CTL_MESSAGE_QUEUE_t m1, m2;
CTL_EVENT_SET_t e;
ctl_message_queue_setup_events(&m1, &e, (1<<0), (1<<1));
ctl_message_queue_setup_events(&m2, &e, (1<<2), (1<<3));
...
switch (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e, (1<<0)|(1<<2), 0, 0))
{
    case 1<<0:
        ctl_message_queue_receive(&m1, ...
        break;
    case 1<<2:
        ctl_message_queue_receive(&m2, ...
        break;
}
```

This example sets up and waits for the notempty event of message queue m1 and the notempty event of message queue m2. When the wait completes it reads from the appropriate message queue. Note that you should **not** use a **_WITH_AUTO_CLEAR** event wait type when waiting on events that are associated with a message queue.

You can test how many messages are in a message queue using [ctl_message_queue_num_used](#) and how many free messages are in a message queue using [ctl_message_queue_num_free](#). You can use these functions to poll the message queue.

```
while (ctl_message_queue_num_free(&m1)<10)
    ctl_task_timeout_wait(ctl_get_current_time()+1000);
ctl_message_queue_post_multi(&m1, 10, ...
```

This example waits for 10 elements to be free before it posts 10 elements.

Byte queues

A byte queue is a structure that enables tasks to post and receive data bytes. The byte queue has a buffer that enables a number of posts to be completed without receives occurring. The buffer keeps the posted bytes in a fifo order so the oldest byte is received first. When the buffer isn't full a post will put the byte at the back of the queue and the calling task continues execution. When the buffer is full a post will block the calling task until there is room for the byte. When the buffer isn't empty a receive will return the byte from the front of the queue and continue execution of the calling task. When the buffer is empty a receive will block the calling task until a byte is posted.

Initialisation

You allocate a byte queue by declaring it as a C variable

```
CTL_BYTE_QUEUE_t m1;
```

A byte queue is initialised using the [ctl_byte_queue_init](#) function.

```
unsigned char queue[20];
...
ctl_byte_queue_init(&m1, queue, 20);
```

This example uses an 20 element array for the byte queue.

Posting

You can post a byte to a byte queue with an optional timeout using the [ctl_byte_queue_post](#) function.

```
ctl_byte_queue_post(&m1, 45, CTL_TIMEOUT_NONE, 0);
```

This example posts the byte 45 onto the byte queue.

You can post multiple bytes to a byte queue with an optional timeout using the [ctl_byte_queue_post_multi](#) function.

```
if (ctl_byte_queue_post(&m1, 4, bytes, CTL_TIMEOUT_ABSOLUTE, ctl_get_current_time()+1000) !=
    4)
{
    // timeout occurred
}
```

This example uses a timeout and tests the return result to see if the timeout occurred.

If you want to post a byte and you don't want to block (e.g from an interrupt service routine) you can use the [ctl_byte_queue_post_nb](#) function (or [ctl_byte_queue_post_multi_nb](#) if you want to post multiple bytes).

```
if (ctl_byte_queue_post_nb(&m1, 45)==0)
{
    // queue is full
}
```

This example tests the return result to see if the post failed.

Receiving

You can receive a byte with an optional timeout using the [ctl_byte_queue_receive](#) function.

```
unsigned char msg;
ctl_byte_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
```

This example receives the oldest byte in the byte queue.

You can receive multiple bytes from a byte queue with an optional timeout using the [ctl_byte_queue_receive_multi](#) function.

```
if (ctl_byte_queue_receive_multi(&m1, 4, bytes, CTL_TIMEOUT_DELAY, 1000) != 4)
{
    // timeout occurred
}
```

This example uses a timeout and tests the return result to see if the timeout occurred.

If you want to receive a byte and you don't want to block (e.g from an interrupt service routine) you can use the [ctl_byte_queue_receive_nb](#) function (or [ctl_byte_queue_receive_multi_nb](#) if you want to receive multiple bytes).

```
if (ctl_byte_queue_receive_nb(&m1, &msg)==0)
{
    // queue is empty
}
```

Example

The following example illustrates usage of a byte queue to implement the producer-consumer problem.

```
CTL_BYTE_QUEUE_t m1;
void *queue[20];

void task1(void)
{
    ...
    ctl_byte_queue_post(&m1, (void *)i, CTL_TIMEOUT_NONE, 0);
    ...
}

void task2(void)
{
    void *msg;
    ...
    ctl_byte_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
    ...
}

int
main(void)
{
    ...
    ctl_byte_queue_init(&m1, queue, 20);
    ...
}
```

Advanced Usage

You can associate event flags with a byte queue that are set (and similarly cleared) when the byte queue is not full and not empty using the function [ctl_byte_queue_setup_events](#).

Using this you can wait (for example) for messages to arrive from multiple byte (or message) queues.

```
CTL_BYTE_QUEUE_t m1, m2;
CTL_EVENT_SET_t e;
ctl_byte_queue_setup_events(&m1, &e, (1<<0), (1<<1));
ctl_byte_queue_setup_events(&m2, &e, (1<<2), (1<<3));
...
switch (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e, (1<<0)|(1<<2), CTL_TIMEOUT_NONE, 0))
{
    case 1<<0:
        ctl_byte_queue_receive(&m1, ...
        break;
    case 1<<2:
        ctl_byte_queue_receive(&m2, ...
        break;
}
```

This example sets up and waits for the notempty event of byte queue m1 and the notempty event of byte queue m2. When the wait completes it reads from the appropriate byte queue. Note that you should **not** use a [_WITH_AUTO_CLEAR](#) event wait type when waiting on events that are associated with a byte queue.

You can test how many bytes are in a byte queue using [ctl_byte_queue_num_used](#) and how many free bytes are in a byte queue using [ctl_byte_queue_num_free](#). You can use these functions to poll the byte queue.

```
while (ctl_byte_queue_num_free(&m1)<10)
    ctl_task_timeout_wait(ctl_get_current_time()+1000);
ctl_byte_queue_post(&m1, 10, ...
```

This example waits for 10 elements to be free before it posts 10 elements.

Global interrupts control

The CTL provides functions that enable and disable the global interrupt enables of the processor. This mechanism is used by CTL when accessing the task list, it can also be used to provide a fast mutual exclusion facility for time critical uses.

You can disable interrupts using [ctl_global_interrupts_disable](#) and enable interrupts using [ctl_global_interrupts_enable](#).

If you don't know if interrupts are currently disabled then you can use [ctl_global_interrupts_set](#). This will either disable or enable interrupts depending on the parameter and will return the previous interrupt enables state.

```
int en=ctl_global_interrupts_set(0); // disable
...
ctl_global_interrupts_set(en); // set to previous state
```

You can call a tasking library function that causes a task switch with global interrupts disabled. The tasking library will ensure that when the next task is scheduled global interrupts are enabled.

Timer support

The current time is held as a 32 bit value in the `ctl_current_time` variable. This variable is incremented by the number held in `ctl_time_increment` each time an ISR calls `ctl_increment_tick_from_isr`.

```
void timerISR(void)
{
    ...
    ctl_increment_tick_from_isr();
    ...
}
void main(...)
..
    ctl_time_increment = 10;
    // set up timerISR to be called every 100 ms
    ..
```

By convention the timer implements a millisecond counter but you can set the timer interrupt and increment rate appropriately for you application.

You can atomically read `ctl_current_time` using the `ctl_get_current_time` function on systems whose word size is not 32 bit.

You can suspend execution of a task for a fixed period using the `ctl_timeout_wait` function.

Note that this function takes the timeout not the duration as a parameter, so you should always call this function with `ctl_get_current_time()+duration`.

```
ctl_timeout_wait(ctl_get_current_time()+100);
```

This example suspends execution of the calling task for 100 ticks of the `ctl_current_time` variable.

Since the counter is implemented as a 32-bit number, to handle wrap around of the counter you can delay for a maximum of a 31-bit number.

```
ctl_timeout_wait(ctl_get_current_time()+0x7fffffff);
```

This example suspends execution of the calling task for the maximum possible time.

Interrupt service routines

Interrupt service routines (ISR) can communicate with CTL tasks using a subset of the CTL programming interface. An ISR should not call the CTL functions that can block ***ctl_byte_queue_post***, ***ctl_byte_queue_receive***, ***ctl_events_wait***, ***ctl_message_queue_post***, ***ctl_message_queue_receive***, ***ctl_timeout_wait*** and ***ctl_semaphore_wait***. If this happens then ***ctl_handle_error*** will be called.

To detect whether a task or an ISR has called a function CTL uses the global variable ***ctl_interrupt_count***. ISR's must increment this variable on entry and decrement it on exit. Any CTL functions that are called by an ISR that require a task reschedule will set the variable ***ctl_reschedule_on_last_isr_exit***. On exit from an ISR ***ctl_interrupt_count*** is decremented to zero and if ***ctl_reschedule_on_last_isr_exit*** is set then (after resetting ***ctl_reschedule_on_last_isr_exit***) a CTL reschedule operation occurs.

The support for writing ISR's differs depending on the target. In general on entry to an ISR the following is needed

```
...
store registers;
ctl_interrupt_count++;
```

and on exit from an ISR

```
ctl_interrupt_count--;
if (ctl_interrupt_count == 0 && ctl_reschedule_on_last_isr_exit);
{
    ctl_reschedule_on_last_isr_exit = 0;
    reschedule
}
else
    restore registers
```

Memory areas

Memory areas provide your application with dynamic allocation of fixed sized memory blocks. Memory areas should be used in preference to the standard C library malloc and free functions if the calling task cannot block or memory allocation is done by an ISR.

You allocate a memory area by declaring it as a C variable

```
CTL_MEMORY_AREA_t m1;
```

A message queue is initialised using the [ctl_memory_area_init](#) function.

```
unsigned mem[20];
...
ctl_memory_area_init(&m1, mem, 2, 10);
```

This example uses an 20 element array for the memory. The array is split into 10 blocks of each of which two words in size.

You can allocate a memory block from a memory area using the [ctl_memory_area_allocate](#) function. If the memory block cannot be allocated then zero is returned.

```
unsigned *block = ctl_memory_area_allocate(&m1);
if (block)
    // block has been allocated
else
    // no block has been allocated
```

When you have finished with a memory block you should return it to the memory area from which it was allocated using [ctl_memory_area_free](#):

```
ctl_memory_area_free(&m1, block);
```

You can associate an event flag with the block available state of a memory queue to be able to wait for a memory block to become available.

```
CTL_MEMORY_AREA_t m0, m1, m2;
...
CTL_EVENT_SET_t e;
...
ctl_memory_area_setup_events(&m0, &e, (1<<0));
ctl_memory_area_setup_events(&m1, &e, (1<<1));
ctl_memory_area_setup_events(&m2, &e, (1<<2));
...
switch (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e, (1<<0)|(1<<1)|(1<<2), 0, 0))
{
    case 1<<0:
        x = ctl_memory_area_allocate(&m0, ...
        break;
    case 1<<1:
        x = ctl_memory_area_allocate(&m1, ...
        break;
    case 1<<2:
        x = ctl_memory_area_allocate(&m2, ...
        break;
}
```

This example sets up and waits for the block available events of memory areas m0, m1 and m2. When the wait completes it attempts to allocate memory from the appropriate memory area. Note that you should **not** use a `_WITH_AUTO_CLEAR` event wait type when waiting on events that are associated with a memory area.

Task scheduling example

An example task list could be

- task1, priority 2, waiting
- task2, priority 1, runnable
- task3, priority 1, executing
- task4, priority 1, runnable
- task5, priority 0, runnable

task2 waits, so task3 is selected to execute

- task1, priority 2, waiting
- task2, priority 1, waiting
- task3, priority 1, executing
- task4, priority 1, runnable
- task5, priority 0, runnable

An interrupt occurs which makes task1 runnable which is higher priority than task3 so task1 executes

- task1, priority 2, executing
- task2, priority 1, waiting
- task3, priority 1, runnable
- task4, priority 1, runnable
- task5, priority 0, runnable

task1 waits, causing task3 to execute

- task1, priority 2, waiting
- task2, priority 1, waiting
- task3, priority 1, executing
- task4, priority 1, runnable
- task5, priority 0, runnable

A interrupt occurs and task3 has used it's timeslice period so task4 is selected to execute

- task1, priority 2, waiting
- task2, priority 1, waiting
- task3, priority 1, runnable
- task4, priority 1, executing
- task5, priority 0, runnable

An interrupt occurs and makes task2 runnable, but task4 hasn't used it's timeslice period so it is left to execute

- task1, priority 2, waiting
- task2, priority 1, runnable

- task3, priority 1, runnable
- task4, priority 1, executing
- task5, priority 0, runnable

A interrupt occurs and task4 has used it's timeslice period

- task1, priority 2, waiting
- task2, priority 1, executing
- task3, priority 1, runnable
- task4, priority 1, runnable
- task5, priority 0, runnable

MSP430 implementation

MSP430 ISR's

The MSP430 CrossWorks C compiler provides the **__ctl_interrupt** modifier that will generate the required code on entry and exit from the ISR. All you have to do is write your interrupt handling code.

```
void basic_timer_irq(void) __ctl_interrupt[BASIC_TIMER_VECTOR]
{
    // Do your interrupt handling here...
}
```

The ISR will run on a dedicated stack which avoids having to allocate stack space for ISR's on task stacks. You must specify the size in (even numbered) bytes of the stack with the linker symbol CTL_IRQ_STACK_SIZE .e.g CTL_IRQ_STACK_SIZE=128 will allocate 128 bytes of stack.

CTL revisions

CTL has been supplied with various 1.x releases of CrossWorks and it's the revision history is available in the corresponding release notes - such releases are termed CTL V1. CTL is supplied in 2.x releases of CrossWorks and is termed CTL V2. This document explains the differences between CTL V1 and CTL V2.

Scheduling implementation

In CTL V1 the executing task was removed from the task list and then put back on when it was descheduled. In CTL V2 the executing task isn't moved from the task list. This change enables CTL to run on the Cortex-M3 and it also has resulted in smaller code with faster context switching and enables interrupt lock out period to be reduced.

In CTL V1 on exit from the last nested interrupt service routine rescheduling would happen. In CTL V2 a reschedule only occurs on exit from the last nested interrupt service routine if the run state of a task has changed.

Mutexes

POSIX thread style mutexes have been added.

Task restore added

A new function ***ctl_task_restore*** has been added that allows tasks that have been removed from the task list (using ***ctl_task_remove***) to be replaced on to the task list.

Suspended task state

A new task state suspended has been added which can be used rather than removing and restoring a task from the task list.

Thread specific data pointer

A new field ***data*** has been added to the task structure which can be used to store thread specific data.

Task execution time

A new global variable ***ctl_last_schedule_time*** has been added and a new field ***execution_time*** has been added to the task structure which keeps the cumulative number of timer ticks that the task has executed for.

Change to global interrupt functions

The functions

- `ctl_global_interrupts_disable`

- `ctl_global_interrupts_enable`

no longer return the previous interrupt enables state. If you need this use **`ctl_global_interrupts_set`** in preference.

Change to task set priority

The function **`ctl_task_set_priority`** returns the old task priority.

Header file changes

In CTL V1 the file **`ctl/include/ctl_api.h`** contained CTL declarations and board support declarations. In CTL V2 the file **`ctl/include/ctl.h`** contains CTL declarations. In CTL V2 the file **`ctl/include/ctl_api.h`** #includes `ctl/source/ctl.h` and has board support declarations for backwards compatibility.

Removed support for interrupt re-enabling

The following functions have been removed from CTL V2

- `ctl_global_interrupts_re_enable_from_isr`
- `ctl_global_interrupts_un_re_enable_from_isr`

These functions are #defined in `ctl_api.h` to use their libarm equivalents. Rather than use these functions it is recommended to re-enable interrupts in the **`irq_handler`**.

Removed programmable interrupt handler support

The following functions have been removed from CTL V2

- `ctl_set_isr`
- `ctl_unmask_isr`
- `ctl_mask_isr`

These functions are now declared in `ctl_api.h` and are implemented in board and CPU support packages.

Removed CPU specific timer functions

The following functions have been removed from CTL V2

- `ctl_start_timer`
- `ctl_get_ticks_per_second`

These functions are now declared in `ctl_api.h` and are implemented in board and CPU support packages.

Removed board specific functions

The following functions have been removed from CTL V2

- `ctl_board_init`
- `ctl_board_set_leds`
- `ctl_board_on_button_pressed`

These functions are now declared in `ctl_api.h` and are implemented in board support packages.

Moved libc mutex

The declaration of the event set **`ctl_libc_mutex`** has been moved into the implementation of the libc multi-threading helper functions.

Byte/Message queue additions

Functions to post and receive multiple bytes/messages.

Functions to query the state of byte/message queues.

Function to associate events that are set when byte/message queue are not empty/not full.

Usage of `ctl_global_interrupts_set`

In CTL V2.1 the usage of **`ctl_global_interrupts_set()`** has been replaced with usage of **`ctl_global_interrupts_disable()`** and **`ctl_global_interrupts_enable()`**. These functions are now implemented using compiler intrinsics in the default CTL build. You can rebuild with the C preprocessor define `__NO_USE_INTRINSICS__` set if you require **`ctl_global_interrupts_set`** to be used.

CTL sources

This section describes the files found in the CTL **source** directory.

ctl.h

Header file containing CTL declarations.

ctl.c

CTL core functionality.

ctl_evt.c

CTL event set implementation.

ctl_sem.c

CTL semaphore implementation.

ctl_bq.c

CTL byte queue implementation.

ctl_mq.c

CTL memory queue implementation.

ctl_mutex.c

CTL mutex implementation.

ctl_mem_blk

CTL memory block implementation.

ctl_libc.c

CrossWorks libc multi-threading support functions.

ctl_impl.h

Header file for functions used within the CTL implementation.

ctl_msp430_s.s

MSP430 specific functions.

main_ctl.c

Boiler plate ctl main function.

threads.js

Javascript for CrossWorks threads window.

MSP430 Library Reference

In addition to the Standard C Library, CrossWorks for MSP430 provides an additional set of library routines that you can use.

In this section

[cross_studio_io.h](#)

Describes the virtual console services and semi-hosting support that CrossStudio provides to help you when developing your applications.

[cruntime.h](#)

The header file `<cruntime.h>` defines the interface to functions that the C compiler calls when generating code. For instance, it contains the runtime routines for all floating point operators and conversion, and shifts, multiplies, and divides for each of the integer types. In general, you do not need to call these routines yourself directly, but they are documented here should you need to call them from assembly language. These functions abide by the standard calling conventions of the compiler.

[ctl.h](#)

Describes the C tasking library, a library of functions that enable you to run multiple tasks in a real-time system.

[in430.h](#)

Describes the intrinsic functions that are mainly compatible with IAR's EW430 v2 product.

[inmsp.h](#)

Describes the intrinsic functions that are mainly compatible with IAR's EW430 v3 product.

<cross_studio_io.h>

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_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_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_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_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
Debug Terminal Input Functions	
debug_getchar	Line-buffered character read
debug_gets	String read
File Functions	
debug_fgetc	Read a character from a stream
debug_fputc	Write a character
Debugger Functions	
debug_abort	Stop debugging
debug_break	Stop target
debug_enabled	Test if debug input/output is enabled
debug_exit	Stop debugging
debug_getargs	Get arguments
debug_loadsymbols	Load debugging symbols
debug_runtime_error	Stop and report error
debug_unloadsymbols	Unload debugging symbols
Misc Functions	
debug_time	get time
Misc Functions	
debug_getenv	Get environment variable value

debug_perror	Display error
debug_system	Execute command

debug_abort

Synopsis

```
void debug_abort();
```

Description

debug_abort causes the debugger to exit and a failure result is returned to the user.

debug_break

Synopsis

```
void debug_break();
```

Description

debug_break causes the debugger to stop the target and position the cursor at the line that called `debug_break`.

debug_clearerr

Synopsis

```
void debug_clearerr(DEBUG_FILE *stream);
```

Description

debug_clearerr clears any error indicator or end of file condition for the **stream**.

debug_enabled

Synopsis

```
int debug_enabled();
```

Description

debug_enabled returns non-zero if the debugger is connected - you can use this to test if a debug input/output functions will work.

debug_exit

Synopsis

```
void debug_exit(int result);
```

Description

debug_exit causes the debugger to exit and **result** is returned to the user.

debug_fclose

Synopsis

```
int debug_fclose(DEBUG_FILE *stream);
```

Description

debug_fclose flushes any buffered output of the **stream** and then closes the stream.

debug_fclose returns 0 on success or -1 if there was an error.

debug_feof

Synopsis

```
int debug_feof(DEBUG_FILE *stream);
```

Description

debug_feof returns non-zero if the end of file condition is set for the **stream**.

debug_ferror

Synopsis

```
int debug_ferror(DEBUG_FILE *stream);
```

Description

debug_ferror returns non-zero if the error indicator is set for the **stream**.

debug_fflush

Synopsis

```
int debug_fflush(DEBUG_FILE *stream);
```

Description

debug_fflush flushes any buffered output of the **stream**.

debug_fflush returns 0 on success or -1 if there was an error.

debug_fgetc

Synopsis

```
int debug_fgetc(DEBUG_FILE *stream);
```

Description

debug_fgetc reads and returns the next character on **stream** or -1 if no character is available.

debug_fgetpos

Synopsis

```
int debug_fgetpos(DEBUG_FILE *stream,  
                  long *pos);
```

Description

debug_fgetpos is equivalent to **debug_fseek**.

debug_fgets

Synopsis

```
char *debug_fgets(char *s,  
                  int n,  
                  DEBUG_FILE *stream);
```

Description

debug_fgets reads at most **n**-1 characters or the characters up to (and including) a newline from the input **stream** into the array pointed to by **s**. A null character is written to the array after the input characters.

debug_fgets returns **s** on success, or 0 on error or end of file.

debug_filesize

Synopsis

```
int debug_filesize(DEBUG_FILE *stream);
```

Description

debug_filesize returns the size of the file associated with the **stream** in bytes.

debug_filesize returns -1 on error.

debug_fopen

Synopsis

```
DEBUG_FILE *debug_fopen(const char *filename,  
                        const char *mode);
```

Description

debug_fopen opens the **filename** on the host PC and returns a stream or **0** if the open fails. The **filename** is a host PC filename which is opened relative to the debugger working directory. The **mode** is a string containing one of:

- **r** open file for reading.
- **w** create file for writing.
- **a** open or create file for writing and position at the end of the file.
- **r+** open file for reading and writing.
- **w+** create file for reading and writing.
- **a+** open or create text file for reading and writing and position at the end of the file.

followed by one of:

- **t** for a text file.
- **b** for a binary file.

debug_fopen returns a stream that can be used to access the file or **0** if the open fails.

debug_fprintf

Synopsis

```
int debug_fprintf(DEBUG_FILE *stream,  
                  const char *format,  
                  ...);
```

Description

debug_fprintf writes to **stream**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. The **format** string is a standard C printf format string. The actual formatting is performed on the host by the debugger and therefore **debug_fprintf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_fprintf returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

debug_fprintf_c

Synopsis

```
int debug_fprintf_c(DEBUG_FILE *stream,  
    __code const char *format,  
    ...);
```

Description

debug_fprintf_c is equivalent to **debug_fprintf** with the format string in code memory.

debug_fputc

Synopsis

```
int debug_fputc(int c,  
                DEBUG_FILE *stream);
```

Description

debug_fputc writes the character **c** to the output **stream**.

debug_fputc returns the character written or -1 if an error occurred.

debug_fputs

Synopsis

```
int debug_fputs(const char *s,  
                DEBUG_FILE *stream);
```

Description

debug_fputs writes the string pointed to by **s** to the output **stream** and appends a new-line character. The terminating null character is not written.

debug_fputs returns -1 if a write error occurs; otherwise it returns a nonnegative value.

debug_fread

Synopsis

```
int debug_fread(void *ptr,  
                int size,  
                int nobj,  
                DEBUG_FILE *stream);
```

Description

debug_fread reads from the input **stream** into the array **ptr** at most **nobj** objects of size **size**.

debug_fread returns the number of objects read. If this number is different from **nobj** then **debug_feof** and **debug_ferror** can be used to determine status.

debug_freopen

Synopsis

```
DEBUG_FILE *debug_freopen(const char *filename,  
                           const char *mode,  
                           DEBUG_FILE *stream);
```

Description

debug_freopen is the same as **debug_open** except the file associated with the **stream** is closed and the opened file is then associated with the **stream**.

debug_fscanf

Synopsis

```
int debug_fscanf(DEBUG_FILE *stream,  
                 const char *format,  
                 ...);
```

Description

debug_fscanf reads from the input **stream**, under control of the string pointed to by **format**, that specifies how subsequent arguments are converted for input. The **format** string is a standard C `scanf` format string. The actual formatting is performed on the host by the debugger and therefore **debug_fscanf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_fscanf returns number of characters read, or a negative value if an output or encoding error occurred.

debug_fscanf_c

Synopsis

```
int debug_fscanf_c(DEBUG_FILE *stream,  
                  __code const char *format,  
                  ...);
```

Description

debug_fscanf_c is equivalent to **debug_fscanf** with the format string in code memory.

debug_fseek

Synopsis

```
int debug_fseek(DEBUG_FILE *stream,  
                long offset,  
                int origin);
```

Description

debug_fseek sets the file position for the **stream**. A subsequent read or write will access data at that position.

The **origin** can be one of:

- **0** sets the position to **offset** bytes from the beginning of the file.
- **1** sets the position to **offset** bytes relative to the current position.
- **2** sets the position to **offset** bytes from the end of the file.

Note that for text files **offset** must be zero. **debug_fseek** returns zero on success, non-zero on error.

debug_fsetpos

Synopsis

```
int debug_fsetpos(DEBUG_FILE *stream,  
                  const long *pos);
```

Description

debug_fsetpos is equivalent to **debug_fseek** with 0 as the **origin**.

debug_ftell

Synopsis

```
long debug_ftell(DEBUG_FILE *stream);
```

Description

debug_ftell returns the current file position of the **stream**.

debug_ftell returns -1 on error.

debug_fwrite

Synopsis

```
int debug_fwrite(void *ptr,  
                 int size,  
                 int nobj,  
                 DEBUG_FILE *stream);
```

Description

debug_fwrite write to the output **stream** from the array **ptr** at most **nobj** objects of size **size**.

debug_fwrite returns the number of objects written. If this number is different from **nobj** then **debug_feof** and **debug_ferror** can be used to determine status.

debug_getargs

Synopsis

```
int debug_getargs(unsigned bufsize,  
                  unsigned char *buf);
```

Description

debug_getargs stores the debugger command line arguments into the memory pointed at by **buf** up to a maximum of **bufsize** bytes. The command line is stored as a C **argc** array of null terminated string and the number of entries is returned as the result.

debug_getch

Synopsis

```
int debug_getch();
```

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();
```

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();
```

Description

debug_kbhit polls the Debug Terminal for a character and returns a non-zero value if a character is available or 0 if not.

debug_loadsymbols

Synopsis

```
void debug_loadsymbols(const char *filename,  
                      const void *address,  
                      const char *breaksymbol);
```

Description

debug_loadsymbols instructs the debugger to load the debugging symbols in the file denoted by **filename**. The **filename** is a (macro expanded) host PC filename which is relative to the debugger working directory. The **address** is the load address which is required for debugging position independent executables, supply **NULL** for regular executables. The **breaksymbol** is the name of a symbol in the filename to set a temporary breakpoint on or **NULL**.

debug_perror

Synopsis

```
void debug_perror(const char *s);
```

Description

debug_perror displays the optional string **s** on the **Debug Terminal** together with a string corresponding to the **errno** value of the last Debug IO operation.

debug_printf

Synopsis

```
int debug_printf(const char *format,  
                ...);
```

Description

debug_printf writes to the **Debug Terminal**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. The **format** string is a standard C printf format string. The actual formatting is performed on the host by the debugger and therefore **debug_printf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_printf returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

debug_printf_c

Synopsis

```
int debug_printf_c(__code const char *format,  
                  ...);
```

Description

debug_printf_c is equivalent to **debug_printf** with the format string in code memory.

debug_putchar

Synopsis

```
int debug_putchar(int c);
```

Description

debug_putchar write the character **c** to the Debug Terminal.

debug_putchar returns the character written or -1 if a write error occurs.

debug_puts

Synopsis

```
int debug_puts(const char *);
```

Description

debug_puts writes the string *s* to the Debug Terminal followed by a new-line character.

debug_puts returns -1 if a write error occurs, otherwise it returns a nonnegative value.

debug_remove

Synopsis

```
int debug_remove(const char *filename);
```

Description

debug_remove removes the filename denoted by **filename** and returns **0** on success or **-1** on error. The **filename** is a host PC filename which is relative to the debugger working directory.

debug_rename

Synopsis

```
int debug_rename(const char *oldfilename,  
                 const char *newfilename);
```

Description

debug_rename renames the file denoted by **oldpath** to **newpath** and returns zero on success or non-zero on error. The **oldpath** and **newpath** are host PC filenames which are relative to the debugger working directory.

debug_rewind

Synopsis

```
void debug_rewind(DEBUG_FILE *stream);
```

Description

debug_rewind sets the current file position of the **stream** to the beginning of the file and clears any error and end of file conditions.

debug_runtime_error

Synopsis

```
void debug_runtime_error(const char *error);
```

Description

debug_runtime_error causes the debugger to stop the target, position the cursor at the line that called **debug_runtime_error**, and display the null-terminated string pointed to by **error**.

debug_scanf

Synopsis

```
int debug_scanf(const char *format,  
               ...);
```

Description

debug_scanf reads from the **Debug Terminal**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for input. The **format** string is a standard C scanf format string. The actual formatting is performed on the host by the debugger and therefore **debug_scanf** consumes only a very small amount of code and data space, only the overhead to call the function.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

debug_scanf returns number of characters read, or a negative value if an output or encoding error occurred.

debug_scanf_c

Synopsis

```
int debug_scanf_c(__code const char *format,  
                  ...);
```

Description

debug_scanf_c is equivalent to **debug_scanf** with the format string in code memory.

debug_system

Synopsis

```
int debug_system(char *command);
```

Description

debug_system executes the **command** with the host command line interpreter and returns the commands exit status.

debug_time

Synopsis

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

Description

debug_time returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC), according to the system clock of the host computer. The return value is stored in ***ptr** if **ptr** is not NULL.

debug_tmpfile

Synopsis

```
DEBUG_FILE *debug_tmpfile();
```

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.

<cruntime.h>

Integer multiplication	
__int16_mul	Multiply two 16-bit signed or unsigned integers forming a 16-bit product
__int16_mul_8x8	Multiply two 8-bit signed integers forming a 16-bit signed product
__int16_mul_asgn	Multiply a 16-bit signed or unsigned integer in memory by a 16-bit integer
__int32_mul	Multiply two 32-bit signed or unsigned integers forming a 32-bit product
__int32_mul_16x16	Multiply two 16-bit signed integers forming a 32-bit signed product
__int32_mul_asgn	Multiply a 32-bit signed or unsigned integer in memory by a 32-bit integer
__int64_mul	Multiply two 64-bit signed or unsigned integers forming a 64-bit product
__int64_mul_32x32	Multiply two 32-bit signed integers forming a 64-bit signed product
__int64_mul_asgn	Multiply a 64-bit signed or unsigned integer in memory by a 64-bit integer
__uint16_mul_8x8	Multiply two 8-bit unsigned integers forming a 16-bit unsigned product
__uint32_mul_16x16	Multiply two 16-bit unsigned integers forming a 32-bit unsigned product
__uint64_mul_32x32	Multiply two 32-bit unsigned integers forming a 64-bit unsigned product
Integer division	
__int16_div	Divide two 16-bit signed integers and return the 16-bit signed quotient
__int16_div_asgn	Divide a 16-bit signed integer in memory by a 16-bit signed integer
__int16_mod	Divide two 16-bit signed integers and return the 16-bit signed remainder after division
__int16_mod_asgn	Divide a 16-bit signed integer in memory by a 16-bit signed integer and assign it the 16-bit remainder
__int32_div	Divide two 32-bit signed integers and return the 32-bit signed quotient
__int32_div_asgn	Divide a 32-bit signed integer in memory by a 32-bit signed integer

__int32_mod	Divide two 32-bit signed integers and return the 32-bit signed remainder after division
__int32_mod_asgn	Divide a 32-bit signed integer in memory by a 32-bit signed integer and assign it the 32-bit remainder
__int64_div	Divide two 64-bit signed integers and return the 64-bit signed quotient
__int64_div_asgn	Divide a 64-bit signed integer in memory by a 64-bit signed integer
__int64_mod	Divide two 64-bit signed integers and return the 64-bit signed remainder after division
__int64_mod_asgn	Divide a 64-bit signed integer in memory by a 64-bit signed integer and assign it the 64-bit remainder
__uint16_div	Divide two 16-bit unsigned integers and return the 16-bit unsigned quotient
__uint16_div_asgn	Divide a 16-bit unsigned integer in memory by a 16-bit unsigned integer
__uint16_mod	Divide two 16-bit unsigned integers and return the 16-bit unsigned remainder after division
__uint16_mod_asgn	Divide a 16-bit unsigned integer in memory by a 16-bit unsigned integer and assign it the 16-bit remainder
__uint32_div	Divide two 32-bit unsigned integers and return the 32-bit unsigned quotient
__uint32_div_asgn	Divide a 32-bit unsigned integer in memory by a 32-bit unsigned integer
__uint32_mod	Divide two 32-bit unsigned integers and return the 32-bit unsigned remainder after division
__uint32_mod_asgn	Divide a 32-bit unsigned integer in memory by a 32-bit unsigned integer and assign it the 32-bit remainder
__uint64_div	Divide two 64-bit unsigned integers and return the 64-bit unsigned quotient
__uint64_div_asgn	Divide a 64-bit unsigned integer in memory by a 64-bit unsigned integer
__uint64_mod	Divide two 64-bit unsigned integers and return the 64-bit unsigned remainder after division
__uint64_mod_asgn	Divide a 64-bit unsigned integer in memory by a 64-bit unsigned integer and assign it the 64-bit remainder
Integer shifts	
__int16_asr	Shift a 16-bit signed integer arithmetically right by a variable number of bit positions

__int16_asr_asgn	Shift a 16-bit signed integer in memory arithmetically right by a variable number of bit positions
__int16_lsl	Shift a 16-bit signed integer left by a variable number of bit positions
__int16_lsl_asgn	Shift a 16-bit signed integer in memory left by a variable number of bit positions
__int16_lsr	Shift a 16-bit unsigned integer logically right by a variable number of bit positions
__int16_lsr_asgn	Shift a 16-bit unsigned integer in memory logically right by a variable number of bit positions
__int32_asr	Shift a 32-bit signed integer arithmetically right by a variable number of bit positions
__int32_asr_asgn	Shift a 32-bit signed integer in memory arithmetically right by a variable number of bit positions
__int32_lsl	Shift a 32-bit signed integer left by a variable number of bit positions
__int32_lsl_asgn	Shift a 32-bit signed integer in memory left by a variable number of bit positions
__int32_lsr	Shift a 32-bit unsigned integer logically right by a variable number of bit positions
__int32_lsr_asgn	Shift a 32-bit unsigned integer in memory logically right by a variable number of bit positions
__int64_asr	Shift a 64-bit signed integer arithmetically right by a variable number of bit positions
__int64_asr_asgn	Shift a 64-bit signed integer in memory arithmetically right by a variable number of bit positions
__int64_lsl	Shift a 64-bit signed integer left by a variable number of bit positions
__int64_lsl_asgn	Shift a 64-bit signed integer in memory left by a variable number of bit positions
__int64_lsr	Shift a 64-bit unsigned integer logically right by a variable number of bit positions
__int64_lsr_asgn	Shift a 64-bit unsigned integer in memory logically right by a variable number of bit positions
Floating-point arithmetic	
__float32_add	Add two 32-bit floating point values
__float32_add_1	Add one to a 32-bit floating point value
__float32_add_asgn	Add a 32-bit floating point value to a 32-bit floating point value in memory
__float32_div	Divide two 32-bit floating point values

__float32_div_asgn	Divide a 32-bit floating point value in memory by a 32-bit floating point value
__float32_mul	Multiply two 32-bit floating point values
__float32_mul_asgn	Multiply a 32-bit floating point value in memory by a 32-bit floating point value
__float32_neg	Negate a 32-bit floating point value
__float32_sqr	Square a 32-bit floating point value
__float32_sub	Subtract two 32-bit floating point values
__float32_sub_asgn	Subtract a 32-bit floating point value from a 32-bit floating point value in memory
__float64_add	Add two 64-bit floating point values
__float64_add_1	Add one to a 64-bit floating point value
__float64_add_asgn	Add a 64-bit floating point value to a 64-bit floating point value in memory
__float64_div	Divide two 64-bit floating point values
__float64_div_asgn	Divide a 64-bit floating point value in memory by a 64-bit floating point value
__float64_mul	Multiply two 64-bit floating point values
__float64_mul_asgn	Multiply a 64-bit floating point value in memory by a 64-bit floating point value
__float64_neg	Negate a 64-bit floating point value
__float64_sqr	Square a 64-bit floating point value
__float64_sub	Subtract two 64-bit floating point values
__float64_sub_asgn	Subtract a 64-bit floating point value from a 64-bit floating point value in memory
Floating point comparison	
__float32_eq	Compare two 32-bit floating point values for equality
__float32_eq_0	Compare 32-bit floating point value to zero
__float32_lt	Compare two 32-bit floating point values
__float32_lt_0	Compare 32-bit floating point value with zero
__float64_eq	Compare two 64-bit floating point values for equality
__float64_eq_0	Compare 64-bit floating point value to zero
__float64_lt	Compare two 64-bit floating point values
__float64_lt_0	Compare 64-bit floating point value with zero
Integer to floating point conversions	
__int16_to_float32	Convert a 16-bit signed integer to a 32-bit floating point value

__int16_to_float64	Convert a 16-bit signed integer to a 64-bit floating point value
__int32_to_float32	Convert a 32-bit signed integer to a 32-bit floating point value
__int32_to_float64	Convert a 32-bit signed integer to a 64-bit floating point value
__int64_to_float32	Convert a 64-bit signed integer to a 32-bit floating point value
__int64_to_float64	Convert a 64-bit signed integer to a 64-bit floating point value
__uint16_to_float32	Convert a 16-bit unsigned integer to a 32-bit floating point value
__uint16_to_float64	Convert a 16-bit unsigned integer to a 64-bit floating point value
__uint32_to_float32	Convert a 32-bit unsigned integer to a 32-bit floating point value
__uint32_to_float64	Convert a 32-bit unsigned integer to a 64-bit floating point value
__uint64_to_float32	Convert a 64-bit unsigned integer to a 32-bit floating point value
__uint64_to_float64	Convert a 64-bit unsigned integer to a 64-bit floating point value
Floating point to integer conversions	
__float32_to_int16	Convert a 32-bit floating point value to a 16-bit signed integer
__float32_to_int32	Convert a 32-bit floating point value to a 32-bit signed integer
__float32_to_int64	Convert a 32-bit floating point value to a 64-bit signed integer
__float32_to_uint16	Convert a 32-bit floating point value to a 16-bit unsigned integer
__float32_to_uint32	Convert a 32-bit floating point value to a 32-bit unsigned integer
__float32_to_uint64	Convert a 32-bit floating point value to a 64-bit unsigned integer
Floating point conversions	
__float32_to_float64	Convert a 32-bit floating point value to a 64-bit floating point value
__float64_to_float32	Convert a 64-bit floating point value to a 32-bit floating point value

__float32_add

Synopsis

```
float32_t __float32_add(float32_t augend,  
                       float32_t addend);
```

Description

__float32_add adds **addend** to **augend** and returns the sum as the result.

__float32_add_1

Synopsis

```
float32_t __float32_add_1(float32_t augend);
```

Description

__float32_add_1 adds one to **augend** and returns the sum as the result.

__float32_add_asgn

Synopsis

```
float32_t __float32_add_asgn(float32_t *augend,  
                             float32_t addend);
```

Description

__float32_add_asgn updates the floating-point value pointed to by **augend** by adding **addend** to it. The stored sum is returned as the result.

__float32_div

Synopsis

```
float32_t __float32_div(float32_t dividend,  
                       float32_t divisor);
```

Description

__float32_div divides **dividend** by **divisor** and returns the quotient as the result.

__float32_div_asgn

Synopsis

```
float32_t __float32_div_asgn(float32_t *dividend,  
                             float32_t divisor);
```

Description

__float32_div_asgn updates the floating-point value pointed to by **dividend** by dividing it by **divisor**. The stored quotient is returned as the result.

__float32_eq

Synopsis

```
int __float32_eq(float32_t arg0,  
                float32_t arg1);
```

Description

__float32_eq compares **arg0** to **arg1**. **__float32_eq** returns zero if **arg0** is different from **arg1**, and a non-zero value if **arg0** is equal to **arg1**.

__float32_eq_0

Synopsis

```
int __float32_eq_0(float32_t arg);
```

Description

__float32_eq_0 compares **arg** to zero. **__float32_eq_0** returns a non-zero value if **arg** is zero, and a zero value if **arg** is non-zero.

__float32_lt

Synopsis

```
int __float32_lt(float32_t arg0,  
                float32_t arg1);
```

Description

__float32_lt compares **arg0** to **arg1**. **__float32_lt** returns a non-zero value if **arg0** is less than **arg1**, and zero if **arg0** is equal to or greater than **arg1**.

__float32_lt_0

Synopsis

```
int __float32_lt_0(float32_t arg0,  
                  float32_t arg1);
```

Description

__float32_lt_0 compares **arg** to zero. **__float32_lt_0** returns a non-zero value if **arg** is less than zero, and zero if **arg0** is equal to or greater than zero.

__float32_mul

Synopsis

```
float32_t __float32_mul(float32_t multiplicand,  
                       float32_t multiplier);
```

Description

__float32_mul multiplies **multiplicand** by **multiplier** and returns the product as the result.

__float32_mul_asgn

Synopsis

```
float32_t __float32_mul_asgn(float32_t *multiplicand,  
                             float32_t multiplier);
```

Description

__float32_mul_asgn updates the floating-point value pointed to by **multiplicand** by multiplying it by **multiplier**. The stored product is returned as the result.

__float32_neg

Synopsis

```
float32_t __float32_neg(float32_t arg);
```

Description

__float32_neg negates **arg** and returns the result.

__float32_sqr

Synopsis

```
float32_t __float32_sqr(float32_t arg);
```

Description

__float32_sqr squares **arg** by multiplying **arg** by itself.

__float32_sub

Synopsis

```
float32_t __float32_sub(float32_t minuend,  
                       float32_t subtrahend);
```

Description

__float32_sub subtracts **subtrahend** from **minuend** and returns the difference as the result.

__float32_sub_asgn

Synopsis

```
float32_t __float32_sub_asgn(float32_t *minuend,  
                             float32_t subtrahend);
```

Description

__float32_sub_asgn updates the floating-point value pointed to by **minuend** by subtracting **subtrahend** from it. The stored difference is returned as the result.

__float32_to_float64

Synopsis

```
float64_t __float32_to_float64(float32_t arg);
```

Description

__float32_to_float64 converts the 32-bit floating value **arg** to a 64-bit floating point value and returns the converted value as the result.

__float32_to_int16

Synopsis

```
int16_t __float32_to_int16(float32_t arg);
```

Description

__float32_to_int16 converts the floating value **arg** to a 16-bit signed integer, truncating towards zero, and returns the truncated value as the result.

__float32_to_int32

Synopsis

```
int32_t __float32_to_int32(float32_t arg);
```

Description

__float32_to_int32 converts the floating value **arg** to a 32-bit signed integer, truncating towards zero, and returns the truncated value as the result.

__float32_to_int64

Synopsis

```
int64_t __float32_to_int64(float32_t arg);
```

Description

__float32_to_int64 converts the floating value **arg** to a 64-bit signed integer, truncating towards zero, and returns the truncated value as the result.

__float32_to_uint16

Synopsis

```
uint16_t __float32_to_uint16(float32_t arg);
```

Description

__float32_to_uint16 converts the floating value **arg** to a 16-bit unsigned integer, truncating towards zero, and returns the truncated value as the result.

__float32_to_uint32

Synopsis

```
uint32_t __float32_to_uint32(float32_t arg);
```

Description

__float32_to_uint32 converts the floating value **arg** to a 32-bit unsigned integer, truncating towards zero, and returns the truncated value as the result.

__float32_to_uint64

Synopsis

```
uint64_t __float32_to_uint64(float32_t arg);
```

Description

__float32_to_uint64 converts the floating value **arg** to a 64-bit unsigned integer, truncating towards zero, and returns the truncated value as the result.

__float64_add

Synopsis

```
float64_t __float64_add(float64_t augend,  
                        float64_t addend);
```

Description

__float64_add adds **addend** to **augend** and returns the sum as the result.

__float64_add_1

Synopsis

```
float64_t __float64_add_1(float64_t augend);
```

Description

__float64_add_1 adds one to **augend** and returns the sum as the result.

__float64_add_asgn

Synopsis

```
float64_t __float64_add_asgn(float64_t *augend,  
                             float64_t addend);
```

Description

__float64_add_asgn updates the floating-point value pointed to by **augend** by adding **addend** to it. The stored sum is returned as the result.

__float64_div

Synopsis

```
float64_t __float64_div(float64_t dividend,  
                        float64_t divisor);
```

Description

__float64_div divides **dividend** by **divisor** and returns the quotient as the result.

__float64_div_asgn

Synopsis

```
float64_t __float64_div_asgn(float64_t *dividend,  
                             float64_t divisor);
```

Description

__float64_div_asgn updates the floating-point value pointed to by **dividend** by dividing it by **divisor**. The stored quotient is returned as the result.

__float64_eq

Synopsis

```
int __float64_eq(float64_t arg0,  
                 float64_t arg1);
```

Description

__float64_eq compares **arg0** to **arg1**. **__float64_eq** returns zero if **arg0** is different from **arg1**, and a non-zero value if **arg0** is equal to **arg1**.

__float64_eq_0

Synopsis

```
int __float64_eq_0(float64_t arg);
```

Description

__float64_eq_0 compares **arg** to zero. **__float64_eq_0** returns a non-zero value if **arg** is zero, and a zero value if **arg** is non-zero.

__float64_lt

Synopsis

```
int __float64_lt(float64_t arg0,  
                float64_t arg1);
```

Description

__float64_lt compares **arg0** to **arg1**. **__float64_lt** returns a non-zero value if **arg0** is less than **arg1**, and zero if **arg0** is equal to or greater than **arg1**.

__float64_lt_0

Synopsis

```
int __float64_lt_0(float64_t arg0,  
                  float64_t arg1);
```

Description

__float64_lt_0 compares **arg** to zero. **__float64_lt_0** returns a non-zero value if **arg** is less than zero, and zero if **arg0** is equal to or greater than zero.

__float64_mul

Synopsis

```
float64_t __float64_mul(float64_t multiplicand,  
                        float64_t multiplier);
```

Description

__float64_mul multiplies **multiplicand** by **multiplier** and returns the product as the result.

__float64_mul_asgn

Synopsis

```
float64_t __float64_mul_asgn(float64_t *multiplicand,  
                             float64_t multiplier);
```

Description

__float64_mul_asgn updates the floating-point value pointed to by **multiplicand** by multiplying it by **multiplier**. The stored product is returned as the result.

__float64_neg

Synopsis

```
float64_t __float64_neg(float64_t arg);
```

Description

__float64_neg negates **arg** and returns the result.

__float64_sqr

Synopsis

```
float64_t __float64_sqr(float64_t arg);
```

Description

__float64_sqr squares **arg** by multiplying **arg** by itself.

__float64_sub

Synopsis

```
float64_t __float64_sub(float64_t minuend,  
                        float64_t subtrahend);
```

Description

__float64_sub subtracts **subtrahend** from **minuend** and returns the difference as the result.

__float64_sub_asgn

Synopsis

```
float64_t __float64_sub_asgn(float64_t *minuend,  
                             float64_t subtrahend);
```

Description

__float64_sub_asgn updates the floating-point value pointed to by **minuend** by subtracting **subtrahend** from it. The stored difference is returned as the result.

__float64_to_float32

Synopsis

```
float32_t __float64_to_float32(float64_t arg);
```

Description

__float64_to_float32 converts the 64-bit floating value **arg** to a 32-bit floating point value and returns the converted value as the result.

`__int16_asr`

Synopsis

```
int16_t __int16_asr(int16_t arg,  
                   int bits);
```

Description

`__int16_asr` shifts **arg** arithmetically right by **bits** bit positions, replicating the sign bit, and returns the shifted result.

__int16_asr_asgn

Synopsis

```
int16_t __int16_asr_asgn(int16_t *arg,  
                        int bits);
```

Description

__int16_asr_asgn updates the 16-bit signed integer pointed to by **arg** by arithmetically shifting it right by **its** bit positions, replicating the sign bit. The shifted value is returned as the result.

`__int16_div`

Synopsis

```
int16_t __int16_div(int16_t dividend,  
                  int16_t divisor);
```

Description

`__int16_div` divides **dividend** by **divisor** and returns the signed quotient, truncated towards zero, as the result.

__int16_div_asgn

Synopsis

```
int16_t __int16_div_asgn(int16_t *dividend,  
                        int16_t divisor);
```

Description

__int16_div_asgn updates the 16-bit signed integer pointed to by **dividend** by dividing it by **divisor** and truncated towards zero. The quotient is returned as the result.

__int16_lsl

Synopsis

```
int16_t __int16_lsl(int16_t arg,  
                   int bits);
```

Description

__int16_lsl shifts **arg** left by **bits** bit positions, shifting zeros in from the left.

`__int16_lsl_asgn`

Synopsis

```
uint16_t __int16_lsl_asgn(uint16_t *arg,  
                          int bits);
```

Description

`__int16_lsl_asgn` updates the 16-bit unsigned integer pointed to by **arg** by shifting it left by **bits** bit positions, shifting in zeros in from the right. The shifted value is returned as the result.

`__int16_lsr`

Synopsis

```
uint16_t __int16_lsr(uint16_t arg,  
                     int bits);
```

Description

`__int16_lsr` shifts **arg** logically right by **bits** bit positions, shifting in zeros from the left, and returns the shifted result.

`__int16_lsr_asgn`

Synopsis

```
uint16_t __int16_lsr_asgn(uint16_t *arg,  
                           int bits);
```

Description

`__int16_lsr_asgn` updates the 16-bit unsigned integer pointed to by **arg** by logically shifting it right by **bits** bit positions, shifting in zeros from the right. The shifted value is returned as the result.

__int16_mod

Synopsis

```
int16_t __int16_mod(int16_t dividend,  
                  int16_t divisor);
```

Description

__int16_mod divides **dividend** by **divisor** and returns the signed remainder after division as the result.

__int16_mod_asgn

Synopsis

```
int16_t __int16_mod_asgn(int16_t *dividend,  
                        int16_t divisor);
```

Description

__int16_mod_asgn updates the 16-bit signed integer pointed to by **dividend** by assigning it the remainder after division of **dividend** by **divisor**. The remainder is returned as the result.

__int16_mul

Synopsis

```
int16_t __int16_mul(int16_t multiplicand,  
                  int16_t multiplier);
```

Description

__int16_mul multiplies **multiplicand** by **multiplier** and returns the product as the result. As only the lower 16 bits of the product are returned, **__int16_mul** returns correct products, modulo 16 bits, for both signed and unsigned arguments.

`__int16_mul_8x8`

Synopsis

```
int16_t __int16_mul_8x8(int8_t multiplicand,  
                       int8_t multiplier);
```

Description

`__int16_mul_8x8` multiplies **`multiplicand`** by **`multiplier`** and returns the 16-bit signed product as the result.

__int16_mul_asgn

Synopsis

```
int16_t __int16_mul_asgn(int16_t *multiplicand,  
                        int16_t multiplier);
```

Description

__int16_mul_asgn updates the 16-bit signed integer pointed to by **multiplicand** by multiplying it by **multiplier**. The product is returned as the result. As only the lower 16 bits of the product are returned, **__int16_mul_asgn** returns correct products, modulo 16 bits, for both signed and unsigned arguments.

__int16_to_float32

Synopsis

```
float32_t __int16_to_float32(int16_t arg);
```

Description

__int16_to_float32 converts the 16-bit signed integer **arg** to a 32-bit floating value and returns the floating value as the result. As all 16-bit integers can be represented exactly in 32-bit floating point format, rounding is never necessary.

__int16_to_float64

Synopsis

```
float64_t __int16_to_float64(int16_t arg);
```

Description

__int16_to_float64 converts the 16-bit signed integer **arg** to a 64-bit floating value and returns the floating value as the result. As all 16-bit integers can be represented exactly in 64-bit floating point format, rounding is never necessary.

`__int32_asr`

Synopsis

```
int32_t __int32_asr(int32_t arg,  
                  int bits);
```

Description

`__int32_asr` shifts **arg** arithmetically right by **bits** bit positions, replicating the sign bit, and returns the shifted result.

__int32_asr_asgn

Synopsis

```
int32_t __int32_asr_asgn(int32_t *arg,  
                        int bits);
```

Description

__int32_asr_asgn updates the 32-bit signed integer pointed to by **arg** by arithmetically shifting it right by **its** bit positions, replicating the sign bit. The shifted value is returned as the result.

`__int32_div`

Synopsis

```
int32_t __int32_div(int32_t dividend,  
                  int32_t divisor);
```

Description

`__int32_div` divides **dividend** by **divisor** and returns the signed quotient, truncated towards zero, as the result.

`__int32_div_asgn`

Synopsis

```
int32_t __int32_div_asgn(int32_t *dividend,  
                        int32_t divisor);
```

Description

`__int32_div_asgn` updates the 32-bit signed integer pointed to by **dividend** by dividing it by **divisor** and truncated towards zero. The quotient is returned as the result.

__int32_lsl

Synopsis

```
int32_t __int32_lsl(int32_t arg,  
                  int bits);
```

Description

__int32_lsl shifts **arg** left by **bits** bit positions, shifting zeros in from the left.

`__int32_lsl_asgn`

Synopsis

```
uint32_t __int32_lsl_asgn(uint32_t *arg,  
                          int bits);
```

Description

`__int32_lsl_asgn` updates the 32-bit unsigned integer pointed to by **arg** by shifting it left by **bits** bit positions, shifting in zeros in from the right. The shifted value is returned as the result.

`__int32_lsr`

Synopsis

```
uint32_t __int32_lsr(uint32_t arg,  
                    int bits);
```

Description

`__int32_lsr` shifts **arg** logically right by **bits** bit positions, shifting in zeros from the left, and returns the shifted result.

`__int32_lsr_asgn`

Synopsis

```
uint32_t __int32_lsr_asgn(uint32_t *arg,  
                          int bits);
```

Description

`__int32_lsr_asgn` updates the 32-bit unsigned integer pointed to by **arg** by logically shifting it right by **bits** bit positions, shifting in zeros from the right. The shifted value is returned as the result.

`__int32_mod`

Synopsis

```
int32_t __int32_mod(int32_t dividend,  
                  int32_t divisor);
```

Description

`__int32_mod` divides **dividend** by **divisor** and returns the signed remainder after division as the result.

`__int32_mod_asgn`

Synopsis

```
int32_t __int32_mod_asgn(int32_t *dividend,  
                        int32_t divisor);
```

Description

`__int32_mod_asgn` updates the 32-bit signed integer pointed to by **dividend** by assigning it the remainder after division of **dividend** by **divisor**. The remainder is returned as the result.

`__int32_mul`

Synopsis

```
int32_t __int32_mul(int32_t multiplicand,  
                  int32_t multiplier);
```

Description

`__int32_mul` multiplies **`multiplicand`** by **`multiplier`** and returns the product as the result. As only the lower 32 bits of the product are returned, `__int32_mul` returns correct products, modulo 32 bits, for both signed and unsigned arguments.

`__int32_mul_16x16`

Synopsis

```
int32_t __int32_mul_16x16(int16_t multiplicand,  
                        int16_t multiplier);
```

Description

this multiplies **multiplicand** by **multiplier** and returns the 32-bit signed product as the result.

__int32_mul_asgn

Synopsis

```
int32_t __int32_mul_asgn(int32_t *multiplicand,  
                        int32_t multiplier);
```

Description

__int32_mul_asgn updates the 32-bit signed integer pointed to by **multiplicand** by multiplying it by **multiplier**. The product is returned as the result. As only the lower 32 bits of the product are returned, **__int32_mul_asgn** returns correct products, modulo 32 bits, for both signed and unsigned arguments.

__int32_to_float32

Synopsis

```
float32_t __int32_to_float32(int32_t arg);
```

Description

__int32_to_float32 converts the 32-bit signed integer **arg** to a 32-bit floating value, rounding if required, and returns the appropriately rounded value as the result.

__int32_to_float64

Synopsis

```
float64_t __int32_to_float64(int32_t arg);
```

Description

__int32_to_float64 converts the 32-bit signed integer **arg** to a 64-bit floating value and returns the floating value as the result. As all 32-bit integers can be represented exactly in 64-bit floating point format, rounding is never necessary.

`__int64_asr`

Synopsis

```
int64_t __int64_asr(int64_t arg,  
                   int bits);
```

Description

`__int64_asr` shifts **arg** arithmetically right by **bits** bit positions, replicating the sign bit, and returns the shifted result.

__int64_asr_asgn

Synopsis

```
int64_t __int64_asr_asgn(int64_t *arg,  
                        int bits);
```

Description

__int64_asr_asgn updates the 64-bit signed integer pointed to by **arg** by arithmetically shifting it right by **its** bit positions, replicating the sign bit. The shifted value is returned as the result.

__int64_div

Synopsis

```
int64_t __int64_div(int64_t dividend,  
                  int64_t divisor);
```

Description

__int64_div divides **dividend** by **divisor** and returns the signed quotient, truncated towards zero, as the result.

__int64_div_asgn

Synopsis

```
int64_t __int64_div_asgn(int64_t *dividend,  
                        int64_t divisor);
```

Description

__int64_div_asgn updates the 64-bit signed integer pointed to by **dividend** by dividing it by **divisor** and truncated towards zero. The quotient is returned as the result.

__int64_lsl

Synopsis

```
int64_t __int64_lsl(int64_t arg,  
                   int bits);
```

Description

__int64_lsl shifts **arg** left by **bits** bit positions, shifting zeros in from the left.

`__int64_lsl_asgn`

Synopsis

```
uint64_t __int64_lsl_asgn(uint64_t *arg,  
                          int bits);
```

Description

`__int64_lsl_asgn` updates the 64-bit unsigned integer pointed to by **arg** by shifting it left by **bits** bit positions, shifting in zeros in from the right. The shifted value is returned as the result.

__int64_lsr

Synopsis

```
uint64_t __int64_lsr(uint64_t arg,  
                    int bits);
```

Description

__int64_lsr shifts **arg** logically right by **bits** bit positions, shifting in zeros from the left, and returns the shifted result.

__int64_lsr_asgn

Synopsis

```
uint64_t __int64_lsr_asgn(uint64_t *arg,  
                          int bits);
```

Description

__int64_lsr_asgn updates the 64-bit unsigned integer pointed to by **arg** by logically shifting it right by **bits** bit positions, shifting in zeros from the right. The shifted value is returned as the result.

`__int64_mod`

Synopsis

```
int64_t __int64_mod(int64_t dividend,  
                   int64_t divisor);
```

Description

`__int64_mod` divides **dividend** by **divisor** and returns the signed remainder after division as the result.

__int64_mod_asgn

Synopsis

```
int64_t __int64_mod_asgn(int64_t *dividend,  
                        int64_t divisor);
```

Description

__int64_mod_asgn updates the 64-bit signed integer pointed to by **dividend** by assigning it the remainder after division of **dividend** by **divisor**. The remainder is returned as the result.

`__int64_mul`

Synopsis

```
int64_t __int64_mul(int64_t multiplicand,  
                  int64_t multiplier);
```

Description

`__int64_mul` multiplies **`multiplicand`** by **`multiplier`** and returns the product as the result. As only the lower 64 bits of the product are returned, `__int64_mul` returns correct products, modulo 64 bits, for both signed and unsigned arguments.

`__int64_mul_32x32`

Synopsis

```
int64_t __int64_mul_32x32(int32_t multiplicand,  
                        int32_t multiplier);
```

Description

this multiplies **multiplicand** by **multiplier** and returns the 64-bit signed product as the result.

__int64_mul_asgn

Synopsis

```
int64_t __int64_mul_asgn(int64_t *multiplicand,  
                        int64_t multiplier);
```

Description

__int64_mul_asgn updates the 64-bit signed integer pointed to by **multiplicand** by multiplying it by **multiplier**. The product is returned as the result. As only the lower 64 bits of the product are returned, **__int64_mul_asgn** returns correct products, modulo 64 bits, for both signed and unsigned arguments.

__int64_to_float32

Synopsis

```
float32_t __int64_to_float32(int64_t arg);
```

Description

__int64_to_float32 converts the 64-bit signed integer **arg** to a 32-bit floating value, rounding if required, and returns the appropriately rounded value as the result.

__int64_to_float64

Synopsis

```
float64_t __int64_to_float64(int64_t arg);
```

Description

__int64_to_float64 converts the 64-bit signed integer **arg** to a 64-bit floating value, rounding if required, and returns the appropriately rounded value as the result.

__uint16_div

Synopsis

```
uint16_t __uint16_div(uint16_t dividend,  
                     uint16_t divisor);
```

Description

__uint16_div divides **dividend** by **divisor** and returns the unsigned quotient, truncated towards zero, as the result.

`__uint16_div_asgn`

Synopsis

```
uint16_t __uint16_div_asgn(uint16_t *dividend,  
                           uint16_t divisor);
```

Description

`__uint16_div_asgn` updates the 16-bit unsigned integer pointed to by **dividend** by dividing it by **divisor** and truncated towards zero. The quotient is returned as the result.

__uint16_mod

Synopsis

```
uint16_t __uint16_mod(uint16_t dividend,  
                     uint16_t divisor);
```

Description

__uint16_mod divides **dividend** by **divisor** and returns the unsigned remainder after division as the result.

__uint16_mod_asgn

Synopsis

```
uint16_t __uint16_mod_asgn(uint16_t *dividend,  
                           uint16_t divisor);
```

Description

__uint16_mod_asgn updates the 16-bit unsigned integer pointed to by **dividend** by assigning it the remainder after division of **dividend** by **divisor**. The remainder is returned as the result.

__uint16_mul_8x8

Synopsis

```
uint16_t __uint16_mul_8x8(uint8_t multiplicand,  
                           uint8_t multiplier);
```

Description

__uint16_mul_8x8 multiplies **multiplicand** by **multiplier** and returns the 16-bit unsigned product as the result.

__uint16_to_float32

Synopsis

```
float32_t __uint16_to_float32(uint16_t arg);
```

Description

__uint16_to_float32 converts the 16-bit unsigned integer **arg** to a 32-bit floating value and returns the floating value as the result. As all 16-bit unsigned integers can be represented exactly in 32-bit floating point format, rounding is never necessary.

__uint16_to_float64

Synopsis

```
float64_t __uint16_to_float64(uint16_t arg);
```

Description

__uint16_to_float64 converts the 16-bit unsigned integer **arg** to a 64-bit floating value and returns the floating value as the result. As all 16-bit unsigned integers can be represented exactly in 64-bit floating point format, rounding is never necessary.

__uint32_div

Synopsis

```
uint32_t __uint32_div(uint32_t dividend,  
                     uint32_t divisor);
```

Description

__uint32_div divides **dividend** by **divisor** and returns the unsigned quotient, truncated towards zero, as the result.

__uint32_div_asgn

Synopsis

```
uint32_t __uint32_div_asgn(uint32_t *dividend,  
                           uint32_t divisor);
```

Description

__uint32_div_asgn updates the 32-bit unsigned integer pointed to by **dividend** by dividing it by **divisor** and truncated towards zero. The quotient is returned as the result.

__uint32_mod

Synopsis

```
uint32_t __uint32_mod(uint32_t dividend,  
                     uint32_t divisor);
```

Description

__uint32_mod divides **dividend** by **divisor** and returns the unsigned remainder after division as the result.

__uint32_mod_asgn

Synopsis

```
uint32_t __uint32_mod_asgn(uint32_t *dividend,  
                           uint32_t divisor);
```

Description

__uint32_mod_asgn updates the 32-bit unsigned integer pointed to by **dividend** by assigning it the remainder after division of **dividend** by **divisor**. The remainder is returned as the result.

__uint32_mul_16x16

Synopsis

```
uint32_t __uint32_mul_16x16(uint16_t multiplicand,  
                             uint16_t multiplier);
```

Description

__uint32_mul_16x16 multiplies **multiplicand** by **multiplier** and returns the 32-bit unsigned product as the result.

__uint32_to_float32

Synopsis

```
float32_t __uint32_to_float32(uint32_t arg);
```

Description

__uint32_to_float32 converts the 32-bit unsigned integer **arg** to a 32-bit floating value, rounding if required, and returns the appropriately rounded value as the result.

__uint32_to_float64

Synopsis

```
float64_t __uint32_to_float64(uint32_t arg);
```

Description

__uint32_to_float64 converts the 32-bit unsigned integer **arg** to a 64-bit floating value and returns the floating value as the result. As all 32-bit unsigned integers can be represented exactly in 64-bit floating point format, rounding is never necessary.

__uint64_div

Synopsis

```
uint64_t __uint64_div(uint64_t dividend,  
                     uint64_t divisor);
```

Description

__uint64_div divides **dividend** by **divisor** and returns the unsigned quotient, truncated towards zero, as the result.

__uint64_div_asgn

Synopsis

```
uint64_t __uint64_div_asgn(uint64_t *dividend,  
                           uint64_t divisor);
```

Description

__uint64_div_asgn updates the 64-bit unsigned integer pointed to by **dividend** by dividing it by **divisor** and truncated towards zero. The quotient is returned as the result.

__uint64_mod

Synopsis

```
uint64_t __uint64_mod(uint64_t dividend,  
                      uint64_t divisor);
```

Description

__uint64_mod divides **dividend** by **divisor** and returns the unsigned remainder after division as the result.

__uint64_mod_asgn

Synopsis

```
uint64_t __uint64_mod_asgn(uint64_t *dividend,  
                           uint64_t divisor);
```

Description

__uint64_mod_asgn updates the 64-bit unsigned integer pointed to by **dividend** by assigning it the remainder after division of **dividend** by **divisor**. The remainder is returned as the result.

__uint64_mul_32x32

Synopsis

```
uint64_t __uint64_mul_32x32(uint32_t multiplicand,  
                             uint32_t multiplier);
```

Description

__uint64_mul_32x32 multiplies **multiplicand** by **multiplier** and returns the 64-bit unsigned product as the result.

__uint64_to_float32

Synopsis

```
float32_t __uint64_to_float32(uint64_t arg);
```

Description

__uint64_to_float32 converts the 64-bit unsigned integer **arg** to a 32-bit floating value, rounding if required, and returns the appropriately rounded value as the result.

__uint64_to_float64

Synopsis

```
float64_t __uint64_to_float64(uint64_t arg);
```

Description

__uint64_to_float64 converts the 64-bit unsigned integer **arg** to a 64-bit floating value, rounding if required, and returns the appropriately rounded value as the result.

<ctl.h>

Task	
ctl_task_die	Terminate the executing task
ctl_task_init	Create the initial task
ctl_task_remove	Remove a task from the task list
ctl_task_reschedule	Cause a reschedule
ctl_task_restore	Put back a task on to the task list
ctl_task_run	Start a task
ctl_task_set_priority	Set the priority of a task
Time	
ctl_get_current_time	Atomically return the current time
ctl_increment_tick_from_isr	Increment tick timer
ctl_timeout_wait	Wait until timeout has occurred
Event Set	
ctl_events_init	Initialize an event set
ctl_events_pulse	Pulse events in an event set
ctl_events_set_clear	Set and clear events in an event set
ctl_events_wait	Wait for events in an event set
Semaphore	
ctl_semaphore_init	Initialize a semaphore
ctl_semaphore_signal	Signal a semaphore
ctl_semaphore_wait	Wait for a semaphore
Message Queue	
ctl_message_queue_init	Initialize a message queue
ctl_message_queue_num_free	Return number of free elements in a message queue
ctl_message_queue_num_used	Return number of used elements in a message queue
ctl_message_queue_post	Post message to a message queue
ctl_message_queue_post_multi	Post messages to a message queue
ctl_message_queue_post_multi_nb	Post messages to a message queue without blocking
ctl_message_queue_post_nb	Post message to a message queue without blocking
ctl_message_queue_receive	Receive message from a message queue
ctl_message_queue_receive_multi	Receive messages from a message queue
ctl_message_queue_receive_multi_nb	Receive messages from a message queue without blocking

ctl_message_queue_receive_nb	Receive message from a message queue without blocking
ctl_message_queue_setup_events	Associate events with the not-full and not-empty state of a message queue
Byte Queue	
ctl_byte_queue_init	Initialize a byte queue
ctl_byte_queue_num_free	Return number of free bytes in a byte queue
ctl_byte_queue_num_used	Return number of used bytes in a byte queue
ctl_byte_queue_post	Post byte to a byte queue
ctl_byte_queue_post_multi	Post bytes to a byte queue
ctl_byte_queue_post_multi_nb	Post bytes to a byte queue without blocking
ctl_byte_queue_post_nb	Post byte to a byte queue without blocking
ctl_byte_queue_receive	Receive a byte from a byte queue
ctl_byte_queue_receive_multi	Receive multiple bytes from a byte queue
ctl_byte_queue_receive_multi_nb	Receive multiple bytes from a byte queue without blocking
ctl_byte_queue_receive_nb	Receive a byte from a byte queue without blocking
ctl_byte_queue_setup_events	Associate events with the not-full and not-empty state of a byte queue
Mutex	
ctl_mutex_init	Initialize a mutex
ctl_mutex_lock	Lock a mutex
ctl_mutex_unlock	Unlock a mutex
Interrupts	
ctl_global_interrupts_disable	Disable global interrupts
ctl_global_interrupts_enable	Enable global interrupts
ctl_global_interrupts_set	Enable/disable interrupts
User-provided functions	
ctl_handle_error	Handle a CTL error condition
Memory Area	
ctl_memory_area_allocate	Allocate a block from a memory area
ctl_memory_area_free	Free a memory area block
ctl_memory_area_init	Initialize a memory area
ctl_memory_area_setup_events	Set memory area events
Types	
CTL_BYTE_QUEUE_t	Byte queue struct definition

CTL_EVENT_SET_t	Event set definition
CTL_MEMORY_AREA_t	Memory area struct definition
CTL_MESSAGE_QUEUE_t	Message queue struct definition
CTL_MUTEX_t	Mutex struct definition
CTL_SEMAPHORE_t	Semaphore definition
CTL_TASK_t	Task struct definition
CTL_TIME_t	Time definition
System state variables	
ctl_current_time	The current time in ticks
ctl_interrupt_count	Nested interrupt count
ctl_last_schedule_time	The time (in ticks) of the last task schedule
ctl_reschedule_on_last_isr_exit	Reschedule is required on last ISR exit
ctl_task_executing	The task that is currently executing
ctl_task_list	List of tasks sorted by priority
ctl_task_switch_callout	A function pointer called on a task switch
ctl_time_increment	Current time tick increment
ctl_timeslice_period	Time slice period in ticks

CTL_BYTE_QUEUE_t

Synopsis

```
typedef struct {
    unsigned char *q;
    unsigned s;
    unsigned front;
    unsigned n;
    CTL_EVENT_SET_t *e;
    CTL_EVENT_SET_t notempty;
    CTL_EVENT_SET_t notfull;
} CTL_BYTE_QUEUE_t;
```

Description

CTL_BYTE_QUEUE_t defines the byte queue structure. The byte queue structure contains:

Member	Description
q	pointer to the array of bytes
s	size of the array of bytes
front	the next byte to leave the byte queue
n	the number of elements in the byte queue
e	the event set to use for the not empty and not full events
notempty	the event number for a not empty event
notfull	the event number for a not full event

CTL_EVENT_SET_t

Synopsis

```
unsigned CTL_EVENT_SET_t;
```

Description

CTL_EVENT_SET_t defines an event set. Event sets are word sized 16 or 32 depending on the machine.

CTL_MEMORY_AREA_t

Synopsis

```
typedef struct {  
    unsigned *head;  
    CTL_EVENT_SET_t *e;  
    CTL_EVENT_SET_t blockavailable;  
} CTL_MEMORY_AREA_t;
```

Description

CTL_MEMORY_AREA_t defines the memory area structure. The memory area structure contains:

Member	Description
head	the next free memory block
e	the event set containing the blockavailable event
blockavailable	the blockavailable event

CTL_MESSAGE_QUEUE_t

Synopsis

```
typedef struct {  
    void ** q;  
    unsigned s;  
    unsigned front;  
    unsigned n;  
    CTL_EVENT_SET_t *e;  
    CTL_EVENT_SET_t notempty;  
    CTL_EVENT_SET_t notfull;  
} CTL_MESSAGE_QUEUE_t;
```

Description

CTL_MESSAGE_QUEUE_t defines the message queue structure. The message queue structure contains:

Member	Description
q	pointer to the array of message queue objects
s	size of the array of message queue objects
front	the next element to leave the message queue
n	the number of elements in the message queue
e	the event set to use for the not empty and not full events
notempty	the event number for a not empty event
notfull	the event number for a not full event

CTL_MUTEX_t

Synopsis

```
typedef struct {  
    unsigned lock_count;  
    CTL_TASK_t *locking_task;  
    unsigned locking_task_priority;  
} CTL_MUTEX_t;
```

Description

CTL_MUTEX_t defines the mutex structure. The mutex structure contains:

Member	Description
lock_count	number of times the mutex has been locked
locking_task	the task that has locked the mutex
locking_task_priority	the priority of the task at the time it locked the mutex

CTL_SEMAPHORE_t

Synopsis

```
unsigned CTL_SEMAPHORE_t;
```

Description

CTL_SEMAPHORE_t defines the semaphore type. Semaphores are held in one word, 16 or 32 bits depending on the machine.

CTL_TASK_t

Synopsys

```
typedef struct {
    unsigned *stack_pointer;
    unsigned char priority;
    unsigned char state;
    unsigned char timeout_occured;
    CTL_TIME_t timeout;
    void *wait_object;
    CTL_EVENT_SET_t wait_events;
    int thread_errno;
    void *data;
    CTL_TIME_t execution_time;
    unsigned *stack_start;
    char *name;
} CTL_TASK_t;
```

Description

CTL_TASK_t defines the task structure. The task structure contains:

Member	Description
stack_pointer	the saved register state of the task when it is not scheduled
priority	the priority of the task
state	the state of task CTL_STATE_RUNNABLE or (CTL_STATE_*_WAIT_* CTL_STATE_TIMER_WAIT) or CTL_STATE_SUSPENDED
timeout_occured	1 if a wait timed out otherwise 0 - when state is CTL_RUNNABLE
next	next pointer for wait queue
timeout	wait timeout value or time slice value when the task is executing
wait_object	the event set, semaphore, message queue or mutex to wait on
wait_events	the events to wait for
thread_errno	thread specific errno
data	task specific data pointer
execution_time	number of ticks the task has executed for
stack_start	the start (lowest address) of the stack
name	task name

CTL_TIME_t

Synopsis

```
unsigned long CTL_TIME_t;
```

Description

CTL_TIME_t defines the base type for times that CTL uses.

ctl_byte_queue_init

Synopsis

```
void ctl_byte_queue_init(CTL_BYTE_QUEUE_t *q,  
                        unsigned char *queue,  
                        unsigned queue_size);
```

Description

ctl_byte_queue_init is given a pointer to the byte queue to initialize in **q**. The array that will be used to implement the byte queue pointed to by **queue** and its size in **queue_size** are also supplied.

ctl_byte_queue_num_free

Synopsis

```
unsigned ctl_byte_queue_num_free(CTL_BYTE_QUEUE_t *q);
```

Description

ctl_byte_queue_num_free returns the number of free bytes in the byte queue **q**.

ctl_byte_queue_num_used

Synopsis

```
unsigned ctl_byte_queue_num_used(CTL_BYTE_QUEUE_t *q);
```

Description

ctl_byte_queue_num_used returns the number of used elements in the byte queue **q**.

ctl_byte_queue_post

Synopsis

```
unsigned ctl_byte_queue_post(CTL_BYTE_QUEUE_t *q,  
                             unsigned char b,  
                             CTL_TIMEOUT_t t,  
                             CTL_TIME_t timeout);
```

Description

ctl_byte_queue_post posts **b** to the byte queue pointed to by **q**. If the byte queue is full then the caller will block until the byte can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value.

ctl_byte_queue_post returns zero if the timeout occurred otherwise it returns one.

Note

ctl_byte_queue_post must not be called from an interrupt service routine.

ctl_byte_queue_post_multi

Synopsis

```
unsigned ctl_byte_queue_post_multi(CTL_BYTE_QUEUE_t *q,  
                                  unsigned n,  
                                  unsigned char *b,  
                                  CTL_TIMEOUT_t t,  
                                  CTL_TIME_t timeout);
```

Description

ctl_byte_queue_post_multi posts **n** bytes to the byte queue pointed to by **q**. The caller will block until the bytes can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value.

ctl_byte_queue_post_multi returns the number of bytes that were posted.

Note

ctl_byte_queue_post_multi must not be called from an interrupt service routine.

ctl_byte_queue_post_multi does not guarantee that the bytes will be all be posted to the byte queue atomically. If you have multiple tasks posting (multiple bytes) to the same byte queue then you may get unexpected results.

ctl_byte_queue_post_multi_nb

Synopsis

```
unsigned ctl_byte_queue_post_multi_nb(CTL_BYTE_QUEUE_t *q,  
                                     unsigned n,  
                                     unsigned char *b);
```

Description

ctl_byte_queue_post_multi_nb posts **n** bytes to the byte queue pointed to by **q**.

ctl_byte_queue_post_multi_nb returns the number of bytes that were posted.

ctl_byte_queue_post_nb

Synopsis

```
unsigned ctl_byte_queue_post_nb(CTL_BYTE_QUEUE_t *q,  
                                unsigned char b);
```

Description

ctl_byte_queue_post_nb posts **b** to the byte queue pointed to by **q**. If the byte queue is full then the function will return zero otherwise it will return one.

ctl_byte_queue_receive

Synopsis

```
unsigned ctl_byte_queue_receive(CTL_BYTE_QUEUE_t *q,  
                               unsigned char *b,  
                               CTL_TIMEOUT_t t,  
                               CTL_TIME_t timeout);
```

Description

ctl_byte_queue_receive pops the oldest byte in the byte queue pointed to by **q** into the memory pointed to by **b**. **ctl_byte_queue_receive** will block if no bytes are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_byte_queue_receive returns zero if a timeout occurs otherwise 1.

Note

ctl_byte_queue_receive must not be called from an interrupt service routine.

ctl_byte_queue_receive_multi

Synopsis

```
unsigned ctl_byte_queue_receive_multi(CTL_BYTE_QUEUE_t *q,  
                                     unsigned n,  
                                     unsigned char *b,  
                                     CTL_TIMEOUT_t t,  
                                     CTL_TIME_t timeout);
```

Description

ctl_byte_queue_receive_multi pops the oldest **n** bytes in the byte queue pointed to by **q** into the memory pointed at by **b**. **ctl_byte_queue_receive_multi** will block until the number of bytes are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_byte_queue_receive_multi returns the number of bytes that have been received.

Note

ctl_byte_queue_receive_multi must not be called from an interrupt service routine.

ctl_byte_queue_receive_multi_nb

Synopsis

```
unsigned ctl_byte_queue_receive_multi_nb(CTL_BYTE_QUEUE_t *q,  
                                         unsigned n,  
                                         unsigned char *b);
```

Description

ctl_byte_queue_receive_multi_nb pops the oldest **n** bytes in the byte queue pointed to by **q** into the memory pointed to by **b**. **ctl_byte_queue_receive_multi_nb** returns the number of bytes that have been received.

ctl_byte_queue_receive_nb

Synopsis

```
unsigned ctl_byte_queue_receive_nb(CTL_BYTE_QUEUE_t *q,  
                                  unsigned char *b);
```

Description

ctl_byte_queue_receive_nb pops the oldest byte in the byte queue pointed to by **m** into the memory pointed to by **b**. If no bytes are available the function returns zero otherwise it returns 1.

ctl_byte_queue_setup_events

Synopsis

```
void ctl_byte_queue_setup_events(CTL_BYTE_QUEUE_t *q,  
                                CTL_EVENT_SET_t *e,  
                                CTL_EVENT_SET_t notempty,  
                                CTL_EVENT_SET_t notfull);
```

Description

ctl_byte_queue_setup_events registers events in the event set **e** that are set when the byte queue **q** becomes **notempty** or becomes **notfull**. No scheduling will occur with this operation, you need to do this before waiting for events.

ctl_current_time

Synopsis

```
CTL_TIME_t ctl_current_time;
```

Description

ctl_current_time holds the current time in ticks. **ctl_current_time** is incremented by **ctl_increment_ticks_from_isr**.

Note

For portable programs without race conditions you should not read this variable directly, you should use **ctl_get_current_time** instead. As this variable is changed by an interrupt, it cannot be read atomically on processors whose word size is less than 32 bits without first disabling interrupts. That said, you can read this variable directly in your interrupt handler as long as interrupts are still disabled.

ctl_events_init

Synopsis

```
void ctl_events_init(CTL_EVENT_SET_t *e,  
                    CTL_EVENT_SET_t set);
```

Description

ctl_events_init initializes the event set **e** with the **set** values.

ctl_events_pulse

Synopsis

```
void ctl_events_pulse(CTL_EVENT_SET_t *e,  
                     CTL_EVENT_SET_t set_then_clear);
```

Description

ctl_events_pulse will set the events defined by **set_then_clear** in the event set pointed to by **e**.

ctl_events_pulse will then search the task list, matching tasks that are waiting on the event set **e**, and make them runnable if the match is successful. The events defined by **set_then_clear** are then cleared.

See Also

[ctl_events_set_clear](#).

ctl_events_set_clear

Synopsis

```
void ctl_events_set_clear(CTL_EVENT_SET_t *e,  
                          CTL_EVENT_SET_t set,  
                          CTL_EVENT_SET_t clear);
```

Description

ctl_events_set_clear sets the events defined by **set** and clears the events defined by **clear** of the event set pointed to by **e**. **ctl_events_set_clear** will then search the task list, matching tasks that are waiting on the event set **e** and make them runnable if the match is successful.

See Also

[ctl_events_pulse](#).

ctl_events_wait

Synopsis

```
unsigned ctl_events_wait(CTL_EVENT_WAIT_TYPE_t type,  
                        CTL_EVENT_SET_t *eventSet,  
                        CTL_EVENT_SET_t events,  
                        CTL_TIMEOUT_t t,  
                        CTL_TIME_t timeout);
```

Description

ctl_events_wait waits for **events** to be set (value 1) in the event set pointed to by **eventSet** with an optional **timeout** applied if **timeoutType** is non-zero.

The **waitType** can be one of:

- **CTL_EVENT_WAIT_ANY_EVENTS** — wait for any of **events** in **eventSet** to be set.
- **CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR** — wait for any of **events** in **eventSet** to be set and reset (value 0) them.
- **CTL_EVENT_WAIT_ALL_EVENTS** — wait for all **events** in ***eventSet** to be set.
- **CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR** — wait for all **events** in **eventSet** to be set and reset (value 0) them.

ctl_events_wait returns the value pointed to by **eventSet** before any auto-clearing occurred or zero if the **imeout** occurred.

Note

ctl_events_wait must not be called from an interrupt service routine.

ctl_get_current_time

Synopsis

```
CTL_TIME_t ctl_get_current_time();
```

Description

ctl_get_current_time atomically reads the value of **ctl_current_time**.

ctl_global_interrupts_disable

Synopsis

```
int ctl_global_interrupts_disable();
```

Description

ctl_global_interrupts_disable disables global interrupts. If **ctl_global_interrupts_disable** is called and interrupts are already disabled then it will return 0. If **ctl_global_interrupts_disable** is called and interrupts are enabled then it will return non-zero which may or may not represent the true interrupt disabled state. **ctl_global_interrupts_disable** is used to provide exclusive access to CTL data structures the implementation of it may or may not disable global interrupts.

ctl_global_interrupts_enable

Synopsis

```
void ctl_global_interrupts_enable();
```

Description

ctl_global_interrupts_enable enables global interrupts. **ctl_global_interrupts_enable** is used to provide exclusive access to CTL data structures the implementation of it may or may not disable global interrupts.

ctl_global_interrupts_set

Synopsis

```
int ctl_global_interrupts_set(int enable);
```

Description

ctl_global_interrupts_set disables or enables global interrupts according to the state **enable**. If **enable** is zero, interrupts are disabled and if **enable** is non-zero, interrupts are enabled. If **ctl_global_interrupts_set** is called and interrupts are already disabled then it will return 0. If **ctl_global_interrupts_set** is called and interrupts are enabled then it will return non-zero which may or may not represent the true interrupt disabled state. **ctl_global_interrupts_set** is used to provide exclusive access to CTL data structures the implementation of it may or may not disable global interrupts.

ctl_handle_error

Synopsis

```
void ctl_handle_error(CTL_ERROR_CODE_t e);
```

Description

ctl_handle_error is a function that you must supply in your application that handles errors detected by the CrossWorks tasking library.

The errors that can be reported in **e** are:

- **CTL_ERROR_NO_TASKS_TO_RUN** — a reschedule has occurred but there are no tasks which are runnable.
- **CTL_WAIT_CALLED_FROM_ISR** — an interrupt service routine has called a tasking library function that could block.
- **CTL_UNSPECIFIED_ERROR** — an unspecified error has occurred.

ctl_increment_tick_from_isr

Synopsis

```
void ctl_increment_tick_from_isr();
```

Description

ctl_increment_tick_from_isr increments **ctl_current_time** by the number held in **ctl_time_increment** and does rescheduling. This function should be called from a periodic interrupt service routine.

Note

ctl_increment_tick_from_isr must only be invoked by an interrupt service routine.

ctl_interrupt_count

Synopsis

```
unsigned char ctl_interrupt_count;
```

Description

ctl_interrupt_count contains a count of the interrupt nesting level. This variable must be incremented immediately on entry to an interrupt service routine and decremented immediately before return from the interrupt service routine.

ctl_last_schedule_time

Synopsis

```
CTL_TIME_t ctl_last_schedule_time;
```

Description

ctl_last_schedule_time contains the time (in ticks) of the last task schedule.

Description

ctl_last_schedule_time contains the time of the last reschedule in ticks.

ctl_memory_area_allocate

Synopsis

```
unsigned *ctl_memory_area_allocate(CTL_MEMORY_AREA_t *memory_area);
```

Description

ctl_memory_area_allocate allocates a block from the initialized memory area **memory_area**.

ctl_memory_area_allocate returns a block of the size specified in the call to [ctl_memory_area_init](#) or zero if no blocks are available.

ctl_memory_area_allocate executes in constant time and is very fast. You can call **ctl_memory_area_allocate** from an interrupt service routine, from a task, or from initialization code.

ctl_memory_area_free

Synopsis

```
void ctl_memory_area_free(CTL_MEMORY_AREA_t *memory_area,  
                          unsigned *block);
```

Description

ctl_memory_area_free is given a pointer to a memory area **memory_area** which has been initialized and a **block** that has been returned by [ctl_memory_area_allocate](#). The block is returned to the memory area so that it can be allocated again.

ctl_memory_area_init

Synopsis

```
void ctl_memory_area_init(CTL_MEMORY_AREA_t *memory_area,  
                          unsigned *memory,  
                          unsigned block_size_in_words,  
                          unsigned num_blocks);
```

Description

ctl_memory_area_init is given a pointer to the memory area to initialize in **memory_area**. The array that is used to implement the memory area is pointed to by **memory**. The size of a memory block is given supplied in **block_size_in_words** and the number of block is supplied in **num_blocks**.

Note

memory must point to a block of memory that is at least **block_size_in_words** × **num_blocks** words long.

ctl_memory_area_setup_events

Synopsis

```
void ctl_memory_area_setup_events(CTL_MEMORY_AREA_t *m,  
                                CTL_EVENT_SET_t *e,  
                                CTL_EVENT_SET_t blockavailable);
```

Description

ctl_memory_area_setup_events registers the events **blockavailable** in the event set **e** that are set when a block becomes available in the the memory area **m**.

ctl_message_queue_init

Synopsis

```
void ctl_message_queue_init(CTL_MESSAGE_QUEUE_t *q,  
                           void **queue,  
                           unsigned queue_size);
```

Description

ctl_message_queue_init is given a pointer to the message queue to initialize in **q**. The array that will be used to implement the message queue pointed to by **queue** and its size in **queue_size** are also supplied.

ctl_message_queue_num_free

Synopsis

```
unsigned ctl_message_queue_num_free(CTL_MESSAGE_QUEUE_t *q);
```

Description

ctl_message_queue_num_free returns the number of free elements in the message queue **q**.

ctl_message_queue_num_used

Synopsis

```
unsigned ctl_message_queue_num_used(CTL_MESSAGE_QUEUE_t *q);
```

Description

ctl_message_queue_num_used returns the number of used elements in the message queue **q**.

ctl_message_queue_post

Synopsis

```
unsigned ctl_message_queue_post(CTL_MESSAGE_QUEUE_t *q,  
                               void *message,  
                               CTL_TIMEOUT_t t,  
                               CTL_TIME_t timeout);
```

Description

ctl_message_queue_post posts **message** to the message queue pointed to by **q**. If the message queue is full then the caller will block until the message can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value. **ctl_message_queue_post** returns zero if the timeout occurred otherwise it returns one.

Note

ctl_message_queue_post must not be called from an interrupt service routine.

ctl_message_queue_post_multi

Synopsis

```
unsigned ctl_message_queue_post_multi(CTL_MESSAGE_QUEUE_t *q,  
                                     unsigned n,  
                                     void **messages,  
                                     CTL_TIMEOUT_t t,  
                                     CTL_TIME_t timeout);
```

Description

ctl_message_queue_post_multi posts **n messages** to the message queue pointed to by **q**. The caller will block until the messages can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value. **ctl_message_queue_post_multi** returns the number of messages that were posted.

Note

ctl_message_queue_post_multi must not be called from an interrupt service routine.

ctl_message_queue_post_multi function does not guarantee that the messages will be all be posted to the message queue atomically. If you have multiple tasks posting (multiple messages) to the same message queue then you may get unexpected results.

ctl_message_queue_post_multi_nb

Synopsis

```
unsigned ctl_message_queue_post_multi_nb(CTL_MESSAGE_QUEUE_t *q,  
                                         unsigned n,  
                                         void **messages);
```

Description

ctl_message_queue_post_multi_nb posts **n messages** to the message queue pointed to by **m**.

ctl_message_queue_post_multi_nb returns the number of messages that were posted.

ctl_message_queue_post_nb

Synopsis

```
unsigned ctl_message_queue_post_nb(CTL_MESSAGE_QUEUE_t *q,  
                                   void *message);
```

Description

ctl_message_queue_post_nb posts **message** to the message queue pointed to by **q**. If the message queue is full then the function will return zero otherwise it will return one.

ctl_message_queue_receive

Synopsis

```
unsigned ctl_message_queue_receive(CTL_MESSAGE_QUEUE_t *q,  
                                   void **message,  
                                   CTL_TIMEOUT_t t,  
                                   CTL_TIME_t timeout);
```

Description

ctl_message_queue_receive pops the oldest message in the message queue pointed to by **q** into the memory pointed to by **message**. **ctl_message_queue_receive** will block if no messages are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_message_queue_receive returns zero if a timeout occurs otherwise 1.

Note

ctl_message_queue_receive must not be called from an interrupt service routine.

ctl_message_queue_receive_multi

Synopsis

```
unsigned ctl_message_queue_receive_multi(CTL_MESSAGE_QUEUE_t *q,  
                                         unsigned n,  
                                         void **messages,  
                                         CTL_TIMEOUT_t t,  
                                         CTL_TIME_t timeout);
```

Description

ctl_message_queue_receive_multi pops the oldest **n** messages in the message queue pointed to by **q** into the memory pointed to by **message**. **ctl_message_queue_receive_multi** will block until all the messages are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_message_queue_receive_multi returns the number of messages that were received.

Note

ctl_message_queue_receive_multi must not be called from an interrupt service routine.

ctl_message_queue_receive_multi_nb

Synopsis

```
unsigned ctl_message_queue_receive_multi_nb(CTL_MESSAGE_QUEUE_t *q,  
                                             unsigned n,  
                                             void **messages);
```

Description

ctl_message_queue_receive_multi_nb pops the oldest **n** messages in the message queue pointed to by **q** into the memory pointed to by **message**.

ctl_message_queue_receive_multi_nb returns the number of messages that were received.

ctl_message_queue_receive_nb

Synopsis

```
unsigned ctl_message_queue_receive_nb(CTL_MESSAGE_QUEUE_t *q,  
                                     void **message);
```

Description

ctl_message_queue_receive_nb pops the oldest message in the message queue pointed to by **q** into the memory pointed to by **message**. If no messages are available the function returns zero otherwise it returns 1.

ctl_message_queue_setup_events

Synopsis

```
void ctl_message_queue_setup_events(CTL_MESSAGE_QUEUE_t *q,  
                                   CTL_EVENT_SET_t *e,  
                                   CTL_EVENT_SET_t notempty,  
                                   CTL_EVENT_SET_t notfull);
```

Description

ctl_message_queue_setup_events registers events in the event set **e** that are set when the message queue **q** becomes **notempty** or becomes **notfull**. No scheduling will occur with this operation, you need to do this before waiting for events.

ctl_mutex_init

Synopsis

```
void ctl_mutex_init(CTL_MUTEX_t *m);
```

Description

ctl_mutex_init initializes the mutex pointed to by **m**.

ctl_mutex_lock

Synopsis

```
unsigned ctl_mutex_lock(CTL_MUTEX_t *m,  
                        CTL_TIMEOUT_t t,  
                        CTL_TIME_t timeout);
```

Description

ctl_mutex_lock locks the mutex pointed to by **m** to the calling task. If the mutex is already locked by the calling task then the mutex lock count is incremented. If the mutex is already locked by a different task then the caller will block until the mutex is unlocked. In this case, if the priority of the task that has locked the mutex is less than that of the caller the priority of the task that has locked the mutex is raised to that of the caller whilst the mutex is locked. If **timeoutType** is non-zero and the current time reaches the **timeout** value before the lock is acquired the function returns zero otherwise it returns one.

Note

ctl_mutex_lock must not be called from an interrupt service routine.

ctl_mutex_unlock

Synopsis

```
void ctl_mutex_unlock(CTL_MUTEX_t *m);
```

Description

ctl_mutex_unlock unlocks the mutex pointed to by **m**. The mutex must have previously been locked by the calling task. If the calling task's priority has been raised (by another task calling **ctl_mutex_unlock** whilst the mutex was locked), then the calling task's priority will be restored.

Note

ctl_mutex_unlock must not be called from an interrupt service routine.

ctl_reschedule_on_last_isr_exit

Synopsis

```
unsigned char ctl_reschedule_on_last_isr_exit;
```

Description

ctl_reschedule_on_last_isr_exit is set to a non-zero value if a CTL call is made from an interrupt service routine that requires a task reschedule. This variable is checked and reset on exit from the last nested interrupt service routine.

ctl_semaphore_init

Synopsis

```
void ctl_semaphore_init(CTL_SEMAPHORE_t *s,  
                        unsigned value);
```

Description

ctl_semaphore_init initializes the semaphore pointed to by **s** to **value**.

ctl_semaphore_signal

Synopsis

```
void ctl_semaphore_signal(CTL_SEMAPHORE_t *s);
```

Description

ctl_semaphore_signal signals the semaphore pointed to by **s**. If tasks are waiting for the semaphore then the highest priority task will be made runnable. If no tasks are waiting for the semaphore then the semaphore value will be incremented.

ctl_semaphore_wait

Synopsis

```
unsigned ctl_semaphore_wait(CTL_SEMAPHORE_t *s,  
                           CTL_TIMEOUT_t t,  
                           CTL_TIME_t timeout);
```

Description

ctl_semaphore_wait waits for the semaphore pointed to by **s** to be non-zero. If the semaphore is zero then the caller will block unless **timeoutType** is non-zero and the current time reaches the **timeout** value. If the timeout occurred **ctl_semaphore_wait** returns zero otherwise it returns one.

Note

ctl_semaphore_wait must not be called from an interrupt service routine.

ctl_task_die

Synopsis

```
void ctl_task_die();
```

Description

ctl_task_die terminates the currently executing task and schedules the next runnable task.

ctl_task_executing

Synopsis

```
CTL_TASK_t *ctl_task_executing;
```

Description

ctl_task_executing points to the **CTL_TASK_t** structure of the currently executing task. The **priority** field is the only field in the **CTL_TASK_t** structure that is defined for the task that is executing. It is an error if **ctl_task_executing** is **NULL**.

ctl_task_init

Synopsis

```
void ctl_task_init(CTL_TASK_t *task,  
                  unsigned char priority,  
                  const char *name);
```

Description

ctl_task_init turns the main program into a task. This function takes a pointer in **task** to the **CTL_TASK_t** structure that represents the main task, its **priority** (0 is the lowest priority, 255 the highest), and a zero-terminated string pointed by **name**. On return from this function global interrupts will be enabled.

The function must be called before any other CrossWorks tasking library calls are made.

ctl_task_list

Synopsis

```
CTL_TASK_t *ctl_task_list;
```

Description

ctl_task_list points to the **CTL_TASK_t** structure of the highest priority task that is not executing. It is an error if **ctl_task_list** is **NULL**.

ctl_task_remove

Synopsis

```
void ctl_task_remove(CTL_TASK_t *task);
```

Description

ctl_task_remove removes the task **task** from the waiting task list. Once you have removed a task the only way to re-introduce it to the system is to call **ctl_task_restore**.

ctl_task_reschedule

Synopsis

```
void ctl_task_reschedule();
```

Description

ctl_task_reschedule causes a reschedule to occur. This can be used by tasks of the same priority to share the CPU without using timeslicing.

ctl_task_restore

Synopsis

```
void ctl_task_restore(CTL_TASK_t *task);
```

Description

ctl_task_restore adds a task **task** that was removed (using **ctl_task_remove**) onto the task list and do scheduling.

ctl_task_run

Synopsis

```
void ctl_task_run(CTL_TASK_t *task,
                  unsigned char priority,
                  void (*entrypoint)(void *),
                  void *parameter,
                  const char *name,
                  unsigned stack_size_in_words,
                  unsigned *stack,
                  unsigned call_size_in_words);
```

Description

ctl_task_run takes a pointer in **task** to the **CTL_TASK_t** structure that represents the task. The **priority** can be zero for the lowest priority up to 255 which is the highest. The **entrypoint** parameter is the function that the task will execute which has the **parameter** passed to it.

name is a pointer to a zero-terminated string used for debug purposes.

The start of the memory used to hold the stack that the task will execute in is **stack** and the size of the memory is supplied in **stack_size_in_words**. On systems that have two stacks (e.g. Atmel AVR) then the **call_size_in_words** parameter must be set to specify the number of stack elements to use for the call stack.

ctl_task_set_priority

Synopsis

```
unsigned char ctl_task_set_priority(CTL_TASK_t *task,  
                                   unsigned char priority);
```

Description

ctl_task_set_priority changes the priority of **task** to **priority**. The priority can be 0, the lowest priority, to 255, which is the highest priority.

You can change the priority of the currently executing task by passing **ctl_task_executing** as the **task** parameter. **ctl_task_set_priority** returns the previous priority of the task.

ctl_task_switch_callout

Synopsis

```
void (*ctl_task_switch_callout)(CTL_TASK_t *);
```

Description

ctl_task_switch_callout contains a pointer to a function that is called (if it is set) when a task schedule occurs. The task that will be scheduled is supplied as a parameter to the function (**ctl_task_executing** will point to the currently scheduled task).

Note that the callout function is called from the CTL scheduler and as such any use of CTL services whilst executing the callout function has undefined behavior.

Note

Because this function pointer is called in an interrupt service routine, you should assign it before interrupts are started or with interrupts turned off.

ctl_time_increment

Synopsis

```
unsigned ctl_time_increment;
```

Description

ctl_time_increment contains the value that **ctl_current_time** is incremented when **ctl_increment_tick_from_isr** is called.

ctl_timeout_wait

Synopsis

```
void ctl_timeout_wait(CTL_TIME_t timeout);
```

Description

ctl_timeout_wait takes the **timeout** (not the duration) as a parameter and suspends the calling task until the current time reaches the timeout.

Note

ctl_timeout_wait must not be called from an interrupt service routine.

ctl_timeslice_period

Synopsis

```
CTL_TIME_t ctl_timeslice_period;
```

Description

ctl_timeslice_period contains the number of ticks to allow a task to run before it will be preemptively rescheduled by a task of the same priority. The variable is set to 0x7ffffff by default so that only higher priority tasks will be preemptively scheduled.

<inmsp.h>

Miscellaneous functions	
__delay_cycles	Delay execution for a number of cycles
__even_in_range	Assert value is a restricted range
__insert_opcode	Insert an opcode
__no_operation	Insert a NOP instruction
Status register manipulation	
__bic_SR_register	Clear bits in status register
__bic_SR_register_on_exit	Clear bits in stacked status register
__bis_SR_register	Set bits in status register
__bis_SR_register_on_exit	Set bits in stacked status register
__disable_interrupt	Disable global interrupts
__enable_interrupt	Enable global interrupts
__low_power_mode_0	Enter low power mode 0
__low_power_mode_1	Enter low power mode 1
__low_power_mode_2	Enter low power mode 2
__low_power_mode_3	Enter low power mode 3
__low_power_mode_4	Enter low power mode 4
__low_power_mode_off_on_exit	Enter active mode when interrupt return
__set_interrupt	Restore global interrupts
Byte order manipulation	
__swap_bytes	Swap order of bytes in a word
__swap_long_bytes	Swap order of bytes in a long
__swap_words	Swap order of words in a long
Bit order manipulation	
__bit_reverse_char	Reverse the order of bits in a char
__bit_reverse_long	Reverse the order of bits in a long
__bit_reverse_long_long	Reverse the order of bits in a long long
__bit_reverse_short	Reverse the order of bits in a short
Bit counting	
__bit_count_leading_zeros_char	Count the number of leading zero bits in a char
__bit_count_leading_zeros_long	Count the number of leading zero bits in a long
__bit_count_leading_zeros_long_long	Count the number of leading zero bits in a long long
__bit_count_leading_zeros_short	Count the number of leading zero bits in a short

Register manipulation	
__get_register	Read from processor register
__set_register	Write to processor register
Binary coded decimal arithmetic	
__bcd_add_long	Add two 32-bit values using decimal arithmetic
__bcd_add_long_long	Add two 64-bit values using decimal arithmetic
__bcd_add_short	Add two 16-bit values using decimal arithmetic
__bcd_negate_long	Negate a 32-bit value using decimal arithmetic
__bcd_negate_long_long	Negate a 64-bit value using decimal arithmetic
__bcd_negate_short	Negate a 16-bit value using decimal arithmetic
__bcd_subtract_long	Subtract two 32-bit values using decimal arithmetic
__bcd_subtract_long_long	Subtract two 64-bit values using decimal arithmetic
__bcd_subtract_short	Subtract two 16-bit values using decimal arithmetic
Extended memory functions	
__read_extended_byte	Read a byte from extend memory
__read_extended_long	Read a long from extended memory
__read_extended_word	Read a word from extended memory
__write_extended_byte	Write a byte to extended memory
__write_extended_long	Write a long to extended memory
__write_extended_word	Write a word to extended memory

__bcd_add_long

Synopsis

```
unsigned long __bcd_add_long(unsigned long x,  
                             unsigned long y);
```

Description

__bcd_add_long adds **x** to **y** using decimal arithmetic. Both **x** and **y** must be binary coded decimal numbers and the result is a binary coded decimal number.

__bcd_add_long is an intrinsic function and produces inline code.

__bcd_add_long_long

Synopsis

```
unsigned long long __bcd_add_long_long(unsigned long long x,  
                                       unsigned long long y);
```

Description

__bcd_add_long_long adds **x** to **y** using decimal arithmetic. Both **x** and **y** must be binary coded decimal numbers and the result is a binary coded decimal number.

__bcd_add_long_long is an intrinsic function and produces inline code.

__bcd_add_short

Synopsis

```
unsigned __bcd_add_short(unsigned x,  
                        unsigned y);
```

Description

__bcd_add_short adds **x** to **y** using decimal arithmetic. Both **x** and **y** must be binary coded decimal numbers and the result is a binary coded decimal number.

__bcd_add_short is an intrinsic function and produces inline code.

__bcd_negate_long

Synopsis

```
unsigned long __bcd_negate_long(unsigned long x);
```

Description

__bcd_negate_long negates (computes the 9's complement of) **x** using decimal arithmetic. **x** must be a binary coded decimal number and the result is a binary coded decimal number.

__bcd_negate_long is an intrinsic function and produces inline code.

__bcd_negate_long_long

Synopsis

```
unsigned long long __bcd_negate_long_long(unsigned long long x);
```

Description

__bcd_negate_long_long negates (computes the 9's complement of) **x** using decimal arithmetic. **x** must be a binary coded decimal number and the result is a binary coded decimal number.

__bcd_negate_long_long is an intrinsic function and produces inline code.

__bcd_negate_short

Synopsis

```
unsigned __bcd_negate_short(unsigned x);
```

Description

__bcd_negate_short negates (computes the 9's complement of) **x** using decimal arithmetic. **x** must be a binary coded decimal number and the result is a binary coded decimal number.

__bcd_negate_short is an intrinsic function and produces inline code.

__bcd_subtract_long

Synopsis

```
unsigned long __bcd_subtract_long(unsigned long x,  
                                unsigned long y);
```

Description

__bcd_subtract_long subtracts **y** from **x** using decimal arithmetic. Both **x** and **y** must be binary coded decimal numbers and the result is a binary coded decimal number.

__bcd_subtract_long is an intrinsic function and produces inline code.

__bcd_subtract_long_long

Synopsis

```
unsigned long long __bcd_subtract_long_long(unsigned long long x,  
                                             unsigned long long y);
```

Description

__bcd_subtract_long_long subtracts **y** from **x** using decimal arithmetic. Both **x** and **y** must be binary coded decimal numbers and the result is a binary coded decimal number.

__bcd_subtract_long_long is an intrinsic function and produces inline code.

__bcd_subtract_short

Synopsis

```
unsigned __bcd_subtract_short(unsigned x,  
                             unsigned y);
```

Description

__bcd_subtract_short subtracts **y** from **x** using decimal arithmetic. Both **x** and **y** must be binary coded decimal numbers and the result is a binary coded decimal number.

__bcd_subtract_short is an intrinsic function and produces inline code.

__bic_SR_register

Synopsis

```
unsigned __bic_SR_register(unsigned mask);
```

Description

__bic_SR_register clears the bits specified in **mask** in the MSP430 status register (i.e. it bitwise ands the complement of **mask** into the status register).

__bic_SR_register returns the value of the MSP430 status register before the update.

__bic_SR_register is an intrinsic function and produces inline code.

__bic_SR_register_on_exit

Synopsis

```
unsigned __bic_SR_register_on_exit(unsigned mask);
```

Description

__bic_SR_register_on_exit clears the bits specified in **mask** in the saved status register of an interrupt function (i.e. it bitwise ands the complement of **mask** into the saved status register). This allows you to change the operating mode of the MSP430 on return from the interrupt service routine, such as changing the low power mode.

__bic_SR_register_on_exit returns the value of the saved MSP430 status register before the update.

__bic_SR_register_on_exit is an intrinsic function and produces inline code.

Note

__bic_SR_register_on_exit can only be used in interrupt functions—an error is reported if it is used outside an interrupt function.

`__bis_SR_register`

Synopsis

```
unsigned __bis_SR_register(unsigned mask);
```

Description

`__bis_SR_register` sets the bits specified in **mask** in the MSP430 status register (i.e. it bitwise-ors **mask** into the status register).

`__bis_SR_register` returns the value of the MSP430 status register before the update.

`__bis_SR_register` is an intrinsic function and produces inline code.

__bis_SR_register_on_exit

Synopsis

```
unsigned __bis_SR_register_on_exit(unsigned);
```

Description

__bis_SR_register_on_exit sets the bits specified in **mask** in the saved status register of an interrupt function (i.e. it bitwise ands the complement of **mask** into the saved status register). This allows you to change the operating mode of the MSP430 on return from the interrupt service routine, such as changing the low power mode.

__bis_SR_register_on_exit returns the value of the saved MSP430 status register before the update.

__bis_SR_register_on_exit is an intrinsic function and produces inline code.

Note

__bis_SR_register_on_exit can only be used in interrupt functions—an error is reported if it is used outside an interrupt function.

__bit_count_leading_zeros_char

Synopsis

```
int __bit_count_leading_zeros_char(unsigned char x);
```

Description

__bit_count_leading_zeros_char counts the number of leading binary zero bits in **x**.

__bit_count_leading_zeros_char is an intrinsic function and produces inline code.

__bit_count_leading_zeros_long

Synopsis

```
int __bit_count_leading_zeros_long(unsigned long x);
```

Description

__bit_count_leading_zeros_long counts the number of leading binary zero bits in **x**.

__bit_count_leading_zeros_long is an intrinsic function and produces inline code.

__bit_count_leading_zeros_long_long

Synopsis

```
int __bit_count_leading_zeros_long_long(unsigned long long x);
```

Description

__bit_count_leading_zeros_long_long counts the number of leading binary zero bits in **x**.

__bit_count_leading_zeros_long_long is an intrinsic function and produces inline code.

__bit_count_leading_zeros_short

Synopsis

```
int __bit_count_leading_zeros_short(unsigned short x);
```

Description

__bit_count_leading_zeros_short counts the number of leading binary zero bits in **x**.

__bit_count_leading_zeros_short is an intrinsic function and produces inline code.

__bit_reverse_char

Synopsis

```
unsigned char __bit_reverse_char(unsigned char x);
```

Description

__bit_reverse_char swaps the order of all bits of **x** and returns that as its result.

__bit_reverse_char is an intrinsic function and produces inline code.

__bit_reverse_long

Synopsis

```
unsigned long __bit_reverse_long(unsigned long x);
```

Description

__bit_reverse_long swaps the order of all bits of **x** and returns that as its result.

__bit_reverse_long is an intrinsic function and produces inline code.

__bit_reverse_long_long

Synopsis

```
unsigned long long __bit_reverse_long_long(unsigned long long x);
```

Description

__bit_reverse_long_long swaps the order of all bits of **x** and returns that as its result.

__bit_reverse_long_long is an intrinsic function and produces inline code.

__bit_reverse_short

Synopsis

```
unsigned short __bit_reverse_short(unsigned int x);
```

Description

__bit_reverse_short swaps the order of all bits of **x** and returns that as its result.

__bit_reverse_short is an intrinsic function and produces inline code.

__delay_cycles

Synopsis

```
void __delay_cycles(unsigned long n);
```

Description

__delay_cycles delays program execution for exactly **n** processor cycles. **n** must be a compile-time constant.

__delay_cycles is an intrinsic function and produces inline code.

__disable_interrupt

Synopsis

```
void __disable_interrupt();
```

Description

__disable_interrupt disables global interrupts by clearing the **GIE** bit in the status register.

__disable_interrupt returns the value of the status register before the **GIE** bit is cleared.

__disable_interrupt is an intrinsic function and produces inline code.

__enable_interrupt

Synopsis

```
void __enable_interrupt();
```

Description

__enable_interrupt enables global interrupts by setting the **GIE** bit in the status register.

__enable_interrupt returns the value of the status register before the **GIE** bit is set.

__enable_interrupt is an intrinsic function and produces inline code.

__even_in_range

Synopsis

```
void __even_in_range(unsigned x,  
                    unsigned r);
```

Description

__even_in_range is provided for IAR compatibility. **__even_in_range** returns **x**.

__get_register

Synopsis

```
unsigned __get_register(unsigned reg);
```

Description

__get_register reads CPU register **reg** and returns its contents. **__get_register** should be used for writing small wrapper routines in C that can be hand checked for correctness.

__get_register is an intrinsic function and produces inline code.

__insert_opcode

Synopsis

```
void __insert_opcode(const unsigned op);
```

Description

__insert_opcode inserts **op** into the code stream and can be used to insert special instructions directly into function code. **op** must be a compile-time constant.

__insert_opcode is an intrinsic function and produces inline code.

__low_power_mode_0

Synopsis

```
void __low_power_mode_0();
```

Description

__low_power_mode_0 enters MSP430 low power mode 0. In this mode the CPU and MCLK are disabled; SMCLK and ACLK are active.

__low_power_mode_0 is an intrinsic function and produces inline code.

__low_power_mode_1

Synopsis

```
void __low_power_mode_1();
```

Description

__low_power_mode_1 enters MSP430 low power mode 1. In this mode the CPU, MCLK, and DCO are disabled; SMCLK and ACLK are active; the DC generator is disabled if the DCO is not used for MCLK or SMCLK in active mode.

__low_power_mode_1 is an intrinsic function and produces inline code.

__low_power_mode_2

Synopsis

```
void __low_power_mode_2();
```

Description

__low_power_mode_2 enters MSP430 low power mode 1. In this mode the CPU, MCLK, SMCLK, and DCO are disabled; ACLK is active and the DC generator is enabled.

__low_power_mode_2 is an intrinsic function and produces inline code.

__low_power_mode_3

Synopsis

```
void __low_power_mode_3();
```

Description

__low_power_mode_3 enters MSP430 low power mode 3. In this mode the CPU, MCLK, SMCLK, and DCO are disabled; ACLK is active and the DC generator is disabled.

__low_power_mode_3 is an intrinsic function and produces inline code.

__low_power_mode_4

Synopsis

```
void __low_power_mode_4();
```

Description

__low_power_mode_4 enters MSP430 low power mode 4. In this mode the CPU and all clocks are disabled.

__low_power_mode_4 is an intrinsic function and produces inline code.

__low_power_mode_off_on_exit

Synopsis

```
void __low_power_mode_off_on_exit();
```

Description

__low_power_mode_off_on_exit turns low power mode off when an interrupt returns, causing the halted CPU to resume in active mode with all clocks enabled.

Note

__low_power_mode_off_on_exit can only be used in interrupt functions—an error is reported if it is used outside an interrupt function.

__low_power_mode_off_on_exit is an intrinsic function and produces inline code.

__no_operation

Synopsis

```
void __no_operation();
```

Description

__no_operation inserts a NOP instruction into the code stream.

__no_operation is an intrinsic function and produces inline code.

__read_extended_byte

Synopsis

```
unsigned __read_extended_byte(unsigned long addr);
```

Description

__read_extended_byte reads a byte from the 20-bit extended address **addre** and returns it as its result.

Note

__read_extended_byte can only be used on MSP430X devices.

__read_extended_long

Synopsis

```
unsigned long __read_extended_long(unsigned long addr);
```

Description

__read_extended_long reads a long from the 20-bit extended address **addr** and returns it as its result.

Note

__read_extended_long can only be used on MSP430X devices.

__read_extended_word

Synopsis

```
unsigned __read_extended_word(unsigned long addr);
```

Description

__read_extended_word reads a word from the 20-bit extended address **addr** and returns it as its result.

Note

__read_extended_word can only be used on MSP430X devices.

__set_interrupt

Synopsis

```
void __set_interrupt(unsigned state);
```

Description

__set_interrupt copies the GIE flag stores held in (by a call to **__disable_interrupt**) into the status register.

__set_interrupt is an intrinsic function and produces inline code.

__set_register

Synopsis

```
void __set_register(unsigned reg,  
                   unsigned value);
```

Description

__set_register sets CPU register **reg** to **value**. **reg** must be a compile-time constant, and **value** can be any expression. Note that the compiler emits a MOVW instruction to move **value** to **reg** and does not save **reg** in the function entry nor restore it on function exit, so you must ensure that you respect the calling conventions of the compiler. Also, using **reg** does not reserve **reg** from the compiler register allocator, so you must be careful not to overwrite a local. **__set_register** should be used for writing small wrapper routines in C that can be hand checked for correctness.

__set_register returns **value**.

__set_register is an intrinsic function and produces inline code.

__swap_bytes

Synopsis

```
unsigned __swap_bytes(unsigned x);
```

Description

__swap_bytes swaps the order of high and low bytes of **x** and returns that as its result.

__swap_bytes is an intrinsic function and produces inline code.

__swap_long_bytes

Synopsis

```
unsigned long __swap_long_bytes(unsigned long x);
```

Description

__swap_long_bytes swaps the order of all bytes of **x** and returns that as its result.

__swap_long_bytes is an intrinsic function and produces inline code.

__swap_words

Synopsis

```
unsigned long __swap_words(unsigned long x);
```

Description

__swap_words swaps the order of the high and low words of **x** and returns that as its result.

__swap_words is an intrinsic function and produces inline code.

__write_extended_byte

Synopsis

```
void __write_extended_byte(unsigned long addr,  
                           unsigned value);
```

Description

__write_extended_byte writes the byte **value** to the 20-bit extended address **addr**.

Note

__write_extended_byte can only be used on MSP430X devices.

__write_extended_long

Synopsis

```
void __write_extended_long(unsigned long addr,  
                           unsigned long value);
```

Description

__write_extended_long writes the long **value** to the 20-bit extended address **addr**.

Note

__write_extended_long can only be used on MSP430X devices.

__write_extended_word

Synopsis

```
void __write_extended_word(unsigned long addr,  
                           unsigned value);
```

Description

__write_extended_word writes the word **value** to the 20-bit extended address **addr**.

Note

__write_extended_word can only be used on MSP430X devices.

<In430.h>

Binary coded decimal arithmetic	
_DADD16	Add two 16-bit values using decimal arithmetic
_DADD32	Add two 32-bit values using decimal arithmetic
_DADD64	Add two 64-bit values using decimal arithmetic
_DNEG16	Negate a 16-bit value using decimal arithmetic
_DNEG32	Negate a 32-bit value using decimal arithmetic
_DNEG64	Negate a 64-bit value using decimal arithmetic
_DSUB16	Subtract two 16-bit values using decimal arithmetic
_DSUB32	Subtract two 32-bit values using decimal arithmetic
_DSUB64	Subtract two 64-bit values using decimal arithmetic
Byte order manipulation	
_LSWPB	Swap order of bytes in a long
_LSWPW	Swap order of words in a long
_SWPB	Swap order of bytes in a word
Miscellaneous functions	
_NOP	Insert a NOP instruction
_OPC	Insert an opcode
Status register manipulation	
_BIC_SR	Clear bits in status register
_BIC_SR_IRQ	Clear bits in stacked status register
_BIS_SR	Set bits in status register
_BIS_SR_IRQ	Set bits in stacked status register
_DINT	Disable global interrupts
_EINT	Enable global interrupts

_BIC_SR

Synopsis

```
#define _BIC_SR(X) __bic_SR_register(X)
```

Description

A synonym for [__bic_SR_register](#).

_BIC_SR_IRQ

Synopsis

```
#define _BIC_SR_IRQ(X) __bic_SR_register_on_exit(X)
```

Description

A synonym for [__bic_SR_register_on_exit](#).

_BIS_SR

Synopsis

```
#define _BIS_SR(X) __bis_SR_register(X)
```

Description

A synonym for [__bis_SR_register](#).

`_BIS_SR_IRQ`

Synopsis

```
#define _BIS_SR_IRQ(X) __bis_sr_register_on_exit(X)
```

Description

A synonym for [__bis_sr_register_on_exit](#).

_DADD16

Synopsis

```
#define _DADD16(X, Y) __bcd_add_short(X, Y)
```

Description

A synonym for [__bcd_add_short](#).

_DADD32

Synopsis

```
#define _DADD32(X, Y) __bcd_add_long(X, Y)
```

Description

A synonym for [__bcd_add_long](#).

_DADD64

Synopsis

```
#define _DADD64(X, Y) __bcd_add_long_long(X, Y)
```

Description

A synonym for [__bcd_add_long_long](#).

_DINT

Synopsis

```
#define _DINT() __disable_interrupt()
```

Description

A synonym for [__disable_interrupt](#).

_DNEG16

Synopsis

```
#define _DNEG16(X) __bcd_negate_short(X)
```

Description

A synonym for [__bcd_negate_short](#).

_DNEG32

Synopsis

```
#define _DNEG32(X) __bcd_negate_long(X)
```

Description

A synonym for [__bcd_negate_long](#).

_DNEG64

Synopsis

```
#define _DNEG64(X) __bcd_negate_long_long(X)
```

Description

A synonym for [__bcd_negate_long_long](#).

_DSUB16

Synopsis

```
#define _DSUB16(X, Y) __bcd_subtract_short(X, Y)
```

Description

A synonym for [__bcd_subtract_short](#).

_DSUB32

Synopsis

```
#define _DSUB32(X, Y) __bcd_subtract_long(X, Y)
```

Description

A synonym for [__bcd_subtract_long](#).

_DSUB64

Synopsis

```
#define _DSUB64(X, Y) __bcd_subtract_long_long(X, Y)
```

Description

A synonym for [__bcd_subtract_long_long](#).

_EINT

Synopsis

```
#define _EINT() __enable_interrupt()
```

Description

A synonym for [__enable_interrupt](#).

_LSWPB

Synopsis

```
#define _LSWPB(X) __swap_long_bytes(X)
```

Description

A synonym for [__swap_long_bytes](#).

_LSWPW

Synopsis

```
#define _LSWPW(X) __swap_words(X)
```

Description

A synonym for [__swap_words](#).

_NOP

Synopsis

```
#define _NOP() __no_operation()
```

Description

A synonym for [__no_operation](#).

_OPC

Synopsis

```
#define _OPC(X) __insert_opcode(X)
```

Description

A synonym for [__insert_opcode](#).

_SWPB

Synopsis

```
#define _SWPB(X) __swap_bytes(X)
```

Description

A synonym for [__swap_bytes](#).

Standard C Library Reference

CrossWorks C provides a library that conforms to the ANSI and ISO standards for C.

In this section

[assert.h](#)

Describes the diagnostic facilities which you can build into your application.

[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.

[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.

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.

Overview

Each conversion specification is introduced by the character ' % '. After the ' % ', the following appear in sequence:

- Zero or more *flags* (in any order) that modify the meaning of the conversion specification.
- An optional *minimum field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag has been given) to the field width. The field width takes the form of an asterisk ' * ' or a decimal integer.
- An optional precision that gives the minimum number of digits to appear for the ' d ', ' i ', ' o ', ' u ', ' x ', and ' X ' conversions, the number of digits to appear after the decimal-point character for ' e ', ' E ', ' f ', and ' F ' conversions, the maximum number of significant digits for the ' g ' and ' G ' conversions, or the maximum number of bytes to be written for s conversions. The precision takes the form of a period ' . ' followed either by an asterisk ' * ' or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an int argument supplies the field width or precision. The arguments specifying field width, or precision, or both, must appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a ' - ' flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

Some CrossWorks library variants do not support width and precision specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Width/Precision Support** property of the project if you use these.

Flag characters

The flag characters and their meanings are:

' - '

The result of the conversion is left-justified within the field. The default, if this flag is not specified, is that the result of the conversion is left-justified within the field.

' + '

The result of a signed conversion *always* begins with a plus or minus sign. The default, if this flag is not specified, is that it begins with a sign only when a negative value is converted.

space

If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the space and ' + ' flags both appear, the space flag is ignored.

' # '

The result is converted to an *alternative form*. For ' o ' conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both zero, a single ' 0 ' is printed). For ' x ' or ' X ' conversion, a nonzero result has ' 0x ' or ' 0X ' prefixed to it. For ' e ', ' E ', ' f ', ' F ', ' g ', and ' G ' conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For ' g ' and ' F ' conversions, trailing zeros are not removed from the result. As an extension, when used in ' p ' conversion, the results has ' # ' prefixed to it. For other conversions, the behavior is undefined.

' 0 '

For ' d ', ' i ', ' o ', ' u ', ' x ', ' X ', ' e ', ' E ', ' f ', ' F ', ' g ', and ' G ' conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the ' 0 ' and ' - ' flags both appear, the ' 0 ' flag is ignored. For ' d ', ' i ', ' o ', ' u ', ' x ', and ' X ' conversions, if a precision is specified, the ' 0 ' flag is ignored. For other conversions, the behavior is undefined.

Length modifiers

The length modifiers and their meanings are:

' hh '

Specifies that a following ' d ', ' i ', ' o ', ' u ', ' x ', or ' X ' conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value will be converted to **signed char** or **unsigned char** before printing); or that a following ' n ' conversion specifier applies to a pointer to a **signed char** argument.

' h '

Specifies that a following ' d ', ' i ', ' o ', ' u ', ' x ', or ' X ' conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integer promotions, but its value is converted to **short int** or **unsigned short int** before printing); or that a following ' n ' conversion specifier applies to a pointer to a **short int** argument.

' l '

Specifies that a following ' d ', ' i ', ' o ', ' u ', ' x ', or ' X ' conversion specifier applies to a **long int** or **unsigned long int** argument; that a following ' n ' conversion specifier applies to a pointer to a **long int** argument; or has no effect on a following ' e ', ' E ', ' f ', ' F ', ' g ', or ' G ' conversion specifier. Some CrossWorks library variants do not support the ' l ' length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

'll'

Specifies that a following '**d**', '**i**', '**o**', '**u**', '**x**', or '**X**' conversion specifier applies to a **long long int** or **unsigned long long int** argument; that a following '**n**' conversion specifier applies to a pointer to a **long long int** argument. Some CrossWorks library variants do not support the '**ll**' length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined. Note that the C99 length modifiers '**j**', '**z**', '**t**', and '**L**' are not supported.

Conversion specifiers

The conversion specifiers and their meanings are:

'd','i'

The argument is converted to signed decimal in the style `[-]dddd`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading spaces. The default precision is one. The result of converting a zero value with a precision of zero is no characters.

'o','u','x','X'

The unsigned argument is converted to unsigned octal for '**o**', unsigned decimal for '**u**', or unsigned hexadecimal notation for '**x**' or '**X**' in the style `dddd`; the letters '**abcdef**' are used for '**x**' conversion and the letters '**ABCDEF**' for '**X**' conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading spaces. The default precision is one. The result of converting a zero value with a precision of zero is no characters.

'f','F'

A double argument representing a floating-point number is converted to decimal notation in the style `[-]ddd.ddd`, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the '**#**' flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. A double argument representing an infinity is converted to '**inf**'. A double argument representing a NaN is converted to '**nan**'. The '**F**' conversion specifier produces '**INF**' or '**NAN**' instead of '**inf**' or '**nan**', respectively. Some CrossWorks library variants do not support the '**f**' and '**F**' conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Floating Point Support** property of the project if you use these conversion specifiers.

'e','E'

A double argument representing a floating-point number is converted in the style `[-]d.ddd e ± dd`, where there is one digit (which is nonzero if the

argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The `'E'` conversion specifier produces a number with `'E'` instead of `'e'` introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero. A double argument representing an infinity is converted to `'inf'`. A double argument representing a NaN is converted to `'nan'`. The `'E'` conversion specifier produces `'INF'` or `'NAN'` instead of `'inf'` or `'nan'`, respectively. Some CrossWorks library variants do not support the `'f'` and `'F'` conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Floating Point Support** property of the project if you use these conversion specifiers.

`'g', 'G'`

A double argument representing a floating-point number is converted in style `'f'` or `'e'` (or in style `'F'` or `'E'` in the case of a `'G'` conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as one. The style used depends on the value converted; style `'e'` (or `'E'`) is used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the `#` flag is specified; a decimal-point character appears only if it is followed by a digit. A double argument representing an infinity is converted to `'inf'`. A double argument representing a NaN is converted to `'nan'`. The `'G'` conversion specifier produces `'INF'` or `'NAN'` instead of `'inf'` or `'nan'`, respectively. Some CrossWorks library variants do not support the `'f'` and `'F'` conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Floating Point Support** property of the project if you use these conversion specifiers.

`'c'`

The argument is converted to an **unsigned char**, and the resulting character is written.

`'s'`

The argument is be a pointer to the initial element of an array of character type. Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null character.

`'p'`

The argument is a pointer to **void**. The value of the pointer is converted in the same format as the `'x'` conversion specifier with a fixed precision of `2*sizeof(void *)`.

`'n'`

The argument is a pointer to 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'**.

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.

Overview

Each conversion specification is introduced by the character ' % '. After the ' % ', the following appear in sequence:

- An optional assignment-suppressing character ' * '.
- An optional nonzero decimal integer that specifies the maximum field width (in characters).
- An optional length modifier that specifies the size of the receiving object.
- A conversion specifier character that specifies the type of conversion to be applied.

The formatted input function executes each directive of the format in turn. If a directive fails, the function returns. Failures are described as input failures (because of the occurrence of an encoding error or the unavailability of input characters), or matching failures (because of inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

- Input white-space characters (as specified by the `isspace` function) are skipped, unless the specification includes a ' [' , ' c ', or ' n ' specifier.
- An input item is read from the stream, unless the specification includes an n specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.
- Except in the case of a ' % ' specifier, the input item (or, in the case of a %n directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a ' * ', the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received

a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

Length modifiers

The length modifiers and their meanings are:

'hh'

Specifies that a following 'd', 'i', 'o', 'u', 'x', 'X', or 'n' conversion specifier applies to an argument with type pointer to **signed char** or pointer to **unsigned char**.

'h'

Specifies that a following 'd', 'i', 'o', 'u', 'x', 'X', or 'n' conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.

'l'

Specifies that a following 'd', 'i', 'o', 'u', 'x', 'X', or 'n' conversion specifier applies to an argument with type pointer to **long int** or **unsigned long int**; that a following 'e', 'E', 'f', 'F', 'g', or 'G' conversion specifier applies to an argument with type pointer to **double**. Some CrossWorks library variants do not support the 'l' length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

'll'

Specifies that a following 'd', 'i', 'o', 'u', 'x', 'X', or 'n' conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**. Some CrossWorks library variants do not support the 'll' length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined. Note that the C99 length modifiers 'j', 'z', 't', and 'L' are not supported.

Conversion specifiers

'd'

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** 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 **strtoul** function with the value zero for the **base** argument. The corresponding argument must be a pointer to signed integer.

'o'

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 18 for the **base** argument. The corresponding argument must be a pointer to signed integer.

'u'

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument must be a pointer to unsigned integer.

'x'

Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 16 for the **base** argument. The corresponding argument must be a pointer to unsigned integer.

'e','f','g'

Matches an optionally signed floating-point number whose format is the same as expected for the subject sequence of the **strtod** function. The corresponding argument shall be a pointer to floating. Some CrossWorks library variants do not support the **'e'**, **'f'** and **'F'** conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Scanf Floating Point Support** property of the project if you use these conversion specifiers.

'c'

Matches a sequence of characters of exactly the number specified by the field width (one if no field width is present in the directive). The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

's'

Matches a sequence of non-white-space characters. The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

'['

Matches a nonempty sequence of characters from a set of expected characters (the *scanset*). The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the format string, up to and including the matching right bracket **']'**. The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex **'^'**, in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with **'[]'** or **'[^']'**, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the

specification. If a ' - ' character is in the scanlist and is not the first, nor the second where the first character is a '^', nor the last character, it is treated as a member of the scanset. Some CrossWorks library variants do not support the '[' conversion specifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Scanf Classes Supported** property of the project if you use this conversion specifier.

'p'

Reads a sequence output by the corresponding '%p' formatted output conversion. The corresponding argument must be a pointer to a pointer to **void**.

'n'

No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the formatted input function. Execution of a '%n' directive does not increment the assignment count returned at the completion of execution of the fscanf function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

'%'

Matches a single '%' character; no conversion or assignment occurs.

Note that the C99 width modifier 'l' used in conjunction with the 'c', 's', and '[' conversion specifiers is not supported and nor are the conversion specifiers 'a' and 'A'.

String handling

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.

<assert.h>

Macros

<code>assert</code>	Allows you to place assertions and diagnostic tests into programs

Functions

<code>__assert</code>	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.

<ctype.h>

Classification functions	
isalnum	Is character alphanumeric?
isalpha	Is character alphabetic?
isblank	Is character a space or horizontal tab?
iscntrl	Is character a control character?
isdigit	Is character a decimal digit?
isgraph	Is character any printing character except space?
islower	Is character a lowercase letter?
isprint	Is character printable?
ispunct	Is character a punctuation mark?
isspace	Is character a whitespace character?
isupper	Is character an uppercase letter?
isxdigit	Is character a hexadecimal digit?
Conversion functions	
tolower	Convert uppercase character to lowercase
toupper	Convert lowercase character to uppercase

isalnum

Synopsis

```
int isalnum(int c);
```

Description

isalnum returns nonzero (true) if and only if the value of the argument **c** is an uppercase letter A through Z.

isalpha

Synopsis

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

Description

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

isblank

Synopsis

```
int isblank(int c);
```

Description

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

isctrl

Synopsis

```
int isctrl(int c);
```

Description

isctrl returns nonzero (true) if and only if the value of the argument *c* is a control character. Control characters have values 0 through 31 and the single value 127.

isdigit

Synopsis

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

Description

isdigit returns nonzero (true) if and only if the value of the argument *c* is a decimal digit 0 through 9.

isgraph

Synopsis

```
int isgraph(int c);
```

Description

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

islower

Synopsis

```
int islower(int c);
```

Description

islower returns nonzero (true) if and only if the value of the argument **c** is an lowercase letter a through z.

isprint

Synopsis

```
int isprint(int c);
```

Description

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

ispunct

Synopsis

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

Description

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

isspace

Synopsis

```
int isspace(int c);
```

Description

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

The standard white-space characters are space (' '), form feed ('\f'), new-line ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v').

isupper

Synopsis

```
int isupper(int c);
```

Description

isupper returns nonzero (true) if and only if the value of the argument *c* is an uppercase letter A through Z.

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 0 through 9, a through f, or A through F.

tolower

Synopsis

```
int tolower(int c);
```

Description

tolower converts an uppercase letter to a corresponding lowercase letter. If the argument **c** is a character for which **isupper** is true, **tolower** returns the corresponding lowercase letter; otherwise, the argument is returned unchanged.

toupper

Synopsis

```
int toupper(int c);
```

Description

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

<errno.h>

Macros

<code>errno</code>	Allows you to access the errno implementation

Error numbers

<code>EDOM</code>	Domain error

<code>EILSEQ</code>	Illegal byte sequence

<code>ERANGE</code>	Result too large or too small

Functions

<code>__errno</code>	User-defined behavior for the errno macro

EDOM

Synopsis

```
#define EDOM      0x01
```

Description

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

EILSEQ

Synopsis

```
#define EILSEQ    0x02
```

Description

EILSEQ - A wide-character code has been detected that does not correspond to a valid character, or a byte sequence does not form a valid wide-character code.

ERANGE

Synopsis

```
#define ERANGE    0x03
```

Description

ERANGE - the result of the function is too large (overflow) or too small (underflow) to be represented in the available space.

__errno

Synopsis

```
int *__errno();
```

Description

There is no default implementation of **__errno**. Keeping **__errno** out of the library means that you can customize its behavior without rebuilding the library. A default implementation could be

```
static int errno;  
int *__errno(void) { return &errno; }
```

errno

Synopsis

```
#define errno (*__errno())
```

Description

errno macro expands to a function call to **__errno** that returns a pointer to an **int**. This function can be implemented by the application to provide a thread specific **errno**.

The value of **errno** is zero at program startup, but is never set to zero by any library function. The value of **errno** may be set to a nonzero value by a library function, and this effect is documented in each function that does so.

Note

The ISO standard does not specify whether **errno** is a macro or an identifier declared with external linkage. Portable programs must not make assumptions about the implementation of **errno**.

<float.h>

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.

<limits.h>

Long integer minimum and maximum values	
LONG_MAX	Maximum value of a long integer
LONG_MIN	Minimum value of a long integer
ULONG_MAX	Maximum value of an unsigned long integer
Character minimum and maximum values	
CHAR_MAX	Maximum value of a plain character
CHAR_MIN	Minimum value of a plain character
SCHAR_MAX	Maximum value of a signed character
SCHAR_MIN	Minimum value of a signed character
UCHAR_MAX	Maximum value of an unsigned char
Long long integer minimum and maximum values	
LLONG_MAX	Maximum value of a long long integer
LLONG_MIN	Minimum value of a long long integer
ULLONG_MAX	Maximum value of an unsigned long long integer
Short integer minimum and maximum values	
SHRT_MAX	Maximum value of a short integer
SHRT_MIN	Minimum value of a short integer
USHRT_MAX	Maximum value of an unsigned short integer
Integer minimum and maximum values	
INT_MAX	Maximum value of an integer
INT_MIN	Minimum value of an integer
UINT_MAX	Maximum value of an unsigned integer
Type sizes	
CHAR_BIT	Number of bits in a character

CHAR_BIT

Synopsis

```
#define CHAR_BIT    8
```

Description

CHAR_BIT is the number of bits for smallest object that is not a bit-field (byte).

CHAR_MAX

Synopsis

```
#define CHAR_MAX    255
```

Description

CHAR_MAX is the maximum value for an object of type **char**.

CHAR_MIN

Synopsis

```
#define CHAR_MIN    0
```

Description

CHAR_MIN is the minimum value for an object of type **char**.

INT_MAX

Synopsis

```
#define INT_MAX    2147483647
```

Description

INT_MAX is the maximum value for an object of type **int**.

INT_MIN

Synopsis

```
#define INT_MIN    (-2147483647 - 1)
```

Description

INT_MIN is the minimum value for an object of type **int**.

LLONG_MAX

Synopsis

```
#define LLONG_MAX 9223372036854775807LL
```

Description

LLONG_MAX is the maximum value for an object of type **long long int**.

LLONG_MIN

Synopsis

```
#define LLONG_MIN  (-9223372036854775807LL - 1)
```

Description

LLONG_MIN is the minimum value for an object of type **long long int**.

LONG_MAX

Synopsis

```
#define LONG_MAX    2147483647L
```

Description

LONG_MAX is the maximum value for an object of type **long int**.

LONG_MIN

Synopsis

```
#define LONG_MIN    (-2147483647L - 1)
```

Description

LONG_MIN is the minimum value for an object of type **long int**.

SCHAR_MAX

Synopsis

```
#define SCHAR_MAX 127
```

Description

SCHAR_MAX is the maximum value for an object of type **signed char**.

SCHAR_MIN

Synopsis

```
#define SCHAR_MIN  (-128)
```

Description

SCHAR_MIN is the minimum value for an object of type **signed char**.

SHRT_MAX

Synopsis

```
#define SHRT_MAX    32767
```

Description

SHRT_MAX is the minimum value for an object of type **short int**.

SHRT_MIN

Synopsis

```
#define SHRT_MIN    (-32767 - 1)
```

Description

SHRT_MIN is the minimum value for an object of type **short int**.

UCHAR_MAX

Synopsis

```
#define UCHAR_MAX 255
```

Description

UCHAR_MAX is the maximum value for an object of type **unsigned char**.

UINT_MAX

Synopsis

```
#define UINT_MAX    4294967295U
```

Description

UINT_MAX is the maximum value for an object of type **unsigned int**.

ULLONG_MAX

Synopsis

```
#define ULLONG_MAX 18446744073709551615ULL
```

Description

ULLONG_MAX is the maximum value for an object of type **unsigned long long int**.

ULONG_MAX

Synopsis

```
#define ULONG_MAX 4294967295UL
```

Description

ULONG_MAX is the maximum value for an object of type **unsigned long int**.

USHRT_MAX

Synopsis

```
#define USHRT_MAX 65535
```

Description

USHRT_MAX is the minimum value for an object of type **unsigned short int**.

<locale.h>

Functions

localeconv	Get current locale data

setlocale	Set Locale

Structures

lconv	Formatting info for numeric values

lconv

Synopsis

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

Description

lconv structure holds formatting information on how numeric values are to be written.

Member	Description
current_symbol	Local currency symbol.
decimal_point	Decimal point separator.
frac_digits	Amount of fractional digits to the right of the decimal point for monetary quantities in the local format.
grouping	Specifies the amount of digits that form each of the groups to be separated by thousands_sep separator for non-monetary quantities.
int_curr_symbol	International currency symbol.
int_frac_digits	Amount of fractional digits to the right of the decimal point for monetary quantities in the international format.
mon_decimal_point	Decimal-point separator used for monetary quantities.
mon_grouping	Specifies the amount of digits that form each of the groups to be separated by mon_thousands_sep separator for monetary quantities.
mon_thousands_sep	Separators used to delimit groups of digits to the left of the decimal point for monetary quantities.
negative_sign	Sign to be used for negative monetary quantities.

n_cs_precedes	Whether the currency symbol should precede negative monetary quantities.
n_sep_by_space	Whether a space should appear between the currency symbol and negative monetary quantities.
n_sign_posn	Position of the sign for negative monetary quantities.
positive_sign	Sign to be used for nonnegative (positive or zero) monetary quantities.
p_cs_precedes	Whether the currency symbol should precede nonnegative (positive or zero) monetary quantities.
p_sep_by_space	Whether a space should appear between the currency symbol and nonnegative (positive or zero) monetary quantities.
p_sign_posn	Position of the sign for nonnegative (positive or zero) monetary quantities.
thousands_sep	Separators used to delimit groups of digits to the left of the decimal point for non-monetary quantities.

localeconv

Synopsis

```
localeconv();
```

Description

localeconv returns a pointer to a structure of type **lconv** with the corresponding values for the current locale filled in.

setlocale

Synopsis

```
char *setlocale(int category,  
               const char *locale);
```

Description

setlocale sets the current locale. The **category** parameter can have the following values

Name	Locale affected
LC_ALL	Entire locale
LC_COLLATE	Affects strcoll and strxfrm
LC_CTYPE	Affects character handling
LC_MONETARY	Affects monetary formatting information
LC_NUMERIC	Affects decimal-point character in I/O and string formatting operations
LC_TIME	Affects strftime

The **locale** parameter contains the name of a C locale to set or if **NULL** is passed the current locale is not changed.

setlocale returns the name of the current locale.

Note

CrossWorks only supports the minimal "C" locale.

<math.h>

Type Generic Macros	
fpclassify	Classify floating type
isfinite	Test for a finite value
isinf	Test for infinity
isnan	Test for NaN
isnormal	Test for a normal value
signbit	Test sign
Trigonometric functions	
cos	Compute cosine of a double
cosf	Compute cosine of a float
sin	Compute sine of a double
sinf	Compute sine of a float
tan	Compute tangent of a double
tanf	Compute tangent of a double
Inverse trigonometric functions	
acos	Compute inverse cosine of a double
acosf	Compute inverse cosine of a float
asin	Compute inverse sine of a double
asinf	Compute inverse sine of a float
atan	Compute inverse tangent of a double
atan2	Compute inverse tangent of a ratio of doubles
atan2f	Compute inverse tangent of a ratio of floats
atanf	Compute inverse tangent of a float
Exponential and logarithmic functions	
cbrt	Compute cube root of a double
cbrtf	Compute cube root of a float
exp	Compute exponential of a double
expf	Compute exponential of a float
frexp	Set exponent of a double
frexpf	Set exponent of a float
ldexp	Adjust exponent of a double
ldexpf	Adjust exponent of a float
log	Compute natural logarithm of a double

log10	Compute common logarithm of a double
log10f	Compute common logarithm of a float
logf	Compute natural logarithm of a float
pow	Raise a double to a power
powf	Raise a float to a power
scalbn	Scale a double
scalbnf	Scale a float
sqrt	Compute square root of a double
sqrtf	Compute square root of a float
Remainder functions	
fmod	Compute remainder after division of two doubles
fmodf	Compute remainder after division of two floats
modf	Break a double into integer and fractional parts
modff	Break a float into integer and fractional parts
Nearest integer functions	
ceil	Compute smallest integer not greater than a double
ceilf	Compute smallest integer not greater than a float
floor	Compute largest integer not greater than a float
floorf	Compute largest integer not greater than a float
Absolute value functions	
fabs	Compute absolute value of a double
fabsf	Compute absolute value of a float
hypot	Compute complex magnitude of two doubles
hypotf	Compute complex magnitude of two floats
Maximum, minimum, and positive difference functions	
fmax	Compute maximum of two doubles
fmaxf	Compute maximum of two floats
fmin	Compute minimum of two doubles
fminf	Compute minimum of two floats
Hyperbolic functions	
cosh	Compute hyperbolic cosine of a double
coshf	Compute hyperbolic cosine of a float
sinh	Compute hyperbolic sine of a double
sinhf	Compute hyperbolic sine of a float
tanh	Compute hyperbolic tangent of a double

tanhf	Compute hyperbolic tangent of a float
Inverse hyperbolic functions	
acosh	Compute inverse hyperbolic cosine of a double
acoshf	Compute inverse hyperbolic cosine of a float
asinh	Compute inverse hyperbolic sine of a double
asinhf	Compute inverse hyperbolic sine of a float
atanh	Compute inverse hyperbolic tangent of a double
atanhf	Compute inverse hyperbolic tangent of a float

acos

Synopsis

```
double acos(double x);
```

Description

acos returns the principal value, in radians, of the inverse circular cosine of **x**. The principal value lies in the interval $[0, \pi]$ radians.

If $|x| > 1$, **errno** is set to **EDOM** and **acos** returns **HUGE_VAL**.

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

acosf

Synopsis

```
float acosf(float x);
```

Description

acosf returns the principal value, in radians, of the inverse circular cosine of **x**. The principal value lies in the interval $[0, \pi]$ radians.

If $|a| > 1$, **errno** is set to **EDOM** and **acosf** returns **HUGE_VAL**.

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

acosh

Synopsis

```
double acosh(double x);
```

Description

acosh returns the non-negative inverse hyperbolic cosine of **x**.

acosh(**x**) is defined as $\log(x + \sqrt{x^2 - 1})$, assuming completely accurate computation.

If **x** < 1, **errno** is set to **EDOM** and **acosh** returns **HUGE_VAL**.

If **x** < 1, **acosh** returns NaN.

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

acoshf

Synopsis

```
float acoshf(float x);
```

Description

acoshf returns the non-negative inverse hyperbolic cosine of **x**.

acosh(x) is defined as $\log(x + \sqrt{x^2 - 1})$, assuming completely accurate computation.

If $x < 1$, **errno** is set to **EDOM** and **acoshf** returns **HUGE_VALF**.

If $x < 1$, **acoshf** returns NaN.

If **x** is NaN, **acoshf** returns that NaN.

asin

Synopsis

```
double asin(double x);
```

Description

asin returns the principal value, in radians, of the inverse circular sine of **x**. The principal value lies in the interval $[-\frac{1}{2}\pi, +\frac{1}{2}\pi]$ 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 $[-\frac{1}{2}\pi, +\frac{1}{2}\pi]$ 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 $+\infty$, $-\infty$, or NaN, **asinh** returns $|x|$. If $|x| > \sim 709.782$, **asinh** returns $+\infty$ or $-\infty$ 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 $+\infty$, $-\infty$, or NaN, **asinhf** returns $|x|$. If $|x| > \sim 88.7228$, **asinhf** returns $+\infty$ or $-\infty$ 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 $[-\frac{1}{2}\pi, +\frac{1}{2}\pi]$ radians.

atan2

Synopsis

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

Description

atan2 returns the value, in radians, of the inverse circular tangent of **y** divided by **x** using the signs of **x** and **y** to compute the quadrant of the return value. The principal value lies in the interval $[-\frac{1}{2}\pi/2, +\frac{1}{2}\pi]$ 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 $\pm \pi$.

atan2($\pm$ (anything but 0 and NaN), 0) is $\pm \frac{1}{2}\pi$.

atan2($\pm$ (anything but ∞ and NaN), $+\infty$) is ± 0 .

atan2($\pm$ (anything but ∞ and NaN), $-\infty$) is $\pm \pi$.

atan2($\pm \infty$, $+\infty$) is $\pm \frac{1}{4}\pi$.

atan2($\pm \infty$, $-\infty$) is $\pm \frac{3}{4}\pi$.

atan2($\pm \infty$, (anything but 0, NaN, and ∞)) is $\pm \frac{1}{2}\pi$.

atan2f

Synopsis

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

Description

atan2f returns the value, in radians, of the inverse circular tangent of **y** divided by **x** using the signs of **x** and **y** to compute the quadrant of the return value. The principal value lies in the interval $[-\frac{1}{2}\pi/2, +\frac{1}{2}\pi]$ 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 $\pm \pi$.

atan2f($\pm$ (anything but 0 and NaN), 0) is $\pm \frac{1}{2}\pi$.

atan2f($\pm$ (anything but ∞ and NaN), $+\infty$) is ± 0 .

atan2f($\pm$ (anything but ∞ and NaN), $-\infty$) is $\pm \pi$.

atan2f($\pm\infty$, $+\infty$) is $\pm \frac{1}{4}\pi$.

atan2f($\pm\infty$, $-\infty$) is $\pm \frac{3}{4}\pi$.

atan2f($\pm\infty$, (anything but 0, NaN, and ∞)) is $\pm \frac{1}{2}\pi$.

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 $[-\frac{1}{2}\pi, +\frac{1}{2}\pi]$ 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 $-\infty$.

atanhf

Synopsis

```
float atanhf(float x);
```

Description

atanhf returns the inverse hyperbolic tangent of **val**.

If $|x| \geq 1$, **errno** is set to **EDOM** and **atanhf** returns **HUGE_VALF**.

If $|val| > 1$ **atanhf** returns NaN. If **val** is NaN, **atanhf** returns that NaN. If **val** is 1, **atanhf** returns ∞ . If **val** is -1 , **atanhf** returns $-\infty$.

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** ($\pm \infty$) is $\pm \infty$.

ceilf

Synopsis

```
float ceilf(float x);
```

Description

ceilf computes the smallest integer value not less than **x**.

ceilf (± 0) is ± 0 . **ceilf** ($\pm \infty$) is $\pm \infty$.

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 $+\infty$, $-\infty$, or NaN, **cosh** returns $|x|$.> If $|x| > \sim 709.782$, **cosh** returns $+\infty$ or $-\infty$ 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 $+\infty$, $-\infty$, or NaN, **coshf** returns $|x|$.

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

If **x** is $-\infty$, **exp** returns 0.

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 ∞ , **expf** returns ∞ .

If **x** is $-\infty$, **expf** returns 0.

fabs

Synopsis

```
double fabs(double x);
```

fabsf

Synopsis

```
float fabsf(float x);
```

Description

fabsf computes the absolute value of the floating-point number **x**.

floor

Synopsis

```
double floor(double);
```

floor computes the largest integer value not greater than **x**.

floor (± 0) is ± 0 . **floor** ($\pm \infty$) is $\pm \infty$.

floorf

Synopsis

```
float floorf(float);
```

floorf computes the largest integer value not greater than **x**.

floorf(± 0) is ± 0 . **floorf**($\pm \infty$) is $\pm \infty$.

fmax

Synopsis

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

Description

fmax determines the maximum of **x** and **y**.

fmax (NaN, **y**) is **y**. **fmax** (**x**, NaN) is **x**.

fmaxf

Synopsis

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

Description

fmaxf determines the maximum of **x** and **y**.

fmaxf (NaN, **y**) is **y**. **fmaxf**(**x**, NaN) is **x**.

fmin

Synopsis

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

Description

fmin determines the minimum of **x** and **y**.

fmin (NaN, **y**) is **y**. **fmin** (**x**, NaN) is **x**.

fminf

Synopsis

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

Description

fminf determines the minimum of **x** and **y**.

fminf (NaN, **y**) is **y**. **fminf** (**x**, NaN) is **x**.

fmod

Synopsis

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

Description

fmod computes the floating-point remainder of **x** divided by **y**. #b #this returns the value $x - n y$, for some integer n such that, if **y** is nonzero, the result has the same sign as **x** and magnitude less than the magnitude of **y**.

If **y** = 0, **fmod** returns zero and **errno** is set to **EDOM**.

fmod (± 0 , **y**) is ± 0 for **y** not zero.

fmod (∞ , **y**) is NaN.

fmod (**x**, 0) is NaN.

fmod (**x**, $\pm \infty$) is **x** for **x** not infinite.

fmodf

Synopsis

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

Description

fmodf computes the floating-point remainder of **x** divided by **y**. **fmodf** returns the value $x - n y$, for some integer n such that, if **y** is nonzero, the result has the same sign as **x** and magnitude less than the magnitude of **y**.

If **y** = 0, **fmodf** returns zero and **errno** is set to **EDOM**.

fmodf (± 0 , **y**) is ± 0 for **y** not zero.

fmodf (∞ , **y**) is NaN.

fmodf (**x**, 0) is NaN.

fmodf (**x**, $\pm \infty$) is **x** for **x** not infinite.

fpclassify

Synopsis

```
#define fpclassify(x) (sizeof(x) == sizeof(float) ? __float32_classify(x) :  
    __float64_classify(x))
```

Description

fpclassify macro shall classify its argument value as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. The fpclassify macro returns the value of the number classification macro one of

- FP_ZERO
- FP_SUBNORMAL
- FP_NORMAL
- FP_INFINITE
- FP_NAN

frexp

Synopsis

```
double frexp(double x,  
             int *exp);
```

Description

frexp breaks a floating-point number into a normalized fraction and an integral power of 2.

frexp stores power of two in the **int** object pointed to by **exp** and returns the value **x**, such that **x** has a magnitude in the interval $[1/2, 1)$ or zero, and value equals $x * 2^{\mathbf{exp}}$.

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 $[\frac{1}{2}, 1)$ 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, **frexpf** returns **x** and stores zero into the int object pointed to by **exp**.

hypot

Synopsis

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

Description

hypot computes the square root of the sum of the squares of **x** and **y**, **sqrt(x*x + y*y)**, without undue overflow or underflow. If **x** and **y** are the lengths of the sides of a right-angled triangle, then **hypot** computes the length of the hypotenuse.

If **x** or **y** is $+\infty$ or $-\infty$, **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 $+\infty$ or $-\infty$, **hypotf** returns ∞ . If **x** or **y** is NaN, **hypotf** returns NaN.

isfinite

Synopsis

```
#define isfinite(x) (sizeof(x) == sizeof(float) ? __float32_isfinite(x) :  
    __float64_isfinite(x))
```

Description

isfinite determines whether **x** is a finite value (zero, subnormal, or normal, and not infinite or NaN). The isfinite macro returns a non-zero value if and only if its argument has a finite value.

isinf

Synopsis

```
#define isinf(x) (sizeof(x) == sizeof(float) ? __float32_isinf(x) : __float64_isinf(x))
```

Description

isinf determines whether **x** is an infinity (positive or negative). The determination is based on the type of the argument.

isnan

Synopsis

```
#define isnan(x) (sizeof(x) == sizeof(float) ? __float32_isnan(x) : __float64_isnan(x))
```

Description

isnan determines whether **x** is a NaN. The determination is based on the type of the argument.

isnormal

Synopsis

```
#define isnormal(x) (sizeof(x) == sizeof(float) ? __float32_isnormal(x) :  
    __float64_isnormal(x))
```

Description

isnormal determines whether **x** is a normal value (zero, subnormal, or normal, and not infinite or NaN). The isnormal macro returns a non-zero value if and only if its argument has a normal value.

ldexp

Synopsis

```
double ldexp(double x,  
             int exp);
```

Description

ldexp multiplies a floating-point number by an integral power of 2.

ldexp returns $x * 2^{\text{exp}}$.

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 ∞ or NaN, **ldexpf** returns x . If the result overflows, **ldexpf** returns ∞ .

log

Synopsis

```
double log(double x);
```

Description

log computes the base-*e* logarithm of **x**.

If **x** = 0, **errno** is set to **ERANGE** and **log** returns **–HUGE_VAL**. If **x** < 0, **errno** is set to **EDOM** and **log** returns **–HUGE_VAL**.

If **x** < 0 or **x** = $-\infty$, **log** returns NaN.

If **x** = 0, **log** returns $-\infty$.

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** = $-\infty$, **log10** returns NaN.

If **x** = 0, **log10** returns $-\infty$.

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** = $-\infty$, **log10f** returns NaN.

If **x** = 0, **log10f** returns $-\infty$.

If **x** = ∞ , **log10f** returns ∞ .

If **x** = NaN, **log10f** returns **x**.

logf

Synopsis

```
float logf(float x);
```

Description

logf computes the base-e logarithm of **x**.

If **x** = 0, **errno** is set to **ERANGE** and **logf** returns **-HUGE_VALF**. If **x** < 0, **errno** is set to **EDOM** and **logf** returns **-HUGE_VALF**.

If **x** < 0 or **x** = $-\infty$, **logf** returns NaN.

If **x** = 0, **logf** returns $-\infty$.

If **x** = ∞ , **logf** returns ∞ .

If **x** = NaN, **logf** returns **x**.

modf

Synopsis

```
double modf(double x,  
            double *iptr);
```

Description

modf breaks **x** into integral and fractional parts, each of which has the same type and sign as **x**.

The integral part (in floating-point format) is stored in the object pointed to by **iptr** and **modf** returns the signed fractional part of **x**.

modff

Synopsis

```
float modff(float x,  
            float *iptr);
```

Description

modff breaks **x** into integral and fractional parts, each of which has the same type and sign as **x**.

The integral part (in floating-point format) is stored in the object pointed to by **iptr** and **modff** returns the signed fractional part of **x**.

pow

Synopsis

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

Description

pow computes x raised to the power y .

If $x < 0$ and $y \leq 0$, **errno** is set to **EDOM** and **pow** returns **-HUGE_VAL**. If $x \leq 0$ 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 $y = +\infty$, **pow** returns $+\infty$.

If $x < -1$ or $1 < x$, and $y = -\infty$, **pow** returns 0.

If $-1 < x < 1$ and $y = +\infty$, **pow** returns $+0$.

If $-1 < x < 1$ and $y = -\infty$, **pow** returns $+\infty$.

If $x = +1$ or $x = -1$ and $y = +\infty$ or $y = -\infty$, **pow** returns NaN.

If $x = +0$ and $y > 0$ and $y \neq \text{NaN}$, **pow** returns $+0$.

If $x = -0$ and $y > 0$ and $y \neq \text{NaN}$ or y not an odd integer, **pow** returns $+0$.

If $x = +0$ and y and $y \neq \text{NaN}$, **pow** returns $+\infty$.

If $x = -0$ and $y > 0$ and $y \neq \text{NaN}$ or y not an odd integer, **pow** returns $+\infty$.

If $x = -0$ and y is an odd integer, **pow** returns -0 .

If $x = +\infty$ and $y > 0$ and $y \neq \text{NaN}$, **pow** returns $+\infty$.

If $x = +\infty$ and $y < 0$ and $y \neq \text{NaN}$, **pow** returns $+0$.

If $x = -\infty$, **pow** returns **pow**(-0 , y)

If $x < 0$ and $x \neq \infty$ and y is a non-integer, **pow** returns NaN.

powf

Synopsis

```
float powf(float,  
           float);
```

Description

powf computes x raised to the power y .

If $x < 0$ and $y \leq 0$, **errno** is set to **EDOM** and **powf** returns **–HUGE_VALF**. If $x \leq 0$ 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 $y = +\infty$, **powf** returns $+\infty$.

If $x < -1$ or $1 < x$, and $y = -\infty$, **powf** returns 0.

If $-1 < x < 1$ and $y = +\infty$, **powf** returns +0.

If $-1 < x < 1$ and $y = -\infty$, **powf** returns $+\infty$.

If $x = +1$ or $x = -1$ and $y = +\infty$ or $y = -\infty$, **powf** returns NaN.

If $x = +0$ and $y > 0$ and $y \neq \text{NaN}$, **powf** returns +0.

If $x = -0$ and $y > 0$ and $y \neq \text{NaN}$ or y not an odd integer, **powf** returns +0.

If $x = +0$ and y and $y \neq \text{NaN}$, **powf** returns $+\infty$.

If $x = -0$ and $y > 0$ and $y \neq \text{NaN}$ or y not an odd integer, **powf** returns $+\infty$.

If $x = -0$ and y is an odd integer, **powf** returns -0 .

If $x = +\infty$ and $y > 0$ and $y \neq \text{NaN}$, **powf** returns $+\infty$.

If $x = +\infty$ and $y < 0$ and $y \neq \text{NaN}$, **powf** returns +0.

If $x = -\infty$, **powf** returns **powf**(-0 , y)

If $x < 0$ and $x \neq \infty$ and y is a non-integer, **powf** returns NaN.

scalbn

Synopsis

```
double scalbn(double x,  
              int exp);
```

Description

scalbn multiplies a floating-point number by an integral power of **DBL_RADIX**.

As floating-point arithmetic conforms to IEC 60559, **DBL_RADIX** is 2 and **scalbn** is (in this implementation) identical to **ldexp**.

scalbn returns $x * \text{DBL\_RADIX}^{\text{exp}}$.

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 shall determine whether the sign of its argument value is negative. The signbit macro returns a non-zero value if and only if its argument value is negative.

sin

Synopsis

```
double sin(double x);
```

Description

sin returns the radian circular sine of **x**.

If $|x| > 10^9$, **errno** is set to **EDOM** and **sin** returns **HUGE_VAL**.

sin returns **x** if **x** is NaN. **sin** returns NaN if $|x|$ is ∞ .

sinf

Synopsis

```
float sinf(float x);
```

Description

sinf returns the radian circular sine of **x**.

If $|x| > 10^9$, **errno** is set to **EDOM** and **sinf** returns **HUGE_VALF**.

sinf returns **x** if **x** is NaN. **sinf** returns NaN if $|x|$ is ∞ .

sinh

Synopsis

```
double sinh(double x);
```

Description

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

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

If **x** is $+\infty$, $-\infty$, or NaN, **sinh** returns $|x|$. If $|x| > \sim 709.782$, **sinh** returns $+\infty$ or $-\infty$ 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 $+\infty$, $-\infty$, or NaN, **sinhf** returns $|x|$. If $|x| > \sim 88.7228$, **sinhf** returns $+\infty$ or $-\infty$ depending upon the sign of **x**.

sqrt

Synopsis

```
double sqrt(double x);
```

Description

sqrt computes the nonnegative square root of **x**. C90 and C99 require that a domain error occurs if the argument is less than zero. CrossWorks C deviates and always uses IEC 60559 semantics.

If **x** is +0, **sqrt** returns +0.

If **x** is −0, **sqrt** returns −0.

If **x** is ∞ , **sqrt** returns ∞ .

If **x** < 0, **sqrt** returns NaN.

If **x** is NaN, **sqrt** returns that NaN.

sqrtf

Synopsis

```
float sqrtf(float x);
```

Description

sqrtf computes the nonnegative square root of **x**. C90 and C99 require that a domain error occurs if the argument is less than zero. CrossWorks C deviates and always uses IEC 60559 semantics.

If **x** is +0, **sqrtf** returns +0.

If **x** is −0, **sqrtf** returns −0.

If **x** is ∞ , **sqrtf** returns ∞ .

If **x** < 0, **sqrtf** returns NaN.

If **x** is NaN, **sqrtf** returns that NaN.

tan

Synopsis

```
double tan(double x);
```

Description

tan returns the radian circular tangent of **x**.

If $|x| > 10^9$, **errno** is set to **EDOM** and **tan** returns **HUGE_VAL**.

If **x** is NaN, **tan** returns **x**. If $|x|$ is ∞ , **tan** returns NaN.

tanf

Synopsis

```
float tanf(float x);
```

Description

tanf returns the radian circular tangent of **x**.

If $|x| > 10^9$, **errno** is set to **EDOM** and **tanf** returns **HUGE_VALF**.

If **x** is NaN, **tanf** returns **x**. If $|x|$ is ∞ , **tanf** returns NaN.

tanh

Synopsis

```
double tanh(double x);
```

Description

tanh calculates the hyperbolic tangent of **x**.

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

tanhf

Synopsis

```
float tanhf(float x);
```

Description

tanhf calculates the hyperbolic tangent of **x**.

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

<setjmp.h>

Functions

[longjmp](#)

Restores the saved environment

[setjmp](#)

Save calling environment for non-local jump

longjmp

Synopsis

```
void longjmp(jmp_buf env,  
             int val);
```

Description

longjmp restores the environment saved by **setjmp** in the corresponding **env** argument. If there has been no such invocation, or if the function containing the invocation of **setjmp** has terminated execution in the interim, the behavior of **longjmp** is undefined.

After **longjmp** is completed, program execution continues as if the corresponding invocation of **setjmp** had just returned the value specified by **val**.

Note

longjmp cannot cause **setjmp** to return the value 0; if **val** is 0, **setjmp** returns the value 1.

Objects of automatic storage allocation that are local to the function containing the invocation of the corresponding **setjmp** that do not have **volatile** qualified type and have been changed between the **setjmp** invocation and **this** call are indeterminate.

setjmp

Synopsis

```
int setjmp(jmp_buf env);
```

Description

setjmp saves its calling environment in the **env** for later use by the **longjmp** function.

On return from a direct invocation **setjmp** returns the value zero. On return from a call to the **longjmp** function, the **setjmp** returns a nonzero value determined by the call to **longjmp**.

The environment saved by a call to **setjmp** consists of information sufficient for a call to the **longjmp** function to return execution to the correct block and invocation of that block, were it called recursively.

<stdarg.h>

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,  
              parmN);
```

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>

Macros

NULL	NULL pointer
offsetof	offsetof

Types

ptrdiff_t	ptrdiff_t type
size_t	size_t type
wchar_t	wchar_t type

NULL

Synopsis

```
#define NULL 0
```

Description

NULL is the null pointer constant.

offsetof

Synopsis

```
#define offsetof(type, member)
```

Description

offsetof returns the offset in bytes to the structure **member**, from the beginning of its structure **type**.

ptrdiff_t

Synopsis

```
__PTRDIFF_T ptrdiff_t;
```

Description

ptrdiff_t is the signed integral type of the result of subtracting two pointers.

size_t

Synopsis

```
__SIZE_T size_t;
```

Description

size_t is the unsigned integral type returned by the sizeof operator.

wchar_t

Synopsis

```
__WCHAR_T wchar_t;
```

Description

wchar_t is the wide character type.

<stdio.h>

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();
```

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.

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.

snprintf

Synopsis

```
int snprintf(char *s,  
             size_t n,  
             const char *format,  
             ...);
```

Description

snprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output.

If **n** is zero, nothing is written, and **s** can be a null pointer. Otherwise, output characters beyond the **n**–1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

snprintf returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

sprintf

Synopsis

```
int sprintf(char *s,  
            const char *format,  
            ...);
```

Description

sprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. A null character is written at the end of the characters written; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

sprintf returns number of characters transmitted (not counting the terminating null), or a negative value if an output or encoding error occurred.

sscanf

Synopsis

```
int sscanf(const char *s,  
           const char *format,  
           ...);
```

Description

sscanf reads input from the string **s** under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

sscanf returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **sscanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

vprintf

Synopsis

```
int vprintf(const char *format,  
            __va_list arg);
```

Description

vprintf writes to the standard output stream using **putchar** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. Before calling **vprintf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vprintf** does not invoke the **va_end** macro.

vprintf returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Note

vprintf is equivalent to **printf** with the variable argument list replaced by **arg**.

vscanf

Synopsis

```
int vscanf(const char *format,  
           __va_list arg);
```

Description

vscanf reads input from the standard input stream under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. Before calling **vscanf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vscanf** does not invoke the **va_end** macro.

If there are insufficient arguments for the format, the behavior is undefined.

vscanf returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **vscanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Note

vscanf is equivalent to **scanf** with the variable argument list replaced **arg**.

vsnprintf

Synopsis

```
int vsnprintf(char *s,  
              size_t n,  
              const char *format,  
              __va_list arg);
```

Description

vsnprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. Before calling **vsnprintf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vsnprintf** does not invoke the **va_end** macro.

If **n** is zero, nothing is written, and **s** can be a null pointer. Otherwise, output characters beyond the **n**–1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

vsnprintf returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

Note

vsnprintf is equivalent to **snprintf** with the variable argument list replaced by **arg**.

vsprintf

Synopsis

```
int vsprintf(char *s,  
             const char *format,  
             __va_list arg);
```

Description

vsprintf writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. Before calling **vsprintf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vsprintf** does not invoke the **va_end** macro.

A null character is written at the end of the characters written; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

vsprintf returns number of characters transmitted (not counting the terminating null), or a negative value if an output or encoding error occurred.

Note

vsprintf is equivalent to **sprintf** with the variable argument list replaced by **arg**.

vsscanf

Synopsis

```
int vsscanf(const char *s,  
            const char *format,  
            __va_list arg);
```

Description

vsscanf reads input from the string **s** under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. Before calling **vsscanf**, **arg** must be initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). **vsscanf** does not invoke the **va_end** macro.

If there are insufficient arguments for the format, the behavior is undefined.

vsscanf returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **vsscanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Note

vsscanf is equivalent to **sscanf** with the variable argument list replaced by **arg**.

<stdlib.h>

Macros	
EXIT_FAILURE	EXIT_FAILURE
EXIT_SUCCESS	EXIT_SUCCESS
RAND_MAX	RAND_MAX
Integer arithmetic functions	
abs	Return an integer absolute value
div	Divide two ints returning quotient and remainder
labs	Return a long integer absolute value
ldiv	Divide two longs returning quotient and remainder
llabs	Return a long long integer absolute value
lldiv	Divide two long longs returning quotient and remainder
Memory allocation functions	
calloc	Allocate space for an array of objects and initialize them to zero
free	Frees allocated memory for reuse
malloc	Allocate space for a single object
realloc	Resizes allocated memory space or allocates memory space
String to number conversions	
atof	Convert string to double
atoi	Convert string to int
atol	Convert string to long
atoll	Convert string to long long
strtod	Convert string to double
strtof	Convert string to float
strtol	Convert string to long
strtoll	Convert string to long long
strtoul	Convert string to unsigned long
strtoull	Convert string to unsigned long long
Pseudo-random sequence generation functions	
rand	Return next random number in sequence
srand	Set seed of random number sequence
Search and sort functions	

bsearch	Search a sorted array
qsort	Sort an array
Environment	
atexit	Set function to be execute on exit
exit	Terminates the calling process
Number to string conversions	
itoa	Convert int to string
lltoa	Convert long long to string
ltoa	Convert long to string
ulltoa	Convert unsigned long long to string
ultoa	Convert unsigned long to string
utoa	Convert unsigned to string
Types	
div_t	Structure containing quotient and remainder after division of an int
ldiv_t	Structure containing quotient and remainder after division of a long
lldiv_t	Structure containing quotient and remainder after division of a long long

EXIT_FAILURE

Synopsis

```
#define EXIT_FAILURE    1
```

Description

EXIT_FAILURE pass to [exit](#) on unsuccessful termination.

EXIT_SUCCESS

Synopsis

```
#define EXIT_SUCCESS    0
```

Description

EXIT_SUCCESS pass to [exit](#) on successful termination.

RAND_MAX

Synopsis

```
#define RAND_MAX 32767
```

Description

RAND_MAX expands to an integer constant expression that is the maximum value returned by [rand](#).

abs

Synopsis

```
int abs(int j);
```

Description

abs returns the absolute value of the integer operand **j**.

atexit

Synopsis

```
int atexit(void (*func)());
```

Description

atexit registers **function** to be called when the application has exited. The functions registered with **atexit** are executed in reverse order of their registration. **atexit** returns 0 on success and non-zero on failure.

atof

Synopsis

```
double atof(const char *nptr);
```

Description

atof converts the initial portion of the string pointed to by **nptr** to a **double** representation. **atof** does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atof** is equivalent to `strtod(nptr, (char **)NULL)`.

atof returns the converted value.

See Also

[strtod](#)

atoi

Synopsis

```
int atoi(const char *nptr);
```

Description

atoi converts the initial portion of the string pointed to by **nptr** to an **int** representation.

atoi does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atoi** is equivalent to `(int)strtol(nptr, (char **)NULL, 10)`.

atoi returns the converted value.

See Also

[strtol](#)

atol

Synopsis

```
long int atol(const char *nptr);
```

Description

atol converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

atol does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atol** is equivalent to `strtol(nptr, (char **)NULL, 10)`.

atol returns the converted value.

See Also

[strtol](#)

atoll

Synopsis

```
long long int atoll(const char *nptr);
```

Description

atoll converts the initial portion of the string pointed to by **nptr** to a **long long int** representation.

atoll does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atoll** is equivalent to `strtoll(nptr, (char **)NULL, 10)`. **atoll** returns the converted value.

See Also

[strtoll](#)

bsearch

Synopsis

```
void *bsearch(const void *key,
              const void *buf,
              size_t num,
              size_t size,
              int (*compare)(const void *, const void *));
```

Description

bsearch searches the array ***base** for the specified **{*key}** and returns a pointer to the first entry that matches or null if no match. The array should have **num** elements of **size** bytes and be sorted by the same algorithm as the **compare** function

The **compare** function should return a negative value if the first parameter is less than second parameter, zero if the parameters are equal, and a positive value if the first parameter is greater than the second parameter.

calloc

Synopsis

```
void *calloc(size_t nobj,  
             size_t size);
```

Description

calloc allocates space for an array of **nmemb** objects, each of whose size is **size**. The space is initialized to all zero bits.

calloc returns a null pointer if the space for the array of object cannot be allocated from free memory; if space for the array can be allocated, **calloc** returns a pointer to the start of the allocated space.

div

Synopsis

```
div_t div(int numer,  
          int denom);
```

Description

div computes **numer** / **denom** and **numer** % **denom** in a single operation.

div returns a structure of type [div_t](#) comprising both the quotient and the remainder. The structures contain the members **quot** (the quotient) and **rem** (the remainder), each of which has the same type as the arguments **numer** and **denom**. If either part of the result cannot be represented, the behavior is undefined.

See Also

[div_t](#)

div_t

Description

`div_t` stores the quotient and remainder returned by `div`.

exit

Synopsis

```
void exit(int exit_code);
```

Description

exit returns to the startup code and performs the appropriate cleanup process.

free

Synopsis

```
void free(void *p);
```

Description

free causes the space pointed to by **ptr** to be deallocated, that is, made available for further allocation. If **ptr** is a null pointer, no action occurs.

If **ptr** does not match a pointer earlier returned by **calloc**, **malloc**, or **realloc**, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

itoa

Synopsis

```
char *itoa(int val,  
           char *buf,  
           int radix);
```

Description

itoa converts **val** to a string in base **radix** and places the result in **buf**.

itoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

If **val** is negative and **radix** is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

See Also

[ltoa](#), [lltoa](#), [ultoa](#), [ulltoa](#), [utoa](#)

labs

Synopsis

```
long int labs(long int j);
```

Description

labs returns the absolute value of the long integer operand **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 operand **j**.

lldiv

Synopsis

```
lldiv_t lldiv(long long int numer,  
             long long int denom);
```

lldiv computes **numer** / **denom** and **numer** % **denom** in a single operation. **lldiv** returns a structure of type **lldiv_t** comprising both the quotient and the remainder. The structures contain the members **quot** (the quotient) and **rem** (the remainder), each of which has the same type as the arguments **numer** and **denom**. If either part of the result cannot be represented, the behavior is undefined.

See Also

[lldiv_t](#)

lldiv_t

Description

`lldiv_t` stores the quotient and remainder returned by [lldiv](#).

lltoa

Synopsis

```
char *lltoa(long long val,  
            char *buf,  
            int radix);
```

Description

lltoa converts **val** to a string in base **radix** and places the result in **buf**.

lltoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

If **val** is negative and **radix** is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

See Also

[itoa](#), [ltoa](#), [ultoa](#), [ulltoa](#), [utoa](#)

ltoa

Synopsis

```
char *ltoa(long val,  
           char *buf,  
           int radix);
```

Description

ltoa converts **val** to a string in base **radix** and places the result in **buf**.

ltoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

If **val** is negative and **radix** is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

See Also

[itoa](#), [lltoa](#), [ultoa](#), [ulltoa](#), [utoa](#)

malloc

Synopsis

```
void *malloc(size_t size);
```

Description

malloc allocates space for an object whose size is specified by 'b size and whose value is indeterminate.

malloc returns a null pointer if the space for the object cannot be allocated from free memory; if space for the object can be allocated, **malloc** returns a pointer to the start of the allocated space.

qsort

Synopsis

```
void qsort(void *buf,  
           size_t num,  
           size_t size,  
           int (*compare)(const void *, const void *));
```

qsort sorts the array ***base** using the **compare** algorithm. The array should have **num** elements of **size** bytes. The **compare** function should return a negative value if the first parameter is less than second parameter, zero if the parameters are equal and a positive value if the first parameter is greater than the second parameter.

rand

Synopsis

```
int rand();
```

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](#) or 'ref rand_max

strtod

Synopsis

```
double strtod(const char *nptr,  
              char **endptr);
```

Description

strtod converts the initial portion of the string pointed to by **nptr** to a **double** representation.

First, **strtod** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by [isspace](#)), a subject sequence resembling a floating-point constant, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtod** then attempts to convert the subject sequence to a floating-point number, and return the result.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **strtod**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtod returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **HUGE_VAL** is returned according to the sign of the value, if any, and the value of the macro [errno](#) is stored in [errno](#).

strtof

Synopsis

```
float strtof(const char *nptr,  
            char **endptr);
```

Description

strtof converts the initial portion of the string pointed to by **nptr** to a **double** representation.

First, **strtof** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace**), a subject sequence resembling a floating-point constant, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtof** then attempts to convert the subject sequence to a floating-point number, and return the result.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtof returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **HUGE_VALF** is returned according to the sign of the value, if any, and the value of the macro **errno** is stored in **errno**.

strtol

Synopsis

```
long int strtol(const char *nptr,  
               char **endptr,  
               int base);
```

Description

strtol converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtol** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by [isspace](#)), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtol** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtol returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **long_min** or **long_max** is returned according to the sign of the value, if any, and the value of the macro **errno** is stored in **errno**.

strtoll

Synopsis

```
long long int strtoll(const char *nptr,  
                     char **endptr,  
                     int base);
```

Description

strtoll converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoll** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by [isspace](#)), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoll** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits, following the optional sign. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtoll returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, [llong_min](#) or [llong_max](#) is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in [errno](#).

strtoul

Synopsis

```
unsigned long int strtoul(const char *nptr,  
                        char **endptr,  
                        int base);
```

Description

strtoul converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoul** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by [isspace](#)), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoul** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant. If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtoul returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, [long_max](#) or [ulong_max](#) is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in [errno](#).

strtoull

Synopsis

```
unsigned long long int strtoull(const char *nptr,  
                               char **endptr,  
                               int base);
```

Description

strtoull converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoull** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by [isspace](#)), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoull** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters "0x" or "0X" may optionally precede the sequence of letters and digits, following the optional sign. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

strtoull returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, [llong_max](#) or [ullong_max](#) is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in [errno](#).

ulltoa

Synopsis

```
char *ulltoa(unsigned long long val,  
             char *buf,  
             int radix);
```

Description

ulltoa converts **val** to a string in base **radix** and places the result in **buf**.

ulltoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

See Also

[itoa](#), [ltoa](#), [lltoa](#), [ultoa](#), [utoa](#)

ultoa

Synopsis

```
char *ultoa(unsigned long val,  
            char *buf,  
            int radix);
```

Description

ultoa converts **val** to a string in base **radix** and places the result in **buf**.

ultoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

See Also

[itoa](#), [ltoa](#), [lltoa](#), [ulltoa](#), [utoa](#)

utoa

Synopsis

```
char *utoa(unsigned val,  
            char *buf,  
            int radix);
```

Description

utoa converts **val** to a string in base **radix** and places the result in **buf**.

utoa returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

See Also

[itoa](#), [ltoa](#), [lltoa](#), [ultoa](#), [ulltoa](#)

<string.h>

Copying functions	
memcpy	Copy memory
memmove	Safely copy overlapping memory
strcat	Concatenate strings
strcpy	Copy string
strdup	Duplicate string
strncat	Concatenate strings up to maximum length
strncpy	Copy string up to a maximum length
strndup	Duplicate string
Comparison functions	
memcmp	Compare memory
strcasecmp	Compare strings ignoring case
strcmp	Compare strings
strncasecmp	Compare strings up to a maximum length ignoring case
strncmp	Compare strings up to a maximum length
Search functions	
memchr	Search memory for a character
strchr	Find character within string
strcspn	Compute size of string not prefixed by a set of characters
strnchr	Find character in a length-limited string
strnlen	Calculate length of length-limited string
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
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)
Miscellaneous functions	
memset	Set memory to character

<code>strerror</code>	Decode error code
<code>strlen</code>	Calculate length of string

memchr

Synopsis

```
void *memchr(const void *s,  
             int c,  
             size_t n);
```

Description

memchr locates the first occurrence of **c** (converted to an **unsigned char**) in the initial **n** characters (each interpreted as **unsigned char**) of the object pointed to by **s**. Unlike **strchr**, **memchr** does *not* terminate a search when a null character is found in the object pointed to by **s**.

memchr returns a pointer to the located character, or a null pointer if **c** does not occur in the object.

memcmp

Synopsis

```
int memcmp(const void *s1,  
           const void *s2,  
           size_t n);
```

Description

memcmp compares the first **n** characters of the object pointed to by **s1** to the first **n** characters of the object pointed to by **s2**. **memcmp** returns an integer greater than, equal to, or less than zero as the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.

memcpy

Synopsis

```
void *memcpy(void *s1,  
             const void *s2,  
             size_t n);
```

Description

memcpy copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The behavior of **memcpy** is undefined if copying takes place between objects that overlap.

memcpy returns the value of **s1**.

memmove

Synopsis

```
void *memmove(void *s1,  
              const void *s2,  
              size_t n);
```

Description

memmove copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1** ensuring that if **s1** and **s2** overlap, the copy works correctly. Copying takes place as if the **n** characters from the object pointed to by **s2** are first copied into a temporary array of **n** characters that does not overlap the objects pointed to by **s1** and **s2**, and then the **n** characters from the temporary array are copied into the object pointed to by **s1**.

memmove returns the value of **s1**.

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.

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**.

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.

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 is an extension commonly found in Linux and BSD C libraries.

See Also

[strtok](#).

<time.h>

Functions	
asctime	Convert a struct tm to a string
asctime_r	Convert a struct tm to a string
ctime	Convert a time_t to a string
ctime_r	Convert a time_t to a string
difftime	Calculates the difference between two times
gmtime	Convert a time_t to a struct tm
gmtime_r	Convert a time_t to a struct tm
localtime	Convert a time_t to a struct tm
localtime_r	Convert a time_t to a struct tm
mktime	Convert a struct tm to time_t
strftime	Format a struct tm to a string
Types	
clock_t	Clock type
time_t	Time type
tm	Time structure

asctime

Synopsis

```
char *asctime(const tm *tp);
```

Description

asctime converts the ***tp** struct to a null terminated string of the form Sun Sep 16 01:03:52 1973. The returned string is held in a static buffer, this function is not re-entrant.

asctime_r

Synopsis

```
char *asctime_r(const tm *tp,  
                char *buf);
```

Description

asctime_r converts the ***tp** struct to a null terminated string of the form Sun Sep 16 01:03:52 1973 in **buf** and returns **buf**. The **buf** must point to an array at least 26 bytes in length.

clock_t

Synopsis

```
long clock_t;
```

Description

clock_t is the type returned by the **clock** function.

ctime

Synopsis

```
char *ctime(const time_t *tp);
```

Description

ctime converts the ***tp** to a null terminated string. The returned string is held in a static buffer, this function is not re-entrant.

ctime_r

Synopsis

```
char *ctime_r(const time_t *tp,  
              char *buf);
```

Description

ctime_r converts the ***tp** to a null terminated string in **buf** and returns **buf**. The **buf** must point to an array at least 26 bytes in length.

difftime

Synopsis

```
double difftime(time_t time2,  
                time_t time1);
```

Description

difftime returns **time1 - time0** as a double precision number.

gmtime

Synopsis

```
gmtime(const time_t *tp);
```

Description

gmtime converts the ***tp** time format to a **struct tm** time format. The returned value points to a static object - this function is not re-entrant.

gmtime_r

Synopsis

```
gmtime_r(const time_t *tp,  
          tm *result);
```

Description

gmtime_r converts the ***tp** time format to a **struct tm** time format in ***result** and returns **result**.

localtime

Synopsis

```
localtime(const time_t *tp);
```

Description

localtime converts the ***tp** time format to a **struct tm** local time format. The returned value points to a static object - this function is not re-entrant.

localtime_r

Synopsis

```
localtime_r(const time_t *tp,  
            tm *result);
```

Description

localtime_r converts the ***tp** time format to a **struct tm** local time format in ***result** and returns **result**.

mktime

Synopsis

```
time_t mktime(tm *tp);
```

Description

mktime validates (and updates) the ***tp** struct to ensure that the **tm_sec**, **tm_min**, **tm_hour**, **tm_mon** fields are within the supported integer ranges and the **tm_m_day**, **tm_mon** and **tm_year** fields are consistent. The validated ***tp** struct is converted to the number of seconds since UTC 1/1/70 and returned.

strftime

Synopsis

```
size_t strftime(char *s,
                size_t smax,
                const char *fmt,
                const tm *tp);
```

Description

strftime formats the ***tp** struct to a null terminated string of maximum size **smax-1** into the array at ***s** based on the **fmt** format string. The format string consists of conversion specifications and ordinary characters. Conversion specifications start with a % character followed by an optional # character. The following conversion specifications are supported:

Specification	Description
%	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for locale
%#c	Long date and time representation appropriate for locale
%d	Day of month [01,31]
%#d	Day of month without leading zero [1,31]
%H	Hour in 24-hour format [00,23]
%#H	Hour in 24-hour format without leading zeros [0,23]
%I	Hour in 12-hour format [01,12]
%#I	Hour in 12-hour format without leading zeros [1,12]
%j	Day of year as a decimal number [001,366]
%#j	Day of year as a decimal number without leading zeros [1,366]
%m	Month as a decimal number [01,12]
%#m	Month as a decimal number without leading zeros [1,12]
%M	Minute as a decimal number [00,59]
%#M	Minute as a decimal number without leading zeros [0,59]
%#p	Locale's a.m or p.m indicator
%S	Second as a decimal number [00,59]

%#S	Second as a decimal number without leading zeros [0,59]
%U	Week number as a decimal number [00,53], Sunday is first day of the week
%#U	Week number as a decimal number without leading zeros [0,53], Sunday is first day of the week
%w	Weekday as a decimal number [0,6], Sunday is 0
%W	Week number as a decimal number [00,53], Monday is first day of the week
%#W	Week number as a decimal number without leading zeros [0,53], Monday is first day of the week
%x	Locale's date representation
%#x	Locale's long date representation
%X	Locale's time representation
%y	Year without century, as a decimal number [00,99]
%#y	Year without century, as a decimal number without leading zeros [0,99]
%z,%Z	Timezone name or abbreviation
%%	%

time_t

Synopsis

```
long time_t;
```

Description

time_t is a long type that represents the time in number of seconds since UTC 1/1/70, negative values indicate time before UTC 1/1/70.

tm

Synopsis

```
typedef struct {  
    int tm_sec;  
    int tm_min;  
    int tm_hour;  
    int tm_mday;  
    int tm_mon;  
    int tm_year;  
    int tm_wday;  
    int tm_yday;  
    int tm_isdst;  
} tm;
```

Description

tm structure has the following fields.

Member	Description
tm_sec	seconds after the minute - [0,59]
tm_min	minutes after the hour - [0,59]
tm_hour	hours since midnight - [0,23]
tm_mday	day of the month - [1,31]
tm_mon	months since January - [0,11]
tm_year	years since 1900
tm_wday	days since Sunday - [0,6]
tm_yday	days since January 1 - [0,365]
tm_isdst	daylight savings time flag

Assembler Reference

The assembler converts assembly source code to relocatable object code written to object code files. The linker takes these object code files and combines them to form an application containing the final instructions. This manual is a reference for the CrossWorks assembler. What we don't do in this manual is explain the architecture of the process machine or how to go about constructing an application in assembly code.

In this section

Command line options

Describes the command line options accepted by the assembler.

Source format

Describes the format of source lines.

Data definition and allocation directives

Describes how to allocate initialized and uninitialized data, align data, and so on.

Labels, variables, and sections

Describes how labels, variables, and sections are defined.

Data types

Describes the strong type system that the assembler uses.

Expressions and operators

Describes the syntax of expressions and how to use the built-in operators.

Compilation units and libraries

Describes how to structure your source code by using relocatable object modules and source inclusion.

Macros, conditions, and loops

Describes how to use the macro processor, conditionals, and loops for repetitive tasks.

Command line options

The assembler understands the following command line options:

Option	Description
-D name[= value]	Define name to be value or -1. See -D (Define macro symbol) .
-g	Generate debug information. See -g (Generate debugging information) .
-I dir	Add dir to the end of the user include search path. See -I (Define include directories) .
-J dir	Add dir to the end of the system include search path. See -J (Define system include directories) .
-o file	Write the object module to file . See -o (Set output file name) .
-Rc , name	Set default code section name to name . See -Rc (Set default code section name) .
-Rd , name	Set default data section name to name . See -Rd (Set default initialized data section name) .
-Rk , name	Set default read-only constant section name to name . See -Rk (Set default read-only data section name) .
-Rv , name	Set default vector section name to name . See -Rv (Set default vector section name) .
-Rz , name	Set default zeroes data section name to name . See -Rz (Set default zeroed data section name) .
-V	Display version information. See -V (Display version) .
-w	Suppress warning messages. See -w (Suppress warnings) .
-we	Treat warnings as errors. See -we (Treat warnings as errors) .

-D (Define macro symbol)

Syntax

-D name

-D name = value

Description

This option instructs the assembler to define a symbol for the compilation unit. If no value is given, the symbol is defined to the value -1.

Setting this in CrossStudio

To define symbols for a project:

- Select the project in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **Preprocessor Definitions** property.

To define symbols for a particular file:

- Select the file in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **Preprocessor Definitions** property.

The **Preprocessor Definitions** property is a semicolon-separated list of symbol definitions, for example "**name1=value1;name2=value2**". Clicking the button at the right of the property displays the **Preprocessor Definitions** dialog which will allow you to easily edit the definitions.

Example

The following defines two macros, **SUPPORT_FLOAT** with a value of 1 and **LITTLE_ENDIAN** with a value of -1.

```
-DSUPPORT_FLOAT=1 -DLITTLE_ENDIAN
```

-g (Generate debugging information)

Syntax

-g

Description

The **-g** option instructs the assembler to insert debugging information into the output file. This allows you to single step through assembly language files at the source level through source files (with all its annotation) rather than a disassembly of the code. And declared, typed data is displayed correctly rather than as simply an address.

Setting this in CrossStudio

To set include debugging information for all files in a project:

- Select the project in the **Project Explorer**.
- In the **General Options** group set the **Include Debug Information** property to **Yes** or **No** as appropriate.

To set include debugging information for a particular file (**not recommended**):

- Select the file in the **Project Explorer**.
- In the **General Options** group set the **Include Debug Information** property to **Yes** or **No** as appropriate.

-I (Define include directories)

Syntax

-I directory

The **-I** option adds **directory** to the end of the list of directories to search for source files included (using quotation marks) by the **INCLUDE** and **INCLUDEBIN** directives.

Setting this in CrossStudio

To set the directories searched for include files for a project:

- Select the project in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **User Include Directories** property.

To set the directories searched for include files for a particular file:

- Select the file in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **User Include Directories** property.

Example

For example, to tell the assembler to search the directories **../include** and **../lib/include** for included files when compiling **file.asm** and writing the output to **file.hzo** you could use the following:

```
has -I../include -I../lib/include file.c -o file.hzo file.asm
```

-J (Define system include directories)

Syntax

-J directory

The **-J** option adds **directory** to the end of the list of directories to search for source files included (using triangular brackets) by the **INCLUDE** and **INCLUDEBIN** directives.

Example

For example, to tell the assembler to search the directories **../include** and **../lib/include** for included system files when compiling **file.asm** and writing the output to **file.hzo** you could use the following:

```
has -J../include -J../lib/include file.c -o file.hzo file.asm
```

-o (Set output file name)

Syntax

-o filename

Description

The **-o** option instructs the assembler to write its object file to **filename**.

-Rc (Set default code section name)

Syntax

-Rc, name

Description

The **-Rc** command line option sets the name of the default code section that the assembler uses for the **.TEXT** and **.CODE** directives. If no other options are given, the default name for the section is **CODE**.

You can control the name of the code section used by the assembler within a source file using the **RSEG** and **.SECTION** directives or by using CrossStudio to set the **Code Section Name** property of the file or project.

Setting this in CrossStudio

To set the default code section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Code Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Code Section Name** property.

Example

The following command line option instructs the assembler to use the name **RAMCODE** as the default code section name for the **.TEXT** and **.CODE** directives.

-Rc, RAMCODE

-Rd (Set default initialized data section name)

Syntax

-Rd, name

Description

The **-Rd** command line option sets the name of the default data section that the assembler uses for the **DATA** directive. If no other options are given, the default name for the section is **IDATA0**.

You can control the name of the data section used by the assembler within a source file using the **RSEG** and **.SECTION** directives or by using CrossStudio to set the **Data Section Name** property of the file or project.

Setting this in CrossStudio

To set the default data section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Data Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Data Section Name** property.

Example

The following command line option instructs the assembler to use the name **NVDATA** as the default initialised section name selected by **.DATA**.

```
-Rd, NVDATA
```

-Rk (Set default read-only data section name)

Syntax

-Rk, name

Description

The **-Rk** command line option sets the name of the default data section that the assembler uses for the **CONST** and **RODATA** directives. If no other options are given, the default name for the section is **CONST**.

You can control the name of the read-only data section used by the assembler within a source file using the **RSEG** and **.SECTION** directives or by using CrossStudio to set the **Constant Section Name** property of the file or project.

Setting this in CrossStudio

To set the default constant section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Constant Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Constant Section Name** property.

Example

The following command line option instructs the assembler to use the name **ROMDATA** as the default read-only data section name.

-Rk, ROMDATA

-Rv (Set default vector section name)

Syntax

-Rv, name

Description

The **-Rv** command line option sets the name of the default vector table section that the assembler uses for the **.VECTORS** directive. If no other options are given, the default name for the section is **INTVEC**.

You can control the name of the vector table section used by the assembler within a source file using the **RSEG** and **.SECTION** directives or by using CrossStudio to set the **Vector Section Name** property of the file or project.

Setting this in CrossStudio

To set the default interrupt vector section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Vector Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Vector Section Name** property.

Example

The following command line option instructs the assembler to use the name **IVDATA** as the default vector section name selected by **.VECTORS**.

```
-Rv, IVDATA
```

-Rz (Set default zeroed data section name)

Syntax

-Rz, name

Description

The **-Rz** command line option sets the name of the default zeroed data section that the assembler uses for the **.BSS** directive. If no other options are given, the default name for the section is **UDATA0**. Uninitialised data in **UDATA0** is set to zero on program startup.

You can control the name of the zeroed data section used by the assembler within a source file using the **RSEG** and **.SECTION** directives or by using CrossStudio to set the **Zeroed Section Name** property of the file or project.

Setting this in CrossStudio

To set the default zeroed section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Zeroed Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Zeroed Section Name** property.

Example

The following command line option instructs the assembler to use the name **ZDATA** as the default zeroed data section name selected by the **.BSS** directive.

-Rz, ZDATA

-V (Display version)

Syntax

-V

Description

The **-V** option instructs the assembler to display its version information.

-we (Treat warnings as errors)

Syntax

-we

Description

This option directs the assembler to treat all warnings as errors.

Setting this in CrossStudio

To suppress warnings for all files in a project:

- Select the project in the **Project Explorer**.
- In the **General Options** group set the **Treat Warnings as Errors** property to **Yes**.

To suppress warnings for a particular file:

- Select the file in the **Project Explorer**.
- In the **General Options** group set the **Treat Warnings as Errors** property to **Yes**.

-w (Suppress warnings)

Syntax

-w

Description

This option instructs the assembler not to issue any warnings.

Setting this in CrossStudio

To suppress warnings for a project:

- Select the project in the **Project Explorer**.
- In the **General Options** group set the **Suppress Warnings** property to **Yes**.

To suppress warnings for a particular file:

- Select the file in the **Project Explorer**.
- In the **General Options** group set the **Suppress Warnings** property to **Yes**.

Source format

A statement is a combination of mnemonics, operands, and comments that defines the object code to be created at assembly time. Each line of source code contains a single statement.

Syntax

Assembler statements take the form:

```
[label] [operation] [operands] [comment]
```

All fields are optional, although the operand or label fields may be required if certain directives or instructions are used in the operation field.

Label Field

The label field starts at the left of the line, with no preceding spaces. A label name is a sequence of alphanumeric characters, starting with a letter. You can also use the dollar sign '\$' and underline character '_' in label names. A colon may be placed directly after the label, or it can be omitted. If a colon is placed after a label, it defines that label to be the value of the location counter in the current section.

Operation field

The operation field contains either a machine instruction or an assembler directive. You must write these in either all upper-case or all lower-case, mixed case is not allowed. The operation field must not start at the left of the line; at least one space must precede it if there is no label field. At least one space must separate the label field and the operation field.

Operand field

The contents of the operand depend upon the instruction or directive in the operation field. Different instructions and directives have different operand field formats. Please refer to the specific section for details of the operand field.

Comment field

The comment field is optional, and contains information that is not essential to the assembler, but is useful for documentation. The comment field must be separated from the previous fields by at least one space.

Comments

To help others better understand some particularly tricky piece of code, you can insert comments into the source program. Comments are simply informational attachments and have no significance for the assembler. Comments come in two forms: single-line comments and multi-line comments.

Single-line comments

A single line comment is introduced either by single character `;` or by the characters `//`.

Syntax

// character...

; character...

The assembler ignores all characters from the comment introducer to the end of the line. This type of comment is particularly good when you want to comment a single assembler line.

Multi-line comments

A multi-line comment resembles a standard C comment as it is introduced by the characters `/*` and is terminated by `*/`.

Syntax

/ character... */*

Anything in between these delimiters is ignored by the assembler. You can use this type of comment to place large amounts of commentary, such as copyright notices or functional descriptions, into your code.

Data definition and allocation directives

You can allocate and initialise memory for data using data definition directives. There are a wide range of data definition directives covering a wide range of uses, and many of these have the same semantics.

You can define **typed** variables using the **DV** directive. The **DS** directive reserves storage but does not initialize variables.

Data alignment is critical in many instances. Defining exactly how your data are arranged in memory is usually a requirement of interfacing with the outside world using devices or communication areas shared between the application and operating system. You can align the current section's location counter with the **ALIGN** and **EVEN** directives.

ALIGN directive

Syntax

ALIGN *type* | *number*

The operand given after the directive defines the alignment requirement. If a type is given the location counter is adjusted to be divisible by the size of the type with no remainder. If a number is given the location counter is adjusted to be divisible by 2^{number} with no remainder.

Example

```
ALIGN LONG
```

This aligns the location counter so that it lies on a 4-byte boundary as the type LONG has size 4.

Example

```
ALIGN 3
```

This aligns the location counter so that it lies on a 8-byte boundary as 2^3 is 8.

DC.A directive

Syntax

DC.A *initialiser* [, *initialiser*]...

DA *initialiser* [, *initialiser*]...

Description

The **DA** directive defines an object as an initialised array of addresses. If the directive is labeled, the label is assigned the location counter of the current section before the data are placed in that section. If a single initializer is present, the label's data type is set to **ADDR**, otherwise it is set to be a fixed array of **ADDR**, the bounds of which are set by the number of elements defined.

Important notes

The location counter is **not** aligned before allocating space.

Example

```
FuncTable DA Method1, Method2, Method3
```

This defines the label **FuncTable** and allocates three addresses initialised to the addresses of **Method1**, **Method2**, and **Method3**. The type of **Mask** is set to **ADDR[3]**, an array of three address, as three values are listed.

DC.B directive

Syntax

DC.B *initialiser* [, *initialiser*]...

DB *initialiser* [, *initialiser*]...

FCB *initialiser* [, *initialiser*]...

BYTE *initialiser* [, *initialiser*]...

Description

The **DC.B** directive defines an object as an initialised array of bytes. If the directive is labeled, the label is assigned the location counter of the current section before the data are placed in that section. If a single initializer is present, the label's data type is set to **BYTE**, otherwise it is set to be a fixed array of **BYTE**, the bounds of which are set by the number of elements defined.

Example

```
Mask DC.B 0x01, 0x03, 0x07, 0x0f, 0x1f, 0x3f, 0x7f, 0xff
```

This defines the label **Mask** and allocates eight bytes with the given values. The type of **Mask** is set to **BYTE[8]**, an array of eight bytes, as eight values are listed.

You can define string data using the **DB** directive. When the assembler sees a string, it expands the string into a series of bytes and places those into the current section.

Example

```
BufOvfL DC.B 13, 10, "WARNING: buffer overflow", 0
```

This emits the bytes 13 and 10 into the current section, followed by the ASCII bytes comprising the string, and finally a trailing zero byte.

DC.L directive

Syntax

DC.L *initialiser* [, *initialiser*]...

DL *initialiser* [, *initialiser*]...

FCL *initialiser* [, *initialiser*]...

LONG *initialiser* [, *initialiser*]...

Description

The **DC.L** directive defines an object as an initialised array of longs. If the directive is labeled, the label is assigned the location counter of the current section before the data are placed in that section. If a single initializer is present, the label's data type is set to **LONG**, otherwise it is set to be a fixed array of **LONG**, the bounds of which are set by the number of elements defined.

Important notes

The location counter is **not** aligned before allocating space.

Example

```
Power10 DC.L 1, 10, 100, 1000, 10000, 100000
```

This defines the label **POWER10** and allocates six long words with the given values. The type of **POWER10** is set to **LONG[6]**, an array of six longs, as six values are listed.

DC.W directive

Syntax

DC.W *initialiser* [, *initialiser*]...

DW *initialiser* [, *initialiser*]...

FCW *initialiser* [, *initialiser*]...

WORD *initialiser* [, *initialiser*]...

Description

The **DC.W** directive defines an object as an initialised array of words. If the directive is labeled, the label is assigned the location counter of the current section before the data are placed in that section. If a single initializer is present, the label's data type is set to **WORD**, otherwise it is set to be a fixed array of **WORD**, the bounds of which are set by the number of elements defined.

The number of bytes per word is defined by the target processor. For 32-bit processors, one word is usually four bytes, and for 8-bit and 16-bit processors, one word is usually two bytes.

Important notes

The location counter is **not** aligned before allocating space.

Example

```
Power10 DC.W 1, 10, 100, 1000, 10000
```

This defines the label **Power10** and allocates five initialised words with the given values. The type of **Power10** is set to **WORD[5]**, an array of five words, as five values are listed.

EVEN directive

Syntax

EVEN

Description

The **EVEN** directive is equivalent to ***ALIGN 1*** and aligns the location counter to the next even address.

FILL directive

Syntax

FILL *size, value*

Description

The **FILL** directive generates **size** bytes of **value** into the current section and adjusts the location counter accordingly.

Example

```
FILL 5, ' '
```

This generates five spaces into the current section.

DS.B directive

Syntax

DS.B *n*

DS *n*

RMB *n*

SPACE *n*

Description

These directives generate *n* bytes of zeroes into the current section and adjusts the location counter accordingly. If the directive is labeled, the label is assigned the location counter of the current section before the space is allocated in that section. If *n* is one, the label's data type is set to **BYTE**, otherwise it is set to be a fixed array of **BYTE[*n*]**.

Example

```
Temp DS.B 10
```

This reserves 10 bytes in the current section, sets them all to zero, and defines **Temp** as the start of the block with type **BYTE[10]**.

DS.W directive

Syntax

DS.W *n*

RMW *n*

Description

These directives generate *n* words of zeroes into the current section and adjusts the location counter accordingly. If the directive is labeled, the label is assigned the location counter of the current section before the space is allocated in that section. If *n* is one, the label's data type is set to **WORD**, otherwise it is set to be a fixed array of **WORD[*n*]**.

The number of bytes per word is defined by the target processor. For 32-bit processors, one word is usually four bytes, and for 8-bit and 16-bit processors, one word is usually two bytes.

Important notes

The location counter is **not** aligned before allocating space.

Example

```
Temp DS.W 10
```

This reserves 10 words in the current section, sets them all to zero, and defines **Temp** as the start of the block with type **WORD[10]**.

DS.L directive

Syntax

DS.L *n*

RML *n*

Description

These directives generate *n* long words of zeroes into the current section and adjusts the location counter accordingly. If the directive is labeled, the label is assigned the location counter of the current section before the space is allocated in that section. If *n* is one, the label's data type is set to **LONG**, otherwise it is set to be a fixed array of **LONG[*n*]**.

Important notes

The location counter is **not** aligned before allocating space.

Example

```
Temp DS.L 10
```

This reserves 10 long words in the current section, sets them all to zero, and defines **Temp** as the start of the block with type **LONG[10]**.

DV directive

Syntax

DV *datatype* [= *initializer*]

Description

This directive reserves space for a data item of type **datatype** and optionally initializes it to a value. The initializer is a comma separated list of numbers and strings.

Important notes

The location counter is **not** aligned before allocating space.

Example

```
x DV WORD = 12
```

This defines the label **x** to be a **WORD** variable initialized to 12.

INCLUDEBIN directive

Syntax

INCLUDEBIN " filename "

INCLUDEBIN < filename >

Description

The **INCLUDEBIN** directive inserts the contents of **filename** into the assembly as binary data. If **filename** is enclosed in quotation marks, the user include directories are searched, and if **filename** is enclosed in triangular brackets the system include directories are searched.

See Also

[Set user include directories](#), [Set system include directories](#)

Labels, variables, and sections

This section explains how to define labels, variables, and other symbols that refer to data locations in sections. The assembler keeps an independent location counter for each section in your application. When you define data or code in a section, the location counter for that section is adjusted, and this adjustment does not affect the location counters of other sections. When you define a label in a section, the current value of the location counter for that section is assigned to the symbol.

Defining sections

You can create/change the current section being assembled using a number of directive styles.

Syntax

```
SECT " name "
PSECT " name "
DSECT " name "
USECT " name "
CSECT " name "
ASECT "name "
```

These directives enable you to create/change sections with your own names.

The **SECT** directive creates/changes to an untyped section called *name*.

The **PSECT** directive creates/changes to a **CODE** typed section called *name*.

The **DSECT** directive creates/changes to a **DATA** typed section called *name*.

The **USECT** directive creates/changes to a **BSS** typed section called *name*.

The **CSECT** directive creates/changes to a **CONST** typed section called *name*.

The **ASECT** directive creates/changes to an **ABS** typed section called *name*.

IAR Style Syntax

```
ASEG [start [(alignment)] ]
RSEG name [:type][(alignment)]
```

The **ASEG** directive creates an **ABSOLUTE** typed section, sets the location counter to the optional *start* value and aligns the section at the optional *alignment*. The start value is an assemble time expression. The alignment value is an assemble time expression that is the power of 2 upon which to align the section - so an alignment value of 1 will cause the section to be aligned on even byte locations.

The **RSEG** directive creates a named section called *name* with an optional type and aligns the section at the optional *alignment*. The section can be one of **CODE**, **DATA**, **BSS**, **CONST**, **ABS** or **UNTYPED**. The alignment value is specified the same as per **ASEG**.

Syntax

```
ORG [expr]
```

The **ORG** directive sets the current location counter to the assemble time expression *expr*. This expression is relative to the currently defined section. If no section has been defined then the **ABSOLUTE** typed section **.abs** is defined making the expression an absolute one.

Syntax

```
SEGEND
```

```
.BREAK  
.KEEP  
.INIT " name "
```

The **SEGEND** and **.BREAK** directives start a new fragment within the current section. A fragment is set of instructions that the linker will elect to include in it's output if a reference is made to one of the instructions in the fragment. If no reference is made to a fragment then the linker will not include that fragment in the output.

The **.KEEP** directive instructs the linker not to throw away the current section. Without this directive if a section is not referenced then the linker will not include it in the output.

The **.INIT** directive places a copy of the section denoted by *name* into the current section. This directive can be used to enable initialised data sections to be copied from read only memory into writable memory.

Symbolic constants and equates

You can define a symbolic name for a constant using the **EQU**, **SET** and **DEFINE** directives. You can also remove the definition of a symbol with the **UNDEF** directive and test if a symbol is defined using the **DEFINED** operator. The **DEFINE** and **UNDEF** directives may be prefixed with a # character which can be in the first column to emulate simple C pre-processor statements. The **SFRB** and **SFRW** directives may start in the first column and are typically used to access memory mapped peripheral registers.

Syntax

symbol **EQU** *expression*

symbol = *expression*

symbol **SET** *expression*

DEFINE *symbol* = *expression*

UNDEF *symbol*

SFRB *symbol* = *addressExpression*

SFRW *symbol* = *addressExpression*

The assembler evaluates the expression and assigns that value to the symbol. For the EQU directive the expression need not be constant or even known at assembly time; it can be any value and may include complex operations involving external symbols. For the SET and DEFINE directive the expression must evaluate to an assemble time constant. The SET directive allows redefinition of an existing symbol, the EQU and DEFINE directives will produce an error if a symbol is redefined. The = operator is equivalent to the SET directive.

Example

```
CR EQU 13
```

This defines the symbol CR to be a symbolic name for the value 13. Now you can use the symbol CR in expressions rather than the constant.

Example

```
ADDRHI EQU (ADDR<<8) & 0FFH
```

This defines the symbol ADDRHI to be equivalent to the value of ADDR shifted right 8 bits and then bitwise-anded with 255. If ADDR is an external symbol defined in another module, then the expression involving ADDR cannot be resolved at assembly time as the value of ADDR isn't known to the assembler. The value of ADDR is only known when linking and the linker will resolve any expression the assembler can't.

Example

```
x = x-1
```

This redefines the symbol x to be the current value of x with 1 subtracted.

Example

```
#if !defined(debug)
```

```
#define debug  
#endif
```

This defines the symbol `debug` if it hasn't already been defined.

Labels

You use labels to give symbolic names to addresses of instructions or data. The most common form of label is a code label where code labels as operands of call, branch, and jump instructions to transfer program control to a new instruction. Another common form of label is a data label which labels a data storage area.

Syntax

label [:] [*directive* | *instruction*]

The label field starts at the left of the line, with no preceding spaces. The colon after the label is optional, but if present the assembler immediately defines the label as a code or data label. Some directives, such as EQU, require that you do not place a colon after the label.

Example

```
ExitPt: RET
```

This defines ExitPt as a code label which labels the RET instruction. You can branch to the RET instruction by using the label ExitPt in an instruction:

```
BRA ExitPt
```

CODE directive

Syntax

CODE

TEXT

Description

The **CODE** and **TEXT** directives select the default code section. The default code section is named **CODE** unless it has been renamed by the **-Rc** command line option.

CONST directive

Syntax

CONST
RODATA

Description

The **CONST** and **RODATA** directives select the default read-only constant section. The default read-only constant section is named **CONST** unless it has been renamed by the **-Rk** command line option.

DATA directive

Syntax

DATA

Description

The **DATA** directive selects the default data section. The default data section is named ***IDATA0*** unless it has been renamed by the **-Rd** command line option.

VECTORS directive

Syntax

VECTORS

Description

The **VECTORS** directive selects the default interrupt vector section. The default interrupt vector section is named **INTVEC** unless it has been renamed by the **-Rv** command line option.

ZDATA directive

Syntax

ZDATA

BSS

Description

The **ZDATA** and **BSS** directives selects the default zeroed data section. The default zeroed data section is named **UDATA0** unless it has been renamed by the **-Rz** command line option.

KEEP directive

Syntax

KEEP

ROOT

Description

The **KEEP** and **ROOT** directives instruct the linker that this is a root fragment and **must not be discarded when constructing the output file**. Normally only the startup code and vector sections use this facility.

Data types

Unlike many assemblers, the CrossWorks orizon assembler fully understands data types. The most well-known and widely used assembler that uses data typing extensively is Microsoft's MASM and its many clones. So, if you've used MASM before you should be pretty well at home with the concept of data types in an assembler and also with the CrossWorks implementation of data typing.

If you haven't used MASM before you may well wonder why data typing should ever be put into an assembler, given that many assembly programs are written without the help of data types at all. But there are many good reasons to do so, even without the precedent set by Microsoft, and the two most valuable benefits are:

- The ability to catch potential or real errors at assembly time rather than letting them through the assembler to go undetected until applications are deployed in the field.
- Data typing is an additional and effective source of program documentation, describing the way data are grouped and represented.

We don't expect you to fully appreciate the usefulness of assembly-level data typing until you've used it in an application and had first-hand experience of both benefits above. Of course, it's still possible to write (almost) typeless assembly code using the CrossWorks assembler if you should wish to do so, but effective use of data typing is a real programmer aid when writing code. Lastly, we should just mention one other important benefit that data typing brings and that is the interaction between properly-typed assembly code and the debugger. If you correctly type your data, the debugger will present the values held in memory using a format based on the type of the object rather than as a string of hexadecimal bytes. Having source-level debugging information displayed in a human-readable format is surely a way to improve productivity.

Built-in types

The CrossWorks assembler provides a number of built-in or pre-defined data types. These data types correspond to those you'd find in a high-level language such as C. You can use these to allocate data storage; for instance the following allocates one byte of data for the symbol ***count***:

```
count    DV      BYTE
```

The directive **DV** allocates one byte of space for ***count*** in the current section and sets ***count***'s type to **BYTE**.

Type name	Size in bytes	Description
BYTE	1	Unsigned 8-bit byte
WORD	processor-dependent	Unsigned word, dependent upon processor word size.
LONG	4	Unsigned 32-bit word
CHAR	1	8-bit character
ADDR	processor-dependent	Address

Structure and union types

Using the **STRUC**, **UNION**, and **FIELD** directives you can define data items which are grouped together. Such a group is called a structure and can be thought of in the same way as a structure or union in C. Structured types are bracketed between **STRUC** and **ENDSTRUC** and should contain only **FIELD** directives; similarly, unions are bracketed between **UNION** and **ENDUNION** and should only contain **FIELD** directives.

Example

We could declare a structure type called **Amount** which has two members, **Pounds** and **Pence** like this:

```
Amount STRUC
Pounds FIELD LONG
Pence  FIELD BYTE
      ENDSTRUC
```

The field **Pounds** is declared to be of type **LONG** and Pennies is of type **BYTE** (we can count lots of Pounds, and a small amount of loose change).

In structures, fields are allocated one after another, increasing the size of the structure for each field added. For a union, all fields are overlaid, and the size of the union is the size of the largest field within the union.

Example

For a 32-bit big-endian machine, we could overlay four bytes over a 32-bit word like this::

```
Word     UNION
asWord   FIELD WORD
asBytes  FIELD BYTE[4]
      ENDUNION
```

The most useful thing about user-defined structures is that they act like any builtin data type, so you can allocate space for variables of structure type:

```
Balance DV      Amount
```

Here we've declared enough storage for the variable **Balance** to hold an **Amount**, and the assembler also knows that **Balance** is of type **Amount**. Because the assembler knows what type **Balance** is and how big an **Amount** is, and because the assembler tracks type information, you can specify which members of **Balance** to operate on. Assuming that the instruction **LDB** loads a byte value and **LDW** loads a word value, then

```
LDB      Balance.Pence
```

will load a single byte which is the **Pence** part of **Balance**. We could equally well have written:

```
LDW      Balance.Pounds
```

which loads the **Pounds** part of **Balance**.

Array types

You can declare arrays of any predefined or user-defined type. Arrays are used extensively in high-level languages, and therefore we decided they should be available in the CrossWorks assembler to make integration with C easier.

An array type is constructed by specifying the number of array elements in brackets after the data type.

Syntax

type [*array-size*]

This declares an array of *array-size* elements each of data type *type*. The array size must be an absolute constant known at assembly time.

Example

The type

BYTE[8]

declares an array of eight bytes.

Pointer types

You can declare pointers to types just like you can in most high-level languages.

Syntax

type **PTR**

This declares a pointer to the data type *type*.

Example

The type

CHAR PTR

declares a pointer to a character. The built-in type **ADDR** is identical to the type **BYTE PTR**.

Combining data types

Arrays, combined with structures, can make complex data structuring simple:

```
BankAccount  STRUC
HolderName   FIELD  CHAR[32]
HolderAddr   FIELD  CHAR[32]
Balance      FIELD  Amount
ODLimit      FIELD  Amount
            ENDSTRUC
```

You can select individual elements from an array by specifying the index to be used in brackets:

```
LDB MyAccount.HolderName[0]
```

The assembler defines arrays as zero-based, so the fragment above loads the first character of MyAccount's HolderName. Because the assembler must know the address to load at assembly time, the expression within the square brackets must evaluate to a constant at assembly time. For example, the following is invalid because Index isn't an assembly-time constant:

```
Index    DV      BYTE
         LDB      MyAccount.HolderName[Index]
```

However, if Index were defined symbolically, the assembler can compute the correct address to encode in the instruction at assembly time:

```
Index    EQU      20
         LDB      MyAccount.HolderName[Index]
```

Expressions and operators

The assembler can manipulate constants and relocatable values at assembly time. If the assembler cannot resolve these to a constant value (for example, an expression involving the value of an external symbol cannot be resolved at assembly time), the expression is passed onto the linker to resolve.

Each operator has a precedence, and the following table lists the precedence of the operators from highest to lowest:

Operator	Group
<code>[] . ::</code>	Postfix operators
<code>DEFINED HBYTE LBYTE HWORD LWORD SIZEOF STARTOF ENDOF SFB SFE NOT ! LNOT !! THIS \$</code>	Monadic prefix operators
<code>* / %</code>	Multiplicative operators
<code>+ -</code>	Additive operators
<code>SHL SHR ASHR << >></code>	Shifting operators
<code>LT GT LE GE < > <= >=</code>	Relational operators
<code>EQ NE == !=</code>	Equality operators
<code>AND &</code>	Bit-wise and
<code>XOR ^</code>	Bit-wise exclusive or
<code>OR </code>	Bit-wise inclusive or
<code>LAND &&</code>	Logical and
<code>LOR </code>	Logical or

Integer constants

Integer constants represent integer values and can be represented in binary, octal, decimal, or hexadecimal. You can specify the radix for the integer constant by adding a radix specified as a suffix to the number. If no radix specifier is given the constant is decimal.

Syntax

decimal-digit digit... [B | O | Q | D | H]

The radix suffix **B** denotes binary, **O** and **Q** denote octal, **D** denotes decimal, and **H** denotes hexadecimal.

Radix suffixes can be given either in lower-case or uppercase letters. Hexadecimal constants must always start with a decimal digit (0 to 9). You must do this otherwise the assembler will mistake the constant for a symbol—for example, **0FCH** is interpreted as a hexadecimal constant but **FCH** is interpreted as a symbol.

Examples

224
224D
17Q
100H

You can specify hexadecimal constants in two other formats which are popular with many assemblers:

Syntax

0x *digit digit...*
\$ *digit digit...*

The **0x** notation is exactly the same as the way hexadecimal constants are written in C, and the **\$** notation is common in many assemblers for Motorola parts.

Examples

0xC0
\$F

String constants

A string constant consists of one or more ASCII characters enclosed in single or double quotation marks.

Syntax

`"character..."`

You can specify non-printable characters in string constants using escape sequences. An escape sequence is introduced by the backslash character `'\'`.

The following escape sequences are supported:

Sequence	Description
<code>\ooo</code>	Octal code of character where <i>o</i> is an octal digit
<code>\"</code>	Double quotation mark
<code>\'</code>	Single quotation mark
<code>\\</code>	Backslash
<code>\b</code>	Backspace, ASCII code 8
<code>\f</code>	Form feed, ASCII code 12
<code>\n</code>	New line, ASCII code 10
<code>\r</code>	Carriage return, ASCII code 13
<code>\v</code>	Vertical tab, ASCII code 11
<code>\xhh</code>	Hexadecimal code of character where <i>h</i> is a hexadecimal digit

Examples

```
"This is a string constant"  
"A string constant with a new line at the end\n"
```

Arithmetic operators

The arithmetic operators perform standard arithmetical operations on signed values.

Operator	Syntax	Description
+	<i>expression</i> ₁ + <i>expression</i> ₂	Add <i>expression</i> ₁ to <i>expression</i> ₂ . See + operator .
-	<i>expression</i> ₁ - <i>expression</i> ₂	Subtract <i>expression</i> ₂ from <i>expression</i> ₁ . See - operator .
*	<i>expression</i> ₁ * <i>expression</i> ₂	Multiply <i>expression</i> ₁ by <i>expression</i> ₂ . See * operator .
/	<i>expression</i> ₁ / <i>expression</i> ₂	The integer quotient of <i>expression</i> ₁ divided by <i>expression</i> ₂ . See / operator .
%	<i>expression</i> ₁ % <i>expression</i> ₂	The integer remainder of <i>expression</i> ₁ divided by <i>expression</i> ₂ . See % operator .
SHL <<	<i>expression</i> ₁ SHL <i>expression</i> ₂ <i>expression</i> ₁ << <i>expression</i> ₂	Shift <i>expression</i> ₁ left by <i>expression</i> ₂ bits. See SHL and << operators .
SHR >>	<i>expression</i> ₁ SHR <i>expression</i> ₂ <i>expression</i> ₁ >> <i>expression</i> ₂	Shift <i>expression</i> ₁ right by <i>expression</i> ₂ bits with zero fill. (This is commonly called a logical shift). See SHR and >> operators .
ASHR	<i>expression</i> ₁ ASHR <i>expression</i> ₂	Shift <i>expression</i> ₁ right by <i>expression</i> ₂ bits with sign fill. (This is commonly called an arithmetic shift). See ASHR operator .

+ operator

Syntax

*expression*₁ + *expression*₂

Description

Add *expression*₁ to *expression*₂ modulo 2^{32} .

Examples

1 + 2 ; evaluates to 3

- operator

Syntax

*expression*₁ - *expression*₂

Description

Subtracts *expression*₂ from *expression*₁ modulo 2³².

Examples

1 - 5 ; evaluates to -4

* operator

Syntax

*expression*₁ * *expression*₂

Description

Multiplies *expression*₁ by *expression*₂ modulo 2^{32} .

Examples

`7 * 5` ; evaluates to 35

/ operator

Syntax

*expression*₁ / *expression*₂

Description

Divides *expression*₁ by *expression*₂ producing an integer quotient. If *expression*₂ is zero, the quotient is zero.

Examples

`7 / 5` ; evaluates to 1

% operator

Syntax

*expression*₁ % *expression*₂

Description

Produces the remainder after division of *expression*₁ by *expression*₂. If *expression*₂ is zero, the remainder is zero.

Examples

7 % 5 ; evaluates to 2

SHL and << operators

Syntax

*expression*₁ **SHL** *expression*₂

*expression*₁ << *expression*₂

Description

Shifts *expression*₁ left by *expression*₂ bits, modulo 2³².

Examples

1 << 7 ; evaluates to 128

SHR and >> operators

Syntax

*expression*₁ **SHR** *expression*₂

*expression*₁ >> *expression*₂

Description

Logically shifts *expression*₁ right by *expression*₂ bits.

Examples

128 >> 6 ; *evaluates to 2*

ASHR operator

Syntax

*expression*₁ ASHR *expression*₂

Description

Arithmetically shifts *expression*₁ right by *expression*₂ bits.

Examples

-10 >> 6 ; evaluates to -1

Logical operators

The logical operators work on truth (Boolean) values and deliver well-formed boolean values. A non-zero value is considered true, and a zero value is considered false. All logical operators return well-formed truth values of either zero or one.

Operator	Syntax	Description
LNOT .LNOT. !!	<i>LNOT expression</i> <i>.LNOT. expression</i> <i>!! expression</i>	Logical negation. See LNOT and !! operators .
LAND .LAND. &&	<i>expression₁ LAND expression₂</i> <i>expression₁ .LAND. expression₂</i> <i>expression₁ && expression₂</i>	Logical conjunction. See LAND and && operators .
LOR .LOR. 	<i>expression₁ LOR expression₂</i> <i>expression₁ .LOR. expression₂</i> <i>expression₁ expression₂</i>	Logical disjunction. See LOR and operators .

LAND and && operators

Syntax

*expression*₁ **LAND** *expression*₂

*expression*₁ **.LAND.** *expression*₂

*expression*₁ **&&** *expression*₂

Description

True if both *expression*₁ and *expression*₂ are true.

Example

```
1 LAND 0 ; evaluates to false (0)
```

LNOT and ! operators

Syntax

`LNOT expression`

`.LNOT. expression`

`!expression`

Description

True if *expression* is false, and false if *expression* is true.

Example

`LNOT 3` ; evaluates to zero, false

LOR and || operators

Syntax

*expression*₁ LOR *expression*₂

*expression*₁ .LOR. *expression*₂

*expression*₁ || *expression*₂

Description

True if either *expression*₁ or *expression*₂ is true.

Examples

1 || 2 ; evaluates to true (1)

Bitwise operators

Bitwise operators perform logical operations on each bit of an expression. Don't confuse these operators with processor instructions having the same names--these operators are used on expressions at assembly time or link time, not at run time.

Operator	Syntax	Description
NOT .NOT. ~	NOT <i>expression</i> .NOT. <i>expression</i> ~ <i>expression</i>	Bit-wise complement. See NOT and ~ operators .
AND .AND. &	<i>expression</i> ₁ AND <i>expression</i> ₂ <i>expression</i> ₁ .AND. <i>expression</i> ₂ <i>expression</i> ₁ & <i>expression</i> ₂	Bit-wise and. See AND and & operators .
OR .OR.	<i>expression</i> ₁ OR <i>expression</i> ₂ <i>expression</i> ₁ .OR. <i>expression</i> ₂ <i>expression</i> ₁ <i>expression</i> ₂	Bit-wise or. See OR and operators .
XOR .XOR. ^	<i>expression</i> ₁ XOR <i>expression</i> ₂ <i>expression</i> ₁ .XOR. <i>expression</i> ₂ <i>expression</i> ₁ ^ <i>expression</i> ₂	Bit-wise exclusive or. See XOR and ^ operators .

AND and & operators

Syntax

*expression*₁ **AND** *expression*₂

*expression*₁ **.AND.** *expression*₂

*expression*₁ **&** *expression*₂

Description

Produces the bit-wise conjunction (and) of *expression*₁ and *expression*₂.

Examples

0AAH AND 0F0H ; evaluates to A0

NOT and ~ operators

Syntax

NOT *expression*

.NOT. *expression*

~ *expression*

Description

Produces the bit-wise (one's) complement of *expression*.

Examples

NOT 0FH ; evaluates to FFFFFFF0

OR and | operators

Syntax

*expression*₁ **OR** *expression*₂

*expression*₁ **.OR.** *expression*₂

*expression*₁ **|** *expression*₂

Description

Produces the bit-wise disjunction (or) of *expression*₁ and *expression*₂.

Examples

0AAH | 0F0H ; evaluates to FA

XOR and ^ operators

Syntax

*expression*₁ **XOR** *expression*₂

*expression*₁ **.XOR.** *expression*₂

*expression*₁ **^** *expression*₂

Description

Produces the bit-wise exclusive or of *expression*₁ and *expression*₂.

Examples

0AAH .XOR. 0FFH ; evaluates to 55

Relational operators

Relational operators compare two expressions and return a true value if the condition specified by the operator is satisfied. The relational operators use the value one (1) to indicate that the condition is true and zero to indicate that it is false.

Syntax

The following table shows the relational operators syntax and their meanings.

Operator	Syntax	Description
EQ .EQ. ==	<i>expression</i> ₁ EQ <i>expression</i> ₂ <i>expression</i> ₁ .EQ. <i>expression</i> ₂ <i>expression</i> ₁ == <i>expression</i> ₂	True if expressions are equal.
NE .NE. !=	<i>expression</i> ₁ NE <i>expression</i> ₂ <i>expression</i> ₁ .NE. <i>expression</i> ₂ <i>expression</i> ₁ != <i>expression</i> ₂	True if expressions are not equal.
LT .LT. <	<i>expression</i> ₁ LT <i>expression</i> ₂ <i>expression</i> ₁ .LT. <i>expression</i> ₂ <i>expression</i> ₁ < <i>expression</i> ₂	True if <i>expression</i> ₁ is less than <i>expression</i> ₂ .
LE .LE. <=	<i>expression</i> ₁ LE <i>expression</i> ₂ <i>expression</i> ₁ .LE. <i>expression</i> ₂ <i>expression</i> ₁ <= <i>expression</i> ₂	True if <i>expression</i> ₁ is less than or equal to <i>expression</i> ₂ .
GT .GT. >	<i>expression</i> ₁ GT <i>expression</i> ₂ <i>expression</i> ₁ .GT. <i>expression</i> ₂ <i>expression</i> ₁ > <i>expression</i> ₂	True if <i>expression</i> ₁ is greater than <i>expression</i> ₂ .
GE .GE. >=	<i>expression</i> ₁ GE <i>expression</i> ₂ <i>expression</i> ₁ .GE. <i>expression</i> ₂ <i>expression</i> ₁ >= <i>expression</i> ₂	True if <i>expression</i> ₁ is greater than or equal to <i>expression</i> ₂ .

Examples

```

1 EQ 2          ; evaluates to false (0)
1 .NE. 2        ; evaluates to true (1)
1 LT 2          ; evaluates to true
1 > 2           ; evaluates to false
1 .LE. 2        ; evaluates to true
1 >= 2          ; evaluates to false

```

Value extraction operators

The value extraction operators extract part of a value from an expression.

Operator	Syntax	Description
HIGH HBYTE	HIGH <i>expression</i> .HIGH. <i>expression</i> HBYTE <i>expression</i>	Extract the high order byte of the 16-bit value <i>expression</i> . See HIGH and HBYTE operators .
HWORD	HWORD <i>expression</i>	Extract the high order 16 bits of <i>expression</i> . See HWORD operator .
LOW LBYTE	LOW <i>expression</i> .LOW. <i>expression</i> LBYTE <i>expression</i>	Extract the low order byte of <i>expression</i> . See LOW and LBYTE operators .
LWORD	LWORD <i>expression</i>	Extract the low order 16 bits of <i>expression</i> . See LWORD operator .

HIGH and HBYTE operators

Syntax

HIGH expression

HBYTE expression

Description

Extract the high order byte of the 16-bit value *expression*.

Examples

```
HIGH $FEDCBA98      ; evaluates to $BA
```

You can combine **HBYTE** with **HWORD** to extract other parts of an expression:

```
HBYTE HWORD $FEDCBA98  ; evaluates to $FE
```

HWORD operator

Syntax

HWORD *expression*

Description

Extract the high order 16 bits of *expression*.

Example

HWORD \$FEDCBA98 ; evaluates to \$FEDC

LOW and LBYTE operators

Syntax

LOW *expression*

LBYTE *expression*

Description

Extract the low order byte of *expression*.

Example

```
LOW    $FEDCBA98      ; evaluates to $98
```

You can combine **LBYTE** with **HWORD** to extract other parts of an expression:

```
LBYTE HWORD $FEDCBA98  ; evaluates to $DC
```

LWORD operator

Syntax

LWORD *expression*

Description

Extract the low order 16 bits of *expression*.

Example

LWORD \$FEDCBA98 ; *evaluates to \$BA98*

THIS operator

Syntax

THIS

\$

The **THIS** operator returns an expression which denotes the location counter at the start of the source line.

Important notes

The location counter returned by **THIS** does not change even if code is emitted.

Example

A typical use of **THIS** is to compute the size of a string or block of memory:

```
MyString      DB  "Why would you count the number of characters"
               DB  "in a string when the assembler can do it?"
MyStringLen   EQU  THIS-MyString
```

DEFINED operator

Syntax

DEFINED *symbol*

The **DEFINED** operator returns a Boolean result which is true if the symbol is defined at that point in the file, and false otherwise. Note that this operator only inquires whether the symbol is known to the assembler, not whether it has a known value: imported symbols are considered as defined even though the assembler does not know their value.

DEFINED cannot be used to detect whether a macro has been defined.

Example

The following show how defined works in a number of cases.

```
.IMPORT X
Y      EQU      10
B1     EQU      DEFINED X    ; true (1)
B2     EQU      DEFINED Y    ; true (1)
B3     EQU      DEFINED Z    ; false (0)  not defined yet
B4     EQU      DEFINED U    ; false (0)  never defined
Z      EQU      100
```

SIZEOF operator

Syntax

SIZEOF expression

The **SIZEOF** operator returns an integer value which is the size of the type associated with the expression. The assembler reports an error if the expression has no type.

Example

```
X      VAR      LONG[100]
XSIZE  EQU      SIZEOF X      ; 400, 100 four-byte elements
X0SIZE EQU      SIZEOF X[0]    ; 4, size of LONG
```

Indexing operator

Syntax

*expression*₁ [*expression*₂]

Description

The index operator indicates addition with a scale factor. It is similar to the addition operator. *expression*₁ can be any expression which has array type. *expression*₂ must be a constant expression. The assembler multiplies *expression*₂ by the size of the array element type and adds it to *expression*₁.

Example

```
ARR    DV    LONG[4]    ; an array of four 32-bit values
W3     EQU   ARR[3]     ; set W4 to the address ARR + 3*(SIZE LONG)
                        ; which is ARR+12
```


Retyping operator

The retype operator `::` allows you to override the data type of an operand, providing the operand with a new type.

Syntax

expression :: type

The expression is evaluated and given the type, replacing whatever type (if any) the expression had.

Example

```
wordvar    DW    2
           LDB    wordvar::BYTE
```

In this example, **wordvar** has the type **WORD** because it is defined using **DW**. The load, however, loads only a single byte because **wordvar** is retyped as a **BYTE**. Because retyping does not alter the value of the expression, it only alters its type, the load will read from the lowest address of **wordvar**.

Compilation units and libraries

When you partition a application into separate compilation units you will need to indicate how a symbol defined in one unit is referenced in other units. This section will show you how to declare symbols exported or imported so they can be used in more than one unit.

When building applications, you often find pieces of code which can be reused in other applications. Rather than duplicating source code, you can package these units together into a *library* which can be reused in different applications.

The CrossWorks tools were designed to be flexible and let you to easily write space-efficient programs using libraries and separate compilation. To that end, the assembler and linker combination provides a number of features which are not found in many compilation systems.

- **Optimum-sized branches** The linker automatically resizes branches to labels where the label is too far away to be reached by a branch instruction. This is completely transparent to you as a programmer, when you use branch instructions your linked program will always use the smallest possible branch instruction. This capability is deferred to the linker so that branches across compilation units are still optimised.
- **Removing dead code and data** The most important features of the linker are its ability to leave all unreferenced code and data out of the final application. The linker automatically discards all code and data fragments in a program that are not reachable from any entry symbols.
- **Whole program optimization** The linker can optimize the application as a whole, rather than on a per-function or per-compilation-unit basis.

INCLUDE directive

Syntax

INCLUDE "filename "

INCLUDE < filename >

Description

The **INCLUDE** directive inserts the contents of the source file **filename** into the assembly. If **filename** is enclosed in quotation marks, the user include directories are searched, and if **filename** is enclosed in triangular brackets the system include directories are searched.

See Also

[Set user include directories](#), [Set system include directories](#)

Exporting symbols

Only symbols exported from a compilation unit can be used by other units. You can export symbols using the **EXPORT** directive. This directive does nothing more than make the symbol visible to other modules: it does not reserve storage for it nor define its type.

Syntax

EXPORT *symbol* [, *symbol*]...

PUBLIC *symbol* [, *symbol*]...

XDEF *symbol* [, *symbol*]...

Description

EXPORT, **PUBLIC**, and **XDEF** are equivalent and are provided in many assemblers: you can use whichever you prefer.

Not all symbols can be exported. Variables, labels, function blocks, and numeric constants defined using **EQU** can be exported, but macro names and local stack-based variables cannot.

The assembler publishes the symbol in the object file so that other modules can access it. If you don't export a symbol you can only use it in the source file it's declared in.

As a convenience, a label can be defined and exported at the same time using double-colon notation.

Example

```
data_ptr::
```

This declares the label **data_ptr** and exports it. This is equivalent to:

```
EXPORT data_ptr
data_ptr:
```

Importing symbols

When you need to use symbols defined in other modules you must import them first. You import symbols using the `.IMPORT` directive.

Syntax

IMPORT *symbol* [*:: type*] [, *symbol* [*:: type*]]...

EXTERN *symbol* [*:: type*] [, *symbol* [*:: type*]]...

XREF *symbol* [*:: type*] [, *symbol* [*:: type*]]...

When importing a symbol you can also define its type. This type information is used by the assembler whenever you reference the imported symbol and acts just like a symbol declared locally within the module. If you don't define a type for the imported variable, no type information is available to the assembler. If you subsequently use such a variable where type information is required, the assembler will report an error.

Example

```
IMPORT CLA::BYTE, La::WORD
IMPORT APDUData::BYTE[256]
IMPORT _myVar
```

The above imports **CLA** as a byte, **La** as a 32-bit word, **APDUData** as an array of 256 bytes, and **_myVar** without type information.

Macros, conditions, and loops

Conditional assembly allows you to control which code gets assembled as part of your application, allowing you to produce variants. Macros and loops automate repetitive tasks, such as constructing tables or duplicating code.

Conditional assembly

Syntax

```

IF expression
    statements
{ ELIF expression
    statements }
[ ELSE
    statements ]
ENDIF

IFDEF symbol
    ....
IFNDEF symbol
    ....

```

The controlling expression must be an absolute assembly-time constant. When the expression is non-zero the true conditional arm is assembled; when the expression is zero the false conditional body, if any, is assembled.

The IFDEF and IFNDEF directives are specialised forms of the IF directive. The IFDEF directive tests the existence of the supplied symbol and the IFNDEF directive tests the non-existence of the supplied symbol.

Example

```

IF type == 1
    CALL type1
ELSE
    IF type == 2
        CALL type2
    ELSE
        CALL type3
    ENDIF
ENDIF

```

The nested conditional can be replaced using the **ELIF** directive which acts like **ELSE IF**:

```

IF type == 1
    CALL type1
ELIF type == 2
    CALL type2
ELSE
    CALL type3
ENDIF

```

Example

Usual practice is to use a symbol, **_DEBUG**, as a flag to either include or exclude debugging code. Now you can use **IFDEF** to conditionally assemble some parts of your application depending upon whether the **_DEBUG** symbol is defined or not.

```

IFDEF _DEBUG
    CALL DumpAppState
ENDIF

```

Macros

The structure of a macro definition consists of a name, some optional arguments, the body of the macro and a termination keyword. The syntax you use to define a macro is:

Syntax

```
name MACRO arg1 , arg2 , ... , argn  
    macro-body  
ENDMACRO | ENDM
```

The name of the macro has the same requirements as a label name (in particular it must start in column one). The arguments are a comma-separated list of identifiers. The body of the macro can have arbitrary assembly language text including other macro definitions and invocations, conditional and file inclusion directives. A macro is instantiated by using its name together with optional actual argument values. A macro instantiation has to occur on its own line, it cannot be used within an expression or as an argument to an assembly code mnemonic or directive. The syntax you use to invoke a macro is

Syntax

```
name actual1 , actual2 , ... , actualn // comment
```

When a macro is instantiated the macro body is inserted into the assembly text with the actual values replacing the arguments that were in the body of the macro definition.

Labels in macros

When labels are used in macros they must be unique for each instantiation to avoid duplicate label definition errors. The assembler provides a label generation mechanism for situations where the label name isn't significant and a mechanism for constructing specific label names.

If a macro definition contains a jump to other instructions in the macro definition it is likely that the actual name of the label isn't important. To facilitate this a label of the form `name?` can be used.

There are situations when a macro invocation should result in the definition of a label. In the simplest case the label can be passed as an argument to the macro, however there are cases when the label name should be constructed from other tokens. The macro definition facility provides two constructs to enable this:

- tokens can be concatenated by putting **##** between them;
- the value of a constant symbol can be used by prefixing the label with **\$\$**.

Loops

If multiple definitions are required a loop structure can be used. This can be achieved either by a recursive macro definitions or by the use of the **LOOP** directive.

Example

```
P2TAB    MACRO    N
        IF      N
        P2TAB   N-1
        ENDF
        DW      1<<N
        ENDMACRO

POWERS:  POWER2TAB 10
```

This creates a table of ten powers of 2, that is 1, 2, 4, 8, and so on, up to 1024.

If the loop counter is a large number then a recursive macro may consume considerable machine resources. To avoid this you can use the **LOOP** directive, which is an iterative rather than recursive solution.

Syntax

LOOP *expression*

loop-body

ENDLOOP

The loop control expression must be a compile time constant. The loop body can contain any assembly text (including further loop constructs) except macro definitions (since it would result in multiple definitions of the same macro). The above recursive definition can be recast in an iterative style:

Example

```
POWERS:
x      SET      0
      LOOP     x <= 10
      DC.W     1<<x
x      SET      x+1
      ENDLOOP
```

Note that the label naming capabilities using **?**, **\$\$**, and **##** are not available within the body of a loop. If the loop body is to declare labels then a recursive macro definition should be used or a combination of using macro invocation to define the labels and the loops to define the text of the label.

Utilities Reference

This section describes the command line tools that CrossStudio uses. You don't have to use CrossStudio to build and manage your applications, you can do it outside of CrossStudio using a make utility which invokes the CrossWorks tools, such as the compiler and linker, individually.

In this section

Compiler driver reference

Describes the compiler driver which can compile, assemble, and link a program.

Linker reference

Describes the operation of the linker and how to place sections into memory.

Hex extractor reference

Describes the hex extractor tool that generates standard load file formats such as Intel Hex and Motorola S-Record.

Librarian reference

Describes the librarian and how to create and manage libraries.

CrossBuild Reference

Describes how to build your application from the command line using CrossBuild.

CrossLoad Reference

Describes downloading applications to your target using the stand-alone loader CrossLoad.

Related sections

Compiler reference

Describes the compiler and its command line options.

Assembler reference

Describes the operation of the assembler and its command line options.

Compiler driver reference

This section describes the switches accepted by the compiler driver, **hcl**. The compiler driver is capable of controlling compilation by all supported language compilers and the final link by the linker. It can also construct libraries automatically.

In contrast to many compilation and assembly language development systems, with you don't invoke the assembler or compiler directly. Instead you'll normally use the compiler driver **hcl** as it provides an easy way to get files compiled, assembled, and linked. This section will introduce you to using the compiler driver to convert your source files to object files, executables, or other formats.

We recommend that you use the compiler driver rather than use the assembler or compiler directly because there the driver can assemble multiple files using one command line and can invoke the linker for you too. There is no reason why you should not invoke the assembler or compiler directly yourself, but you'll find that typing in all the required options is quite tedious-and why do that when **hcl** will provide them for you automatically?

File naming conventions

The compiler driver uses file extensions to distinguish the language the source file is written in. The compiler driver recognises the extension **.c** as C source files, **.s** and **.asm** as assembly code files, and **.hzo** as object code 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.

Java language files

When the compiler driver finds a file with a **.java** extension, it runs a Java compiler followed by the java byte code translator to convert the class file to object code.

Assembly language files

When the compiler driver finds a file with a **.s** or **.asm** extension, it runs the assembler to convert it to object code.

Object code files

When the compiler driver finds a file with a **.hzo** extension, it passes it to the linker to include it in the final application.

-ansi (Warn about potential ANSI problems)

Syntax

-ansi

Description

Warn about potential problems that conflict with the relevant ANSI or ISO standard for the files that are compiled.

-ar (Archive output)

Syntax

-ar

Description

This switch instructs the compiler driver to archive all output files into a library. Using **-ar** implies **-c**.

Example

The following command compiles **file1.c**, **file2.asm**, and **file3.c** to object code and archives them into the library file **libfunc.hza**.

```
hcl file1.c file2.asm file3.c -o libfunc.hza
```

-c (Compile to object code, do not link)

Syntax

-c

Description

All named files are compiled to object code modules, but are not linked.

-D (Define macro symbol)

Syntax

-D name

-D name = value

Description

You can define preprocessor macros using the **-D** option. The macro definitions are passed on to the respective language compiler which is responsible for interpreting the definitions and providing them to the programmer within the language.

The first form above defines the macro **name** but without an associated replacement value, and the second defines the same macro with the replacement value **value**.

Setting this in CrossStudio

To define preprocessor macros for a project:

- Select the project in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **Preprocessor Definitions** property.

To define preprocessor macros for a particular file:

- Select the file in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **Preprocessor Definitions** property.

The **Preprocessor Definitions** property is a semicolon-separated list of macro definitions, for example "**name1=value1;name2=value2**". Clicking the button at the right of the property displays the **Preprocessor Definitions** dialog which will allow you to easily edit the definitions.

Example

The following defines two macros, **SUPPORT_FLOAT** with a value of 1 and **LITTLE_ENDIAN** with no replacement value.

```
-DSUPPORT_FLOAT=1 -DLITTLE_ENDIAN
```


-e (Override entry symbol)

Syntax

-e name

Description

This option allows you to override the default entry point selected by the linker. If no entry symbol is provided with **-e**, the linker searches for the symbols **start** and then **_main** in that order, and selects the first found.

-F (Set output format)

Syntax

-F format

Description

The **-F** option instructs the linker to write its output in the format **fmt**. The linker supports the following formats:

- **-Fsrec** Motorola S-record format
- **-Fhex** Intel extended hex format
- **-Ftek** Tektronix hex format
- **-Ttxt** Texas Instruments hex format
- **-Flst** Hexadecimal listing
- **-Fhzx** Rowley native format

The default format, if no other format is specified, is **-Fhzx**.

Setting this in CrossStudio

To set the output format for a project:

- Select the project in the **Project Explorer**.
- In the **Linker Options** group select the required format from the **Output Format** property.

-g (Generate debugging information)

Syntax

-g

Description

The **-g** option instructs the compiler and assembler to generate debugging information (line numbers and data type information) for the debugger to use.

Setting this in CrossStudio

To set include debugging information for all files in a project:

- Select the project in the **Project Explorer**.
- In the **General Options** group set the **Include Debug Information** property to **Yes** or **No** as appropriate.

To set include debugging information for a particular file (**not recommended**):

- Select the file in the **Project Explorer**.
- In the **General Options** group set the **Include Debug Information** property to **Yes** or **No** as appropriate.

-h (Display help information)

Syntax

-help

Description

Displays a short summary of the options accepted by the compiler driver.

-I (Define user include directories)

Syntax

-I directory

In order to find include files the compiler driver arranges for the compilers to search a number of standard directories. You can add directories to the search path using the **-I** switch which is passed on to each of the language processors. The current directory is automatically added to the user include path before any directory added by the **-I** command line option.

Setting this in CrossStudio

To set the directories searched for include files for a project:

- Select the project in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **User Include Directories** property.

To set the directories searched for include files for a particular file:

- Select the file in the **Project Explorer**.
- In the **Preprocessor Options** group edit the **User Include Directories** property.

Example

To tell the compiler to search the current directory (automatically provided by the compiler driver), **../include** and **../lib/include** for included files when compiling **file.c** use the following:

```
hcl -I../include -I../lib/include file.c
```

You can specify more than one include directory by separating each directory component with either a comma or semicolon, so the following command lines have the same effect as the one above.

```
hcl -I../include,../lib/include file.c  
hcl -I../include;../lib/include file.c
```

See Also

[Exclude Standard Include Directories \(-I-\)](#)

-J (Define system include directories)

Syntax

-J directory

The **-J** option adds **directory** to the end of the list of directories to search for source files included by the **#include** preprocessor command for C files and the **INCLUDE** assembler directive. The directory **\$(InstallDir)/include** is automatically added to the system include path before any directory added by the **-J** command line option.

Setting this in CrossStudio

For example, to tell the compiler to search the directories **../include** and **../lib/include** for included system files when compiling **file.c** and writing the output to **file.hzo** you could use the following:

```
hcc -J../include -J../lib/include file.c -o file.hzo file.c
```

-K (Keep linker symbol)

Syntax

-K name

Description

The CrossWorks linker removes unused code and data from the output file. This process is called **deadstripping**. To prevent the linker from deadstripping unreferenced code and data you wish to keep, you must use the **-K** command line option to force inclusion of symbols.

Example

If you have a C function, **contextSwitch** that must be kept in the output file (and which the linker will normally remove), you can force its inclusion using:

```
-K_contextSwitch
```

Because **-K** is passed to the linker as an option, you must prefix the C function or variable name with an underscore as the CrossWorks C compiler prefixes all external symbols with an underscore when constructing linker symbols.

-l (Link library)

Syntax

-l x

Description

Link the library '**lib x.hza**' from the library directory. The library directory is, by default ***\$(InstallDir)/lib***, but can be changed with the **-L** option.

-L (Set library directory path)

Syntax

-L dir

Description

Sets the library directory to **dir**. If **-L** is not specified on the command line, the default location to search for libraries is set to ***\$(InstallDir)/lib***.

-I- (Exclude standard include directories)

Syntax

-I-

Description

Usually the compiler and assembler search for include files in the standard include directory created when the product is installed. If for some reason you wish to exclude these system locations from being searched when compiling a file, the **-I-** option will do this for you.

Setting this in CrossStudio

To exclude all search directories for a project:

- Select the project in the **Project Explorer**.
- In the **Preprocessor Options** group set the **Ignore Includes** property to **Yes**.

To exclude all search directories for a particular file:

- Select the file in the **Project Explorer**.
- In the **Preprocessor Options** group set the **Ignore Includes** property to **Yes**.

Example

To instruct the compiler to search *only* the directories **../include** and **../lib/include** for included files when compiling **file.c**:

```
hcl -I- -I../include -I../lib/include file.c
```

Special Notes

The **-I-** option will clear any include directories previously set with the **-I** option, so you must ensure that **-I-** comes before setting any directories you wish to search. Therefore, the following command line has a different effect to the command line above:

```
hcl -I../include -I../lib/include -I- file.c
```

See Also

[Define Include Directories \(-I\)](#)

-I- (Do not link standard libraries)

Syntax

-I-

Description

The **-I-** option instructs the linker not to link standard libraries automatically included by the compiler or by the assembler **INCLUDELIB** directive. If you use this options you must supply your own library functions or provide the names of alternative sets of libraries to use.

Setting this in CrossStudio

To exclude standard libraries from the link:

- Select the project in the **Project Explorer**.
- In the **Linker Options** group set the **Include Standard Libraries** property to **No**.

-mmpy (Enable hardware multiplier)

Syntax

-mmpy

Description

This option instructs the compiler to generate code that can use the MSP430 hardware multiplier. By default, the hardware multiplier is not used and all integer and floating-point multiplications are carried out by software loops.

When using the hardware multiplier, the compiler ensures that no interrupts occur during the time the multiplier is in use. Global interrupts are disabled during a multiplication to prevent, for instance, an interrupt being taken immediately after the multiplication is complete but before the result has been loaded which could possibly corrupt the result of the multiplication. Because interrupts are disabled during hardware-assisted multiplication, interrupt latency is increased—if you wish to have the lowest possible interrupt latency, then do not enable the hardware multiplier and use soft multiplication instead.

The CrossWorks compiler generates inline code to use the hardware multiplier for 16-bit multiplications and calls out-of-line subroutines for all other multiplications. The runtime library also uses the hardware multiplier to accelerate multiplication of floating-point values.

Setting this in CrossStudio

To use the hardware multiplier for a project:

- Select the project in the **Project Explorer**.
- In the **Compiler Options** group set the **Use Hardware Multiplier** property to **Yes**.

It is not possible to set the **Use Hardware Multiplier** property on a per-file basis.

Special notes

There is no means to prevent a non-maskable interrupt from occurring, so you must be very careful not to use the hardware multiplier in any NMI interrupt service routines.

See also

[-mmpyinl \(Enable inline hardware multiplier\)](#)

-mmpyinl (Enable inline hardware multiplier)

Syntax

-mmpy

Description

This option instructs the compiler to generate inline code that can use the MSP430 hardware multiplier. By default, the hardware multiplier is not used and all integer and floating-point multiplications are carried out by software loops.

When using the hardware multiplier, the compiler ensures that no interrupts occur during the time the multiplier is in use. Global interrupts are disabled during a multiplication to prevent, for instance, an interrupt being taken immediately after the multiplication is complete but before the result has been loaded which could possibly corrupt the result of the multiplication. Because interrupts are disabled during hardware-assisted multiplication, interrupt latency is increased—if you wish to have the lowest possible interrupt latency, then do not enable the hardware multiplier and use soft multiplication instead.

The CrossWorks compiler generates inline code to use the hardware multiplier for 16-bit multiplications and calls out-of-line subroutines for all other multiplications. The runtime library also uses the hardware multiplier to accelerate multiplication of floating-point values.

Setting this in CrossStudio

To use the hardware multiplier for a project:

- Select the project in the **Project Explorer**.
- In the **Compiler Options** group set the **Use Hardware Multiplier** property to **Inline**.

It is not possible to set the **Use Hardware Multiplier** property on a per-file basis.

Special notes

There is no means to prevent a non-maskable interrupt from occurring, so you must be very careful not to use the hardware multiplier in any NMI interrupt service routines.

See also

[-mmpy \(Enable hardware multiplier\)](#)

-msd (Treat double as float)

Syntax

-msd

Description

This option directs the compiler to treat **double** as **float** and not to support 64-bit floating point arithmetic.

Setting this in CrossStudio

To treat **double** as **float** for a project:

- Select the project in the **Project Explorer**.
- In the **Compiler Options** group set the **Treat 'double' as 'float'** property to **Yes**.

It is not possible to set the **Treat 'double' as 'float'** property on a per-file basis

-M (Print linkage map)

Syntax

-M-

-M *filename*

Description

The **-M** option prints a linkage map to standard output; **-M file** prints a linkage map to *filename*.

-n (Dry run, no execution)

Syntax

-n

Description

When **-n** is specified, the compiler driver processes options as usual, but does not execute any subprocesses to compile, assemble, or link applications.

-o (Set output file name)

Syntax

-o filename

Description

The **-o** option instructs the compiler to write its object file to **filename**.

-O (Optimize output)

Syntax

-O x

Description

Pass the optimization option **-O x** to the compiler and linker. Specific **-O** optimization options are described in the compiler and linker reference section.

-Rc (Set default code section name)

Syntax

-Rc, name

Description

The **-Rc** command line option sets the name of the default code section that the compiler emits code into. If no other options are given, the default name for the section is **CODE**.

You can control the name of the code section used by the compiler within a source file using the [codeseg pragma](#) or by using CrossStudio to set the **Code Section Name** property of the file or project.

Setting this in CrossStudio

To set the default code section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Code Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Code Section Name** property.

Example

The following command line option instructs the compiler to use the name **RAMCODE** as the default code section name and to initially generate all code into that section.

-Rc, RAMCODE

-Rd (Set default initialised data section name)

Syntax

-Rd, name

Description

The **-Rd** command line option sets the name of the default data section that the compiler emits initialized data into. If no other options are given, the default name for the section is **IDATA0**.

You can control the name of the data section used by the compiler within a source file using the [dataseg pragma](#) or by using CrossStudio to set the **Data Section Name** property of the file or project.

Setting this in CrossStudio

To set the default data section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Data Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Data Section Name** property.

Example

The following command line option instructs the compiler to use the name **NVDATA** as the default initialised section name and to initially generate all initialised data into that section.

-Rd, NVDATA

-Rk (Set default read-only data section name)

Syntax

-Rk, name

Description

The **-Rk** command line option sets the name of the default data section that the compiler emits read-only data into. If no other options are given, the default name for the section is **CONST**.

You can control the name of the read-only data section used by the compiler within a source file using the [constseg pragma](#) or by using CrossStudio to set the **Constant Section Name** property of the file or project.

Setting this in CrossStudio

To set the default constant section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Constant Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Constant Section Name** property.

Example

The following command line option instructs the compiler to use the name **ROMDATA** as the default read-only data section name and to initially generate all read-only data into that section.

-Rk, ROMDATA

-Rv (Set default vector section name)

Syntax

-Rv, name

Description

The **-Rv** command line option sets the name of the default vector table section that the compiler emits interrupt vectors into. If no other options are given, the default name for the section is **INTVEC**.

You can control the name of the vector table section used by the compiler within a source file using the [vectorseg pragma](#) or by using CrossStudio to set the **Vector Section Name** property of the file or project.

Setting this in CrossStudio

To set the default interrupt vector section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Vector Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Vector Section Name** property.

Example

The following command line option instructs the compiler to use the name **IVDATA** as the default vector section name and to initially generate all interrupt vectors into that section.

-Rv, IVDATA

-Rz (Set default zeroed data section name)

Syntax

-Rz, name

Description

The **-Rz** command line option sets the name of the default zeroed data section that the compiler emits uninitialized data into. If no other options are given, the default name for the section is **UDATA0**. Uninitialised data in **UDATA0** is set to zero on program startup.

You can control the name of the zeroed data section used by the compiler within a source file using the [zeroedseg pragma](#) or by using CrossStudio to set the **Zeroed Section Name** property of the file or project.

Setting this in CrossStudio

To set the default zeroed section name for a project:

- Select the project in the **Project Explorer**.
- In the **Section Options** group edit the **Zeroed Section Name** property.

To set the default code section name for a particular file:

- Select the file in the **Project Explorer**.
- In the **Section Options** group edit the **Zeroed Section Name** property.

Example

The following command line option instructs the compiler to use the name **ZDATA** as the default zeroed data section name and to initially generate all uninitialised into that section.

-Rz, ZDATA

-s- (Exclude standard startup code)

Syntax

-s-

Description

C code requires a small startup file containing system initialization code to be executed before entering **main**. The standard startup code is found in the object file **\$(InstallDir)/lib/crt0.hzo** and the compiler driver automatically links this into your program. If, however, you do not require the standard startup code because you have a pure assembly language application, you can request the compiler driver to exclude this standard startup code from the link using the **-s-** option.

You will find the source code for the standard startup module crt0 in the file **\$(InstallDir)/src/crt0.asm**.

Setting this in CrossStudio

To exclude the standard startup code for a project:

- Select the project in the **Project Explorer**.
- In the **Linker Options** group set the **Include Startup Code** property to **No**.

Example

To instruct the compiler to assemble **file1.asm** and **file2.asm** into **app.hzx** and not link the standard startup code:

```
hcl file1.asm file2.asm -s- -o app.hzx
```


-s (Set startup code file)

Syntax

-s name

Description

C code requires a small startup file containing system initialization code to be executed before entering **main**. The standard startup code is found in the object file **\$(InstallDir)/lib/crt0.hzo** and the compiler driver automatically links this into your program. If, however, you have special requirements for system initialization, or have changed the default names of the data sections, you can customize the standard startup code and use this alternative code rather than the standard code.

You will find the source code for the standard startup module **crt0** in the file **\$(InstallDir)/src/crt0.asm**.

Setting this in CrossStudio

To provide customized startup code for your C application:

- Select the project in the **Project Explorer**.
- In the **Linker Options** group set the **Include Startup Code** property to **No**.
- Add the assembly source file containing the replacement startup code to the project with **Project | Add Existing Item**.

The startup code it is just like any other source file in your project and is rebuilt when out of date, for instance.

Examples

To instruct the compiler to assemble **file1.c** and link the replacement system startup code **\$(InstallDir)/lib/mystartup.hzo** (rather than the standard startup code) into **app.hzx**:

```
hcl -smystartup -o app.hzx file1.c
```

To instruct the compiler to assemble **file1.c** and link the replacement system startup code **mystartup.asm** in the current directory (rather than the standard startup code) into **app.hzx**:

```
hcl -s- -o app.hzx file1.c mystartup.asm
```

-v (Verbose execution)

Syntax

-v

Description

The compiler driver and other tools usually operate without displaying any information messages or banners, only diagnostics such as errors and warnings are displayed.

If the **-V** switch is given, the compiler driver displays its version and it passes the switch on to each compiler and the linker so that they display their respective versions.

The **-v** switch displays command lines executed by the compiler driver.

-V (Version information)

Syntax

-V

Description

The compiler driver and other tools usually operate without displaying any information messages or banners, only diagnostics such as errors and warnings are displayed.

If the **-V** switch is given, the compiler driver displays its version and it passes the switch on to each compiler and the linker so that they display their respective versions.

The **-v** switch displays command lines executed by the compiler driver.

-w (Suppress warnings)

Syntax

-w

Description

This option instructs the compiler, assembler, and linker not to issue any warnings.

Setting this in CrossStudio

To suppress warnings for a project:

- Select the project in the **Project Explorer**.
- In the **General Options** group set the **Suppress Warnings** property to **Yes**.

To suppress warnings for a particular file:

- Select the file in the **Project Explorer**.
- In the **General Options** group set the **Suppress Warnings** property to **Yes**.

-we (Treat warnings as errors)

Syntax

-we

Description

This option directs the compiler, assembler, and linker to treat all warnings as errors.

Setting this in CrossStudio

To suppress warnings for all files in a project:

- Select the project in the **Project Explorer**.
- In the **General Options** group set the **Treat Warnings as Errors** property to **Yes**.

To suppress warnings for a particular file:

- Select the file in the **Project Explorer**.
- In the **General Options** group set the **Treat Warnings as Errors** property to **Yes**.

-Wa (Pass option to assembler)

Syntax

-Wa, option

Description

The **-Wa** command line option passes **option** directly to the assembler.

Example

The following command line option passes **-V** directly to the assembler—the effect of this is to force the compiler to display its version information.

-Wa, -V

-Wc (Pass option to C compiler)

Syntax

-Wc, option

Description

The **-Wc** command line option passes **option** directly to the C compiler.

Example

The following command line option passes **-V** directly to the C compiler—the effect of this is to force the compiler to display its version information.

-Wc, -V

-Wl (Pass option to linker)

Syntax

-Wl, option

Description

The **-Wl** command line option passes **option** directly to the linker.

Example

The following command line option passes **-Dstack_size=0x100** directly to the linker—the effect of this is to define a linker symbol **stack_size** with the value 256.

```
-Wl, -Dstack_size=0x100
```


Linker reference

The linker **hld** is responsible for linking together the object files which make up your application together with some run-time startup code and any support libraries.

Although the compiler driver usually invokes the linker for you, we fully describe how the linker can be used stand-alone. If you're maintaining your project with a make-like program, you may wish to use this information to invoke the linker directly rather than using the compiler driver.

The linker performs the following functions:

- resolves references between object modules;
- extracts object modules from archives to resolve unsatisfied references;
- combines all fragments belonging to the same section into a contiguous region;
- removes all unreferenced code and data;
- runs an architecture-specific optimizer to improve the object code;
- fixes the size of span-dependent instructions;
- computes all relocatable values;
- produces a linked application and writes it in a number of formats.

Command line syntax

You invoke the linker using the following syntax:

hld [**option** | **file**]...

Files

file is either an object file or library file to include in the link and it must be in CrossWorks object or library format.

Options

option is a command-line option. Options are case sensitive and cannot be abbreviated. The linker supports the following command line options:

Option	Description
-A name [= val]	Define the attribute name and optionally set it to val
-D name [= number symbol]	Define the symbol name and set it equal to number or symbol
-g	Propagate debugging information into the linked image
-G sect = flags	Set sect 's section flags to flags
-L dir	Set the library directory to dir
-I-	Disable automatic inclusion of standard libraries
-I x	Search library x to resolve symbols
-M [- <i>file</i>]	Print linkage map to standard output or to <i>file</i>
-o file	Write output to file
-O [level]	Run optimizer at level level
-T section ,...= start [- end]	Locate sections between start and end
-v	Verbose mode
-V	Display version information
-w	Suppress warning messages
-we	Treat warnings as errors

-D (Define linker symbol)

Syntax

-D name =[symbol | number]

Description

This option instructs the linker to define the symbol **name** as either the value **number** or the low-level symbol **symbol**. You can specify **number** in either decimal or hexadecimal notation using a '**0x**' prefix.

Setting this in CrossStudio

To define the linker symbols for a project:

- Select the project in the **Project Explorer**.
- In the **Linker Options** group edit the **Linker Symbol Definitions** property.

The **Linker Symbol Definitions** property is a semicolon-separated list of symbol definitions, for example "**name1=value1;name2=value2**". Clicking the button at the right of the property displays the **Linker Symbol Definitions** dialog which will allow you to easily edit the definitions.

Example

The following defines two linker symbols, **stack_size** with a value of 512 (0x200) and **__vfprintf** with a value of the symbol **__vfprintf_int**.

```
-Dstack_size=0x200 -D__vfprintf=__vfprintf_int
```

-F (Set output format)

Syntax

-F format

Description

The **-F** option instructs the linker to write its output in the format **fmt**. The linker supports the following formats:

- **-Fsrec** Motorola S-record format
- **-Fhex** Intel extended hex format
- **-Ftek** Tektronix hex format
- **-Ttxt** Texas Instruments hex format
- **-Flst** Hexadecimal listing
- **-Fhzx** Rowley native format

The default format, if no other format is specified, is **-Fhzx**.

Setting this in CrossStudio

To set the output format for a project:

- Select the project in the **Project Explorer**.
- In the **Linker Options** group select the required format from the **Output Format** property.

-g (Propagate debugging information)

Syntax

-g

Description

The **-g** option instructs the linker to propagate debugging information contained in the individual object files into the linked image. If you intend to debug your application at the source level, you must use this option when linking your program.

-H (checksum sections)

Syntax

-H section,...=method

Description

This option generates checksums of the sections in the section list into the **CHECKSUM** segment. Only the used content of the section is checksummed, not any of the unused part.

The method can be one of:

- **sum** A simple sum of all the data items.
- **lrc** or **xor** A longitudinal redundancy check of the data bytes, that is all bytes are exclusive-ored together.
- **crc16** A CRC16 of the data using the generating polynomial 0x11021.
- **crc32** A CRC32 of the data using the generating polynomial 0x104C11DB7.

Setting this in CrossStudio

- Select the project in the **Project Explorer**.
- In the **Linker Options** group edit the **Checksum Sections** property.

The **Checksum Sections** property is a list of comma-separated sections followed by '=' followed by the checksum method. If the method is omitted, CRC16 is assumed.

Example

To checksum the **CODE** and **CONST** sections using CRC16:

```
-HCODE,CONST=crc16
```

The **CHECKSUM** section generated by the linker for this will be as if the following assembly code had been written:

```
.PSECT "CHECKSUM"

__begin_CHECKSUM:

; CODE section range and checksum
    DW      __begin_CODE
    DW      __end_CODE
__checksum_CODE::
    DW      crc-of-CODE-section

; CONST section range and checksum
    DW      __begin_CONST
    DW      __end_CONST
```

```

__checksum_CONST::
    DW      crc-of-CONST-section

; Sentinel
    DW      0
    DW      0
__end_CHECKSUM:

```

Note that the order of the checksums in the **CHECKSUM** section is undefined, so do not assume the order chosen by the linker will remain fixed. Always access the checksums using the public labels provided.

You can verify the individual checksums of the **CODE** and **CONST** sections checksummed above by using code along the following lines:

```

// External, provided as source code in the "src" directory
unsigned int __checksum_crc16(unsigned char *begin, unsigned char *end);

// Symbols generated by the linker to delimit the CODE and CONST sections
extern unsigned char __start_CODE[], __end_CODE[];
extern unsigned char __start_CONST[], __end_CONST[];

// Symbols generated by the linker for the CODE and CONST checksums
extern unsigned int __checksum_CODE, __checksum_CONST;

void main(void)
{
    assert(__checksum_crc16(__start_CODE, __end_CODE) == __checksum_CODE);
    assert(__checksum_crc16(__start_CONST, __end_CONST) == __checksum_CONST);
}

```

However, because the linker generates a table that has the addresses of the ranges checksummed and the expected checksum, you can check all the checksummed regions very simply:

```

// External, provided as source code in the "src" directory
unsigned int __checksum_crc16(unsigned char *begin, unsigned char *end);

typedef struct {
    unsigned char *begin;
    unsigned char *end;
    unsigned int checksum;
} checksum_info;

// Symbols generated by the linker for the CHECKSUM section
extern checksum_info __begin_CHECKSUM[];

void main(void)
{
    checksum_info *p;
    for (p = __begin_CHECKSUM; p->begin || p->end; ++p)
        assert(__checksum_crc16(p->begin, p->end) == p->checksum);
}

```

-I- (Do not link standard libraries)

Syntax

-I-

Description

The **-I-** option instructs the linker not to link standard libraries automatically included by the compiler or by the assembler **INCLUDELIB** directive. If you use this options you must supply your own library functions or provide the names of alternative sets of libraries to use.

Setting this in CrossStudio

To exclude standard libraries from the link:

- Select the project in the **Project Explorer**.
- In the **Linker Options** group set the **Include Standard Libraries** property to **No**.

-l (Link library)

Syntax

-l x

Description

Link the library '**lib x.hza**' from the library directory. The library directory is, by default **\$(InstallDir)/lib**, but can be changed with the **-L** option.

-L (Set library directory path)

Syntax

-L dir

Description

Sets the library directory to **dir**. If **-L** is not specified on the command line, the default location to search for libraries is set to ***\$(InstallDir)/lib***.

-M (Display linkage map)

Syntax

-M-

-M *filename*

Description

The **-M** option prints a linkage map to standard output; **-M file** prints a linkage map to *filename*.

-o (Set output file name)

Syntax

-o filename

Description

The **-o** option instructs the linker to write its linked output to **filename**.

-O (Optimize output)

Syntax

-O[level]

Description

Optimize at level **level**. **level** must be between **-9** and **+9**. Negative values of **level** optimize code space at the expense of speed, whereas positive values of **level** optimize for speed at the expense of code space. The '+' sign for positive optimization levels is accepted but not required.

The exact strategies used by the compiler to perform the optimization will vary from release to release and are not described here.

-Obl (Enable block localization optimization)

Syntax

-Obl

Description

Enables the block locality improvement optimization. This optimization moves blocks of code in order reduce span-dependent jump sizes on many architectures.

This optimization is extremely compute intensive but delivers very good results on many applications. It will always reduce code size and execution time because the size of span-dependent jumps are reduced.

-Ocm (Enable code motion optimization)

Syntax

-Ocm

Description

Enables the code motion optimization. Code motion moves blocks of instructions from one place to another to reduce the number of jump instructions in the final program. Code motion will always reduce code size and increase execution speed.

It is extremely difficult to debug a program which has been linked with code motion enabled because parts of functions will be moved around the program and merged with other functions.

Setting this in CrossStudio

To enable this optimization for a project:

- Select the project in the **Project Explorer**.
- In the **Optimization Options** group set the **Code Motion Optimization** property to **Yes**.

-Ocp (Enable copy propagation optimization)

Syntax

-Ocp

Description

Enables the copy propagation optimization. Copy propagation tracks the values in registers and tries to eliminate register-to-register moves. Copy propagation will always reduce code size and increase execution speed wherever it is applied.

Setting this in CrossStudio

To enable this optimization for a project:

- Select the project in the **Project Explorer**.
- In the **Optimization Options** group set the **Copy Propagation Optimization** property to **Yes**

-Ojc (Enable jump chaining optimization)

Syntax

-Ojc

Description

Enables the jump chaining optimization. Jump chaining reduces the size of span-dependent jumps by finding a closer jump instruction to the same target address and reroutes the original jump to that jump. This optimization always reduces code size at the expense of execution speed because of the jump chains introduced.

Jump chaining delivers its best results with the Jump Threading optimization enabled.

Setting this in CrossStudio

To enable this optimization for a project:

- Select the project in the **Project Explorer**.
- In the **Optimization Options** group set the **Jump Chaining Optimization** property to **Yes**.

See Also

[-Ojt \(Jump threading optimization\)](#)

-Ojt (Enable jump threading optimization)

Syntax

-Ojt

Description

Enables the jump threading optimization. Jump threading finds jumps to jump instructions and reroutes the original jump instruction to the final destination. Jump threading will always increase execution speed and may reduce code size. A jump will not be rerouted if, in doing so, the size of the jump instruction increases.

Setting this in CrossStudio

To enable this optimization for a project:

- Select the project in the **Project Explorer**.
- In the **Optimization Options** group set the **Jump Threading Optimization** property to Yes.

See Also

[-Ojc \(Jump chaining optimization\)](#)

-Oph (Enable peephole optimizations)

Syntax

-Oph

Description

Enables peephole optimizations. Peephole optimizations transform local code sequences into more efficient code sequences using a collection of common idioms. Peephole optimization will always reduce code size and increase execution speed.

Setting this in CrossStudio

To enable this optimization for a project:

- Select the project in the **Project Explorer**.
- In the **Optimization Options** group set the **Peephole Optimization** property to **Yes**

-Osf (Enable flattening optimizations)

Syntax

-Osf

Description

Enables flattening optimizations. Calls to subroutines immediately followed by an unconditional return instruction are converted into jumps to the subroutine.

Setting this in CrossStudio

To enable this optimization for a project:

- Select the project in the **Project Explorer**.
- In the **Optimization Options** group set the **Flattening Optimization** property to **Yes**

-Otm (Enable tail merging optimization)

Syntax

-Otm

Description

Enables the tail merging optimization. Tail merging finds identical code sequences at the end of functions that can be shared, deletes all copies and reroutes control flow to exactly one instance of the code sequence. Tail merging will always reduce code size at the expense of executing an additional jump instruction.

Setting this in CrossStudio

To enable this optimization for a project:

- Select the project in the **Project Explorer**.
- In the **Optimization Options** group set the **Tail Merging Optimization** property to Yes

-Oxc (Enable code factoring optimization)

Syntax

-Oxc[= *n*]

Description

Enables the code factoring optimization. Code factoring is also commonly called *common block subroutine packing*, **cross calling**, and *procedure abstraction*. Code factoring finds common instruction sequences and replaces each common sequence with a subroutine call to one instance of that sequence.

Code factoring will always reduce the size of a program at the expense of execution speed as there is an overhead for the additional subroutine call and return instructions.

The option parameter *n* defines the number of bytes that must the common instruction sequence must contain before it is abstracted into a subroutine. Smaller values of *n* are likely to find more common sequences and will transform the code into a smaller, but slower, program. Larger values of *n* will find fewer common sequences, where each of those sequences are longer and will transform the code in to a slightly larger, and slightly faster, program.

The time complexity of the algorithm use depends upon *n*. Smaller values of *n* require more time for optimization to find and transform the small code sequences, whereas larger values of *n* requires less time to run as fewer common code sequences will be identified. You can also limit the number of code factoring passes using the **-Oxcp** option.

It is extremely difficult to debug a program which has been linked with code factoring enabled because parts of functions will be extracted and placed into their own subroutine.

See Also

[-Oxcx \(Extreme code factoring optimization\)](#), [-Oxcp \(Set maximum code factoring passes\)](#)

-Oxcx (Enable extreme code factoring)

Syntax

-Oxcx[= *n*]

Description

This optimization is identical to the Code Factoring optimization except that it works much harder to find common code sequences and, consequently, is much slower than the standard cross calling optimization. We recommend that you do not use this optimization unless you wish to reduce code size to the smallest possible as this optimization takes a long time to run for large programs.

Code factoring will always reduce the size of a program at the expense of execution speed as there is an overhead for the additional subroutine call and return instructions.

The option parameter *n* defines the number of bytes that must the common instruction sequence must contain before it is abstracted into a subroutine. Smaller values of *n* are likely to find more common sequences and will transform the code into a smaller, but slower, program. Larger values of *n* will find fewer common sequences, where each of those sequences are longer and will transform the code in to a slightly larger, and slightly faster, program.

The time complexity of the algorithm use depends upon *n*. Smaller values of *n* require more time for optimization to find and transform the small code sequences, whereas larger values of *n* requires less time to run as fewer common code sequences will be identified.

It is extremely difficult to debug a program which has been linked with cross calling enabled because parts of functions will be extracted and placed into their own subroutine.

See Also

[-Oxc \(Code factoring optimization\)](#)

-Oxcp (Set maximum code factoring passes)

Syntax

-Oxcp[= *n*]

Description

Sets the maximum number of code factoring passes to *n*, or sets unlimited code factoring if *n* is omitted.

Each pass of the code factoring optimization may increase the maximum subroutine depth required by the linked application by one call. If stack space is at a premium, you can limit the additional subroutine depth introduced by the code factoring optimization to *n*. For instance, specifying **-Oxcp=1** will cause the application to use only a single depth of subroutines, with no other calls, when performing code factoring; specifying **-Oxcp=2** will introduce up to two additional subroutines (mainline code calls a subroutine which then calls another subroutine) and will require up to two additional return addresses on the call stack.

Setting *n* higher leads to higher code compression but introduces more subroutines and makes the code slower to execute—you may wish to limit the number of subroutines, their size, and the subroutine depth to strike a balance between speed, code space, and stack requirements.

For processors with a small hardware stack, it may be appropriate to limit the code factoring optimization to only a few levels of subroutines so that the hardware stack does not overflow, or even to disable code factoring completely if stack space is at a premium.

See Also

[-Oxc \(Code factoring optimization\)](#), [-Oxcx \(Extreme code factoring optimization\)](#)

-Oxj (Enable cross jumping optimization)

Syntax

-Oxj

Description

Enables the cross jumping optimization. Cross jumping finds identical code sequences that can be shared, deletes all copies and reroutes control flow to exactly one instance of the code sequence. Cross jumping will always reduce code size at the expense of executing an additional jump instruction.

Setting this in CrossStudio

To enable this optimization for a project:

- Select the project in the **Project Explorer**.
- In the **Optimization Options** group set the **Cross Jumping Optimization** property to **Yes**

-T (Locate sections)

Syntax

-T section ,...=**start**[- **end**]

Description

This option sets the way that sections are aggregated and laid out in memory. The **start** and **end** addresses are inclusive and define the memory segment into which sections in the list are placed. Sections are allocated in the order that they are specified in the list.

Setting this in CrossStudio

Section layout is configured using the XML-format memory map file. To set the memory map file for a project:

- Select the project in the **Project Explorer**.
- In the **Linker Options** group set the **Memory Map File** to the location of the appropriate memory map file.

Example

To aggregate and place the **CODE** and **CONST** sections into the memory segment 0x1000 through 0xffff (inclusive) with **CODE** placed before **CONST**, to aggregate the **IDATA0** and **UDATA0** sections into the memory segment 0x200 through 0xaff placing **IDATA0** before **UDATA0**:

```
-TCODE,CONST=0x1000-0xffff -TIDATA0,UDATA0=0x200-0xaff
```

-we (Treat warnings as errors)

Syntax

-we

Description

This option directs the linker to treat all warnings as errors.

Setting this in CrossStudio

To suppress warnings for all files in a project:

- Select the project in the **Project Explorer**.
- In the **General Options** group set the **Treat Warnings as Errors** property to **Yes**.

To suppress warnings for a particular file:

- Select the file in the **Project Explorer**.
- In the **General Options** group set the **Treat Warnings as Errors** property to **Yes**.

-w (Suppress warnings)

Syntax

-w

Description

This option instructs the linker not to issue any warnings.

Setting this in CrossStudio

To suppress warnings for a project:

- Select the project in the **Project Explorer**.
- In the **General Options** group set the **Suppress Warnings** property to **Yes**.

To suppress warnings for a particular file:

- Select the file in the **Project Explorer**.
- In the **General Options** group set the **Suppress Warnings** property to **Yes**.

-v (Verbose execution)

Syntax

-v

Description

The **-v** option instructs the linker to display progress information whilst linking.

-V (Version information)

Syntax

-V

Description

The **-V** switch instructs the linker to display its version information.

Hex extractor reference

The hex extractor **hex** is used to prepare images in a number of formats to burn into EPROM or flash memory. Although the linker is capable of writing all the formats described here, it doesn't have the capability of splitting files for different bus or device sizes.

When you prepare to download applications to a monitor held in ROM, you usually need the application in a single industry-standard format such as S-records or Intel hex format. What you don't need to do is split high and low bytes, and you won't need to split across ROMs. The extractor generates files in this format by default—all you need to provide is the format you need the file in.

Example

```
hex -Fhex app.hzx
```

This will generate a single Intel hex, **app.hzx.hex**, which contains all code and data in the application. The addresses in the output file are the physical addresses of where the code and data are to be loaded.

Command line option summary

The hex extractor supports the following command-line options:

Option	Description
-F fmt	Select output format fmt
-o file	Write output with file as a prefix
-T name	Extract section name from input file
-V	Display tool version information
-W width	Set bus width to width

-F (Set output format)

Syntax

-F fmt

Description

The **-F** option sets the output format to **fmt**. The supported output formats are:

- **-Fsrec** Motorola S-record format
- **-Fhex** Intel extended hex format
- **-Ftek** Tektronix hex format
- **-Ftxt** Texas Instruments hex format
- **-Flst** Hexadecimal listing

Example

```
hex app.hzx -Ftxt -o app
```

This reads the application file **app.hzx** and writes all sections in Texas Instruments hex format to **app.txt**.

-o (Set output prefix)

Syntax

-o prefix

Description

The **-o** option sets the prefix to use for the created files. If **-o** is not specified on the command line, the input file name is used as the output prefix.

Example

```
hex -w2 app.hzx -Fhex -o app
```

This splits the application file **app.hzx** into high and low bytes for a processor with a 16-bit bus. The bytes at even addresses are placed in the file **app.0.hex** and the bytes at odd addresses in the file **app.1.hex**.

-T (Extract named section)

Syntax

-T name

Description

The **-T** option extracts the named section from the input file. By default, all loadable sections are extracted from the input file. You can specify multiple **-T** options on the command line to extract more than one section.

Example

```
hex app.hzx -Ftxt -TIDATA0 -TCODE
```

This reads the application file **app.hzx**, extracts the sections IDATA0 and CODE (ignoring all others) and writes them to the file **app.hzx.txt**.

-V (Display version)

Syntax

-V

Description

The **-V** option instructs the hex extractor to display its version information.

-W (Set bus width)

Syntax

-W[1|2|4]

Description

The **-W** option splits a file into multiple streams. You can use this if your EPROM programmer cannot split files into different devices when programming.

Example

```
hex -w2 app.hzx -Fhex
```

This splits the application file **app.hzx** into high and low bytes for a processor with a 16-bit bus. The bytes at even addresses are placed in the file **app.hzx.0.hex** and the bytes at odd addresses in the file **app.hzx.1.hex**.

Librarian reference

The librarian, or archiver, creates and manages object code libraries. Object code libraries are collections of object files that are consolidated into a single file, called an archive. The benefit of an archive is that you can pass it to the linker and the linker will search the archive to resolve symbols needed during a link.

By convention, archives have the extension **.hza**. In fact, the format used for archives is compatible with PKWare's popular Zip format with deflate compression, so you can manipulate and browse archives using many readily available utilities for Windows.

Automatic archiving

The compiler driver **hcl** can create archives for you automatically using the **-ar** option. You will find this more convenient than manipulating archives by hand and we recommend that you use the compiler driver to construct archives.

Command syntax

You invoke the archiver using the following syntax:

```
har [ option ] archive file...
```

archive is the archive to operator on. *file* is an object file to add, replace, or delete from the archive according to *option*. *option* is a command-line option. Options are case sensitive and cannot be abbreviated.

Options

Option	Description
-c	Create archive
-d	Delete member from archive
-r	Replace member in archive/Add member to existing archive
-t	List members of archive
-v	List members of archive
-V	Verbose mode

-c (Create archive)

Syntax

har -c archive-name object-file...

Description

This option creates a new archive. The archive will be created, overwriting any archive which already exists with the same name.

Example

To create an archive called **cclib.hza** which initially contains the two object code files **ir.hzo** and **cg.hzo** you would use:

```
har -c cclib.hza ir.hzo cg.hzo
```

The archiver expands wildcard file names so you can use

```
har -c cclib.hza *.hzo
```

to create an archive containing all the object files found in the current directory.

-r (Add or replace archive member)

Syntax

har -r archive-name [object-file...]

Description

You can replace members in an archive using the **-r** switch. If you specify a file to add to the archive, and the archive doesn't contain that member, the file is simply appended to the archive as a new member.

Example

To replace the member **ir.hzo** in the archive **cclib.hza** with the file **ir.hzo** on disk you would use:

```
har cclib.hza -r ir.hzo
```

-d (Delete archive members)

Syntax

har -d archive-name [object-file...]

Description

You can remove members from the archive using the **-d** switch, short for *delete*.

Example

To remove the member **ir.hzo** from the archive **cclib.hza** you'd use:

```
har cclib.hza -d ir.hzo
```


-t (List archive members)

Syntax

har -t archive-name [object-file...]

Description

To show the members which comprise an archive, you use the **-t** switch. The member's names are listed together with their sizes. If you only give the archive name on the command line, the archiver lists all the members contained in the archive. However, you can list the attributes of specific members of the archive by specifying on the command line the names of the members you're interested in.

Example

To list all the members of the archive **cclib.hza** created above you'd use:

```
har -t cclib.hza
```

To list only the attributes of the member **ir.hzo** contained in the archive **cclib.hza** you'd use:

```
har -t cclib.hza ir.hzo
```

CrossBuild

The command line program **crossbuild** enables your software to be built without using **CrossStudio**. You can use this for production and batch build purpose

Using CrossBuild with a CrossStudio project file

You can use a CrossStudio project file (.hzip)

```
crossbuild [options] project-file
```

You must specify a configuration to build in using the **-config** option, for instance:

```
crossbuild -config "V5T Thumb LE Release" arm.hzip
```

This example will build all projects in the solution contained in **arm.hzip** in the configuration **V5T Thumb LE Release**.

If you want to build a specific project in the solution then you can specify it using the **-project** option.

```
crossbuild -config "V5T Thumb LE Release" -project "libm" libc.hzip
```

This example will build the project **libm** contained in **libc.hzip** in the configuration **V5T Thumb LE Release**.

If your project file imports other project files (using the <import..> mechanism) then denoting projects requires you to specify the solution names as a comma separated list in brackets after the project name.

```
crossbuild -config "V5T Thumb LE Release" -project "libc(C Library)" arm.hzip
```

With this example **libc(C Library)** specifies the **libc** project in the **C Library** solution that has been imported by the project file **arm.hzip**.

If you want 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.

```
crossbuild -config "ARM Debug" -solution "ARM Targets,EB55" arm.hzip
```

With this example **ARM Targets,EB55** specifies the **EB55** solution imported by the **ARM Targets** solution which in turn was imported by the project file **arm.hzip**.

You can do a batch build using the **-batch** option.

```
crossbuild -config "ARM Debug" -batch libc.hzip
```

With this example the projects in **libc.hzip** which are marked to batch build in the configuration **ARM Debug** will be built.

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 file is up to date. You can force a complete build by using the **-rebuild** option. Alternatively you can remove all output files using the **-clean** option.

You can see the commands that are being used in the build if you use the **-echo** option and you can also see why commands are being executed using the **-verbose** option. You can see what commands will be executed without executing them using the **-show** option.

Using crossbuild without a crosstudio project file

You can use crossbuild without a crosstudio project by specifying the name of an installed project template, the name of the project and files to build.

```
crossbuild -config .. -template LM3S_EXE -project myproject -file main.c
```

alternatively you can specify a project type rather than a template

```
crossbuild -config .. -type "Library" -project myproject -file main.c
```

You can specify project properties using the **-property** option.

```
crossbuild .. -property Target=LM3S811
```

CrossBuild Options

-batch	Do a batch build.
-config 'name'	Specify the configuration to build in. If the 'name' configuration can't be found crossbuild will list the set of configurations that are available.
-clean	Remove all the output files of the build process.
-D macro=value	Define a crossworks macro value for the build process.
-echo	Show the command lines as they are executed.
-file 'name'	Build the named file - use with -template or -type.
-packagesdir 'name'	Override the default value of the \$(PackagesDir) macro.
-project 'name'	Specify the name of the project to build. When used with a project file if crossbuild can't find the specified project then a list of project names is shown.
-property 'name'='value'	Specify the value of a project property - use with -template or -type. If crossbuild can't find the specified property then a list of the properties is shown.
-rebuild	Always execute the build commands.
-show	Show the command lines but don't execute them.
-solution 'name'	Specify the name of the solution to build. If crossbuild can't find the specified solution then a list of solution names is shown.
-studiodir 'name'	Override the default value of the \$(StudioDir) macro.
-template 'name'	Specify the project template to use. If crossbuild can't find the specified template then a list of template names is shown.
-type 'name'	Specify the project type to use. If crossbuild can't find the specified project type then a list of project type names is shown.
-verbose	Show build information.

CrossLoad

CrossLoad is a command line program that allows you to download and debug applications without using **CrossStudio**.

Usage

```
crossload [options] [files...]
```

Options

-break <i>symbol</i>	When used with the -debug option this will stop execution at <i>symbol</i> .
-config <i>configuration</i>	Specify the build configuration to use.
-debug	Command line debugging mode is entered. A command prompt is displayed in which debugger commands can be entered. The command prompt has a simple history and editing mechanism. See Command Line Debugger for an overview of the command line debugger.
-filetype <i>filetype</i>	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. Use the -listfiletypes option to display the list of supported file types.
-help	Display the command line options.
-listfiletypes	List all of the supported file types.
-listprops	List the target properties of the target specified by the -target option.
-listtargets	List all of the supported target interfaces.
-loadaddress <i>address</i>	When downloading a load file that doesn't contain any address information, such a binary file, this option specifies the base address the file should be downloaded to.
-nodisconnect	Do not disconnect the target interface when finished.
-nodownload	Do not carry out download, just verify.
-noverify	Do not carry out verification of download.
-packagesdir <i>directory</i>	Set $\$(PackagesDir)$ to be <i>directory</i> .
-project <i>name</i>	Specify the name of the project to use.
-quiet	Do not output any progress messages.
-script <i>file</i>	When used with the -debug option this will execute the debug commands in <i>file</i> .

-serve	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 <code>debug_getargs()</code> or by compiling <code>crt0.s</code> with the <code>FULL_LIBRARY</code> C preprocessor definition defined so that <code>argc</code> and <code>argv</code> are passed to <code>main</code> .
-setprop <i>property</i> = <i>value</i>	Set the target property <i>property</i> to <i>value</i> .
-solution <i>file</i>	Specify the CrossWorks solution file to use.
-studiodir <i>directory</i>	Set $\$(StudioDir)$ to be <i>directory</i> .
-target <i>target</i>	Specify the target interface to use. Use the -listtargets option to display the list of supported target interfaces.
-verbose	Produce verbose output.

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 produce a list of the currently supported target interfaces use the **-listtargets** option.

```
crossload -listtargets
```

This command will produce a list of target interface names and descriptions:

```
usb          USB CrossConnect
parport      Parallel Port Interface
sim          Simulator
```

Use the **-target** option followed by the target interface name to specify which target interface to use:

```
crossload -target usb ...
```

CrossLoad is normally used to download and/or verify projects created and built with **CrossStudio**. To do this you need to 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 might not have been created using **CrossStudio** using the settings from an existing **CrossStudio** project. You might want to do this if your existing project describes specific loaders or scripts that are required in order to download the application. To do this you simply need to add the name of the file you want to download to the command line. For example the following command line will download the HEX file *ExternalApp.hex* using the release settings of the project *MyProject* using a USB CrossConnect:

```
crossload -target usb -solution MySolution.hzp -project MyProject -config
Release ExternalApp.hex
```

CrossLoad is able to download and verify a range of file types. The supported file types vary between systems, to display a list of the file types supported by **CrossLoad** use the **-listfiletypes** option:

crossload -listfiletypes

This command will produce 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 carry out a download without the need to specify a **CrossStudio** project. In this case all you need to specify is the target interface and the load file. For example the following command line will download **myapp.s19** using a USB CrossConnect:

```
crossload -target usb myapp.s19
```

Each target interface has a range of configurable properties that allow you to customize the default behaviour. To produce a list of the target properties and their current value use the **-listprops** option:

```
crossload -target parport -listprops
```

This command will produce a list of the *parport* target interfaces properties, a description of what the properties are and their current value:

```
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"="Ltp2" ...
```

Memory Map file format

CrossStudio memory map files are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type which is 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 attributes. Names should be unique within a hierarchy level. Within a Root element there are MemorySegment and RegisterGroup elements which represent memory 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 0x prefixed hexadecimal number.
- **size** The size of the memory segment. A simple expression usually a 0x prefixed hexadecimal number.
- **access** The permissible access types of the memory segment. One of **ReadOnly**, **Read/Write**, **WriteOnly**, **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 group sets of registers. Register elements are used to define peripheral registers.

```
<Root name="My Board" >
  <RegisterGroup name="Peripheral1" start="0x2100" size="0x10" >
    <Register name="Register1" start="+0x8" size="4" >
```

Register group 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 and underscores but not starting with a number.
- **start** The start address of the memory segment. Either a 0x prefixed hexadecimal number or a + prefixed 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 MemorySegment element.
- **address_symbol** The same as the MemorySegment element.

A Register element can contain BitField elements that represent bits within a peripheral register.

```
<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" />
    </Register>
  </RegisterGroup>
</Root>
```

BitField elements have the following attributes.

- **name** BitFields names should be valid C/C++ identifier names i.e. alphanumeric and underscores but not starting with a number.
- **start** The starting bit position 0-31.
- **size** The number of bits 1-31.

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" />
    </Register>
  </RegisterGroup>
</Root>
```

Enum elements have the following attributes.

- **name** Enum name
- **start** The enumeration value.

You can import [CMSIS](#) svd files 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 which is 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. Names should be unique within a hierarchy level. Within a Root element there are MemorySegment elements. These correspond to memory regions that are defined in a memory map file that will be used in conjunction with the section placement file when linking a program.

```
<Root name="Flash Placement" >
  <MemorySegment name="FLASH" >
```

A MemorySegment contains ProgramSection elements which 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 ProgramSection will be placed last.

```
<Root name="My Board" >
  <MemorySegment name="FLASH" >
    <ProgramSection name=".text" ... >
```

ProgramSection elements have the following attributes.

- **alignment** The required alignment of the program section - a decimal number specifying the byte alignment.
- **load** If "Yes" then the section is loaded. If "No" then the section isn't loaded.
- **start** The optional start address of the program section - a 0x prefixed hexadecimal number.
- **size** The optional size of the program section in bytes - a 0x prefixed hexadecimal number.
- **address_symbol** A symbolic name for the start address of the section.
- **end_symbol** A symbolic name for the end address of the section.
- **size_symbol** A symbolic name for the size of the section.

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. We anticipate that you may want to edit project files and perhaps generate them so they are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type which is 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 name of solution displayed in the project explorer and also has a version attribute which defines the file format version of the project file. Solutions can contain projects, projects can contain folder and /files, and folders can contain folder 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, names of folder elements must be unique to the project however 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 you cannot have file paths containing the \$ character. For example

```
<file file_name="$(StudioDir)/source/crt0.s" Name="crt0.s" />
```

will be expanded using the value of the \$(StudioDir) when the file is referenced from CrossStudio.

Project properties are held in configuration elements with the Name attribute of the configuration element corresponding to the configuration name e.g. "Debug". At a given project level (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 link projects together using the import element.

```
<import file_name="target/libc.hzip" />
```

Project Templates file format

The CrossStudio New Project Dialog works from a file called `project_templates.xml` which is held in the `targets` subdirectory of the CrossStudio installation directory. We anticipate that you may want to add your own new project types so they are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type which is 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 project 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`).

```
<project caption="ARM Evaluator7T" name="Executable" description="An executable for an ARM
  Evaluator7T." type="Executable"/>
```

The project type can be one of

- "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 that are to be created for the project are defined using the `configuration` element. The configuration element 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 then you can specify this using the `value` attribute and optionally set the property in a defined configuration.

```
<property name="target_reset_script" configuration="RAM"
  value="Evaluator7T_ResetWithRamAtZero()"/>
```

Alternatively you can include a property that will be shown to the user who can 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.

```
<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](#) to specify files that are in the CrossStudio installation directory. Files will be copied to the project directory or just left as references based on the value of the `expand` attribute.

```
<file name="$(StudioDir)/source/crt0.s" expand="no"/>
```

You can define the set of configurations that can be referred to in the 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.

```
<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 project property settings and can also set C pre-processor definitions when selected. Such properties are called property groups and are defined in a property groups file. The property group file to use for a project is defined by the `Property Groups File` property. These files usually define target specific properties and are structured using XML syntax to enable simple construction and parsing.

The first entry of the property groups file defines the XML document type which is used to validate the file format.

```
<!DOCTYPE CrossStudio_Group_Values>
```

The next entry is the `propertyGroups` element; which is used to group a set of property groups entries into an XML hierarchy.

```
<propertyGroups>
  <group....
  ....
  <group....
</propertyGroups>
```

Each group has the name of the group (attribute `name`), the name of the options category (attribute `group`), short (attribute `short`) and long (attribute `long`) help descriptions and a default value (attribute `default`).

```
<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`)

```
<groupEntry name="STR910FW32">
```

A `groupEntry` has the property values and C pre-processor definitions that are set when the `groupEntry` is selected which are specified with `property` and `cdefine` elements.

```
<groupEntry...>
  <property....
  <cdefine....
  <property....
</groupEntry>
```

A property element has the property 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 should contain one **package** element that describes the package. Optionally, the **package** element can contain a collection of **file**, **history** and **documentation** elements, these are used by CrossStudio for documentation purposes.

The filename of the package description file should match that of the package and end in *_package.xml*.

Here is an example of two package description files, the first for a base chip support package for the LPC2000, the second for a board support package dependent on the first:

[Filename: 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>
```

[Filename: 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 Element

The *package* element describes the support package, its contents and any dependencies it has on other packages.

Attribute Name	Description
author	The author of the package.
board_manufacturer	The manufacturer of the board supported by the package (<i>CPU manufacturer will be used if omitted</i>).
board_name	The name of the specific board supported by the package (<i>only required for board support packages</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 CPU family is specified</i>).
crossstudio_versions	A string describing which version of CrossStudio the package is supported on (<i>optional</i>). The format of the string is target_id_number:version_range_string .
description	A description of the package (<i>optional</i>).
dependencies	Semicolon separated list of packages that the package requires to be installed in order to work.
installation_directory	The directory the package should be installed to (<i>optional, defaults to "\$(PackagesDir)" if undefined</i>).
title	A short description of the package (<i>optional</i>).
version	The package version number.

***file* Element**

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, documentation, etc.

Attribute Name	Description
file_name	The file path of the file.
title	A description of the file.

Optionally, **file** elements can be grouped into categories using the **group** element.

group Element

The **group** element is used for categorising files described by **file** elements into a particular group.

Attribute Name	Description
title	A title of the group.

***history* Element**

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 Name	Description
name	The name of the version being described.

The **version** element should contain a collection of **description** elements.

***description* Element**

Each **description** element contains text that describes a feature of the package version.

***documentation* Element**

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* Element**

The **section** element contains the documentation in XHTML format.

Attribute Name	Description
name	The title of the documentation section.

target_id_number

The following table lists the possible target ID numbers:

Target	ID Number
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.

Project Property Reference

Property categories

Build Properties	Properties that are generally applicable.
Combining Properties	Properties that apply to combining projects.
Compilation Properties	Properties that apply to compilation.
Debugging Properties	Properties that apply to debugging executable and externally built executable projects.
External Build Properties	Properties that apply to externally built executable project.
General Properties	Properties that apply to project files and folders.
Library Properties	Properties that apply to library project types.
Linker Properties	Properties that apply to executable projects.
Staging Properties	Properties that apply to staging projects.

General Build Properties

Build Options

Property	Description
Always Rebuild <code>build_always_rebuild</code> – Boolean	Specifies whether or not to always rebuild the project/folder/file.
Build Quietly <code>build_quietly</code> – Boolean	Suppress the display of startup banners and information messages.
Enable Unused Symbol Removal <code>build_remove_unused_symbols</code> – Boolean	Enable the removal of unused symbols from the executable.
Exclude From Build <code>build_exclude_from_build</code> – Boolean	Specifies whether or not to exclude the project/folder/file from the build.
Include Debug Information <code>build_debug_information</code> – Boolean	Specifies whether symbolic debug information is generated.
Intermediate Directory <code>build_intermediate_directory</code> – FileName	Specifies a relative path to the intermediate file directory. This property will have macro expansion applied to it. The macro <code>\$(IntDir)</code> is set to this value.
Memory Map File <code>linker_memory_map_file</code> – ProjFileName	The name of the file containing the memory map description.
Memory Map Macros <code>linker_memory_map_macros</code> – StringList	Macro values to substitute in memory map nodes. Each macro is defined as name=value and are separated by <code>;</code> .
Output Directory <code>build_output_directory</code> – FileName	Specifies a relative path to the output file directory. This property will have macro expansion applied to it. The macro <code>\$(OutDir)</code> is set to this value. The macro <code>\$(RootRelativeOutDir)</code> is set relative to the Root Output Directory if specified.
Project Dependencies <code>project_dependencies</code> – StringList	Specifies the projects the current project depends upon.
Project Directory <code>project_directory</code> – String	Path of the project directory relative to the directory containing the project file. The macro <code>\$(ProjectDir)</code> is set to the absolute path of this property.
Project Macros <code>macros</code> – StringList	Specifies macro values which are expanded in project properties. Each macro is defined as name=value and are separated by <code>;</code> .
Project Type <code>project_type</code> – Enumeration	Specifies the type of project to build. The options are Executable , Library , Object file , Staging , Combining , Externally Built Executable .

Property Groups File property_groups_file_path – ProjFileName	The file containing the property groups for this project. This is applicable to Executable and Externally Built Executable project types only.
Root Output Directory build_root_output_directory – FileName	Allows a common root output directory to be specified that can be referenced using the \$(RootOutDir) macro.
Suppress Warnings build_suppress_warnings – Boolean	Don't report warnings.
Treat Warnings as Errors build_treat_warnings_as_errors – Boolean	Treat all warnings as errors.

General Options

Property	Description
Batch Build Configurations batch_build_configurations – StringList	The set of configurations to batch build.
Inherited Configurations inherited_configurations – StringList	The list of configurations that are inherited by this configuration.

Combining Project Properties

Combining Options

Property	Description
Combine Command combine_command – String	The command to execute. This property will have macro expansion applied to it with the macro \$(CombiningOutputFilePath) set to the output filepath of the combine command and the macro \$(CombiningRelInputPaths) is set to the (project relative) names of all of the files in the project.
Combine Command Working Directory combine_command_wd – String	The working directory in which the combine command is run. This property will have macro expansion applied to it.
Output File Path combine_output_filepath – String	The output file path the stage command will create. This property will have macro expansion applied to it.
Set To Read-only combine_set_readonly – Boolean	Set the output file to read only or read/write.

Compilation Properties

Assembler Options

Property	Description
Additional Assembler Options asm_additional_options – StringList	Enables additional options to be supplied to the assembler. This property will have macro expansion applied to it.
Additional Assembler Options From File asm_additional_options_from_file – ProjFileName	Enables additional options to be supplied to the assembler from a file. This property will have macro expansion applied to it.

Code Generation Options

Property	Description
Block Localization Optimization optimize_block_locality – Boolean	Reduce spand-dependent jumps by rearranging basic blocks to improve their locality.
Code Factoring Optimization optimize_cross_calling – Enumeration	Reduces code size at the expense of execution speed by factoring common code sequences into subroutines.
Code Factoring Passes linker_cross_call_maximum_passes – IntegerRange	The maximum number of passes to perform when code factoring (0=unlimited).
Code Factoring Subroutine Size linker_cross_call_minimum_subroutine_size – IntegerRange	The minimum size of subroutine created by code factoring.
Code Motion Optimization optimize_code_motion – Boolean	Rearrange code to reduce the number of jump instructions.
Copy Propagation Optimization optimize_copy_propagation – Boolean	Tries to eliminate copy instructions and reduce code size.
Cross Jumping Optimization optimize_cross_jumping – Boolean	Always reduces code size at the expense of a single jump instruction.
Dead Code Elimination optimize_dead_code – Boolean	Remove code that is not referenced by the application.
Enable Exception Support cpp_enable_exceptions – Boolean	Specifies whether exception support is enabled for C++ programs.
Enable RTTI Support cpp_enable_rtti – Boolean	Specifies whether RTTI support is enabled for C++ programs.
Flattening Optimization optimize_flattening – Boolean	Subroutines that are just a call followed by a return are flattened into a jump.

Jump Chaining Optimization <code>optimize_jump_chaining</code> – Boolean	Reduce spand-dependent jump sizes by chaining jumps together.
Jump Threading Optimization <code>optimize_jump_threading</code> – Boolean	Follow jump chains and retarget jumps to jump instructions.
Optimization Strategy <code>compiler_optimization_strategy</code> – Enumeration	Minimize code size or maximize execution speed (positive values).
Peephole Optimization <code>optimize_peepholes</code> – Boolean	Find instruction sequences that can be replaced with faster, smaller code.
Register Allocation <code>optimize_register_allocation</code> – Enumeration	Whether to allocate registers to locals or locals and global addresses.
Tail Merging Optimization <code>optimize_tail_merging</code> – Boolean	Always reduces code size at the expense of a single jump instruction.
Treat 'double' as 'float' <code>double_is_float</code> – Boolean	Forces the compiler to make 'double' equivalent to 'float'.
Use Hardware Multiplier <code>build_use_hardware_multiplier</code> – Enumeration	Enables code generation for the 16-bit or 32-bit hardware multiplier.

Compiler Options

Property	Description
Additional C Compiler Only Options <code>c_only_additional_options</code> – StringList	Enables additional options to be supplied to the C compiler only. This property will have macro expansion applied to it.
Additional C Compiler Only Options From File <code>c_only_additional_options_from_file</code> – ProjFileName	Enables additional options to be supplied to the C compiler only from a file. This property will have macro expansion applied to it.
Additional C Compiler Options <code>c_additional_options</code> – StringList	Enables additional options to be supplied to the C compiler. This property will have macro expansion applied to it.
Additional C Compiler Options From File <code>c_additional_options_from_file</code> – ProjFileName	Enables additional options to be supplied to the C compiler from a file. This property will have macro expansion applied to it.
Additional C++ Compiler Only Options <code>cpp_only_additional_options</code> – StringList	Enables additional options to be supplied to the C++ compiler only. This property will have macro expansion applied to it.
Additional C++ Compiler Only Options From File <code>cpp_only_additional_options_from_file</code> – ProjFileName	Enables additional options to be supplied to the C++ compiler only from a file. This property will have macro expansion applied to it.
Enforce ANSI Checking <code>c_enforce_ansi_checking</code> – Boolean	Ensure programs conform to the ANSI-C/C++ standard.

Object File Name <code>build_object_file_name</code> – FileName	Specifies a name to override the default object file name.

Preprocessor Options

Property	Description
Ignore Includes <code>c_ignore_includes</code> – Boolean	Ignore the include directories properties.
Preprocessor Definitions <code>c_preprocessor_definitions</code> – StringList	Specifies one or more preprocessor definitions. This property will have macro expansion applied to it.
Preprocessor Undefinitions <code>c_preprocessor_undefinitions</code> – StringList	Specifies one or more preprocessor undefinitions. This property will have macro expansion applied to it.
System Include Directories <code>c_system_include_directories</code> – StringList	Specifies the system include path. This property will have macro expansion applied to it.
Undefine All Preprocessor Definitions <code>c_undefine_all_preprocessor_definitions</code> – Boolean	Does not define any standard preprocessor definitions.
User Include Directories <code>c_user_include_directories</code> – StringList	Specifies the user include path. This property will have macro expansion applied to it.

Section Options

Property	Description
Code Section Name <code>default_code_section</code> – String	Specifies the default name to use for the program code section.
Constant Section Name <code>default_const_section</code> – String	Specifies the default name to use for the read-only constant section.
Data Section Name <code>default_data_section</code> – String	Specifies the default name to use for the initialized, writable data section.
Vector Section Name <code>default_vector_section</code> – String	Specifies the default name to use for the interrupt vector section.
Zeroed Section Name <code>default_zeroed_section</code> – String	Specifies the default name to use for the zero-initialized, writable data section.

User Build Step Options

Property	Description

Post-Compile Command compile_post_build_command – String	A command to run after the compile command has completed. This property will have macro expansion applied to it with the additional \$(TargetPath) macro set to the output filepath of the compiler command.
Post-Compile Working Directory compile_post_build_command_wd – String	The working directory where the post-compile command is run. This property will have macro expansion applied to it.
Pre-Compile Command compile_pre_build_command – String	A command to run before the compile command. This property will have macro expansion applied to it.
Pre-Compile Working Directory compile_pre_build_command_wd – String	The working directory where the pre-compile command is run. This property will have macro expansion applied to it.

Debugging Properties

Debugger Options

Property	Description
Additional Load File <code>debug_additional_load_file</code> – ProjFileName	Additional file to load on debug load. This property will have macro expansion applied to it.
Additional Load File Address <code>debug_additional_load_file_address</code> – String	The address to load the additional load file.
Additional Load File Type <code>debug_additional_load_file_type</code> – Enumeration	The file type of the additional load file. The options are Detect, hzx, bin, ihex, hex, tihex, srec .
Command Arguments <code>debug_command_arguments</code> – String	The command arguments passed to the executable. This property will have macro expansion applied to it.
Entry Point Symbol <code>debug_entry_point_symbol</code> – String	Debugger will start execution at symbol if defined.
Startup Completion Point <code>debug_startup_completion_point</code> – String	Specifies the point in the program where startup is complete. Software breakpoints and debugIO will be enabled after this point has been reached.
Working Directory <code>debug_working_directory</code> – DirPath	The working directory for a debug session.

JTAG Chain Options

Property	Description
JTAG Data Bits After <code>arm_linker_jtag_pad_post_dr</code> – IntegerRange	Specifies the number of bits to pad the JTAG data register after the target.
JTAG Data Bits Before <code>arm_linker_jtag_pad_pre_dr</code> – IntegerRange	Specifies the number of bits to pad the JTAG data register before the target.
JTAG Instruction Bits After <code>arm_linker_jtag_pad_post_ir</code> – IntegerRange	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_ir</code> – IntegerRange	Specifies the number of bits to pad the JTAG instruction register with the BYPASS instruction before the target.

Target Script Options

Property	Description

Attach Script target_attach_script – String	The script that is executed when the target is attached to.
Connect Script target_connect_script – String	The script that is executed when the target is connected to.
Debug Begin Script target_debug_begin_script – String	The script that is executed when the debugger begins a debug session.
Debug End Script target_debug_end_script – String	The script that is executed when the debugger ends a debug session.
Debug Interface Reset Script target_debug_interface_reset_script – String	The script that is executed to reset the debug interface. If not specified the default debug interface reset will be carried out instead.
Disconnect Script target_disconnect_script – String	The script that is executed when the target is disconnected from.
Reset Script target_reset_script – String	The script that is executed when the target is reset.
Run Script target_go_script – String	The script that is executed when the target is run.
Stop Script target_stop_script – String	The script that is executed when the target is stopped.
Target Extras Script target_extras_script – String	The script that is executed to supply extra menu entries in the targets window context menu.

Externally Built Executable Project Properties

External Build Options

Property	Description
Build Command external_build_command – String	The command line to build the executable.
Clean Command external_clean_command – String	The command line to clean the executable.
Debug Symbols File external_debug_symbols_file_name – ProjFileName	The name of the debug symbols file. This property will have macro expansion applied to it.
Debug Symbols Load Address external_debug_symbols_load_address – String	The (code) address to be added to the debug symbol (code) addresses.
Executable File external_build_file_name – ProjFileName	The name of the externally built executable. This property will have macro expansion applied to it.
Load Address external_load_address – String	The address to load the externally built executable.
Load File Type external_load_file_type – Enumeration	The file type of the externally built executable. The options are Detect , hzx , bin , ihex , hex , tihex , srec .
Start Address external_start_address – String	The address to start the externally built executable running from.

File and Folder Properties

(Information)

Property	Description
File Name file_name – String	The name of the file. This property will have global macro expansion applied to it. The following macros are set based on the value: \$(InputDir) relative directory of file, \$(InputName) file name without directory or extension, \$(InputFileName) file name, \$(InputExt) file name extension, \$(InputPath) absolute path to the file name, \$(RelInputPath) relative path from project directory to the file name.
Name Name – String	Names the item. The macro \$(ProjectNodeName) is set to this value.
Num Elements num_elements – Integer	The number of project elements in the node.
Platform Platform – String	Specifies the platform for the project. The macro \$(Platform) is set to this value.

File Options

Property	Description
File Encoding file_codec – Enumeration	Specifies the encoding to use when reading and writing the file.
File Open Action file_open_with – Enumeration	Specifies how to open the file when it is double clicked.
File Type file_type – Enumeration	The type of file. Default setting uses the file extension to determine file type.
Tag file_tag – Enumeration	File tag which you can use to draw attention to important files in your project.

Folder Options

Property	Description
Dynamic Folder Directory path – DirPath	Dynamic folder directory specification.
Dynamic Folder Filter filter – String	Dynamic folder directory specification.

Dynamic Folder Recurse recurse – Boolean	Dynamic folder recurse into subdirectories.
Filter filter – StringList	A filter used when adding new files to the project.

Library Project Properties

Library Options

Property	Description
Library File Name build_output_file_name – FileName	Specifies a name to override the default library file name.

Executable Project Properties

Library Options

Property	Description
I/O Library Name <code>link_IOLibraryName</code> – Enumeration	Specifies the IO library (printf etc) to use.
Include Standard Libraries <code>link_include_standard_libraries</code> – Boolean	Specifies whether the standard libraries should be linked into your application.
Standard Libraries Directory <code>link_standard_libraries_directory</code> – String	Specifies where to find the standard libraries
Use Multi Threaded Libraries <code>link_use_multi_threaded_libraries</code> – Boolean	Specifies whether to use thread safe standard libraries.

Linker Options

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> – ProjFileName	Enables additional options to be supplied to the linker from a file.
Additional Output Format <code>linker_output_format</code> – Enumeration	The format used when creating an additional linked output file.
Checksum Algorithm <code>linker_checksum_algorithm</code> – Enumeration	The algorithm used to checksum sections.
Checksum Sections <code>linker_checksum_sections</code> – StringList	The list of sections to checksum using the set checksum algorithm.
DebugIO Supported <code>linker_DebugIO_enabled</code> – Boolean	Is DebugIO supported.
Executable File Name <code>build_output_file_name</code> – FileName	Specifies a name to override the default executable file name.
Generate Absolute Listing <code>linker_absolute_listing</code> – Boolean	Generate an absolute listing of the application.
Generate Map File <code>linker_map_file</code> – Boolean	Specifies whether to generate a linkage map file.

Heap Size linker_heap_size – IntegerRange	The number of bytes to allocate for the application's heap.
Keep Symbols linker_keep_symbols – StringList	Specifies the symbols that should be kept by the linker even if they are not reachable.
Linker Symbol Definitions link_symbol_definitions – StringList	Specifies one or more linker symbol definitions.
Section Placement File linker_section_placement_file – ProjFileName	The name of the file containing section placement description.
Section Placement Macros linker_section_placement_macros – StringList	Macro values to substitute in section placement nodes - MACRO1=value1;MACRO2=value2.
Stack Size linker_stack_size – IntegerRange	The number of bytes to allocate for the application's stack.

Printf/Scanf Options

Property	Description
Printf Floating Point Supported linker_printf_fp_enabled – Boolean	Are floating point numbers supported by the printf function group.
Printf Integer Support linker_printf_fmt_level – Enumeration	The largest integer type supported by the printf function group.
Printf Supported linker_printf_enabled – Boolean	Is printf supported.
Printf Width/Precision Supported linker_printf_width_precision_supported – Boolean	Enables support for width and precision specification in the printf function group.
Scanf Classes Supported linker_scanf_character_group_matching_enabled – Boolean	Enables support for %[...] and %[^...] character class matching in the scanf functions.
Scanf Floating Point Supported linker_scanf_fp_enabled – Boolean	Are floating point numbers supported by the scanf function group.
Scanf Integer Support linker_scanf_fmt_level – Enumeration	The largest integer type supported by the scanf function group.
Scanf Supported linker_scanf_enabled – Boolean	Is scanf supported.

User Build Step Options

Property	Description

Link Patch Command linker_patch_build_command – String	A command to run after the link but prior to additional binary file generation. This property will have macro expansion applied to it with the additional \$(TargetPath) macro set to the output filepath of the linker command.
Link Patch Working Directory linker_patch_build_command_wd – String	The working directory where the link patch command is run. This property will have macro expansion applied to it.
Post-Link Command linker_post_build_command – String	A command to run after the link command has completed. This property will have macro expansion applied to it with the additional \$(TargetPath) macro set to the output filepath of the linker command.
Post-Link Working Directory linker_post_build_command_wd – String	The working directory where the post-link command is run. This property will have macro expansion applied to it.
Pre-Link Command linker_pre_build_command – String	A command to run before the link command. This property will have macro expansion applied to it.
Pre-Link Working Directory linker_pre_build_command_wd – String	The working directory where the pre-link command is run. This property will have macro expansion applied to it.

Staging Project Properties

Staging Options

Property	Description
Output File Path stage_output_filepath – String	The output file path the stage command will create. This property will have macro expansion applied to it.
Set To Read-only stage_set_readonly – Boolean	Set the output file to read only or read/write.
Stage Command stage_command – String	The command to execute. This property will have macro expansion applied to it with the additional \$(StageOutputFilePath) macro set to the output filepath of the stage command.
Stage Command Working Directory stage_command_wd – String	The working directory in which the stage command is run. This property will have macro expansion applied to it.
Stage Post-Build Command stage_post_build_command – String	The command to execute after staging commands have executed. This property will have macro expansion applied to it.
Stage Post-Build Command Working Directory stage_post_build_command_wd – String	The working directory where the post build command runs. This property will have macro expansion applied to it.

Build Macros

Build Macro Values

Property	Description
\$(CombiningOutputFilePath) \$(CombiningOutputFilePath) – String	The full path of the output file of the combining command.
\$(CombiningRelInputPaths) \$(CombiningRelInputPaths) – String	The relative inputs to the combining command.
\$(Configuration) \$(Configuration) – String	The build configuration e.g. ARM Flash Debug.
\$(EXE) \$(EXE) – String	The default file extension for an executable file including the dot e.g. .elf.
\$(InputDir) \$(InputDir) – String	The absolute directory of the input file.
\$(InputExt) \$(InputExt) – String	The extension of an input file not including the dot e.g. cpp.
\$(InputFileName) \$(InputFileName) – String	The name of an input file relative to the project directory.
\$(InputName) \$(InputName) – String	The name of an input file relative to the project directory without the extension.
\$(InputPath) \$(InputPath) – String	The absolute name of an input file including the extension.
\$(IntDir) \$(IntDir) – String	The macro-expanded value of the Intermediate Directory project property.
\$(LIB) \$(LIB) – String	The default file extension for a library file including the dot e.g. .lib.
\$(OBJ) \$(OBJ) – String	The default file extension for an object file including the dot e.g. .o.
\$(OutDir) \$(OutDir) – String	The macro-expanded value of the Output Directory project property.
\$(PackageExt) \$(PackageExt) – String	The file extension of a CrossWorks package file e.g. hzq.
\$(ProjectDir) \$(ProjectDir) – String	The absolute value of the Project Directory project property of the current project. If this isn't set then the directory containing the solution file.
\$(ProjectName) \$(ProjectName) – String	The project name of the current project.
\$(ProjectNodeName) \$(ProjectNodeName) – String	The name of the selected project node.

\$(RelInputPath) \$(RelInputPath) – String	The relative path of the input file to the project directory.
\$(RootOutDir) \$(RootOutDir) – String	The macro-expanded value of the Root Output Directory project property.
\$(RootRelativeOutDir) \$(RootRelativeOutDir) – String	The relative path to get from the path specified by the Output Directory project property to the path specified by the Root Output Directory project property.
\$(SolutionDir) \$(SolutionDir) – String	The absolute path of the directory containing the solution file.
\$(SolutionExt) \$(SolutionExt) – String	The extension of the solution file without the dot.
\$(SolutionFileName) \$(SolutionFileName) – String	The filename of the solution file.
\$(SolutionName) \$(SolutionName) – String	The basename of the solution file.
\$(SolutionPath) \$(SolutionPath) – String	The absolute path of the solution file.
\$(StageOutputFilePath) \$(StageOutputFilePath) – String	The full path of the output file of the stage command.
\$(TargetPath) \$(TargetPath) – String	The full path of the output file of the link or compile command.

System Macros

System Macro Values

Property	Description
\$(Date) \$(Date) – String	Day Month Year e.g. 21 June 2011.
\$(DateDay) \$(DateDay) – String	Year e.g. 2011.
\$(DateMonth) \$(DateMonth) – String	Month e.g. June.
\$(DateYear) \$(DateYear) – String	Day e.g. 21.
\$(DesktopDir) \$(DesktopDir) – String	Path to users desktop directory e.g. c:/users/mpj/Desktop.
\$(DocumentsDir) \$(DocumentsDir) – String	Path to users documents directory e.g. c:/users/mpj/Documents.
\$(HomeDir) \$(HomeDir) – String	Path to users home directory e.g. c:/users/mpj.
\$(HostArch) \$(HostArch) – String	The CPU architecture that CrossStudio is running on e.g. x86.
\$(HostDLL) \$(HostDLL) – String	The file extension for dynamic link libraries on the CPU that CrossStudio is running on e.g. .dll.
\$(HostEXE) \$(HostEXE) – String	The file extension for executables on the CPU that CrossStudio is running on e.g. .exe.
\$(HostOS) \$(HostOS) – String	The name of the operating system that CrossStudio is running on e.g. win.
\$(Micro) \$(Micro) – String	The CrossWorks target e.g. ARM.
\$(PackagesDir) \$(PackagesDir) – String	Path to the users packages directory e.g. c:/users/mpj/AppData/Local/Rowley Associates Limited/CrossWorks for ARM/packages.
\$(Platform) \$(Platform) – String	The CrossWorks target e.g. arm
\$(SamplesDir) \$(SamplesDir) – String	Path to the samples subdirectory of the packages directory e.g. c:/users/mpj/AppData/Local/Rowley Associates Limited/CrossWorks for ARM/packages/samples.

\$(StudioDir) \$(StudioDir) – String	The install directory of CrossStudio e.g. c:/Program Files(x86)/Rowley Associates Limited/CrossWorks for ARM 2.0.
\$(StudioMajorVersion) \$(StudioMajorVersion) – String	The major release version of CrossStudio e.g. 2.
\$(StudioMinorVersion) \$(StudioMinorVersion) – String	The minor release version of CrossStudio e.g. 0.
\$(StudioName) \$(StudioName) – String	The name of CrossStudio e.g. CrossStudio for ARM.
\$(StudioRevision) \$(StudioRevision) – String	The release revision of CrossStudio e.g. 11.
\$(StudioUserDir) \$(StudioUserDir) – String	The directory containing the packages directory e.g. c:/users/mpj/AppData/Local/Rowley Associates Limited/CrossWorks for ARM.
\$(TargetID) \$(TargetID) – String	ID number representing the CrossWorks target.
\$(TargetsDir) \$(TargetsDir) – String	Path to the targets subdirectory of the packages directory e.g. c:/users/mpj/AppData/Local/Rowley Associates Limited/CrossWorks for ARM/packages/targets.
\$(Time) \$(Time) – String	Hour:Minutes:Seconds e.g. 15:34:03.
\$(TimeHour) \$(TimeHour) – String	Hour e.g. 15.
\$(TimeMinute) \$(TimeMinute) – String	Hour e.g. 34.
\$(TimeSecond) \$(TimeSecond) – String	Hour e.g. 03.

JavaScript Classes Reference

Script Classes

Scripting	Scripting utility class.
Debug	Debugger access class.
WScript Compatibility	Windows Script Host compatibility class.

CWSys

The following table lists the CWSys object's member functions.

CWSys.copyFile(srcPath, destPath) copies file <i>srcPath</i> to <i>destPath</i> .
CWSys.crc32(array) returns the CRC32 value of the byte array <i>array</i> .
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.packU32Signature(array, offset, number, le) packs <i>number</i> into the <i>array</i> at <i>offset</i> .
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.unpackU32Signature(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.getlinenumber() 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.wait(ms) wait <i>ms</i> milliseconds for a breakpoint and return the number of the breakpoint that hit.
Debug.where() display call stack.

WScript

The following table lists the WScript object's member functions.

WScript.Echo(s) echos string <i>s</i> to the output terminal.

Code editor command summary

The following table summarizes the keystrokes and corresponding menu items for code editor commands:

Keystrokes	Menu	Description
Up		Move the caret one line up.
Down		Move the caret one line down.
Left		Move the caret one character to the left.
Right		Move the caret one character to the right.
Home		Move the caret to the start of the current line.
End		Move the caret to the end of the current line.
PageUp		Move the caret on page up.
PageDown		Move the caret one page down.
Ctrl+Up		Scroll the document down one line.
Ctrl+Down		Scroll the document up one line.
Ctrl+Left		Move the caret to the start of the previous word.
Ctrl+Right		Move the caret to the start of the next word.
Ctrl+Home		Move the caret to the start of the document.
Ctrl+End		Move the caret to the end of the document.
Ctrl+PageUp		Move the caret to the top of the window.
Ctrl+PageDown		Move the caret to the bottom of the window.
Enter Return		Insert a new line and move the caret to an appropriate position on the next line dependant on the indent settings.
Shift+Up		Extend the current selection up by one line.
Shift+Down		Extend the current selection down by one line.
Shift+Left		Extend the current selection left by one character.

Shift+Right		Extend the current selection right by one character.
Shift+Home		Extend the current selection to the beginning of the current line.
Shift+End		Extend the current selection to the end of the current line.
Shift+PageUp		Extend the current selection up by one page.
Shift+PageDown		Extend the current selection down by one page.
Ctrl+Shift+Left		Extend the current selection to the beginning of the previous word.
Ctrl+Shift+Right		Extend the current selection to the end of the next word.
Ctrl+Shift+Home		Extend the current selection to the beginning of the file.
Ctrl+Shift+End		Extend the current selection to the end of the file.
Ctrl+Shift+PageUp		Extend the current selection to the top of the window.
Ctrl+Shift+PageDown		Extend the current selection to the end of the window.
Ctrl+Shift+]		Select the text contained within the nearest delimiter pair.
Ctrl+A		Select the entire document.
Ctrl+F8		Select the current line.
	Edit Advanced Sort Ascending	Sort the lines contained within the current selection into ascending order.
	Edit Advanced Sort Descending	Sort the lines contained within the current selection into descending order.
Ctrl+C Ctrl+Insert	Edit Copy	Copy the current selection into the clipboard.
Ctrl+X Shift+Delete	Edit Cut	Copy the current selection into the clipboard and remove the selected text from the document.
Ctrl+V Shift+Insert	Edit Paste	Insert the contents of the clipboard into the document at the current caret position.
Ctrl+L		Cut the current line <i>or selection</i> .
Ctrl+Shift+L		Delete the current line <i>or selection</i> .

	Edit Clipboard Clear Clipboard	Empty the current contents of the clipboard.
Ctrl+F2	Edit Bookmarks Toggle Bookmark	Add or remove a bookmark to the current line.
F2	Edit Bookmarks Next Bookmark	Move the caret to the next bookmark.
Shift+F2	Edit Bookmarks Previous Bookmark	Move the caret to the previous bookmark.
	Edit Bookmarks First Bookmark	Move the caret to the first bookmark in the document.
	Edit Bookmarks Last Bookmark	Move the caret to the last bookmark in the document.
Ctrl+Shift+F2	Edit Bookmarks Clear All Bookmarks	Remove all bookmarks from the document.
Alt+F2		Add a permanent bookmark on the current line.
Ctrl+F	Edit Find	Display the find dialog.
Ctrl+H	Edit Replace	Display the replace dialog.
F3		Find the next occurrence of the previous search ahead of the current caret position.
Shift+F3		Find the next occurrence of the previous search behind the current caret position.
Ctrl+]		Find the matching delimiter character for the nearest delimiter character on the current line.
Ctrl+F3		Search up the document for currently selected text.
Ctrl+Shift+F3		Search down the document for the currently selected text.
Ctrl+G, Ctrl+L		Display the goto line dialog.
Backspace		Delete the character to the left of the caret position.
Delete		Delete the character to the right of the caret position.
Ctrl+Backspace		Delete from the caret position to the start of the current word.
Ctrl+Delete		Delete from the caret position to the end of the current word.
Ctrl+L		Delete current line.

Ctrl+Alt+L		Delete from the caret position to the end of the line.
Alt+Shift+L		Delete from the caret position to the next blank line.
Tab	Edit Advanced Increase Line Indent	Either advance the caret to the next indent position or, if there is selected text, indent each line of the selection.
Shift+Tab	Edit Advanced Decrease Line Indent	Either move the caret to the previous indent position or, if there is selected text, unindent each line of the selection.
Alt+Right		Indent the current line.
Alt+Left		Unindent the current line.
Ctrl+S	File Save	Save the current file.
	File Save As	Save the current file under a different file name.
Ctrl+Shift+S	File Save All	Save all the files.
Ctrl+P	File Print	Print the current file.
Ctrl+U	Edit Advanced Make Selection Lowercase	Either change the current character to lowercase or, if there is selected text, change all characters within the selection to lowercase.
Ctrl+Shift+U	Edit Advanced Make Selection Uppercase	Either change the current character to uppercase or, if there is selected text, change all characters within the selection to uppercase.
Ctrl+/ 	Edit Advanced Comment	If there is a selection, adds a comment to the start of each selected line. If there is no selection, adds a comment to the start of the line the caret is on.
Ctrl+Shift+/ 	Edit Advanced Uncomment	If there is a selection, removes any comment from the start of each selected line. If there is no selection, removes any comment from the start of the line the caret is on.
Ctrl+Z or Alt+Backspace	Edit Undo	Undoes the last operation.
Ctrl+Y	Edit Redo	Redoes the last operation.
Insert		Enable or disable overwrite mode.
Ctrl+Shift+T		Swap the current word with the previous word or, if there is no previous word, the next word.

Alt+Shift+T		Swap the current line with the previous line or, if there is no previous line, the next line.
Ctrl+Alt+J		Appends the line below the caret onto the end of the current line.
	Edit Advanced Tabify Selection	Replace whitespace with appropriate tabs within the current selection.
	Edit Advanced Untabify Selection	Remove tabs from within the current selection.
	Edit Advanced Visible Whitespace	Enable or disable visible whitespace.
	Edit Advanced Toggle Read Only	Toggle the write permissions of the current file.

Binary editor command summary

The following table summarizes the keystrokes and corresponding menu items for binary editor commands:

Keystrokes	Menu	Action
Up		Move the caret 16 bytes back.
Down		Move the caret 16 bytes forward.
Left		Move the caret one byte back.
Right		Move the caret one byte forward.
Home		Move the caret to the start of the current line of bytes.
End		Move the caret to the end of the current line of bytes.
Page Up		Move the caret one page up.
Page Down		Move the caret one page down.
Ctrl+Home		Move the caret to address 0.
Ctrl+End		Move the caret to the address of the last byte in the file.
Ctrl+Up		Move the view up one line.
Down		Move the view down one line.
Ctrl+Left		Move the caret 4 bytes back.
Ctrl+Right		Move the caret 4 bytes forward.
Ctrl+F	Edit Find	Display the find dialog.
F3		Find the next occurrence of the value most recently searched for.
Backspace		Removes the byte in the address before the caret position.
Delete		Removes the currently selected byte.
Ctrl+S	File Save	Save the current file.
	File Save As	Save the current file under a different file name. WARNING the current version of the binary editor continues to display the original file name after a "Save as" operation.
Ctrl+Z	Edit Undo	Undoes the last operation.
Ctrl+Y	Edit Redo	Redoes the last undone operation.
Insert		Enable or disable overwrite mode.

Ctrl+T		When in text input mode the currently selected byte will be replaced with the ASCII character code for the input key. e.g. 0F would become 66 when 'f' is pressed. When not in text input mode the currently selected byte is replaced with the HEX value of the input keys. e.g. 00 would become 0F when 'f' is pressed. 00 would become FA if the 'f' then 'a' keys were pressed.
	Edit Advanced Toggle Read Only	Toggle the write permissions of the current file.
Ctrl+E		Allows the file to be a fixed size or a variable size
Down or Right		Extends the file when the last address is selected and the file is write enabled. the new bytes will be initialised to 00.

Frequently asked questions

Assembly Code

Can I put inline assembly code into my C source files?

No, we don't support inline assembly code. You can use the compiler intrinsic `_OPC` which will emit a word into the instruction stream.

For example:

```
#include <in430.h>
_OPC(0x4302); // mov.w #0, sr
```

Memory Usage

How can I find out how big my program is?

You can use the symbol browser window to find out the size of your program:

- Open up the **Symbol Browser** using **View | Symbol Browser** or typing **Ctrl+Alt+Y**.
- Click the arrow on the first button on the symbol browser's toolbar.
- Select **Group By Section** from the menu.

You will now see the sections that comprise your program together with their addresses and sizes. You can use the symbol browser to view the range of data items and see how much code each individual function takes up.

How can I find out where functions or variables have been placed in memory by the linker?

You can use the symbol browser window to find out where symbols are placed in memory:

- Open up the **Symbol Browser** using **View | Symbol Browser** or typing **Ctrl+Alt+Y**.
- Click the arrow on the first button on the symbol browser's toolbar.
- Select **Group By Type** from the menu.

You will see a list of groups each containing symbols of a particular type, such as functions, variables and labels. Within each of these groups will be listed each symbol and the address range it occupies in memory.

Glossary

The following terms are in common use and are used throughout the CrossWorks documentation:

Active project

The project that is currently selected in the **Project Explorer**. The **Build** tool bar contains a dropdown and the **Project > Set Active Project** menu contains an item that display the active project. You can change the active project using either of these elements.

Active configuration

The configuration that is currently selected for building. The **Build** too bar contains a dropdown and the **Build > Set Active Build Configuration** menu display the active configuration. You can change the active configuration using either of these elements.

Assembler

A program that translates low-level assembly language statements into executable machine code. See [Assembler Reference](#).

Compiler

A program that translates high-level statement into executable machine code. See [C Compiler Reference](#).

Integrated development environment

A program that supports editing, managing, building, and debugging your programs within a single environment.

Linker

A program that combines multiple relocatable object modules and resolves inter-module references to produce an executable program. See [Linker Reference](#).

Project explorer

A docking indow that contains a visual representation of the project. See [Project Explorer](#).